Unprivileged Black-box Detection of User-space Keyloggers

Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo, *Senior Member, IEEE*

Abstract—Software keyloggers are a fast growing class of invasive software often used to harvest confidential information. One of the main reasons for this rapid growth is the possibility for unprivileged programs running in user space to eavesdrop and record all the keystrokes typed by the users of a system. The ability to run in unprivileged mode facilitates their implementation and distribution, but, at the same time, allows one to understand and model their behavior in detail. Leveraging this characteristic, we propose a new detection technique that simulates carefully crafted keystroke sequences in input and observes the behavior of the keylogger in output to unambiguously identify it among all the running processes. We have prototyped our technique as an unprivileged application, hence matching the same ease of deployment of a keylogger executing in unprivileged mode. We have successfully evaluated the underlying technique against the most common free keyloggers. This confirms the viability of our approach in practical scenarios. We have also devised potential evasion techniques that may be adopted to circumvent our approach and proposed a heuristic to strengthen the effectiveness of our solution against more elaborated attacks. Extensive experimental results confirm that our technique is robust to both false positives and false negatives in realistic settings.

Index Terms—Invasive Software, Keylogger, Security, Black-box, PCC

1 Introduction

Keyloggers are implanted on a machine to intentionally monitor the user activity by logging keystrokes and eventually delivering them to a third party [1]. While they are seldom used for legitimate purposes (e.g., surveillance/parental monitoring infrastructures), keyloggers are often maliciously exploited by attackers to steal confidential information. Many credit card numbers and passwords have been stolen using keyloggers [2], [3], which makes them one of the most dangerous types of spyware known to date.

Keyloggers can be implemented as tiny hardware devices or more conveniently in software. Software-based keyloggers can be further classified based on the privileges they require to execute. Keyloggers implemented by a kernel module run with full privileges in kernel space. Conversely, a fully unprivileged keylogger can be implemented by a simple user-space process. It is important to notice that a user-space keylogger can easily rely on documented sets of unprivileged APIs commonly available on modern operating systems. This is not the case for a keylogger implemented as a kernel module. In kernel space, the programmer must rely on kernel-level facilities to intercept all the messages dispatched by the keyboard driver, undoubtedly requiring a considerable effort and knowledge for an effective and bug-free implementation. Furthermore, a keylogger implemented as a user-space process is much easier to deploy since no special permission is required. A user can erroneously regard the keylogger as a harmless piece of software and being deceived in executing it. On the contrary, kernel-space keyloggers require a user with super-user privileges to consciously install and execute unsigned code within the kernel, a practice often forbidden by modern operating systems such Windows Vista or Windows 7.

In light of these observations, it is no surprise that 95% of the existing keyloggers run in user space [4]. 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.

In this paper, we propose a new approach to detect keyloggers running as unprivileged user-space processes. To match the same deployment model, our technique is entirely implemented in an unprivileged process. As a result, our solution is portable, unintrusive, easy to install, and yet very effective. In addition, the proposed detection technique is completely black-box, i.e., based on behavioral characteristics common to all keyloggers. In other words, our technique does not rely on the internal structure of the keylogger or the particular set of APIs used. For this reason, our solution is of general applicability. We have prototyped our approach and evaluated it against the most common free keyloggers [5]. Our approach has proven effective in all the cases. We have also evaluated the impact of false positives in practical scenarios. In the final part of this paper, we further validate our approach with

• *S. Ortolani and C. Giuffrida are with Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081HV Amsterdam, The Netherlands. E-mail:* {ortolani,giuffrida}@cs.vu.nl
• *B. Crispo is with University of Trento, Via Sommarive 14, 38050 Povo, Trento, Italy. E-mail:* crispo@disi.unitn.it

a homegrown keylogger that attempts to thwart our detection technique. Albeit already robust against the large majority of evasive behaviors, we also present and evaluate a heuristic against elaborated evasion strategies.

The structure of the paper is as follows. We start with an in-depth analysis of modern keyloggers in Section 2. We then introduce our approach in Section 3, detail its architecture in Section 4, and evaluate the resulting prototype in Section 5. Section 6 discusses the robustness against evasion techniques. We conclude with related work in Section 7 and final remarks in Section 8.

2 Internals of Modern Keyloggers

Breaching the privacy of an individual by logging his keystrokes can be perpetrated at many different levels. For example, an attacker with physical access to the machine might wiretap the hardware of the keyboard. A dishonest owner of an Internet café, in turn, may find it more convenient to purchase a software solution, install it on all the terminals, and have the logs dropped on his own machine. Depending on the setting, a keylogger can be implemented in many different ways. For instance, *external keyloggers* rely on some physical property, either the acoustic emanations produced by the user typing [6], or the electromagnetic emanations of a wireless keyboard [7]. *Hardware keyloggers* are still external devices, but are implemented as dongles placed in between keyboard and motherboard. All these strategies, however, require physical access to the target machine.

To overcome this limitation, software approaches are more commonly used. *Hypervisor-based keyloggers* (e.g., BluePill [8]) 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 in the chain and are often implemented as part of more complex rootkits. In contrast to hypervisor-based approaches, hooks are directly used to intercept buffer-processing events or other kernel messages.

Albeit 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 (even a single bug may lead to a kernel panic). *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. This is a feat 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.

User-space keyloggers can be further classified based on the scope of the hooked message/data structures. Since a system hosts multiple applications, keystrokes

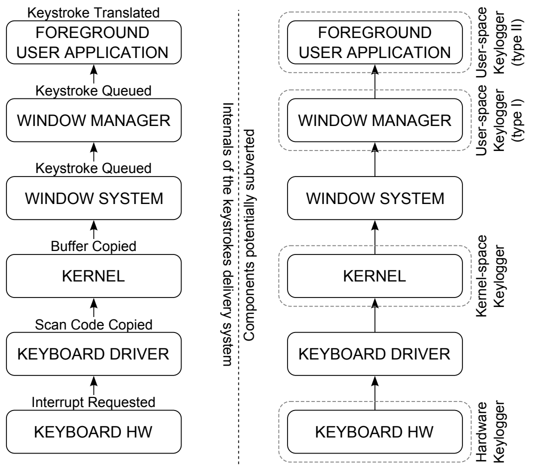


Fig. 1. The delivery phases of a keystroke, and the components potentially subverted (we omit hypervisor-based approaches for the sake of clarity).

can be intercepted either globally (i.e., for all the applications) or locally (i.e., within the application). We term these two classes of user-space keyloggers *type I* and *type II*. Figure 1 shows the proposed classification: the left pane shows the process of delivering a keystroke to the intended application, whereas the right pane highlights the particular component subverted by each type of keylogger. Both types can be easily implemented in Windows, while the facilities available in Unix-like OSes—`X11` and `GTK` required—allow for a straightforward implementation of the more invasive *type I* keyloggers.

Table 1 presents a list of all the APIs that can be used to implement a user-space keylogger. 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 last parameter (`thread_id`) to 0 (which subscribes to any keyboard event). For `gdk_window_add_filter()`, it is sufficient to set the handler of the monitored window to `NULL`. The class of functions `Get*State()`, `XQueryKeymap()`, and `inb(0x60)` query the state of the keyboard and return a vector with the state of all (one in case of `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. We have not found any example of this particular class of keyloggers in Unix-like OSes.

Since some of the APIs have just 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

TABLE 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.

Type	API	Comments
	Windows APIs	
Type I	`SetWindowsHookEx(WH_KEYBOARD_LL, ..., 0)`	The keylogging procedure is given as an argument.
	`GetAsyncKeyState()`	Poll-based.
Type II	`SetWindowsHookEx(WH_KEYBOARD, ..., 0)`	The keylogging procedure is given as an argument.
	`GetKeyboardState()`	Poll-based.
	`GetKeyState()`	Poll-based.
	`SetWindowLong(..., GWL_WNDPROC, ...)`	Overwrites the default procedure that deals with application messages.
	Intercepting {`Dispatch,Get,Translate`}`Message()`	Manual instrumentation of Win32 APIs.
	Unix-like APIs	
Type I	`gdk_window_add_filter(NULL, ...)`	The keylogging procedure is given as argument. `GTK` API.
	`inb(0x60)`	Poll-based and available only to the super-user.
	`XQueryKeymap()`	Poll-based. `X11` API.

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.

We can draw three 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. This design choice is dictated by the necessity to support such functionalities for legitimate applications. The following are three simple scenarios in which the ability to intercept arbitrary keystrokes is a functional requirement: (1) keyboards with additional special-purpose keys; (2) window managers with system-defined shortcuts; (3) background user applications whose execution is triggered by user-defined shortcuts (for instance, an application handling multiple virtual workspaces requires hot keys that must not be overridden by other applications). All these functionalities can be implemented with all the APIs we presented so far (with the exception of `inb(0x60)`, which is reserved to the super user and tailored to low-level tasks). 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 100 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 privilege is required to install a keylogger. There is no need to look for rather specific exploits to execute arbitrary privileged code.
- **Unprivileged execution**. The keylogger is hardly noticeable at all during normal execution. The executable does not need to acquire privileged rights.

3 Our Approach

Our approach is explicitly focused on designing a detection technique for unprivileged user-space keyloggers. Unlike other classes of keyloggers, a user-space keylogger is a background process which registers 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. 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 outside our threat model and cannot be identified using our detection technique. Also note that a background keylogger cannot spawn a foreground application and steal the current application focus on demand without the user immediately noticing.

Our model is based on these observations and 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. 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 forcefully move 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.

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.

4 Architecture

Our design is based on five different components as depicted in Figure 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

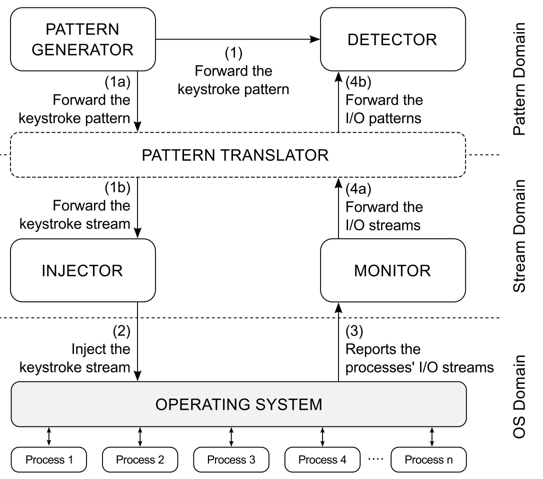

PATTERN GENERATOR
DETECTOR
(1) Forward the keystroke pattern
(1a) Forward the keystroke pattern
(4b) Forward the I/O patterns
PATTERN TRANSLATOR
(1b) Forward the keystroke stream
(4a) Forward the I/O streams
INJECTOR
MONITOR
(2) Inject the keystroke stream
(3) Reports the processes' I/O streams
OPERATING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

Fig. 2. The different components of our architecture.

recorded to the pattern translator for further analysis. In the *Pattern Domain*, the input stream and the output stream are both represented in a more abstract form, termed *Abstract Keystroke Pattern* (AKP). A pattern in the AKP form is a discretized and normalized representation of a stream. Adopting a compact and uniform representation is advantageous for several reasons. First, this allows the pattern generator to exclusively focus on generating an input pattern that follows a desired distribution of values. Details on how to inject a particular distribution of keystrokes into the system are offloaded to the pattern translator and the injector. Second, the same input pattern can be reused to produce and inject several input streams with different properties but following the same underlying distribution. Finally, the ability to reason over abstract representations simplifies the role of the detector that only receives an input pattern and an output pattern and makes the final decision on whether detection should or should not be triggered.

4.1 Injector

The role of the injector is to inject the input stream into the system, simulating the behavior of a 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 real user. 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`.

4.2 Monitor

The monitor is responsible to record 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.

4.3 Pattern Translator

The role of the pattern translator is to transform an AKP into a stream and vice-versa, given a set of 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 in the interval $[0, 1]$, where 0 and 1 reflect the predefined minimum and maximum number of keystrokes in a given time interval. 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 $\bar{R}_i = \frac{P_i \cdot (K_{max} - K_{min}) + K_{min}}{T}$. 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 $P_i = \frac{\bar{R}_i \cdot T - K_{min}}{K_{max} - K_{min}}$ 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 4.4.

4.4 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), one of the most widely used correlation measures [9]. Given two discrete sequences described by two patterns P and Q with N samples, the PCC is defined as [9]:

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

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 [10]. 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 other correlation metrics, the PCC measures the strength of a linear relationship between two series of samples, ignoring any non-linear association. In our setting, a linear dependence well approximates the relationship between the input pattern and an output pattern produced by a keylogger. The 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, 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. 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.

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 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 to several possible data transformations.

A fundamental factor to consider is, however, the number of samples collected. While we would like to shorten the duration of the detection algorithm, there is a clear tension between the length of the patterns and the reliability of the resulting value of the PCC. A very small number of samples can lead to unstable results. A larger number of samples is beneficial especially in case of disturbing factors. As reported in [11], 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 in Section 5. 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 [12]. In other words, to avoid false positives, 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.

4.5 Pattern Generator

Our pattern generator is designed to support several pattern generation algorithms. More specifically, the pattern generator can leverage any algorithm producing a valid pattern in AKP form. We now 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 [11]. 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 when the two patterns tend to grow apart from their respective means. As a consequence, the more closely to their respective means the patterns are distributed, the less accurate the resulting PCC. In the extreme case of no variability, 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 coherent under the assumption that the selected input pattern presents a reasonable level of variability, and poor correlation should naturally be expected against other low-variability patterns.

A robust pattern generation algorithm should allow for a minimum number of false positive. When the chosen input pattern happens to closely resemble the I/O behavior of some benign process, 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 realistic I/O workloads for PC users is generally considerably low over small time intervals. The results presented in [13] 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. This suggests that the I/O behavior of one or more processes is in general very poorly correlated with other I/O distributions.

The problem of designing a pattern generation algorithm that minimizes 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:

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

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 [14].

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:

$$c\,(i, j) = \sum_t \frac{\left(S_i - \bar{S}\right)\left(P_j^t - \bar{P}^t\right)}{\sigma_S \sigma_{P^t}}, \tag{3}$$

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 [14]. 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 [15]. 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)**. Each sample is generated at random.

- **Random with fixed range (RFR)**. The pattern is a random permutation of a series of samples uniformly distributed over the interval $[0,1]$. This is 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 and idle periods at minimum.
- **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.

5 Evaluation

To evaluate the proposed detection technique, we implemented a prototype based on the ideas described in the paper. Written in `C#` in 7000 LoC, it 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 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. 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 showed 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.

5.1 Keylogger detection

To evaluate the ability to detect real-world keyloggers, we experimented with all the keyloggers from the top monitoring free software list [5], 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 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. 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 5.2 and Section 5.3.

TABLE 2
Detection results.

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

Table 2 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. 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 5.2 demonstrates, our approach deal with these nuisances transparently with no effect on the resulting PCC.

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.

5.2 False negatives

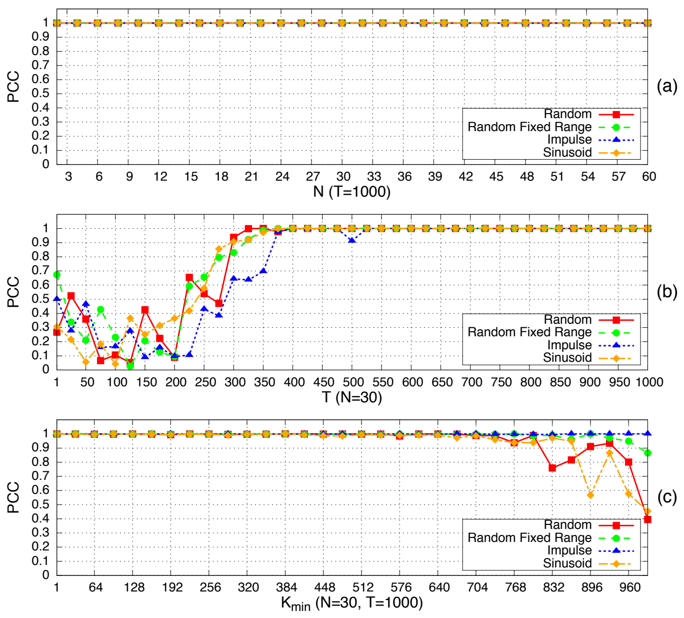


Fig. 3. Impact of N, T, and K_{min} on the PCC.

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. 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 3a and 3b 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 false negatives.

In contrast, Figure 3b 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 3c, in turn, shows the impact of K_{min} on the PCC (with K_{max} still constant). The results confirm our observations in Section 4.4, 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

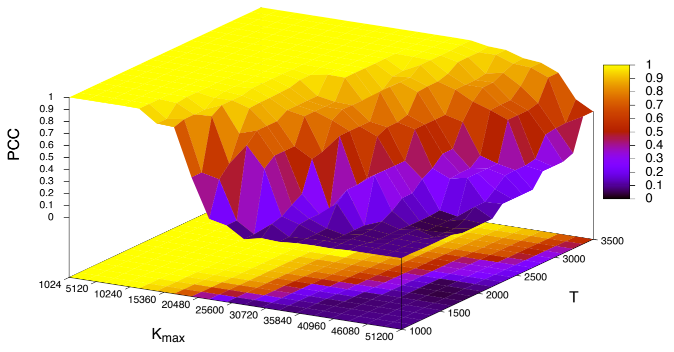


Fig. 4. Impact of K_{max} and T on the PCC.

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 4 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 several possible input-output transformations. First, we experimented with a keylogger using a nontrivial fixed-length encoding for keystrokes. Figure 5a 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 4.4, 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

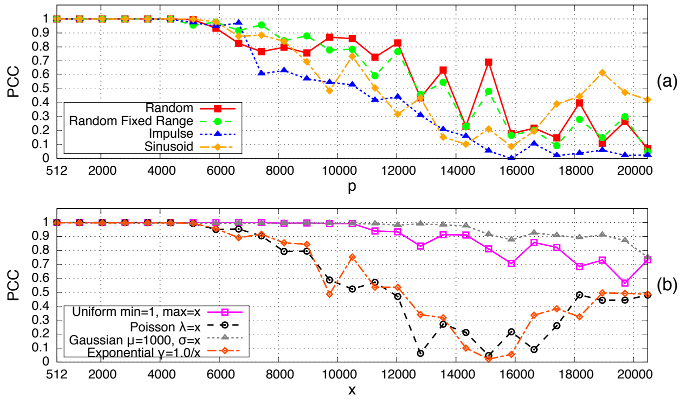


Fig. 5. Impact of different classes of noise on the PCC.

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 experiment increasing the value of a characteristic parameter (reported on the x-axis). Results are depicted in Figure 5b. As observed in Figure 5a, 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 4 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.

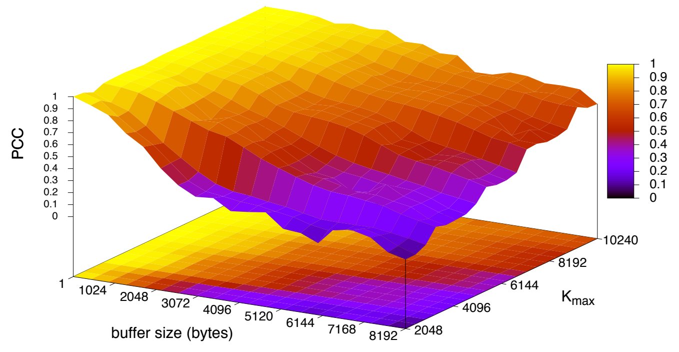


Fig. 6. Detection of a keylogger buffering its output.

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 6 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.

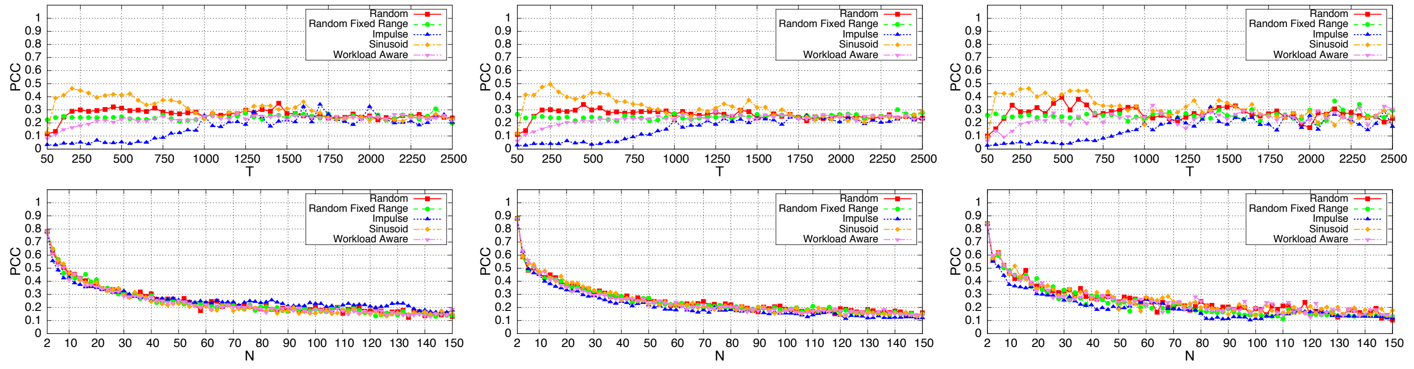
5.3 False positives

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 [16]. 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 [1] running in the background, and no input allowed by the user.

For each scenario, we repeatedly reproduced the synthetic workloads on a number of different machines and collected I/O traces of all the running processes for several possible sampling intervals T. Each trace was stored as a set of output patterns and broken down into k consecutive chunks of N samples. Every experiment was repeated over $k/2$ rounds, once for each pair of consecutive chunks. At each round, the output patterns from the first chunk were used to train our workload-aware pattern generation algorithm, while the second chunk was used for testing. In the testing phase, we measured the maximum PCC between every generated input pattern of length N and every output pattern

1. Skype 4, Pidgin 2.6.3, Dropbox 0.6, Firefox 3.5.7, Google Chrome 5, Avira Antivir Personal 9, Comodo Firewall 3.13, and VideoLAN 1.0.5.



(a) Idle workload. (b) Internet workload. (c) Office workload.

Fig. 7. Impact of N and T on the PCC measured with our prototype keylogger against different workloads.

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 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 strongly suggests that, independently of the value of N, 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 [13]. 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 cases, 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.

6 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.

6.1 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.

6.2 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 [17]. 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 used in real-world scenarios.

6.3 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, and therewith perform a separate detection run thwarting this 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. Recent work demonstrates that building such a system is feasible in practice (with reasonable overhead) using standard operating system facilities [18].

6.4 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 pattern 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. 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 nonrandom 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 [19]. This technique, often used to compare time series, warps sequences in the time dimension to determine a measure of similarity independent of non-linear variations in the time dimensions.

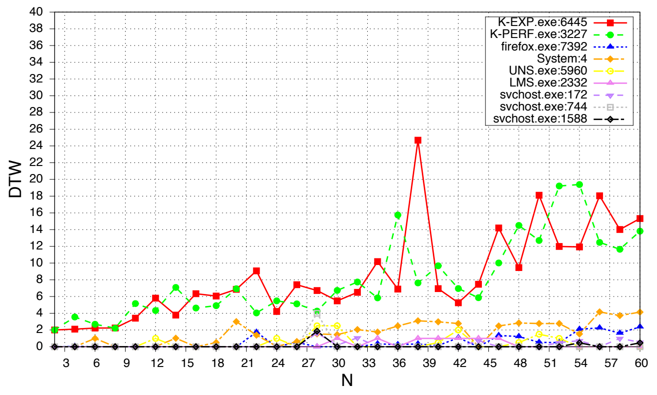


Fig. 8. Impact of N on the DTW.

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 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 8 depicts the DTW computed by our two-step heuristic for different processes and increasing values of N. 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 that we are investigating as part of our ongoing work [20]. 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 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.

7 Related Work

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. Behavior-based spyware detection has been first introduced by Kirda et al. in [21]. 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 malware. Their analysis models malicious behavior in terms of API calls invoked in response to browser events. Those used by keyloggers, however, are also commonly used by legitimate programs. Their approach is therefore prone to false positives, which can only be mitigated with continuously updated whitelists.

Other keylogger-specific approaches have suggested detecting the use of well-known keystroke interception APIs. Aslam et al. [22] 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. [23] 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 a new message type or periodically registering a new hook to obtain the highest priority in the hook chain.

Closer to our approach is the solution proposed by AlHammadi et al. in [24]. 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.

A similar quantitative and privileged technique is sketched by Han et al. in [25]. Unlike the solution presented in [24], 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 [17], [26] the authors propose a combination of concrete and symbolic execution for the task. Their strategy aims to explore all the possible execution paths that a malware can possibly exhibit during execution. As the authors in [17] 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.

8 Conclusions

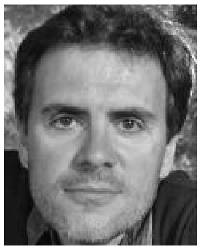
In this paper, we presented an unprivileged black-box approach for accurate detection of the most common keyloggers, i.e., user-space keyloggers. We modeled the behavior of a keylogger by surgically correlating the input (i.e., the keystrokes) with the output (i.e., the I/O patterns produced by the keylogger). In addition, we augmented our model with the ability to artificially inject carefully crafted keystroke patterns. We then discussed the problem of choosing the best input pattern to improve our detection rate. Subsequently, we presented an implementation of our detection technique on Windows, arguably the most vulnerable OS to the threat of keyloggers. To establish an OS-independent architecture, we also gave implementation details for other operating systems. We successfully evaluated our prototype system against the most common free keyloggers [5], with no false positives and no false negatives reported. Other experimental results with a homegrown keylogger demonstrated the effectiveness of our technique in the general case. While attacks to our detection technique are possible and have been discussed at length in Section 6, we believe our approach considerably raises the bar for protecting the user against the threat of keyloggers.

References

[1] T. Holz, M. Engelberth, and F. Freiling, "Learning more about the underground economy: A case-study of keyloggers and dropzones," *Proc. of the 14th European Symposium on Research in Computer Security*, pp. 1–18, 2009.
[2] San Jose Mercury News, "Kinkois spyware case highlights risk of public internet terminals," http://www.siliconvalley.com/mld/siliconvalley/news/6359407.htm.
[3] N. Strahija, "Student charged after college computers hacked," http://www.xatrix.org/article2641.html.
[4] N. Grebennikov, "Keyloggers: How they work and how to detect them," http://www.viruslist.com/en/analysis?pubid=204791931.
[5] Security Technology Ltd., "Testing and reviews of keyloggers, monitoring products and spyware," http://www.keylogger.org.
[6] L. Zhuang, F. Zhou, and J. D. Tygar, "Keyboard acoustic emanations revisited," *ACM Trans. on Information and System Security*, vol. 13, no. 1, pp. 1–26, 2009.
[7] M. Vuagnoux and S. Pasini, "Compromising electromagnetic emanations of wired and wireless keyboards," *Proc. of the 18th USENIX Security Symposium*, pp. 1–16, 2009.
[8] J. Rutkowska, "Subverting vista kernel for fun and profit," *Black Hat Briefings*, 2007.
[9] J. L. Rodgers and W. A. Nicewander, "Thirteen ways to look at the correlation coefficient," *The American Statistician*, vol. 42, no. 1, pp. 59–66, feb 1988.
[10] J. Benesty, J. Chen, and Y. Huang, "On the importance of the pearson correlation coefficient in noise reduction," *IEEE Trans. on Audio, Speech, and Language Processing*, vol. 16, no. 4, p. 757, 2008.
[11] L. Goodwin and N. Leech, "Understanding correlation: Factors that affect the size of r," *Experimental Education*, vol. 74, no. 3, pp. 249–266, 2006.
[12] J. Aldrich, "Correlations genuine and spurious in pearson and yule," *Statistical Science*, vol. 10, no. 4, pp. 364–376, 1995.
[13] W. Hsu and A. Smith, "Characteristics of I/O traffic in personal computer and server workloads," *IBM System Journal*, vol. 42, no. 2, pp. 347–372, 2003.
[14] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.
[15] G. Kochenberger, F. Glover, and B. Alidaee, "An effective approach for solving the binary assignment problem with side constraints," *Information Technology and Decision Making*, vol. 1, pp. 121–129, May 2002.
[16] BAPCO, "SYSmark 2004 SE," http://www.bapco.com.
[17] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," *Proc. of the 28th IEEE Symposium on Security and Privacy*, pp. 231–245, May 2007.
[18] S. Ortolani and B. Crispo, "Noisykey: Tolerating keyloggers via keystrokes hiding," *Proc. of the 7th USENIX Workshop on Hot Topics in Security*, p. to appear, 2012.
[19] H. Sakoe and S. Chiba, *Readings in speech recognition*, A. Waibel and K.-F. Lee, Eds. Morgan Kaufmann Publishers Inc., 1990.
[20] S. Ortolani, C. Giuffrida, and B. Crispo, "Klimax: Profiling memory write patterns to detect keystroke-harvesting malware," *Proc. of the 14th Intl. Symposium on Recent Advances in Intrusion Detection*, pp. 81–100, 2011.
[21] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-based spyware detection," *Proc. of the 15th USENIX Security Symposium*, pp. 273–288, 2006.
[22] M. Aslam, R. Idrees, M. Baig, and M. Arshad, "Anti-Hook Shield against the Software Key Loggers," *Proc. of the National Conference on Emerging Technologies*, p. 189, 2004.
[23] M. Xu, B. Salami, and C. Obimbo, "How to protect personal information against keyloggers," *Proc. of the 9th Intl. Conf. on Internet and Multimedia Systems and Applications*, 2005.
[24] Y. Al-Hammadi and U. Aickelin, "Detecting bots based on keylogging activities," *Proc. of the Third International Conference on Availability, Reliability and Security*, pp. 896–902, 2008.
[25] J. Han, J. Kwon, and H. Lee, "Honeyid: Unveiling hidden spywares by generating bogus events," *Proc. of the IFIP 23rd Intl. Information Security Conference*, pp. 669–673, 2008.
[26] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," *Advances in Information Security*, vol. 36, pp. 65–88, 2008.

Stefano Ortolani is a PhD student in the Computer Systems Section of the Department of Computer Science at the Vrije Universiteit, Amsterdam. His research covers security in computers and networks, privacy-enhancing technologies, and intrusion detection. Ortolani received a MSc in Computer Science from the Ca' Foscari University, Venice, Italy.

Cristiano Giuffrida is a PhD student in the Computer Systems Section of the Department of Computer Science at the Vrije Universiteit, Amsterdam. His research focuses on the design and implementation of secure and reliable systems. Giuffrida received a MEng in Computer Engineering from the University of Rome "Tor Vergata", Italy.



Bruno Crispo is an associate professor in the Department of Computer Science at Vrije Universiteit and at the University of Trento, Italy. His research interests are networks, distributed systems, cryptography, and security protocols. Crispo received a PhD in computer science from the University of Cambridge, United Kingdom. He is a senior member of the IEEE.