Bait Your Hook: A Novel Detection Technique for Keyloggers*

Stefano Ortolani[1], Cristiano Giuffrida[1], and Bruno Crispo[2]

[1] Vrije Universiteit, De Boelelaan 1081, 1081HV Amsterdam, The Netherlands
{ortolani,giuffrida}@cs.vu.nl
[2] University of Trento, Via Sommarive 14, 38050 Povo, Trento, Italy
crispo@disi.unitn.it

Abstract. Software keyloggers are a fast growing class of malware 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 of the users of the system. Such an ability to run in unprivileged mode facilitates their implementation and distribution, but, at the same time, allows to understand and model their behavior in detail. Leveraging this property, we propose a new detection technique that simulates carefully crafted keystroke sequences (the bait) in input and observes the behavior of the keylogger in output to univocally identify it among all the running processes. We have prototyped and evaluated this technique with some of the most common free keyloggers. Experimental results are encouraging and confirm the viability of our approach in practical scenarios.

Keywords: Keylogger, Malware, Detection, Black-box.

1 Introduction

Keyloggers are implanted on a machine to intentionally monitor the user activity by logging keystrokes and eventually sending them to a third party. While they are sometimes used for legitimate purposes (i.e. child computer monitoring), keyloggers are often maliciously exploited by attackers to steal confidential information. Many credit card numbers and passwords have been stolen using keyloggers [17,19], which makes them one of the most dangerous types of spyware. Keyloggers can be implemented as tiny hardware devices or more conveniently in software. A software acting as a keylogger can be implemented by means of two different techniques: as a kernel module or as a user-space process. It is important to notice that, while a kernel keylogger requires a privileged access to the system, a user-space keylogger can easily rely on documented sets of unprivileged API commonly available on modern operating systems. A user can

* This work has been partially funded by the EU FP7 IP Project MASTER (contract no. 216917) and by the PRIN project "Paradigmi di progettazione completamente decentralizzati per algoritmi autonomici".

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 198–217, 2010.

be easily deceived in installing it, and, since no special permission is required, the user can erroneously regard it as a harmless piece of software. On the contrary, kernel-level keyloggers require a considerable effort and knowledge for an effective and bug-free implementation. It is therefore no surprise that 95% of the existing keyloggers are user-space keyloggers [9]. Despite the number of frauds exploiting keyloggers (i.e. identity theft, password leakage, etc.) has increased rapidly in recent years, not many effective and efficient solutions have been proposed to address this problem. Preventing keyloggers to be implanted without limiting the behavior of the user is hardly an option in real-world scenarios. Traditional defensive mechanisms use fingerprinting or heuristic-based strategies similar to those used to detect viruses and worms. Unfortunately, results have been poor due to keyloggers' small footprint and their ability to hide.

In this paper, we propose a new approach to detect keyloggers running as unprivileged user-space processes. Our technique is entirely implemented in an unprivileged process. As a result, our solution is portable, unintrusive, easy to install, and yet very effective. Moreover, the proposed detection technique does not depend on the internal structure of the keylogger or the particular set of APIs used to capture the keystrokes. On the contrary, our solution is of general applicability, since it is based on behavioral characteristics common to all the keyloggers. We have prototyped our approach and evaluated it against the most common free keyloggers [15]. Our approach has proven effective in all the cases. We have also evaluated the impact of false positives and false negatives in practical scenarios.

The structure of the paper is as follows. We first present our approach and compare it to analogous solutions (Sec. 2). We then detail the architecture of our solution in Sec. 3 and evaluate the resulting prototype in Sec. 4. Sec. 5 discusses how a keylogger may counter our approach, and why our underlying model would still be valid. We conclude with related work in Sec. 6 and final remarks in Sec. 7.

2 Our Approach

Common misuse-based approaches rely on the ability to build a profile of the malicious system activity. Once the profile is available, any behavior that matches any known malicious patterns is reported as anomalous. However, applying analogous approaches to malware detection is not a trivial task. Building a malicious profile requires the ability to identify what a malicious behavior is. Unfortunately, such a behavior is normally triggered by factors that are neither easy to analyze nor feasible to control. In our approach, we explore the opposite direction. Rather than targeting the general case, we focus on designing a detection technique for a very peculiar category of malware, the keyloggers. In contrast to many other malicious programs, a keylogger has a very well defined behavior that is easy to model. In its simplest form, a keylogger eavesdrops each keystroke issued by the user and logs the content on a file on the disk. In this scenario, the events triggering the malicious activities are always known in advance and could be reproduced and controlled to some extent.

Our model is based on these observations and investigates the possibility to isolate 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 activities generated by the keylogger in output. To detect malicious behavior, 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. Whatever 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, there is potential to identify common patterns and trigger a detection. Furthermore, if we can select and enforce the input pattern preventively, we can better avoid possible evasion attempts. The key advantage of our approach is that it is centered around a black-box model that completely ignores the internals of a keylogger. As a result, our technique can deal with a large number of keyloggers transparently and has the potential to realize a fully-unprivileged detection system.

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 Architecture

Our design is based on five different components as depicted in Fig. 1: 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 keystrokes on 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, that acts as bridge between the *Stream Domain* and the *Pattern Domain*. Similarly, the monitor delivers the output stream recorded to the

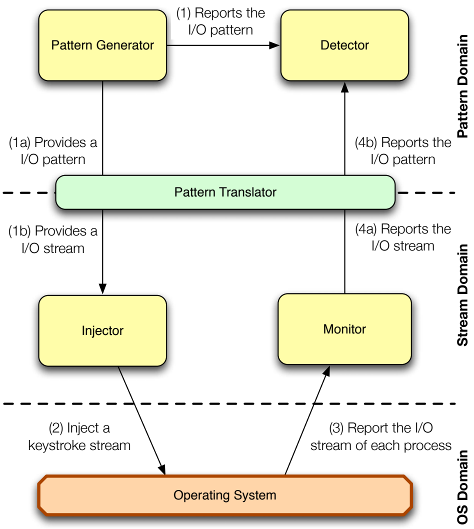

Pattern Generator
(1) Reports the I/O pattern
Detector
Pattern Domain
(1a) Provides a I/O pattern
(4b) Reports the I/O pattern
Pattern Translator
(1b) Provides a I/O stream
(4a) Reports the I/O stream
Injector
Monitor
Stream Domain
(2) Inject a keystroke stream
(3) Report the I/O stream of each process
Operating System
OS Domain

Fig. 1. The prototype's architecture divided in components and domains

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, we allow 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 whether detection should be triggered.

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, for example, this functionality is provided by the API call SendInput, available for several versions of the OS. All the other OSes supporting the X11 window server, the same functionality is available via the API call XTestFakeKeyEvent, part of the XTEST extension library.

3.2 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, that supports an efficient query-based interface. All the performance counters are constantly maintained up-to-date by the kernel. In WMI, the counter WriteTransferCount contains the total number of bytes the process wrote since its creation. 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. Linux, for instance, supports analogous performance counters since the 2.6.19 version.

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 predefined number of keystrokes per sample allowed.
K_{max} – the maximum predefined 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 $\overline{R}_i = \frac{P_i \cdot (K_{max} - K_{min}) + K_{min}}{T}$. 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 $P_i = \frac{\overline{R}_i \cdot T - K_{min}}{K_{max} - K_{min}}$ where $\overline{R}_i$ 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 more detail in Sec. 3.4.

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 [18]. Given two discrete sequences described by two patterns P and Q with N samples, the PCC is defined as [18]:

$$r = \frac{\mathrm{cov}(P,Q)}{\sigma_p \cdot \sigma_q} = \frac{\sum_{i=1}^{N}(P_i - \overline{P})(Q_i - \overline{Q})}{\sqrt{\sum_{i=1}^{N}(P_i - \overline{P})^2}\sqrt{\sum_{i=1}^{N}(Q_i - \overline{Q})^2}} \tag{1}$$

where $\mathrm{cov}(P,Q)$ is the sample covariance, σ_p and σ_q are sample standard deviations, and $\overline{P}$ and $\overline{Q}$ 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 [5]. 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. This scenario occurs, for instance, when a keylogger uses special byte sequences to encode particular classes of keystrokes or encrypts keystrokes with a variable number of bytes. Even under these circumstances, the resulting data transformation can still be approximated as linear. The scale invariance property makes also 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 [8]. 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 experimented this circumstance to rarely occur in practice, 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 [8], 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 [8], 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 Sec. 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.

3.5 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.

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 [8]. 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 Sec. 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 regular system processes. Fortunately, studies show that the correlation between different realistic I/O workloads for PC users is generally considerably low different I/O workloads for PC users is generally considerably low over small time intervals [11]. The results presented in [11] 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 0.0462 on average and never exceeds 0.0708 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 [11]. 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 average 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 constraint 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 average PCC. This problem is a variant of the standard assignment problem for N objects and N tasks, 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 can be modeled by the samples in the target set S and the tasks reflect the N slots in the input pattern each sample has to be assigned to. In addition, the cost of assigning a sample S_i from the set S to a particular slot j is $c(i,j) = \sum_t \frac{(S_i - \overline{S})(P_{t_j} - \overline{P}_t)}{\sigma_s \cdot \sigma_{p_t}}$, where P_t are the patterns in the training set, and $\overline{S}$ 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 resulting expression of the average PCC we want to minimize. 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 easily addressed by leveraging well-known algorithms that solve the linear 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 indeed attempt to find a pattern that results in an average PCC maximally close to -1. In contrast, the ultimate goal of our analysis is to produce a maximally uncorrelated pattern, thereby aiming at an average 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 [13]. In our solution, we have implemented a similar approach limiting the approach to a maximum number of iterations to guarantee convergence in bounded time 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, 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 with the first sample having the value of 1. The sine wave grows or drops with a fixed step of 0.1 at every sample. This algorithm explores the effect of constant increments (and decrements) in the input pattern.

4 Evaluation

To demonstrate the viability of our approach and evaluate the proposed detection technique, we implemented a prototype system based on the ideas described in the paper. Our prototype is entirely written in `C#` and runs as an unprivileged application for the Windows operating system.

In the following, we present several experiments to evaluate our approach. The ultimate goal is to understand the effectiveness of our technique and whether it can be used in realistic settings. We experimented our prototype with many publicly available keyloggers. We have also developed our own keylogger to evaluate the effect of special features or particular conditions more throughly. Finally, we have collected traces for different realistic PC workloads to evaluate the strength of our approach in real-life scenarios. We ran all of our experiments on PCs with a 2.53 GHz Core 2 Duo processor, 4 GB memory, and 7200 rpm SATA II hard drives. Every test was performed under Windows XP Professional SP3, while the workload traces were gathered from a number of PCs running several different versions of Windows.

4.1 Keylogger Detection

To evaluate the ability to detect real-world keyloggers, we experimented all the keyloggers from the top monitoring free software list [15], an online repository continuously updated with reviews and latest developments in the area. At the moment of writing, eight keyloggers were listed in the free software list. To carry out the experiments, we manually installed each keylogger, launched our detection system for $N \cdot T$ ms, and recorded the results.

In the experiments, we used arbitrary settings for the threshold and the parameters N, T, K_{min}, K_{max}. The reason is that we observed the same results for several reasonable combinations of parameters in most cases. We have also solely selected the RFR algorithm as the pattern generation algorithm for the experiments. More details on how to select a pattern generation algorithm and

Table 1. Detection results for the keyloggers used in the evaluation. PCC's threshold set to 0.80.

Keylogger	Detection	Notes
Refog Keylogger Free 5.4.1	✔	uses focus-based buffering
Best Free Keylogger (BFK) 1.1	✔	
Iwantsoft Free Keylogger 3.0	✔	
Actual Keylogger 2.3	✔	uses focus-based buffering
Revealer Keylogger Free 1.4	✔	uses focus-based buffering
Virtuoza Free Keylogger 2.0	✔	uses time-based buffering
Quick Keylogger 3.0.031	N/A	unable to test it properly
Tesline KidLogger 1.4	N/A	unable to test it properly

tune parameters and threshold in the general case are given in Sec. 4.2 and Sec. 4.3. Table 1 shows the keyloggers used in the evaluation and summarizes the detection results. All the keyloggers were detected without generating any false positives. For the last two keyloggers in the list, we were not able to provide any detection result since no consistent log file was ever generated in the two cases even after repeated experiments[1]. In every other case, our detection system was able to detect the keylogger correctly within a few seconds.

Virtuoza Free Keylogger required a longer window of observation to be detected. The *Virtuoza Free Keylogger* was indeed the only keylogger to use some form of aggressive buffering, with keystrokes stored in memory and flushed out to disk at regular time intervals. Nevertheless, we were still able to collect consistent samples from buffer flush events and report high values of PCC with the normalized version of the input pattern.

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 buffering strategy efficiently, our detection system enforces a change of focus every time a sample is injected into the system. Other buffering strategies and possible evasion techniques are discussed in detail in Sec. 5.

Furthermore, some of the keyloggers examined included support for encryption and most of them used variable-length encoding to store special keys. As Sec. 4.2 demonstrates with experimental results, our algorithm can deal with these nuisances transparently with no effect on the resulting PCC measured.

Another potential issue arises from most 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 a 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.

[1] Both keyloggers were installed on Windows XP SP3 and instructed to output their log in a specific directory. However, since no logs have been subsequently produced, we assumed they were not fully compatible with the underlying environment.

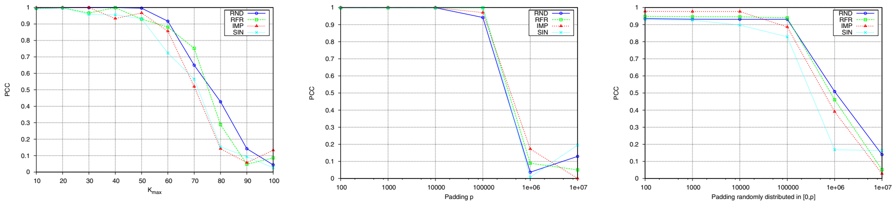
4.2 False Negatives

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 with respect to false negatives, we made several experiments with our own artificial keylogger. In its basic version, our prototype keylogger merely logs each keystroke on a text file on the disk.

Our evaluation starts with 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 reproduce a more distinct distribution in the output stream of a keylogger. Nevertheless, the keystroke rate is clearly bound by the size of the time interval T. Figure 2(a) depicts this scenario with $N = 50$ and $T = 1000$ ms. For each pattern generation algorithm, we plot the PCC measured with our prototype keylogger. This graph shows very high values of PCC for $K_{max} < 50$. For $K_{max} > 50$, regardless of the pattern generation algorithm, the PCC linearly decreases. The effect observed is due to the inability of the system to absorb more than $K_{max} \approx 50$ in the given time interval. We observe analogous results whether we plot the PCC against different values of T. Our results (hereby not reported) shows that the PCC value becomes steady for $T \geq 150$.

We conducted further experiments to analyze the impact of the number of samples N. As expected, the PCC is steady regardless of the value of N. This behavior should not suggest, however, 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 accurate measures of the PCC and avoid potential false negatives.

We have also simulated the effect of several possible input-output transformations. First, we experimented with a keylogger using a non-trivial fixed-length encoding for keystrokes. Figure 2(b) depicts the results for different values of padding $100 < p < 10000000$. A value of $p = 100$ simulates a keylogger writing 100 bytes on the disk for each eavesdropped keystroke. As discussed in Sec. 3.4, the PCC should be unaffected in this case and presumably exhibit a constant behavior. The graph confirms this basic intuition, but shows the PCC dropping linearly after around $p = 100000$ bytes. This behavior is due to the limited I/O throughput that can be achieved within a single time interval. Let us now



(a) PCC in function of K_{max}. (b) Effect of constant padding. (c) Effect of random padding.

Fig. 2. The effect of the parameters on the PCC measured with our keylogger

consider a scenario where the keylogger writes a random amount of characters r, with $0 \leq r \leq p$, for each eavesdropped keystroke. This is interesting to evaluate the impact of several different conditions. First, the experiment simulates a keylogger randomly dropping keystrokes with a certain probability. Second, the experiment simulates a keylogger encoding a number of keystrokes with special sequences, e.g. `CTRL` logged as `[Ctrl]`. Finally, this scenario investigates the impact of a keylogger performing variable-length encryption or other variable-length transformations. Results for different values of p are depicted in Fig. 2(c). As observed in Fig. 2(b), the PCC only drops at saturation. The graph still reveals a steady behavior with the stable value of the PCC only slightly affected (PCC ≈ 0.95), despite the extreme level of noise introduced. Experiments with non-uniform distributions of r in the same interval yield similar results. We believe these results are very encouraging to demonstrate the strength of our detection technique with to respect to false negatives, even in presence of severe data transformations.

4.3 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 particular threshold selected, a false detection is triggered. In this section, we evaluate our prototype system to understand how often such circumstances may occur in practice.

To generate representative synthetic workloads for the PC user, we relied on the widely-used SYSmark 2004 SE suite [4]. The suite leverages common Windows interactive applications[2] to generate realistic workloads that mimic common user scenarios with input and think time. In its 2004 SE version, SYSmark supports two individual 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 have also experimented with another workload that simulates an idle Windows system with common user applications[3] running in the background. In the Idle workload scenario, we allow no user input and focus on the I/O behavior of a number of typical background processes.

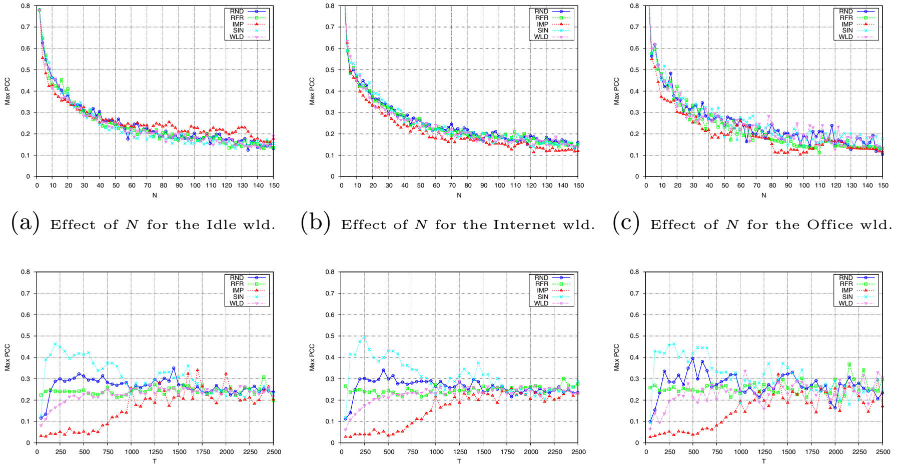
For each scenario, we repeatedly reproduced the synthetic workload on a number of machines and collected I/O traces of all the running processes for several possible sampling intervals T. Each trace was stored as a set of output patterns and broken down into k consecutive chunks with N samples. Every 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.

[2] The set of user programs is available at the following web site `http://www.bapco.com/products/sysmark2004se/applications.php`.

[3] Skype 4.1, Pidgin 2.6.3, Dropbox 0.6.556, Firefox 3.5.7, Google Chrome 5.0.307, Avira Antivir Personal 9.0, Comodo Firewall 3.13, and VideoLAN 1.0.5.

In the testing phase, we measured the maximum absolute PCC between every generated input pattern of length N and every output pattern in the testing set. At the end of each experiment, we averaged all the results. We tested all the workload-agnostic and workload-aware pattern generation algorithms introduced earlier.

We start with an analysis of the pattern length N, evaluating its effect while fixing T to 1000 ms. Similar results can be obtained with other values of T. Figures 3(a), 3(b), 3(c) depict the results of the experiments for the Idle, Internet, and Office workload. As aforementioned, 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.



(a) Effect of N for the Idle wld.
(b) Effect of N for the Internet wld.
(c) Effect of N for the Office wld.
(d) Effect of T for the Idle wld.
(e) Effect of T for the Internet wld.
(f) Effect of T for the Office wld.

Fig. 3. The effect of the parameters and the workload on the maximum PCC measured with regular system processes.

As shown in the graphs, 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. For example, using input patterns of length $N < 10$ typically results in misleading PCC values in the range of 0.5-0.8 for one or more regular system process. 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 generally reflects the low level of predictability in the I/O behavior of a process.

From Figures 3(d), 3(e), 3(f) we can observe the effect of the parameter T on input patterns generated by the IMP algorithm. The experiments shown here have been conducted by fixing the pattern length to an arbitrary stable value of $N = 50$. For small values of T, IMP constantly outperforms all the other algorithms by producing extremely anomalous I/O patterns for the given T in any workload scenario. As T increases, the irregularity becomes less evident and IMP matches more closely the behavior of the other algorithms.

In general, for reasonable values of T, all the pattern generation algorithms reveal a similar and constant distribution of the PCC. This confirms the property of self-similarity of the I/O traffic. As expected, the PCC measured is generally independent of the interval T. Notably, RFR and WLD reveal a more steady distribution of the PCC. This is probably due to the use of a fixed range of values in both algorithms. This also confirms the intuition that more variability in the input pattern leads to more accurate and stable results.

For very small values of T, we also 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. For small values of T, we also found the SIN algorithm to be more prone to generation of false positives. In our analysis, we found that similar PCC distributions can be obtained with very different types of workload. This suggests 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 $0.5 - 0.6$ 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.

5 Evasion Techniques

Despite we were able to detect all the existing keyloggers, there are some evasion techniques that keyloggers may employ to make detection more difficult. For instance, a keylogger may rely on some kind of aggressive buffering, namely flushing its buffer every 12 hours. In this case, since we would need 12 hours to collect a single sample, increasing the amount of samples is not desired solution. We point out that the model underlying our detection technique is not accountable for this limitation. In fact, monitoring the memory accesses of the running processes would promptly make the detection process immune to such behavior. However, since monitoring the memory accesses is not available by means of unprivileged APIs, we reckon that such benefits are mitigated by the need of running the OS in a virtualized environment. A more complex behavior is a keylogger actively performing I/O activities. Although this class of keylogger is hypothetical, our model can easily be augmented to handle this type of keyloggers. The solution is to inject rates of keystrokes higher than the I/O generated by the disguisement activities. In this scenario the component able to inject most of the keystrokes would make its pattern to emerge. Further research is advised to assess the viability of such a countermeasure against this evasion technique.

6 Related Work

Despite our approach is the first and only technique to solely rely on unprivileged execution environments, several works recently dealt with the detection of privacy-breaching malware.

The technique of detecting malware by means of modeling their behavior has been previously proposed by Kirda et al. [12]. Their approach is tailored to detect malware running as Internet Explorer loadable modules. Modules both monitoring the user's activity and disclosing such data to other processes are flagged as malware. Their analysis in fact defines a malware behavior in terms of API calls invoked in response to browser events. However, as we previously discussed, the API calls a keylogger leverages are commonly used by legitimate components. Their approach is therefore prone to false positives that only a continuously updated white-list may be able to counter.

Slightly more sophisticated approaches are the ones detecting when known APIs are exploited. Since user-space keyloggers are known to target a little set of APIs, this approach perfectly fits our case. Aslam et al. [3] adopt the approach to disassemble executable in order to look for the mentioned API calls. Unfortunately, all these calls are commonly used by legitimate applications; detecting keyloggers by such means would produce a remarkable amount of false positives. Xu et al. [20] push this technique a little further. They interpose a function between the API and any program calling it; this function denies the delivering of the keystroke to the keylogger by means of altering its type (from `WM_KEYDOWN` to `WM_CHAR`). However, since they rely on the ability to interpose the function before the keylogger, a malware aware of this countermeasure can easily elude it.

A step a little closer to our approach is discussed by AlHammadi et al. in [1]. Their approach defines a malware behavior in terms of the invoked API functions. To be more precise, they collect the frequency of API calls invoked to (i) intercept keystrokes, (ii) writing to a file, and (iii) sending bytes over the network. A malware is then flagged as such whether two frequencies are found to be highly correlated. Since no bogus events are sent to the system (no injection of crafted input), the correlation may be not be as strong as expected. The correlation value would be even more impaired in case of any delay introduced by the malware. Moreover, since the whole analysis is focused on a specific bot, it lacks a proper discussion on both false positive and false negatives of their quantitative analysis. In our approach we focus on the actual written bytes and consequently adopt a different correlation metric, i.e. PCC, that instead is linear. Due to its linearity, any data transformation (such as encryption) would not help the malware in evading our detection. Our approach is also immune to malware reasonably buffering the collected data: as long as the minimum rate of injected keystrokes flushes the buffer in question, our approach preserves its effectiveness. In case such a requirement can not be met, our technique can be easily extended by means of aggregating consecutive samples as we explain in Sec. 5.

A similar technique comprising of both quantitative analysis and injection routine is sketched by Han et al. in [10]. However, besides being a privileged approach like [1] and [6], it merely relies on the amount of API calls triggered in response to a certain amount of keystrokes. However, the assumption that a certain amount of keystrokes implies a fixed amount of API calls is not always true. It is how a program is implemented that determines in how many chunks a stream of data is written to disk. In our approach we rely on more precise measurements that are also available by means of unprivileged APIs, namely the amount of bytes a process writes.

In conclusion, notable approaches recently attempted to generalize the behavior deemed malicious. In particular, in [16,7] the authors attempt to identify trigger-based behavior by means of mixing concrete and symbolic execution. In such a way they aim to explore all the possible execution paths that a malware may reproduce during execution. As the authors in [16] admit, however, automating the detection of trigger-based behavior is an extremely challenging task requiring advanced privileged tools. The problem is also undecidable in the general case.

7 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 means of correlating the input, i.e. the keystrokes, to the I/O pattern produced by the keylogger. Moreover, since the input to the system was known, we augmented our model by introducing the ability to artificially inject keystrokes. We then discussed the problem of choosing the best input pattern to improve our detection rate. Subsequently we implemented our architecture on the operating system most vulnerable to the threat of keyloggers, i.e. Windows. We also gave implementation details to accommodate different operating systems, thus obtaining an OS independent architecture. We then tested the prototype against a real case scenario; the results met our expectations: given a proper threshold, we were able to detect 100% of the most common free keyloggers [15] completely avoiding any false positive.

As future works, we will investigate the tradeoff of giving up the constraint of an unprivileged execution environment; accessing privileged APIs would allow our detector to monitor memory accesses of running processes.

References

1. Al-Hammadi, Y., Aickelin, U.: Detecting bots based on keylogging activities. In: Proceedings of the Third International Conference on Availability, Reliability and Security, pp. 896–902 (2008)

2. Aldrich, J.: Correlations genuine and spurious in pearson and yule. Statistical Science 10(4), 364–376 (1995)
3. Aslam, M., Idrees, R., Baig, M., Arshad, M.: Anti-Hook Shield against the Software Key Loggers. In: Proceedings of the 2004 National Conference on Emerging Technologies, p. 189 (2004)
4. BAPCO: SYSmark 2004 SE (2004), `http://www.bapco.com/products/sysmark2004se/`
5. Benesty, J., Chen, J., Huang, Y.: On the importance of the pearson correlation coefficient in noise reduction. IEEE Transactions on Audio, Speech, and Language Processing 16(4), 757–765 (2008)
6. Borders, K., Zhao, X., Prakash, A.: Siren: Catching evasive malware (short paper). In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 76–85 (2006)
7. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. Advances in Information Security 36, 65–88 (2008)
8. Goodwin, L., Leech, N.: Understanding correlation: Factors that affect the size of r. The Journal of Experimental Education 74(3), 249–266 (2006)
9. Grebennikov, N.: Keyloggers: How they work and how to detect them, `http://www.viruslist.com/en/analysis?pubid=204791931`
10. Han, J., Kwon, J., Lee, H.: Honeyid: Unveiling hidden spywares by generating bogus events. In: Proceedings of The Ifip Tc 11 23rd International Information Security Conference, pp. 669–673 (2008)
11. Hsu, W., Smith, A.: Characteristics of I/O traffic in personal computer and server workloads. IBM System Journal 42(2), 347–372 (2003)
12. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based spyware detection. In: Proceedings of the 15th USENIX Security Symposium (USENIX Security 2006) (2006)
13. Kochenberger, G., Glover, F., Alidaee, B.: An effective approach for solving the binary assignment problem with side constraints. Internation Journal of Information Technology and Decision Making 1, 121–129 (2002)
14. Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly 2, 83–97 (1955)
15. Security Technology Ltd.: Testing and reviews of keyloggers, monitoring products and spy software (spyware) (2009), `http://www.keylogger.org/monitoring-free-software-review/`
16. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceeding of the 28th IEEE Symposium on Security and Privacy (SP 2007), pp. 231–245 (May 2007)
17. San Jose Mercury News: Kinkois spyware case highlights risk of public internet terminals (2009), `http://www.siliconvalley.com/mld/siliconvalley/news/6359407.htm`
18. Rodgers, J.L., Nicewander, W.A.: Thirteen ways to look at the correlation coefficient. The American Statistician 42(1), 59–66 (1988)
19. Strahija, N.: Student charged after college computers hacked (2003), `http://www.xatrix.org/article2641.html`
20. Xu, M., Salami, B., Obimbo, C.: How to protect personal information against keyloggers. In: Proceedings of the 9th International Conference on Internet and Multimedia Systems and Applications, IASTED 2005 (2005)