ISSN: 2278-0181
NCICCT 14 Conference Proceedings

Anti-Hacking Mechanism for Keylogger using Blackbox Detection

Saiganesan N
Dept. of IT
Roever Engg College
Perambalur
Tamilnadu, India

Dheenadhayalan A
Dept. of IT
Roever Engg College
Perambalur
Tamilnadu, India

Arulmani M
Dept. of IT
Roever Engg College
Perambalur
Tamilnadu, India

Suresh K (Guide)
Dept. of IT
Roever Engg College
Perambalur
Tamilnadu, India

***Abstract*— Key logging or Keyboard Capturing is the action of recording the keys struck on a keyboard, typically in a covert manner so that the person using the keyboard is unaware that their actions are being monitored. Keyloggers are hardware and software tools that capture characters, numbers sent from keyboard to an attached computer. Software Key logger is a stealth surveillance application, which is used to keep record of user activities on the computer in various ways like keyboard logging, screen logging, mouse logging and voice logging completely undetected to ANY user because it is designed to capture what is done on a PC. 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. A keylogger is a hardware device or a software program that records the real time activity of a computer user including the keyboard keys they press. Keyloggers are used in IT organizations to troubleshoot technical problems with computers and business networks. Keyloggers can also be used by a family (or business) to monitor the network usage of people without their direct knowledge. Finally, malicious individuals may use keyloggers on public computers to steal passwords or credit card information. 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 key-loggers 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 (OSs). 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.

Furthermore, a keylogger implemented as a user-space process is much easier to deploy since no special permission is required. In light of these observations, it is no surprise that 95 percent 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, un intrusive, 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. 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 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.

2 EXISTING SYSTEM

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 cafe´, 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 are the straightforward software evolution of hardware-based keyloggers, literally performing a man-in-the-middle attack between the hardware and the operating system. Kernel keyloggers come second in the chain and are often implemented as part of more complex root kits. 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 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. Fig. 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.

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 functions of the last class apply only to Windows and are typically used to overwrite the default address of key-stroke-related functions in all the Win32 graphical applications. We have not found any example of this particular class of keyloggers in Unix-like OSes.

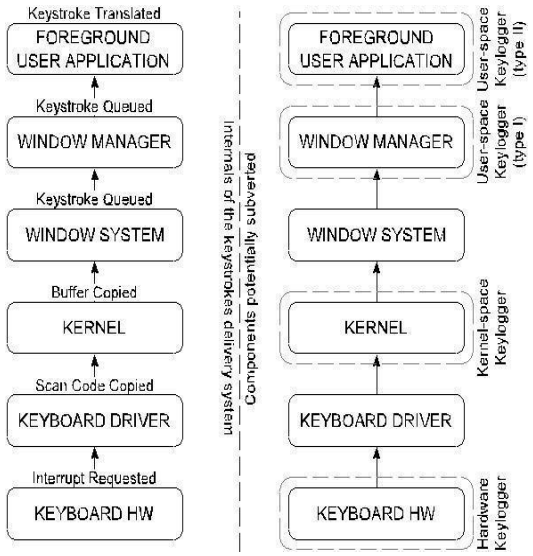

Figure 1-The delivery phases of a keystroke, and the components potentially subverted (we omit hypervisor-based approaches for the sake of clarity).

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 GetKey-boardState(). In these cases, the keylogging process can attach its input queue (i.e., the queue of events used to control a graphical user application) to other threads by using the procedure AttachtreadInput(). As a tentative counter-measure, Windows Vista recently eliminated the ability to share the same input queue for processes running 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 key-strokes 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. 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.

3 PROPOSED SYSTEM

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-sup-ported 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 fore-ground applications surreptitiously logging user-issued keystrokes (e.g., a keylogger spoofing a trusted word processor application) and application-specific keyloggers (e.g., browser plug-in surreptitiously performing key logging 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 in two different integrity levels. Unfortunately, since higher 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, pre-selecting 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.

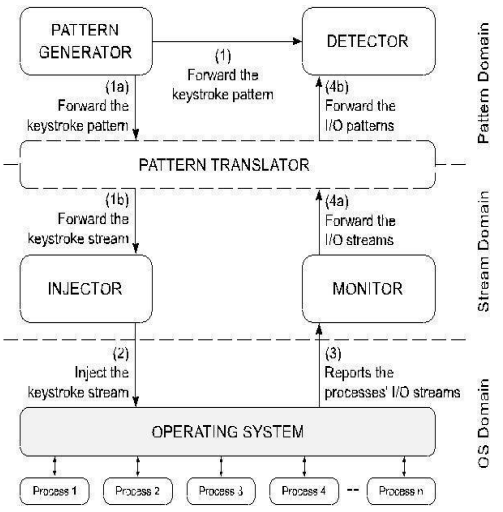

Fig. 2 The different components of our architecture

The key advantage of our approach is that it is centered on a black-box model that completely ignores the keylogger internals. Also, I/O monitoring is a nonintrusive 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 Fig. 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.

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 keyboard event. In all Unix-like OSes supporting X11 the same functionality is available via the API call XTestFake-KeyEvent.

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 real time monitoring with minimal overhead and the best level of resolution possible. Finally, we are interested in application-level statistics of I/O activities, to avoid dealing with file system level caching or other potential nuisances. Fortunately, most modern operating systems provide unprivileged API calls to access performance counters on a per-process basis. On all the versions of Windows since Windows NT 4.0, this functionality is provided by the Windows Management Instrumentation (WMI). In particular, the performance counters of each process are made available via the class Win32_Process, which supports an efficient query-based interface. The counter WriteTransferCount contains the total number of bytes written by the process 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; both Linux and OSX, in fact, support analogous performance counters which can be accessed in an un-privileged manner; the reader may refer to the iotop utility for usage examples. Adopting a compact and uniform representation is advantageous for several reasons.

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 ½0; 1&, where 0 and 1 reflect the predefined minimum and maximum number of key-strokes 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.

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. 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 on a particular value z. The statistical meaning of the value z is the average number of keystrokes dropped per time interval.

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. 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. A larger number of samples are beneficial especially in case of disturbing factors. As reported, 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 detection, a thresholding mechanism is used. Our detection algorithm is conceived to infer a causal relationship between two patterns by analyzing their correlation. Detection can be done by identifying the input and output correlation appearing to be different and we come to the conclusion that somebody has traced our activities and they had seen the output in the monitor. After identifying the hacker, detection algorithm is used to detect the keylogger from the system, this will provide enhanced security features to the user. 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 nontrivial nonlinear optimization problem.

Partner 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. 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 ½0; 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.

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 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 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. 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.

Table 1
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	✔	-

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 develop-ments in the area. To carry out the experiments, we manually installed each keylogger, launched our detection system for N _ T ms, and recorded the results; we asserted successful detection for PCC _ 0:7. In the experiments, we found that arbitrary choices of N, T, K_{min}, and K_{max} were possible; the reason is that we observed the same results for several reasonable combinations of the parameters.

Table 1 shows the keyloggers used in the evaluation and summarizes the detection results. All the keyloggers were detected within a few seconds without generating any false positives; in particular, no legitimate process scored PCC values _ 0:3. Virtuoza Free Keylogger required a longer window of observation to be detected; this sample was indeed the only keylogger to store keystrokes in memory and flush out to disk at regular time intervals. Nevertheless, we were still able to collect consistent samples from flush events and report high PCC values.

In a few other cases, keystrokes were kept in memory but flushed out to disk as soon as the keylogger detected a change of focus. This was the case for Actual Keylogger, Revealer Keylogger Free, and Refog Keylogger Free. To deal with this common strategy, our detection system enforces a change of focus every time a sample is injected. In addition, some of the keyloggers examined included support for encryption and most of them used variable-length encoding to store special keys.

Another potential issue arises from keyloggers dumping a fixed-format header on the disk every time a change of focus is detected.

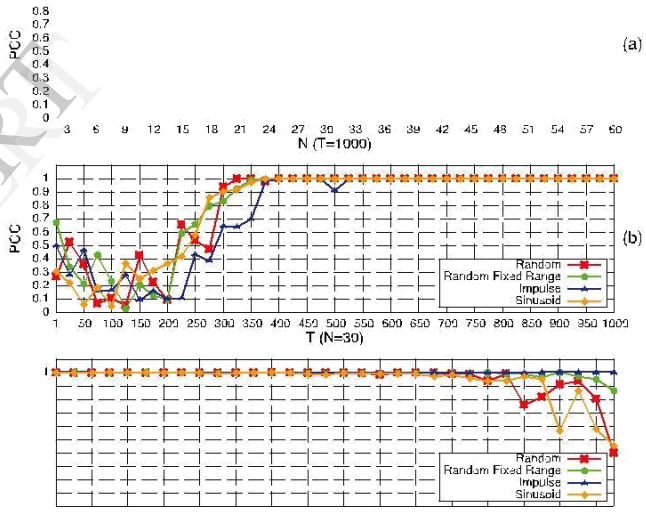

Figure 3-Impact of N, T , and Kmin on the PCC

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 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. 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 white lists.

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 [4], 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.

6 FUTURE WORKS

Future work will focus on extending our approach to spyware programs that do not rely on the Browser Helper Object or toolbar interfaces to monitor the user's behavior. We also plan to extend our characterization with more sophisticated data-flow analysis that would allow one to characterize the type of information accessed (and leaked) by the spyware program. This type of characterization would enable a tool to provide an assessment of the level of "maliciousness" of a spyware program.

7 CONCLUSION

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.

REFERENCES

[1] San Jose Mercury News, "Kinkois Spyware Case Highlights Risk of Public Internet Terminals," http://www.siliconvalley.com/mld/siliconvalley/news/6359407.htm, 2012.

[2] N. Strahija, "Student Charged After College computers acked,"http://www.xatrix.org/article2641.html, 2012.

[3] N. Grebennikov, "Keyloggers: How They Work and How to Detect Them," www.viruslist.com/en/analysis?pubid=204791931, 2012.

[4] Security Technology Ltd., "Testing and Reviews of Keyloggers, Monitoring Products and Spyware," http://www.keylogger.org, 2012.

[5]S. Ortolani and B. Crispo, "Noisykey: Tolerating Keyloggers via Keystrokes Hiding," Proc. Seventh USENIX Workshop Hot Topics in Security, 2012.

[6]S. Ortolani, C. Giuffrida, and B. Crispo, "Klimax: Profiling Memory Write Patterns to Detect Keystroke-Harvesting Malware," Proc. 14th Int'l Symp. Recent Advances in Intrusion Detection, pp. 81-100, 2011.

[7]C.Wood and R. K. Raj, "Sample keylogging programming projects," 2010 (accessed May 8, 2010), http://www.cs.rit.edu/~rkr/ keylogger2010.

[8] M.Agarwal, M.Mehra "Secure Authentication using Dynamic Virtual Keyboard Layout", ICWET – TCET, Mumbai, India, 2011.

[9] S.Gong "Design and Implementation of Anti-Screenshot Virtual Keyboard Applied in Online Banking "E-Business and E- Government(ICEE), 2010 International Conf., 7-9 May 2010, pp-1320 – 1322.

[10] Taehwan, Choi, Sooel, Son, Mophamed Gouda, and Jorge Cobb, In Symposium on Stabilization, Safety, and Security of Distributed System(SSS), 2008.