Enhancing Keylogger Detection Performance of the Dendritic Cell Algorithm by an Enticement Strategy

Jun Fu and Huan Yang
The 28th Research Institute of China Electronics Technology Group Corporation, Nanjing 210007, China
Email: {doctorfj, happyfairy106}@163.com

Yiwen Liang and Chengyu Tan
Computer School, Wuhan University, Wuhan 430079, China
Email: ywliang@whu.edu.cn, nadinetan@163.com

***Abstract*—Evasive software keyloggers hide their malicious behaviors to defeat run-time detection. In this paper, based on the analysis of the evasion mechanisms used by common software keyloggers, we established a framework for their detection. Using an enticement strategy, the framework we built could induce keyloggers exhibited more obvious malicious activities by mimicking user keystrokes. These 'amplified' activities are then correlated by the dendritic cell algorithm (an immune-inspired algorithm) to final determine the existence of a keylogger in a host. Preliminary experimental results showed that the framework could improve the performance of keylogger detection and hard to evade.**

***Index Terms*—keylogger, keystroke simulation, dendritic cell algorithm (DCA), correlation**

I. Introduction

With the development of e-commerce and online games, software keyloggers which steal confidential information by monitoring a user's keyboard actions are becoming a new trend for malware [1] [2]. They intercept and log all keystrokes, and transmit this information to profit-driven attackers. Unlike other types of malicious programs, keyloggers are designed to capture what is done on a PC without attracting the attention of users and present no threat to the system [3]. This makes them largely undetectable by most anti-virus and anti-keylogger applications [4] [5].

To overcome the problems above, security experts are trying to use behavior-based detection techniques that analyze API calls of a process to classify it as keylogger or not [6][7]. However, these methods all have some shortcomings to some extent. Detection [6] relies on single behavior (setting Windows hooks) has a high rate of false positives (FP) [7]. Though correlation of multiple behaviors (keystroke tracking, file access and network communication) reduces the FP rate, it seems that the detection is prone to be evaded when specified time window and simple correlation algorithm are used [7].

For the purpose of improving the detection performance, Fu [8] uses an immune-inspired algorithm - dendritic cell algorithm (DCA) to correlate the behaviors mentioned by [7]. As an algorithm, the DCA performs multi-sensor data fusion on a set of input signals, and these signals are correlated with potential 'suspects', leading to information which will state not only if an anomaly is detected, but in addition the culprit responsible for it. By using variable time windows and time sequence of the different behaviors, the DCA improves the detection performance to a certain degree. But every coin has two sides. The correlating feature of the DCA can be exploited by crafty attackers to evade detection by reducing the frequency of the malicious behaviors [9]. The experimental results of [8] support the above claim.

Man-to-machine interfaces cannot be ignored when fight against keyloggers [3]. In this paper, we analyzed the evasion mechanisms used by common software keyloggers. We discovered that the keystroke (especially keystroke of the special key, such as '*Enter*' key) frequency is an important trigger for keyloggers to log and send captured information. As a result, we built an induction-correlation framework for keylogger detection. In this framework, we synthesized man-to-machine interactions by implementing a keystrokes simulation program. The program can induce keyloggers to exhibit more malicious activities without disturbing normal applications. Then, the 'amplified' behaviors are correlated by the DCA in order to identify the keylogger as early as possible. Experiments were conducted to test capabilities of our framework to improve the detection rate and reduce the possibility of successful evasion.

II. Related Work

There are only a few existing techniques for software keylogger detection. Most of these techniques use signature-based approaches. Since signature-based detection has nothing to do against keylogger variants [10], security experts are now focusing their attentions to

doi:10.4304/jcp.9.6.1347-1354

behavior-based detection techniques that analyze API calls of a process to classify it as benign or malicious.

Some approaches based on API calls focus on searching only those APIs that can be used to intercept keystrokes, either statically [6] or dynamically [11] [12]. Unfortunately, these APIs are also used by legitimate applications, which makes these approaches heavily prone to false positives.

Rather than relying on single type of API (keylogging APIs), Al-Hammadi and Aickelin [7] detects keylogging activities with correlations between multiple types of API (keystroke tracking, file access and network communication APIs), that is detecting when both interception and leakage of keystrokes are taking place. Although the technique has a relatively low false positive rate, the detection rate is not high because specified time windows and simple correlation algorithm (an algorithm using Spearman's Rank Correlation [13]) are used.

Based on the work above, Fu [8] uses an immune-inspired algorithm - dendritic cell algorithm (DCA) to correlate multiple types of API described in [7]. The DCA is based on an abstract model of the behaviors of dendritic cells which are natural intrusion detection agents of the human body. These cells collect antigens and signals (environmental conditions of the antigens), and combine the evidence of damage (signals) with the collected suspect antigen to provide information about how 'dangerous' a particular antigen is. The DCA performs multi-sensor data fusion on a set of input signals and antigens, leading to information which states not only if an anomaly is detected, but in addition the culprit responsible for it [14]. More information about the DCA please refers to [15].

The input signals defined in [8] are derived from the frequency of invocations of keystroke tracking functions (PAMPs and safe signal-2), the time difference between two consecutive *WriteFile* calls (danger signal-1), the relation between different categories of function calls (danger signal-2) and the time difference between two outgoing consecutive communication functions (safe signal-1). The process (identified by Process ID) which causes the calls is defined as antigens [8]. The DCA correlates these antigens with input signals, resulting in a pairing between signal evidences and antigen suspects, and the identification of the keylogger process in the end. However, as the DCA distinguishes between normal and potentially malicious antigens on the basis of neighboring antigens, the crafty attackers can exploit this correlating feature to evade detection by reducing the 'concentration' of antigens in DCs [9].

The experimental results of [8] confirmed the above conclusion. The keylogger they used in experiments hid its behaviors by logging and sending keys only when enough keystrokes were intercepted or special keys (such as '*Enter*' key) were pressed. In experiments that long sentences were entered, the frequency of the malicious activities generated by the keylogger decreased significantly compared to the one observed in short sentence scenarios. The same trends were also found in detection performances of the DCA because of the reducing of the 'concentration' of the malicious antigens. In the real world, we believe users keystroke patterns are similar with the long sentence scenarios described in [7] and [8]. This challenges the DCA to detect keyloggers in the real environment.

III. Keylogger Analysis

In this paper, we analyzed the source code of some typical open source keyloggers running on *Windows NT* operating systems, such as *Keymail V0.7*, *Spybot V1.2* and *Morsa-Keylogger V1.8*. Then we compiled and executed these source codes to find their run-time features. Based on the static and dynamic analysis, we discovered the relationships between different behaviors generated by these keyloggers, and revealed the evasion mechanisms often used by them.

Through static analysis, we found that all keyloggers worked in a similar manner. They all firstly tracked keystrokes and then wrote them to a file or/and send them to a destination across the Internet (via Email, FTP and etc.). The most important difference between these keyloggers was the timing that triggered file access and communication activities. These activities were performed when:

1) intercepted every keystroke;
2) the keystrokes intercepted reached a certain amount;
3) special keys (such as '*Enter*' key) were pressed.

After running the compiled source codes, we found the keyloggers (such as *Keymail*) using the 1) trigger condition generated more file access and communication behaviors. However, these behaviors were relatively rarely observed when the keyloggers (such as *Spybot* and *Morsa-Keylogger*) with the 2) and the 3) trigger conditions were executed. So we could make a conclusion that it is the 2) and the 3) trigger conditions that gave keyloggers capabilities to evade the correlation-based detection (such as the DCA).

Fortunately, the evasion mechanism above is a double-edged sword. Besides hiding keylogger behaviors, it greatly exposes the existence of the keylogger when high frequency of keystrokes (especially special keystrokes) is encountered. That is why we use the keystroke simulation to enhance the detection performance of the DCA.

IV. Induction-Correlation Framework

We assume that the host to be monitored is infected with a keylogger without a user's awareness. The installed keylogger logs and sends the captured information when a user types his/her privacy via keyboard. In this paper, we propose an enhanced approach to detect software keyloggers on a host. The approach consists of two steps: 1) the induction of the keyloggers, 2) the correlation of the behaviors exhibited by them. We emphasize on the first step, describing how it improves the performance of the second step. The framework of our approach is shown in fig. 1.

Because the frequency of keystrokes in real environment is not high, the behaviors of the keylogger are not evident enough [8]. Therefore, we design a

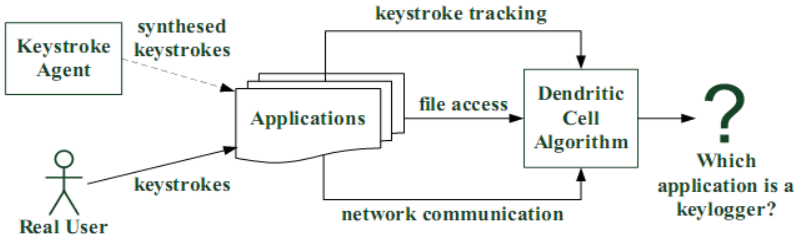


Figure 1. The induction-correlation framework for software keylogger detection by the DCA.

keystroke agent application to frequently generate random keystrokes, hoping that these keystrokes will be seen by the keylogger, but will not affect normal applications. As a result, the behavior of the keylogger will be more obvious in the stimulation of a large number of random keystrokes in a short time. Meanwhile, the keystroke agent holds the simulated keystrokes within a hidden application it creates to avoid them passing to the other applications. Thus the normal applications will not be affected by the simulated keystrokes since they only focus on the keystrokes passed to them.

We also focus on keystroke tracking, file access and network communication behaviors exhibited by applications. By correlating API calls generated by these behaviors, the DCA can classify the application running in a host as a keylogger or not. For example, file access shortly after the keystroke tracking strongly indicates that there exist keylogging activities. The more active the application is in that period, the more likely it is a keylogger.

A. Induction Phase

In the induction phase, we synthesize random keystrokes to induce keyloggers. In order to find a way to simulate keystrokes which will be seen by a keylogger, we must understand how an operating system generates and handles keyboard events. We also must be clear that how a keylogger intercepts keystrokes in this process.

As in fig. 2, a *Windows NT* operating system generates a keyboard interrupt when a key was pressed. Then the keyboard driver transforms the interrupt to a system-defined message and puts it into the '*system level message queue*'. Tracking the focused application at the time when the keyboard interrupt was generated, the operating system passes the message to the '*application level message queue*' of that specific focused application. Now it's the responsibility of that application to handle this key accordingly. If the operating system does not find any specific focused application, it simply discards that key. In this process, keyloggers employ very low level operating system calls [14], such as *GetKeyboardState* or *GetAsyncKeyState*, to intercept keystroke messages or detect keyboard interrupts directly. So the keyloggers see everything whenever a key is pressed.

In this paper, we design a keystroke agent according to the mechanisms described above. By invoking system kernel (*keybd_event*), the agent simulates keyboard event completely. Because a keylogger tracks keystrokes from all applications (including keystroke agent application) in order to log sensitive data entered in them, it could see the simulated keystrokes since they are the same with the real keystrokes. But the keylogger doesn't understand what it sees and it can't tell the keystrokes generated by real users via keyboard from the ones generated by phantom users via our keystroke agent. When we simulate keystrokes frequently, in order to log and send these plentiful keys, the keylogger has to perform more file access and communication behaviors.

In this case, the keystroke stream generated by the keystroke agent can be described as follows:

$$KeystrokeStream(N, T_L, T_I, f) \quad (1)$$

The keystroke stream can be modeled as a sequence of N samples with a uniform time interval T_I. Each sample lasts for a particular length of time T_L, and synthesizes keystrokes with an rate changes according to function $f(t) : (0, T_L) \rightarrow (0,+\infty)$. $f(t)$ determines the pattern of a sample. In this paper, we design the following patterns:

1) Fixed Square Wave. All samples synthesize keystrokes with a fixed rate R_0. This pattern attempts to minimize the variability of the keystroke rate.
2) Random Square Wave. All samples synthesize keystrokes with a rate uniformly distributed over the range [0, R_0]. This pattern attempts to maximize the variability of samples.
3) Sawtooth Wave. Each sample linearly increases the rate based on an initial rate R_0. This pattern explores the effect of constant increments in the rate.

On the other hand, the simulated keystrokes must not affect normal applications. Before starting to simulation, the agent creates a hidden window and sets the current active window to it. Then the simulated keystrokes are generated and passed to the hidden window which simply discards the keys received. After the simulation, the active window is set back to the active window before simulation. The keystroke agent regularly performs the procedure above. Since the execution time of this process is very short, the active window switches are almost imperceptible to users.

B. Correlation Phase

In this paper, we use the dendritic cell algorithm (DCA) to correlate API calls generated by all running applications to identify keylogger applications. In order to obtain the API calls, we implement a hook program to monitor three types of function calls:

1) **Keyboard Tracking**: *GetKeyboardState*, *GetAsyncKeyState* and *GetKeyNameText* [17].

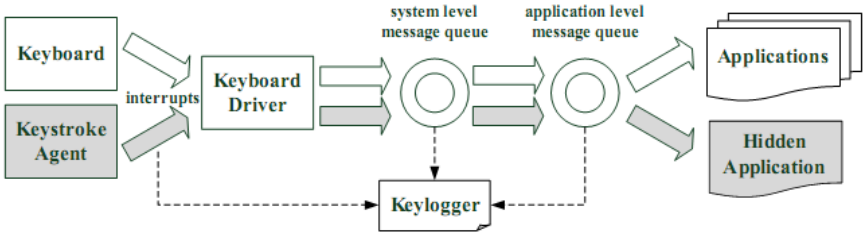


Figure 2. How keystrokes are handled by a *Windows NT* operating system and intercepted by a keylogger.

2) **File Access**: *CreateFile*, *OpenFile*, *ReadFile* and *WriteFile* [18].
3) **Communication**: *socket*, *send*, *recv*, *sendto* and *recvfrom* [19].

These API functions are often employed by keyloggers to implement their keylogging and other features, but also may form part of legitimate usage. Therefore, an intelligent correlation method such as the DCA is required to determine if the invocations of such functions are indeed anomalous. Signals and antigens are vital input to the DCA. To facilitate comparison, we use the same definitions of the signal and antigen described in [7].

Five signals, namely one PAMP signal (PAMP), two danger signals (DS) and two safe signals (SS), are used for the input of the DCA. They are derived from the API calls captured, and then analyzed by the DCA, following a signal normalization process.

PAMP is a signature based signal. This signal is derived from the rate (r) of keyboard tracking function calls. A large number of these function calls indicate the potential existence of a keylogger. Let the signal value be 100 (danger) when $r > N_{ph}$ and be 0 (not danger) when $r < N_{pl}$ ($N_{pl} < N_{ph}$). We normalize the PAMP base on these values by applying linear scale between 0 and 100.

Danger signal is a measure of an attribute which increases in value to indicate an abnormality. Low values of this signal may not be anomalous.

DS-1 is derived from the time difference (Δt_1) between two consecutive *WriteFile* function calls. Because a keylogger saves the keystrokes captured to log files continuously, a small Δt1 will be observed. In contrast, a normal application will have a higher value of Δt_1 between writing activities. Let the signal value be 0 (not danger) when $\Delta t_1 > N_{d1h}$ and be 100 (danger) when $\Delta t_1 < N_{d1l}$ ($N_{d1l} < N_{d1h}$). We normalize DS-1 base on these values by applying linear scale between 0 and 100.

DS-2 is derived from the correlation between different categories of function calls. Based on the behavioral characteristics of keyloggers, we generate this signal when file access or communication functions are invoked shortly after the invocations of keyboard tracking functions. The value of the signal lies on the sum of the number of the file access and communication function calls within specified time-window. Let the signal value be 100 (danger) when this sum exceeds N_{d2h}. DS-2 is normalized base on this value by applying linear scale between 0 and 100.

Safe signal is a confident indicator of normal or steady-state system behavior. This signal is used to counteract the effects of PAMP and danger signals.

SS-1 is derived from the time difference (Δt_2) between two outgoing consecutive communication functions including *send*, *sendto* and *socket* functions. This is needed as keyloggers send information to attackers after the keylogging activity. In normal situation, we expect to have a large Δt_2 between two consecutive functions. In comparison, we expect to have a short period of this action when the keylogger sends information to the attacker. Let the signal value be 100 (not danger) when the $\Delta t_2 \geqslant N_{s1h}$ and be 0 (danger) when the $\Delta t_2 < N_{s1l}$ ($N_{s1l} < N_{s1h}$). We normalize SS-1 base on these values by applying linear scale between 0 and 100.

SS-2 is derived from the small amount of the keyboard tracking function calls within a specified time-window. As legitimate applications such as *notepad* or *wordpad* invoke much fewer keyboard tracking functions than keyloggers. So, small amount of invocations is considered to be safe in the host. Let the signal value be 0 (danger) when this amount exceeds N_{s2l}. SS-2 is normalized base on this value by applying linear scale between 0 and 100.

Antigens are potential culprits responsible for any observed changes in the status of the system. As any process executed one of the selected API functions, the process id (PID) which causes the calls and thus generates signals is defined as antigens. Observing which processes are active when signal context is danger, the DCA can find the existing keylogger in the system.

V. EXPERIMENS

The aim of our experiments is to verify that the keystroke simulation can enhance the visibility of a keylogger's behaviors, and thus can improve the detection performance of the DCA. To achieve this goal, we chose the same keylogger instance (*spybot*) and benign instances (*notepad* and *mirc* [20]) used in the experiments in [8], and set up the same environment for their running.

The experiments are divided into two groups to show the detection performance differences between the DCA with the keystroke agent (E2) and the DCA without (E1) the keystroke agent. The agent works in Fixed Square Wave pattern ($R_0 = 200$, $T_I = 5$s, $T_L = 50$ms), synthesizing some random (numbers and letters) and special keys ('*Enter*' key). Each experiment was repeated for 10 times and lasted 600 seconds. Without any operations in the

first 60 seconds, we used *notepad* and *mirc* to input sentences for 180 seconds respectively with an interval of 60 seconds. We had no operations in the final 120 seconds.

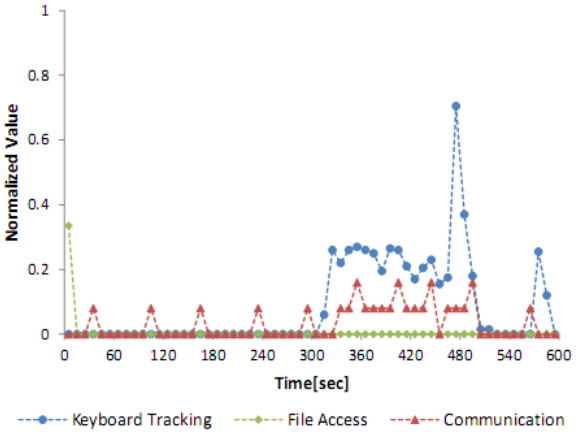
In both groups of experiments, we monitor two scenarios of typing. We type short sentences in one scenario (E1.1 and E2.1) and long sentences in the other (E1.2 and E2.2). The sentence ends with '*Enter*' key. By monitoring two typing scenarios, we are able to show the effects of different keystroke patterns on our detection scheme and the effects of keystroke simulation in different input mode.

In [8], the contents of the sentences are random, and the lengths of the sentences are not mentioned. To get closer to real user keystroke patterns, we collect 200 commonly used English sentences: 100 long (19-48 characters) and 100 short (9-23 characters) sentences, and type them one by one in corresponding scenarios. Because of the changes in typing mode, we adjust the signal threshold values used in [8]: N_{ph} = 1301(times/s), N_{pl} = 1105(times/s), N_{d1l} = 7000(ms), N_{d1h} = 20000(ms), N_{d2h} = 2(times/s), N_{s1l} = 5000(ms), N_{s1h} = 10000(ms), N_{s2l} = 55(times/s). They are set based on the statistical results of the frequency of API calls. Take keyboard tracking API calls for example, the average frequency generated by *spybot* is 1203(times/s), the standard deviation is 88(times/s). And the max frequency concerning *notepad* or *mirc* is 55(times/s). Therefore, we set N_{ph} = 1203 + 88 = 1301(times/s), N_{pl} = 1203 − 88 = 1105(times/s), N_{s2l} = 55(times/s). The maliciousness of keyboard tracking API calls is depend on whether they are in a high frequency (greater than N_{ph}) or a low frequency (less than N_{s2l}). The population of DCs is set to 100, the DCA chooses 10 DCs every time an antigen or a signal is arrived for their storage. The fuzzy migration threshold value of DCs is between 1500 and 2000. The weight matrix is the same with [7].

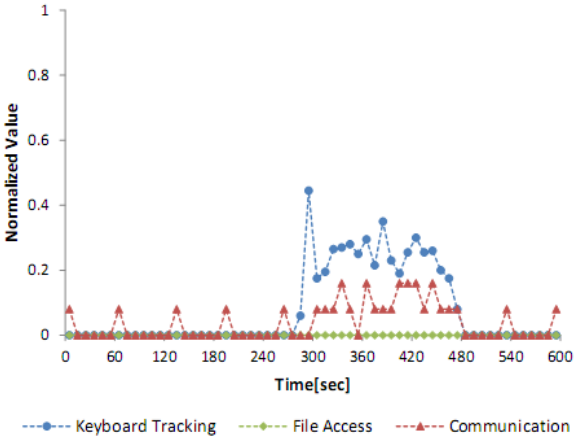
A. Results

We first give a look at the frequency of API calls generated by all applications in E1 and E2. The x-axis represents time in seconds while the y-axis represents the normalized value of API call frequencies. The normalized API call frequency values represent the total value we get during 10 seconds divided by the maximum value of the whole period (600 seconds).

Keystroke agent does not interfere with normal applications by design. This means that normal application instances are not affected by simulated keystrokes, as shown in Fig. 3 (take *mirc* program for example). From Fig. 3, we notice that the difference between the API call frequencies of *mirc* instance in E1.1 and E2.1 is very small. When *mirc* program is used for online chatting (301-480 seconds), most of the values of the keyboard tracking and communication API call frequencies are in the range of (0.2-0.3) and (0.1-0.2) respectively in both experiments. When *mirc* program is idle, we also notice that there is a burst in the network traffic in both experiments. This burst is generated due to *mirc* program which sends a bulk of words to its servers every specified time intervals.



(a) API functions invoked by *mirc* in E1.1



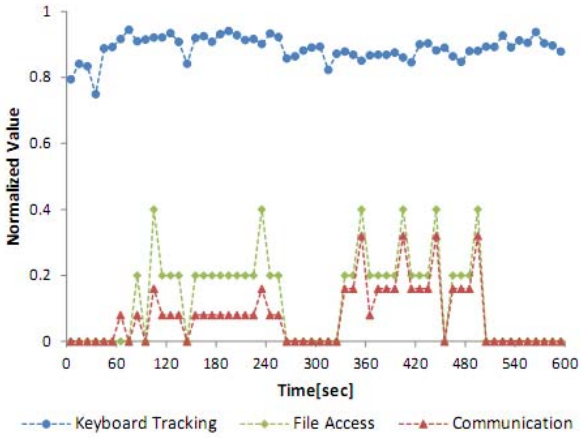
(b) API functions invoked by *mirc* in E2.1

Figure 3. API functions invoked by *mirc* in E1.1 (without keystroke simulation) and E2.1 (with keystroke simulation). The *mirc* program is executed at the beginning of the experiments and is used for online chatting during 301-480s.

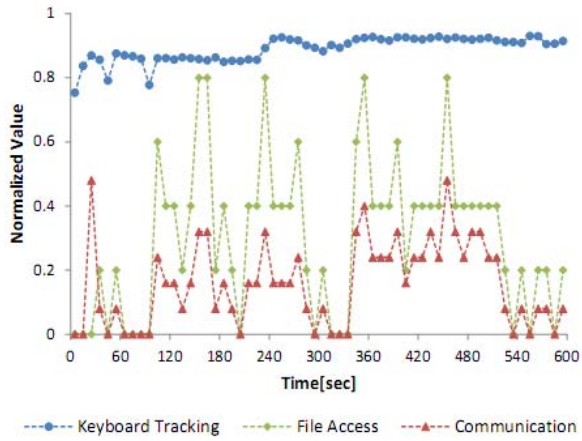
For *spybot* instance, the results from E1 and E2 show that there are significant differences between API call frequency without keystroke simulation and the one with keystroke simulation in both two typing scenarios, as depicted in Fig. 4 and Fig. 5.

From Fig. 4(a) and Fig. 4(b), we can see that although the keyboard tracking API calls generated by *spybot* maintain a high frequency in both E1.1 and E2.1, the frequencies of file access and communication API calls in E2.1 are much higher than the ones in E1.1, especially when short sentences for text editing (using *notepad* from 61 to 240 second) or online chatting (using *mirc* from 301 to 480 second) are typed. The same differences are also shown in Fig. 5(a) and Fig. 5(b) when long sentences are entered. Therefore, we can conclude that the keystroke agent does have the ability to amplify the malicious behavior exhibited by *spybot*.

The intercepted API calls invoked by *spybot*, *notepad* and *mirc* are used to generate corresponding signals using the method described in section IV-B. The DCA then processes and analyzes these signals to determine which

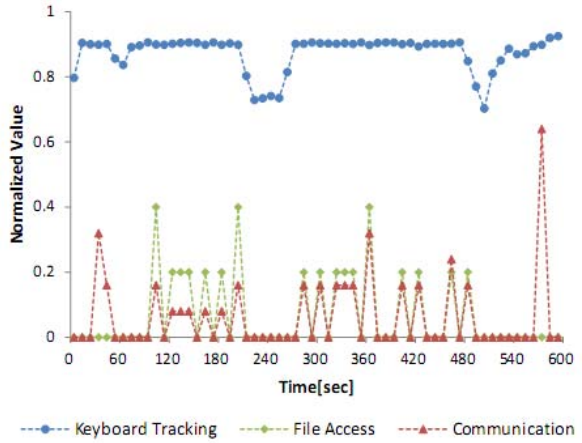


(a) API functions invoked by *spybot* in E1.1

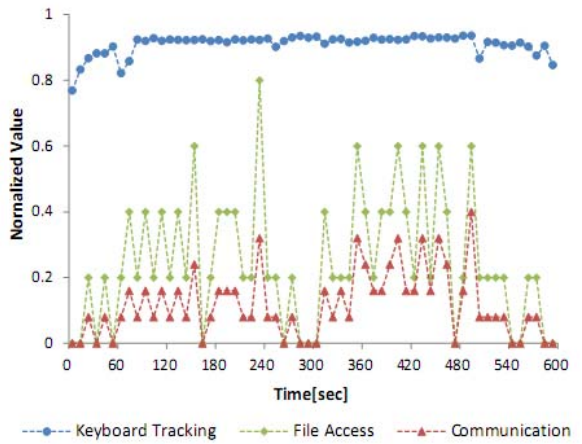


(b) API functions invoked by *spybot* in E2.1

Figure 4. API functions invoked by *spybot* in long sentences scenarios (E1.1 and E2.1). We use *notepad* program for text editing during 61-240s and *mirc* program for online chatting during 301-480s.



(a) API functions invoked by *spybot* in E1.2



(b) API functions invoked by *spybot* in E2.2

Figure 5. API functions invoked by *spybot* in long sentences scenarios (E1.2 and E2.2). We use *notepad* program for text editing during 61-240s and *mirc* program for online chatting during 301-480s.

process has keylogger behaviors. Table I gives the results from the DCA. The values in the last two columns are the mean values and standard deviation values (in parentheses) in 10 repeated experiments.

A threshold (T) is applied to *MAC* to make the final classification decision. The process whose *MAC* is higher than T is termed malicious, and vice-versa. Because the dataset contains one malicious instance and two benign instances, we can define $T = 1 / (1+2)$. So the process with $MAC > 1/3$ is considered to be a keylogger process. From the table, we observe that the DCA detects the keylogger process in all scenarios except in E1.2. This means that the keylogger is more difficult to be detected when user inputs long sentences. However, with the help of the keystroke agent, the DCA detects keylogger process with no false negatives no matter which input mode is used. It is also noteworthy that no false positives are generated in all scenarios.

B. Discussions

From the results of the experiments, we can find that although the *MAC* values in Table I are relatively low compared to the results from the experiments in [8], the basic experimental conclusions do not change:

- Keystroke simulation can enhance the visibility of keylogger behaviors, and thus can improve the detection performance of the DCA.
- Keystroke patterns have an obvious impact on the detection performance of the DCA. And the effects of keystroke simulation vary in different input mode.

Fig. 4 and Fig. 5 demonstrate that the keystroke agent we implemented can induce *spybot* to perform more file access and communication behaviors. And the simulation method improves the *spybot* detection performance of the DCA to some extent in the same environment, as depicted in Table I. The *MAC* value increases by about 58% and 11% in average when we type long sentences (E1.2 and E2.2) and short sentences (E1.1 and E2.1) respectively. In 20 experiments without simulation (E1.1 and E1.2), only 10 experiments detects *spybot*, detection rate is 50%. In contrast, all 20 experiments report *spybot* detection when keystroke agent is used.

TABLE I.
RESULTS FROM THE DCA

Scenario	Process Name	The Number of Antigens	MAC
E1.1	spybot notepad mirc	6767(23.0) 1470(0) 1804(3.8)	0.540(0.0049) 0.087(0.0022) 0.087(0.0027)
E1.2	spybot notepad mirc	5164(35.5) 1140(0) 1795(5.9)	0.271(0.0111) 0.024(0.0020) 0.037(0.0030)
E2.1	spybot notepad mirc	8045(23.2) 1520(0) 1718(1.4)	0.599(0.0065) 0.064(0.0030) 0.109(0.0024)
E2.2	spybot notepad mirc	6819(43.4) 1640(0) 1804(5.4)	0.427(0.0060) 0.056(0.0033) 0.071(0.0029)

The improvement of detection efficiency in long sentences scenarios is much higher than short sentences scenarios. That may because the *spybot* behaviors are already obvious enough since the '*Enter*' key is typed frequently when the user type short sentences. So there is not much room for improvement in short sentences scenarios.

The keystroke agent passes synthesized random keys to a hidden window created by it. The keylogger not only can intercept these keys, but also can know the destination of them (We find *spybot* has this ability in experiments). As a result, the keylogger can defeat the keystroke agent by the way that not to handle the keys send to the hidden window created by the agent. The future work will find a possible way to solve it.

In addition, only Fixed Square Wave pattern keys are generated by the keystroke agent in experiments because we are more concerned about the feasibility of our approach and it is easy to be implemented and analyzed. In future work, we will further analyze the effects of the other two patterns so as to identify the application range of these patterns.

VI. CONCLUSION

The success of any keylogger is determined by its ability to evade detection. In this paper, we analyzed the evasion mechanisms used by common software keyloggers and proposed an induction-correlation framework for keylogger detection. In this framework, keystrokes simulation raises the frequency of the keystrokes, and thus induces keyloggers produce more malicious behaviors to deal with these synthesized keystrokes. Then the 'amplified' behaviors are correlated by the DCA in order to find the keylogger process as early as possible to reduce the loss of privacies. Experimental results showed that the framework we built can improve the keylogger detection rate and reduce the possibility of successful evasion.

ACKNOWLEDGMENT

This work was supported by the Defense Industrial Technology Development Program of PR China (GrantNo. A1420080183).

REFERENCES

[1] N. Patterson and M. Hobbs, "Virtual World Security Inspection", *Journal of Networks*, Vol. 7, No. 6, 2012, pp. 895–907.

[2] T. Holz, M. Engelberth, and F. Freiling, "Learning More about the Underground Economy: A Case-Study of Keyloggers and Dropzones", in *Proc. of 14th European Symposium on Research in Computer Security*, 2009, pp. 1–18.

[3] S. Sagiroglu and G. Canbek, "Keyloggers", *Technology and Society Magazine*, Vol. 28, No. 3, 2009, pp. 10–17.

[4] M. Baig and W. Mahmood, "A Robust Technique of Anti Key-Logging Using Key-Logging Mechanism", in *Proc. of Digital EcoSystems and Technologies Conference*, 2007, pp. 314–318.

[5] W. Luo, N. Li, and Y. Tang, "Reverse Analysis of Malwares: A Case Study on QQ Passwords Collection", *Journal of Software*, Vol. 7, No. 8, 2012, pp. 1706–1712.

[6] M. Aslam, R. Idrees, M. Baig, and M. Arshad, "Anti-Hook Shield against the Software Key Loggers", in *Proc. of the National Conference on Emerging Technologie*s, 2004, pp. 189–191.

[7] Y. Al-Hammadi and U. Aickelin, "Detecting Bots Based on Keylogging Activities", in *Proc. of the 3rd International Conference on Availability, Reliability and Security*, 2008, pp. 896–902.

[8] J. Fu, Y. Liang, C. Tan, and X. Xiong, "Detecting Software Keyloggers with Dendritic Cell Algorithm", in *Proc. of the International Conference on Communications and Mobile Computing*, 2010 , pp. 111–115.

[9] S. Manzoor, M. Shafiq, S. Tabish, and M. Farooq, "A Sense of 'Danger' for Windows Processes", in *Proc. of the 8th International Conference of Artificial Immune System*, 2009, pp. 220–233.

[10] S. Qi, M. Xu, and N. Zheng, "A Malware Variant Detection Method Based on Byte Randomness Test", *Journal of Computers*, Vol. 8, No. 10, 2013, pp. 2469–2477.

[11] M. Xu, B. Salami, and C. Obimbo, "How to Protect Personal Information against Keyloggers", in *Proc. of the 9th International Conference on Internet and Multimedia Systems and Applications*, 2005, pp. 275–280.

[12] K. Nasaka, T. Takami, T. Yamamoto, and M. Nishigaki, "A Keystroke Logger Detection Using Keyboard-Input-Related API Monitoring", in *Proc. of 14th International Conference on Network-Based Information Systems*, 2011, pp. 651–656.

[13] G. Bancroft and G. O'Sullivan, *Maths and Statistics for Accounting and Business Studies*, 2nd ed., McGraw-Hill, 1988.

[14] J. Greensmith, U. Aickelin, and G. Tedesco, "Information Fusion for Anomaly Detection with the Dendritic Cell Algorithm", *Information Fusion*, Vol. 11, No. 1, 2010, pp. 21–34.

[15] L. Ding, F. Yu, and Z. Yang, "Survey of DCA for Abnormal Detection", *Journal of Software*, Vol. 8, No. 8, 2013, pp. 2087–2094.

[16] C. Herley and D. Florencio, "How to Login from an Internet Cafe without Worrying about Keyloggers", in *Proc. of Symposium on Usable Privacy and Security*, Vol. 6, 2006, pp. 1–2.

[17] MSDN-Keyboard Input. http://msdn2.microsoft.com/en-us/library/ms645530.aspx.
[18] MSDN-File Management Functions. http://msdn2.microsoft.com/en-us/library/aa364232.aspx.
[19] MSDN-Winsock Functions. http://msdn2.microsoft.com/en-us/library/ms741394.aspx.
[20] mIRC client application. http://www.mirc.com

Jun Fu received a PhD. degree on computer software and theory from Wuhan University in 2011. Now he is an engineer of the 28th Research Institute of China Electronics Technology Group Corporation. His research interests include artificial immune system, network security and stealthy malware detection.

Huan Yang received a PhD. degree on computer software and theory from Wuhan University in 2012. Now she is an engineer of the 28th Research Institute of China Electronics Technology Group Corporation. Her research interests are in the field of software reliability, artificial immune system, and software health management. Currently, she focuses on applying the AIS on web server aging.

Yiwen Liang is currently a Professor and Ph.D. advisor in Computer School, Wuhan University. His research interests include artificial immune system and anomaly detection. He has been awarded three NSFC research funding as principal investigator on topics including AIS, Network Security and Anomaly Detection. Prof. Liang is an associate Secretary General of the Natural Computing Committee of Chinese Association for Artificial Intelligence (CAAI), a member of the council of the IEEE education association China sub-commission.

Chengyu Tan is currently an associate professor and master advisor in Computer School, Wuhan University. She received a B.E. (1990) and a M.S. (1996) degrees from Wuhan University of Hydraulic and Electrical Engineering, and completed a PhD in software and theory at Wuhan University in 2007. Her research interests include artificial immune system and natural computation.