Virtual Machine Introspection for Anomaly-Based Keylogger Detection

Huseyn Huseynov[*], Kenichi Kourai[†], Tarek Saadawi[*] and Obinna Igbe[*]
[*]Department of Electrical Engineering
City University of New York, City College, New York, United States
Emails: hhuseynov@ccny.cuny.edu, saadawi@ccny.cuny.edu, obiigbe91@gmail.com
[†]Department of Computer Science and Networks
Kyushu Institute of Technology, Fukuoka, Japan
Email: kourai@ksl.ci.kyutech.ac.jp

***Abstract*—Software Keyloggers are dominant class of malicious applications that surreptitiously logs all the user activity to gather confidential information. Among many other types of keyloggers, API-based keyloggers can pretend as unprivileged program running in a user-space to eavesdrop and record all the keystrokes typed by the user. In a Linux environment, defending against these types of malware means defending the kernel against being compromised and it is still an open and difficult problem. Considering how recent trend of edge computing extends cloud computing and the Internet of Things (IoT) to the edge of the network, a new types of intrusion-detection system (IDS) has been used to mitigate cybersecurity threats in edge computing. Proposed work aims to provide secure environment by constantly checking virtual machines for the presence of keyloggers using cutting edge artificial immune system (AIS) based technology. The algorithms that exist in the field of AIS exploit the immune system's characteristics of learning and memory to solve diverse problems. We further present our approach by employing an architecture where host OS and a virtual machine (VM) layer actively collaborate to guarantee kernel integrity. This collaborative approach allows us to introspect VM by tracking events (interrupts, system calls, memory writes, network activities, etc.) and to detect anomalies by employing negative selection algorithm (NSA).**

***Index Terms*—Artificial Immune System, Edge Computing, Virtual Machine Introspection, Keylogger, Spyware, Invasive Software, Genetic Algorithm.**

I. Introduction

Software keyloggers are one of the most serious types of malware that surreptitiously log keyboard activity, and in most cases exfiltrate the recorded data to third parties. Despite many conducted research and commercial efforts, keyloggers can still pose a significant threat of stealing personal and financial information. Depending on which part of the computer they are embedded into and what operating system used, all keyloggers can be categorized as either hardware-based or software-based. The latter is the most common and in turn divided into several categories that can be found in Section II of this paper. In comparison with other types of malware, such as viruses and worms, the goal of keyloggers is generally not to cause damage or to spread to other systems. Instead, software keyloggers monitor the behavior of users and steal private information, such as keystrokes and browsing patterns. This information is then sent back to third parties and in best cases can be used as a basis for targeted advertisement or marketing analysis, while in the worst-case malicious application can steal all the private information, bank account passwords or any confidential information.

Since the last few years, hackers have started paying more attention to Linux and mac-OS platforms, making them a new target for viruses, trojans, spyware, adware, ransomware, and other nefarious threats. This tendency is primarily associated with proliferation of Linux based IoT devices, as well as increased interest for Linux OS among regular users and organizations. Despite the fact that the attack surface for Linux is much much smaller, it has its own share of vulnerabilities and malware threats, which is why proactive monitoring is important to keep the system safe. Therefore, providing a single AI based solution as comprehensive protection for multiple platforms is crucial, contrary to the existing signature-based threat detection technique. The biggest disadvantage of signature-based keyloggers is that, while using them user can be sure only being protected from keyloggers that are in their signature-base list, thus staying absolutely vulnerable to others. For this reason, security experts are now focusing on using anomaly-based (or behavior-based) detection techniques, which analyze system calls and network utilization of a process to classify it as benign or malicious.

Our work focused on development of a distributed application by using negative selection algorithm (NSA) derived from artificial immune system (AIS). This software will reside on a host machine and constantly introspect multiple virtual machines. Once the suspicious process(es) detected based on anomalous behavior, the application in real-time will notify the system administrator about potential threat. The crucial part of this software is Virtual Machine Introspection (VMI), which addresses several security issues from outside the guest OS without relying on functionality that can be rendered unreliably by advanced malware. VMI tool (KVMonitor) acts by tracking the events (interrupts, memory read/writes, network activities, and so on). Collected data is being processed by AIS based Intrusion Detection System (IDS) for anomaly detection.

II. Linux Keyboard Driver Structure and Classification of Keyloggers

The main idea behind keyloggers is to get in between any two links in the chain of events between when a key is pressed and when information about that keystroke is displayed on the monitor. One of the ways how this can be achieved in Linux OS is by running user-space based keystroke loggers. In order to understand closely how this process works it's important to find out how Linux handles the keyboard driver.

A. Linux Keyboard Driver

One of the basic component of any Linux environment is *Display Server* – an application which sits between the graphical interface and the kernel. Its primary task is to coordinate the input and output of its clients (programs and applications running GUI interface) to and from the rest of the OS, the hardware, and each other. It communicates with its clients over the display server protocol which can be network-transparent and network capable. Commonly known display server communications protocols include X11, Wayland, Mir, etc. In this paper we have focused on employing Linux with X11 protocol [10].

When user presses a key on the keyboard, the keyboard sends corresponding scancodes to keyboard driver. A single key press can produce a sequence of up to six scancodes [8], [10].

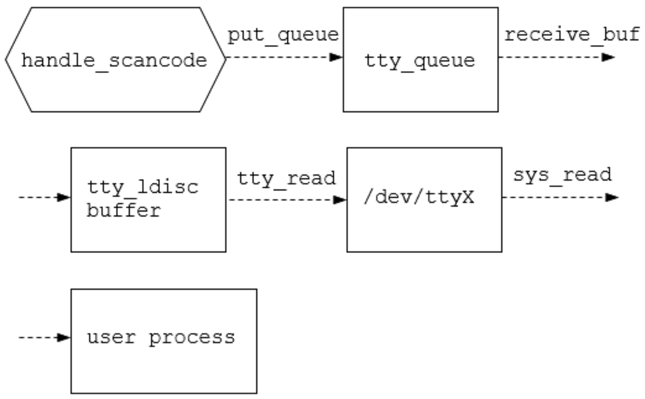


Fig. 1. Processing of keystrokes from Linux terminal.

The *handle_scancode()* function in the keyboard driver parses the stream of scancodes and converts them into a series of key-press and key-release events called keycode by using a translation-table via function *kbd_translate()*. Each key is provided with a unique keycode *k* in the range 1-127. Pressing key *k* produces keycode *k*, while releasing it produces keycode *k+128*. For instance, pressing key "a" returns keycode 30, while releasing "a" produces keycode 158 (128+30). On the next step, keycodes are being converted to key symbols by scanning them on the appropriate keymap. There are eight possible modifiers (shift keys - Shift, AltGr, Control, Alt, ShiftL, ShiftR, CtrlL and CtrlR), and depending on which modifiers are currently active the appropriate keymap is used. Once the characters obtained they are placed into the raw tty queue – *tty_flip_buffer*. In this queue, *receive_buf()* function is called periodically in order to get characters from *tty_flip_buffer* and put them into tty read queue, as shown in Fig. 1.

The keyboard driver can be in one of four modes depending on what type of data the application will get as keyboard input:

- RAW MODE (*scancode*): the application receives scancodes for input. It is used by application that implement their own keyboard (ex. X11).
- MEDIUMRAW MODE (*keycode*): the application receives information on which keys (identified by their keycodes) get pressed and released.
- XLATE MODE (*ASCII*): the application effectively receives characters as defined by the keymap, using an 8-bit encoding.
- UNICODE MODE (*unicode*): the only difference between this mode and XLATE MODE is by allowing the user to compose UTF-8 unicode characters by their decimal value, using Ascii_0 to Ascii_9, or their hexadecimal value, using Hex_0 to Hex_9. A keymap can be set up to produce UTF-8 sequences (with U+DDDD pseudo-symbol, where D is a hexadecimal digit).

When user process want to get user input, it calls function *read()* on stdin (standard input stream) of the process. Function *sys_read()* calls *read()* defined in *file_operations* structure (linux/fs.h), which is pointed to tty_read of corresponding tty (/dev/ttyX on Fig.1) to read input characters and return to the process.

B. Classification of Software Keyloggers

Kernel based: A program that obtains root access to hide itself in the OS and intercepts keystrokes that pass through the kernel [7]. Such keyloggers reside at the kernel level, which makes them difficult to detect, especially for user-mode applications that don't have root access. They can often be implemented as rootkits that subvert the operating system kernel to gain unauthorized access to the hardware. The fact that a keylogger using this method can act as a keyboard device driver makes them a powerful tool in the hands of intruders.

User-space or API based: These keyloggers hook keyboard APIs inside a running application. The keylogger registers keystroke events, as if it was a normal piece of the application instead of malware. The keylogger receives an event each time the user presses or releases a key and quietly records it. These types of keyloggers follow the routine of how Linux keyboard driver is processing keystrokes described in the previous section. Typically, they can be written in a known application or pretending to be a separate system process. Depending on the logic of malware, these keyloggers are mostly trying to establish a network connection in order to send their logs. Provided work also focused on detection of user-space based keyloggers.

Form grabbing or formjacking: These types of keyloggers logs web form submissions by recording the web browsing on submit events. This happens when the user completes a form and submits it, usually by clicking a button or hitting enter key.

Javascript based: A malicious script tag is injected into a targeted web page and listens for key events such as *onKeyUp()*. Script can be injected via a variety of methods, including cross-site scripting, man-in-the-browser, man-in-the-middle, or as a result of compromise on the remote server.

Provided classification shows the most common types of keyloggers that can be found in Windows, Mac or Linux operating systems. The list can be continued, since there are many other types of keyloggers such as ***CSS based***, ***memory-injection based*** and so on. Experiments were conducted to test capabilities of proposed framework to improve the detection rate and reduce the possibility of successful evasion.

III. Keylogger Detection

Our approach is explicitly focused on designing a detection technique for *unprivileged user-space* keyloggers running on Linux-based virtual machines. Unlike other classes of keyloggers, a user-space keylogger is a background process which registers operating system supported hooks to surreptitiously eavesdrop (and log) every keystroke issued by the user into the current foreground application. Our goal is to prevent user-space keyloggers from stealing confidential data originally intended for a (trusted) legitimate foreground application. Designed software consists of two major parts: virtualization tool (*KVMonitor*), which constantly introspects virtual machine, and genetic algorithm based application, which analyze captured data for potential anomalies.

Proposed work employs artificial immune system (AIS) based algorithm for anomaly detection. One significant feature of the theory immunology is the ability to adapt to changing environments and dynamically learning. AIS is inspired by the human immune system (HIS), which has the ability to distinguish internal cells and molecules of the body against diseases [1], [4].

A. Artificial Immune System (AIS)

Anomaly-based intrusion detection system monitors network traffic and user/system activity for abnormal behavior. Unlike the signature-based detection method, the anomaly-based IDS can detect both known attacks and unknown (zero-day) attacks. Hence, it is a better solution than the signature-based detection technique if its system is well designed [4]. Therefore, efficiency of anomaly-based IDS depends on multiple requirements such as what kind of algorithm has been deployed, what is the main target, understanding generated input data, application run-time and so on.

The algorithms that exist in the field of AIS exploit the immune system's characteristics of learning and memory to solve diverse problems. These algorithms are based on human immune system (HIS) models taken from the field of immunology. Immunology uses models for understanding the structure and function of the immune system. The self-nonself (SNS) model is an immunology model that has been successfully utilized in AIS in the design of IDS systems to detect network attacks; both insider and outsider [1], [6]. Our approach lies on application of Negative Selection Algorithm (NSA) as part of the AIS to detect and classify suspicious processes.

The negative selection algorithm is based on the self-nonself model of the human immune system (HIS). The first step of the NSA according to Forest et al. [15] involves randomly generating detectors (which is the AIS's equivalent of B cell in HIS) in the complementary space (i.e., space which contains no seen self elements) and then to apply these detectors to classify new (unseen) data as self (no data manipulation) or non-self (data manipulation). For this purpose, the whole shape-space U is divided into a self set S and a non-self set N with

$$U = S \cup N \ and \ S \cap N = \emptyset \tag{1}$$

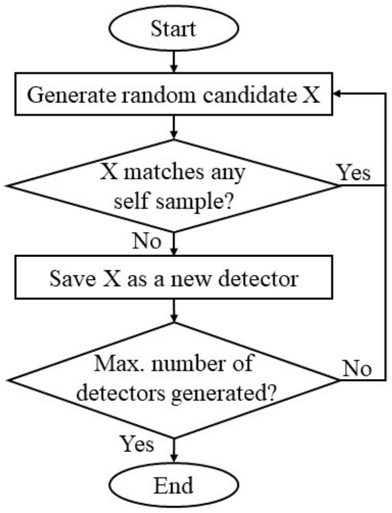
The steps undertaken by NSA can be subdivided into 6 phases shown on Table I. First, at the *Data Capture Phase*, normal profiles (also called self profiles or self samples) are extracted from the training data. Each data instance in the normal profile is obtained from the data instances captured by the system during periods of normal virtual machine activity (i.e., during the absence of any keyloggers). The process of *Data Capture* has been obtained by employing multiple times *KVMonitor* – our virtual machine introspection (VMI) software [2]. *KVMonitor* resides in the host machine and can introspect multiple virtual machines. More detailed about KVMonitor and the process of VMI is described in the next subsection.

TABLE I
Six modules of Artificial Immune System based IDS [1]

Steps	Phases
1	DATA CAPTURE
2	FEATURE SELECTION
3	DATA PREPROCESSING
4	DETECTOR DISTRIBUTION
5	MONITORING
6	DETECTOR GENERATION

On the phase called *Detector Generation* we employ an evolutionary approach using a genetic algorithm (GA). GAs are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Each generation consists of a population of character strings that are analogous to the chromosome that we see in our DNA. Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution [6].

A detector is defined as $d = (C, r_d)$, where $C = \{c_1, c_2, ..., c_m\}, c_i \in \mathbb{R}$, as an m-dimensional point that corresponds to the center of a unit hypersphere with $r_d \in \mathbb{R}$ as its unit radius. Fig. 2 shows a generic flowchart of this detector generation process [4]. Randomly generated detectors that match any self sample is discarded. The detector generation process is halted when the desired number of detectors is obtained.



Start
Generate random candidate X
X matches any self sample?
Yes
No
Save X as a new detector
Max. number of detectors generated?
No
Yes
End

Fig. 2. NSA detector generation phase.

To find out if a detector $d = (C, r_d)$ matches any normal profile, the distance (D) between this detector and it's nearest self profile neighbor $(X^{normal}, r_s) \in S$ is computed, where X^{normal} is also an m-dimensional point $\{x_1^{normal}, x_2^{normal}, ..., x_m^{normal}\}$ and corresponds to the center of a unit hypersphere with r_s as its unit radius. The distance (D) is obtained using *Euclidean distance measure* given by equation (2).

$$\sqrt{(\sum_{i=1}^{m}(c_i - x_i^{normal}))^2} \tag{2}$$

A variable radius is assigned to the new detector sample based on the minimum distance from the detector that is going to be retained and its nearest self/normal profile (i.e., $(D) - r_s$).

The *NSA keylogger detection* phase matches system calls, interrupts, traffic records from testing data with the stored detectors. These data obtained by constantly running *KVMonitor* with given time interval. Application collects data based on following three Linux API categories:

1. **Keyboard Tracking**: *XkbGetState()*, *XKeysymToString()*, *Read()* and *Write()* system calls [10].
2. **File Access**: *CreateFile*, *OpenFile*, *ReadFile* and *WriteFile* [8].
3. **Network**: *socket*, *tcp_socket*, *udp_socket*, *send*, *sendto*, *sendmsg* [14].

After the data has been collected with respect to given time interval, Euclidean distance matching used for matching purposes. For any instance in the testing data, if the radius of it's hypersphere falls within the radius covered by any stored detector, this instance is considered to be *anomaly*, otherwise, it is considered to be *normal*.

B. Virtual Machine Introspection (VMI)

The performance becomes important factor when users choose virtualization software, e.g. Xen and KVM. Proposed application for efficient VM introspection (KVMonitor) was 32 times faster than existing LibVMI during memory introspection comparison. The experimental results showed that checking the kernel memory was 118 times faster than in Xen [2], [3].

KVMonitor is a library linked to IDSes that allows offloaded IDSes to introspect the memory, disks and network of VMs. As illustrated in Fig. 3, an IDS is offloaded onto the host operating system. An offloaded IDS introspects a VM by accessing virtual devices managed by QEMU-KVM via KVMonitor [2]. One advantage of the KVMonitor mechanism is efficiency. KVMonitor communicates with QEMU-KVM only at once because it can translate a series of addresses using the same value of the CR3 register. Another advantage is that the KVMonitor can naturally translate virtual addresses of a specific process by traversing its page directory [2].

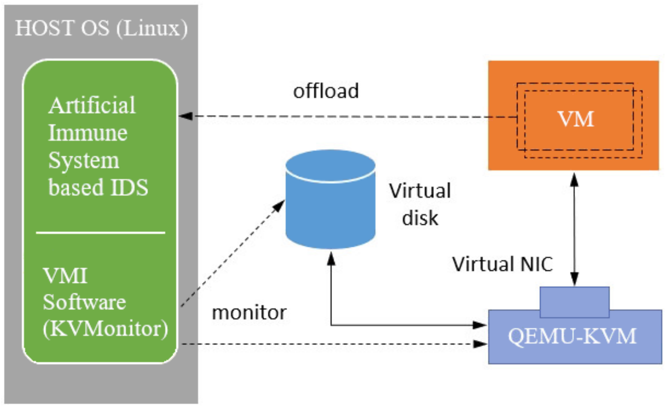


Fig. 3. The architecture of proposed VM introspection [2].

Proposed IDS application resides in the host machine and utilizes KVMonitor N times with given time intervals T. Each time it collects necessary data about VM such as interrupts, system calls, memory writes, network activities and other data. As shown in Table I, once the data has been collected the application starts to perform negative selection algorithm (NSA) in order to distinguish normal processes from suspicious. This has been achieved on the last *"Detector Generation"* phase (Table I) where we apply a genetic algorithm (GA) illustrated in Fig. 2.

Once the malicious application has been detected based on its anomalous behavior, proposed IDS in real-time will send a notification to system administrator. The notification includes, but is not limited to, the process id number of the suspicious application, detected time, system utilization characteristics and other important information.

IV. Experiments

To evaluate the ability of detection real-world keyloggers, we experimented with more than dozens of keyloggers from the top open source software list [9]. To carry out the experiments, we manually installed each keylogger in a virtual machine, launched our detection system from the host machine for $N \cdot T$ ms, and recorded the results. In this paper, we provide experiments conducted on three different types of open source keyloggers provided on Table II. Among the listed keyloggers *Blueberry* has been slightly modified, so it can send the log files to remote server over TCP protocol once the number of entered characters become 250. On the oher hand, *EKeylogger* keeps data in the buffer and sends it by email every 10 seconds. Provided keylogger detection system was able to successfully analyze and detect all anomaly correlations, as well as to return process ID(s) (*PID*) of keyloggers running in the VM.

TABLE II
Detection Results

Keylogger	**Detection**	**Note**
Logkeys	✓	Multi functional GNU/Linux keylogger. Logs all common character and function keys [11].
Blueberry	✓	Opens a stream to the keyboard event handler and gets every key press. Create logs when the buffer gets 250 characters and sends it to remote server over TCP protocol [12].
EKeylogger	✓	Sends recorded keystrokes every 10 sec directly to email address [13].

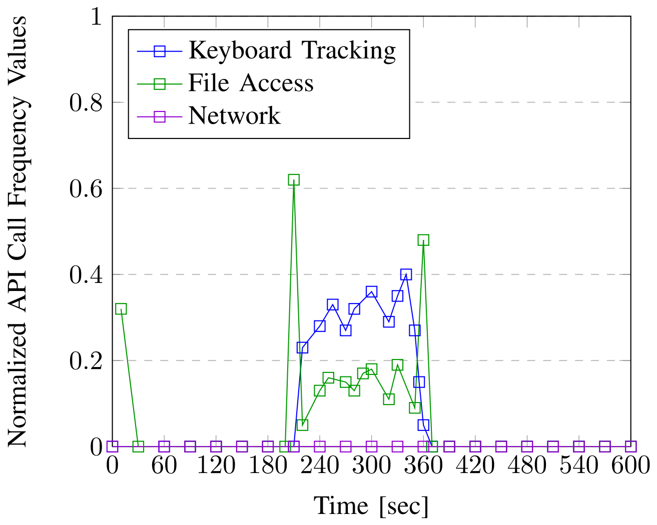
System configuration is provided as follows:

- **HOST**: Intel® Core™ i5 2.5 GHz CPU, Memory 16 GB DDR4-2400 PC4 SO-DIMM, OS Ubuntu 18.04 LTS
- **GUEST**: QEMU/KVM, Allocated CPUs “3”, Allocated memory 2 GB, Virtual Network Interface “virtio” over bridge, Channel Device “spicevmc”, Virtual Input Device “Generic PS2 Keyboard”, OS Ubuntu 18.04 LTS

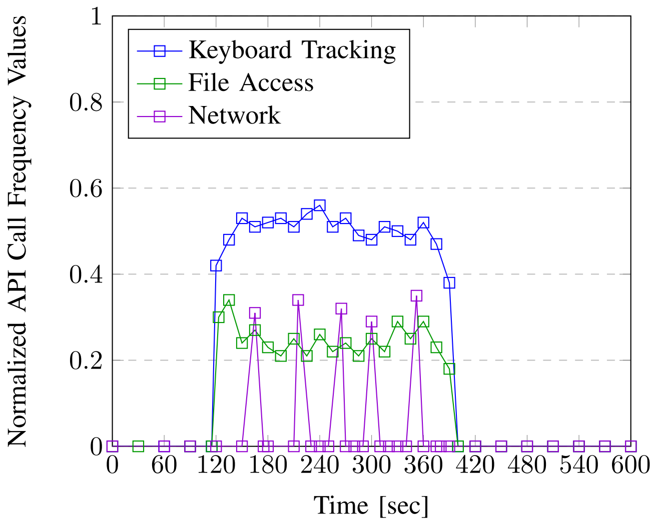
The experiments are divided into two cases to show the detection performance of proposed system. In the first case we monitor each keylogger for scenario of typing short sentences (30-85 characters) in address bar of *Mozilla Firefox* browser (*Chart (a)*). In the second case, we type long sentences (300-1350 characters) using default text editor *gedit* (*Chart (b)*). In both cases, after starting the keylogger in VM we wait first 60 seconds and then start typing process.

The result of virtual machine introspection with activated *Logkeys* keylogger provided on *Chart (a)*. On this example, we typed a short sentence (30-85 characters) in address bar of *Firefox* browser. The x-axis represents time in seconds while the y-axis represents 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).

(a) API calls invoked by *Firefox* using *Logkeys*



(b) API calls invoked by *gedit* using *Blueberry*



```
...
Gen 8
   1      0       4652     0    0
   577    0       2920     0    0
   601    0       620      0    0
   795    24576   10932    1    0
   4436   98304   0        1    0
   5200   45602   0        1    T
Detected
Process ID: 4436 Fitness: 28.90
Process ID: 5200 Fitness: 17.70
...
```

Listing 1. Results after completion of 8^{th} generation.

Chart (b) shows an example of running *Blueberry* keylogger in our guest machine. As shown from the chart, on each time frame when number of entered characters become 250, keylogger saves data from the buffer to log file, establishes TCP connection and sends the log file to remote server. Similar result has been achieved from running *EKeylogger* in VM. To get closer to real user keystroke patterns, we collect 200 commonly used English sentences and type them one bye one in corresponding scenarios. Output shown on *Listing 1* represents detection process while running two keyloggers on the guest machine – *Logkeys* (PID=4436) and *Blueberry* (PID=5200). On this example, we started *Blueberry* with delay of 120 seconds after *Logkeys* has been executed. As shown from the output, captured in the middle of running process, application was able to detect both of the keyloggers on 8^{th} generation.

In all provided examples we set $T = 10$ sec and $N = 60$, which means that our detection system will run 60 generations to complete genetic algorithm (GA) and will execute *KVMonitor* with the time interval of 10 sec on each generation. On average, application was able to detect anomalies within the first 15 seconds of typing process. However, accuracy and overall completion time depends on number of generations N.

A. Future Works

Detecting user-space keyloggers in Linux based VMs is a challenging process. To improve the detection accuracy and speed-up overall run-time we outlined necessary future works:

1. **Hidden Keyboard Simulator**: Provided examples shows that induction of API calls happens during the typing process. If we start keylogger and do not type anything, system call *Write()* and other crucial parameters won't correlate. Therefore, in order to "amplify" detection process by inducing keyloggers, we're working on developing a hidden keyboard simulator, which will be running on VM, acting like a real user.
2. **Expanding Area of Detection**: As noted on Section II, there are many other types of keyloggers. One of our next steps will be expanding existing detection system by adding up Linux *Kernel-based* keyloggers.
3. **Performing Wide Tests**: Another important factor is to implement tests on real-world cloud computers. Proposed system reveals maximum detection accuracy of 99.68% by testing it on one virtual machine. Running this system on cloud based VPN server is on our checklist.

V. Conclusion

When some virus or malware succeed in taking control of a given digital machine, its first task is to deactivate any anti-malware software on the digital machine and prevent installation of such software, so that the malware can keep on controlling the system. Thus, classic anti-malware software must reliably block malware upon entry; if they miss one, they have lost. On the other hand, with a virtual machine (VM), the anti-malware could run on the host, outside of the guest system, thus impossible to deactivate by malware who subverted the guest system. This is one of the main concepts behind virtual machine introspection (VMI). Another key factor of proposed work is employing evolutionary technique using a genetic algorithm. Encapsulating VMI and artificial immune system based approach for detecting malicious activities on Linux stationed virtual machines provides efficient way of monitoring VMs, and therefore cloud-based systems.

References

[1] O. Igbe, I. Darwish, T. Saadawi "Distributed Network Intrusion Detection Systems: An Artificial Immune System Approach," Dept. of Electrical Engineering, City University of New York, City College, 2016.

[2] K. Kourai and K. Nakamura "Efficient VM Introspection in KVM and Performance Comparison with Xen," Department of Creative Informatics, Kyushu Institute of Technology, Fukuoka, Japan 2014.

[3] K. Kourai and K. Juda "Secure Offloading of Legacy IDS Using Remote VM Introspection in Semi-trusted Clouds," Department of Creative Informatics, Kyushu Institute of Technology, Fukuoka, Japan 2016.

[4] O. Igbe, O. Ajayi, T. Saadawi "Detecting Denial of Service Attacks Using a Combination of Dendritic Cell Algorithm and the Negative Selection Algorithm," The 2nd IEEE International Conference on Smart Cloud (SmartCloud 2017), New York, NY, 2017.

[5] D. Wang, L. He, Y. Xue, and Y. Dong "Exploiting artificial immune systems to detect unknown dos attacks in real time," in Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on, vol. 2. IEEE, 2012, pp. 646–650.

[6] X. Z. Gao, S. J. Ovaska, and X. Wang "Genetic Algorithms-based Detector Generation in Negative Selection Algorithm," IEEE Mountain Workshop on Adaptive and Learning Systems, Institute of Intelligent Power Electronics, Helsinki University of Technology, Finland 2006.

[7] S. Ortolani, C. Giuffrida, C. Crispo "Unprivileged Black-Box Detection of User-Space Keyloggers," IEEE Transactions on Dependable and Secure Computing 99(1):1, 2011.

[8] D. Le, C. Yue, T. Smart, H. Wang "Detecting Kernel Level Keyloggers Through Dynamic Taint Analysis," Technical Report, College of William & Mary, Department of Computer Science, 2008, pp 3–5.

[9] Top Open Source Keylogger Projects https://awesomeopensource.com/projects/keylogger

[10] A. Benson, G. Aitken, E. Fortune, D. Converse, G. Sachs, W. Walker "The X Keyboard Extension: Library Specification,", https://www.x.org/releases/X11R7.7/doc/libX11/XKB/xkblib.html

[11] Logkeys – a GNU/Linux Keylogger. The source code for the index construction and search is available at https://github.com/kernc/logkeys. It is implemented in C and dual licensed under the terms of either GNU GPLv3 or later, or WTFPLv2 or later.

[12] Blueberry – Simple Open Source Keylogger for Linux. The source code for the index construction and search is available at https://github.com/PRDeving/blueberry. It is implemented in C and has open license.

[13] EKeylogger or simply Keylogger. The source code for the index construction and search is available at https://github.com/aydinnyunus/Keylogger. It is implemented in Python for the purpose of testing the security of information systems.

[14] C. Benvenuti, *Understanding Linux Network Internals*, 2nd ed. Sebastopol, CA: O'Reilly, 2006, pp 541–555.

[15] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrimination in a computer," in Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on. Ieee, 1994, pp. 202–212.