



# Detection of Virtual Machines Based on Thread Scheduling

Zhi Lin<sup>1(✉)</sup>, Yubo Song<sup>2,3</sup>, and Junbo Wang<sup>1,4</sup>

<sup>1</sup> School of Information Science and Engineering, Southeast University, Nanjing 211111, China

<sup>2</sup> Key Laboratory of Computer Network Technology of Jiangsu Province, School of Cyber Science and Engineering, Southeast University, Nanjing 211111, China

<sup>3</sup> Purple Mountain Laboratories, Nanjing 211111, China

<sup>4</sup> National Mobile Communications Research Laboratory, Southeast University, Nanjing 211111, China

**Abstract.** With the rapid development of cloud computing, virtual machines are now attracting more and more attention. Virtual machines used at malicious motivation cause enormous threats to the security of computer systems. Virtual machine detection is crucial for honeypot systems and software that provide free trials. Various strategies based on local register values affected by virtualization have been proposed. However, these strategies have a limited scope of application since they can only run natively. What's more, the values they depend on can be modified with ease. In this paper, we propose a new remote virtual machine detection strategy applying to different types of virtual machines and different operating systems based on time difference in thread scheduling. Our main contribution is to set up a probability-based thread scheduling analysis model to describe the time difference between physical machines and virtual machines. This paper shows that the probability distribution of execution time of a piece of CPU-bound code in virtual machines has higher variance along with lower kurtosis and skewness, which make up our index system for detection. Results of Numeric simulation and real test show good agreement and provide a clear criterion for detection. In the real test all the virtual machines and 97.2% of the physical machines were identified correctly.

**Keywords:** Virtual machine · Remote detection · Probability · Thread scheduling

## 1 Introduction

Since virtual machines have been widely used not only by professionals but also by normal people, it is significant for software or a website to detect a virtual machine in order to escape unfriendly acts. Some software needs to change their activities in honeypot systems to evade analysis [4]. Commercial software wants to prevent the overuse of free trial. A virtual-machine based rootkit (VMBR) is

a virtual machine monitor (VMM) installed underneath an existing OS, which controls the victim OS as a virtual machine [9]. The detection of virtual machines is crucial for finding a VMBR. Some special attacks are targeted at virtual machines on cloud infrastructures, and the detection of virtual machines is their first step [13]. Asviya listed three types of attacks in cloud computing including attacks on I/O channels, side-channel/covert channel attacks, and attacks on trusted execution and secured boot technologies [2]. Studies in virtualization detection strategies could help improve the security of computers, mobile devices, and cloud infrastructures.

Code segments that can detect a virtual machine environment is also named as a red pill, on the contrary with a blue pill. There have been many kinds of red pills, mostly run natively because of the dependency on some local fingerprints like Windows registry and CPU registers, or local operations like executing special instructions and reading from special ports. Klein implemented a tool ScoopyNG which detects VMM by SIDT, SGDT instruction, reading value in the Interrupt Descriptor Table Register (IDTR) and Global Descriptor Table Register (GDTR) [10]. Keith proposed a method using Translation Lookaside Buffer (TLB) capacity to detect VMM by revealing the fact that the TLB size will be affected by executing some particular instructions including CPUID in the virtual machine [8]. These methods cause little false positive but usually can be prevented by virtual machine providers or VMBR producers. Sierra-Arriaga listed three types of avoiding detection strategies including the use of thin hypervisors, Timestamp Counter offset manipulation, and the “Blue Chicken” Technique [12]. Moreover, they cannot handle each type of virtual machine with a single standard. For example, virtual machines from different companies may have different SDTR values, and VMBR developers can even modify them at will.

Some researchers proposed detection strategies based on the time difference of some special instructions. These strategies are usually aimed at detecting hardware-assisted virtualization, which is believed to be stealthier to attackers. Brengel used instructions that can provoke a VM Exit and measured the CPU cycles used by the operating system [3]. These instructions cost physical machines less time because they don’t need to perform VM Exit on physical machines. Zhang used Last-Level Cache (LLC) and Level-1 Data (L1D) Cache to detect hardware-assisted virtualization [14]. These strategies are effective most of the time regardless of the types of virtual machines, but they also need to run natively.

Some related works focused on remotely detecting the existence of VMM. Jämthagen used information in IP packets and HTTP requests to infer the source of an Internet connection [7]. But it only works when the virtual machine works under NAT and the host and guest OS should be completely different. Franklin provides a fuzzy benchmarking strategy by calculating the execution time of a piece of code on the remote system [5]. However, the result may be affected by the efficiency of hardware. Ho applied redpill to browsers and combined four types of operations to distinguish virtual machines, hardware-assisted virtual machines with different OS versions, which is also based on the time difference [6].

It also reveals that less running time in physical machines is not a certainty. In this paper, we talk about a remote detection strategy based on the time difference. However, we do not use the absolute value of time difference, which is highly correlated with hardware performance. Instead, this paper cares about the distribution characteristics. In section two we explain some assumptions and definitions involved, and proposes the thread scheduling model. Then in section three numeric simulations and real tests are performed and their results are analyzed. In section four we put brief conclusions and future outlook. Our contributions mainly include:

- We set up a new probability-based thread scheduling model to analyze the difference between virtual machines and physical machines.
- We propose a new remote virtual machine detection strategy based on characteristics of probability distributions instead of the absolute values. It depends slightly on hardware efficiency.
- We design and implement the remote detection system on websites, making it functional on browsers.
- We do numeric simulations and real tests to prove the validity of theories and provide a clear index system in practical use.

## 2 Probability Model of Thread Scheduling

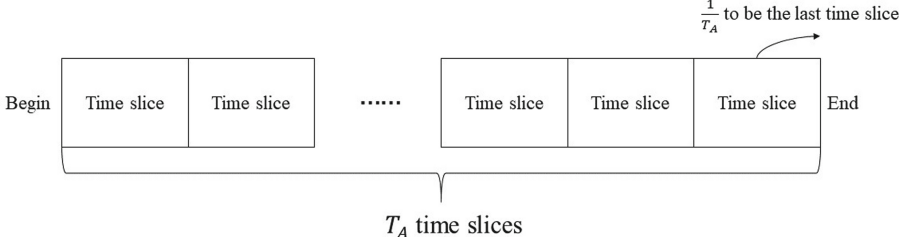
### 2.1 Assumptions and Definitions

**MLFQ to RR.** Thread scheduling is a base activity of an OS. There are many types of scheduling algorithms like First Come First Service (FCFS), Shortest Job First (SJF), Round Robin (RR), etc. But the most frequently used algorithm on PC operating systems is Multi-Level Feedback Queue (MLFQ), the base for Windows and some Linux scheduling algorithms. Different OS also applies their own priority models to achieve more refined control on processes or threads, which makes it difficult to set up a unified model for most of the OS.

However, in some cases, MLFQ in various forms can be simplified to RR. Take Windows for example, if we run a piece of CPU-bound code in thread  $A$  with expected execution time  $T_A$ , and suppose that all the CPU-bound threads share the priority range (normal priority), some characteristics of MLFQ will become similar to RR after a limited period of time. That's because inserted thread, mostly system call with real-time priority, cost far less time. And the scheduler, in most time slices, concentrates on the priority queues where  $A$  and other CPU-bound threads appear. In a single queue, threads are scheduled as RR. In Linux, the case is similar. Real time processes and I/O-bound processes cost little CPU time compared to CPU-bound normal processes. Therefore, the following analysis is based on RR.

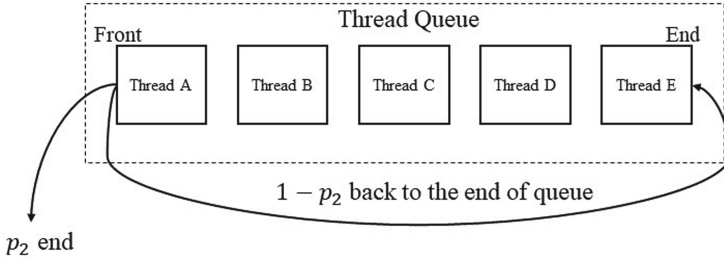
**Probability Model.** At the macro level, a RR operating system is a probability system with thread inserting probability  $p_1$  and thread releasing probability  $p_2$ .

$p_2$  can also be denoted by the average expected execution time of threads  $T_A$  because  $p_2 = \frac{1}{T_A}$ . In other words, the probability that the current time slice is the last one is  $p_2$ . Figure 1 shows the timeline of the execution of a thread.



**Fig. 1.** Thread execution timeline

From the position of operating systems or schedulers, in a certain time slice, there's a  $p_2$  chance that the current thread ends and  $1 - p_2$  chance it goes to the back of the queue, as Fig. 2 shows. The time interval between two runs of thread A on the CPU is a random variable decided by the number of existing threads to be scheduled and we call this time interval a **round**.



**Fig. 2.** Scheduling activity from OS position

Assume at the start of the  $k_{th}$  round, the number of threads in the queue is  $n_k$ , so we have:

$$n_{k+1} = n_k + B(1, p_1) - B(n_k, p_2) \quad (1)$$

$B(n, p)$  is the Binomial distribution with total number of experiments  $n$  and success probability of each experiment  $p$ . This equation shows a simple recurrence relation of thread numbers in a stable OS: some new threads are inserted and some old threads end. Therefore, given the initial number of threads

in the system, the total needed time of thread A could be calculated by adding the numbers of threads in all the rounds together, as Eq. 2 shows:

$$T = \sum_{i=1}^{T_A} n_i \quad (2)$$

## 2.2 Two-Level Scheduling

Scheduling strategies of virtual machines are decided by VMMs. VMMs are divided into two types. Type 1 hypervisors run directly on the hardware while Type 2 hypervisors depend on a host OS. Based on this fact, the thread scheduling patterns need separate discussions [1].

When creating Type 2 virtual machines, vCPUs are created, and the guest OS schedules tasks on the vCPUs. However, threads are executed on physical CPUs in real. For example, in KVM, each vCPU is a thread in the host machine. Host OS can schedule each vCPU independently. Although some advanced virtual machine scheduling strategies have been proposed [11], generally applied algorithms are similar to the thread scheduling in the OS. This pattern can also be seen as a superposition of two thread scheduling activities, the thread scheduling of guest OS, and the thread scheduling of host OS. Note that only CPU tasks are considered here, other devices, like I/O devices and GPU are not included.

As to Type 1 virtual machines, the hypervisor manages several virtual machines and schedules the resource among them. Xen has different VMM schedulers, and credit is the default one. The changing credit value decides the real CPU time that the VM could gain. If we see the virtual machines on Xen as processes/threads and the VMM scheduler as the host machine thread scheduler, any activity of Type 1 VMM and virtual machines could be mapped to that of Type 2 VMM.

Intuitively speaking, this two-level scheduling means longer running time in the virtual machine than in a real machine, because other threads in the virtual machine cost extra time. But it is not always the truth especially when running I/O operations [6]. Anyway, we are focusing on CPU-bound operations which have a small relationship to virtualized I/O devices. Assuming a piece of CPU-bound code requires  $N$  CPU time units and a virtual machine could run  $M$  CPU time units. Threads in the same queue of the host machine cost  $L$  CPU time units in total. Therefore, this piece of code needs about  $N \frac{L}{M}$  CPU time units in the virtual machine. It is in line with some researches or virtual machine detection methods that virtual machines usually cost more time, but it is too ideal. Firstly, there doesn't exist a standard of the length of time. It is only a relative relationship on the same hardware. On an old computer, the running time may be very long so that the computer will be identified as a virtual machine. Secondly, even on the same machine, it could be affected much by inserted high-priority threads.

### 2.3 Probability-Based Two-Level Scheduling

As is explained above, the probability model can reflect the start and end of threads in the operating system, and now we care about combining the probability model with two-level scheduling to analyze the behavior of virtual machines.

If we run a CPU-bound thread A for enough times and get the set of time it costs  $T_1, T_2, \dots, T_n$ , we can say they observe a probability distribution  $D(p_1, p_2, T_A, n_1)$  where  $n_1$  is the initial number of threads in the queue. Note that  $D(p_1, p_2, T_A, n_1)$  is decided by the state of an OS, whether it's a host OS or a guest OS. Therefore, for a guest machine with average thread insert probability  $p'_1$ , average thread end probability  $p'_2$  and initial thread number  $n'_1$ , there exists a distribution  $D(p'_1, p'_2, T'_A, n'_1)$ .

If we run the test code in the virtual machine,  $D(p'_1, p'_2, T'_A, n'_1)$  means the expected vCPU time between the start and end of the test thread. However, vCPU should join the scheduling of threads in the host OS to share CPU resources. In that case, A in the host OS is the vCPU thread, and the real running time is:

$$Z = X \cdot Y, X \sim D(p_1, p_2, T_A, n_1), Y \sim \frac{D(p'_1, p'_2, T'_A, n'_1)}{T'_A} \quad (3)$$

$Y$  is called Expand Factor, which refers to the ratio of used CPU time unit by VM and used CPU time unit by thread  $A'$ . The PDF of  $Z$  could be calculated if  $X$  and  $Y$  could be expressed mathematically. However, they are virtual distributions and we cannot find their real mathematical representations. A Monte Carlo Simulation method is used to simulate the probability density function (PDF) of  $Z$ .

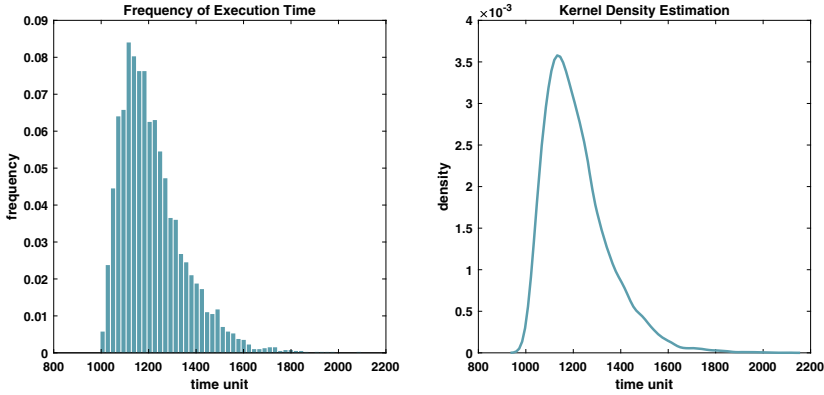
## 3 Numeric Simulations and Real Tests

Based on Eq. 3, we could run a numerical simulation for a host machine and a virtual machine to discover features that could be used to identify them.  $p_1$ ,  $p_2$  and  $n_1$  are specified manually. In our simulation,  $p_1 = 0.002$ ,  $p_2 = 0.01$ ,  $n_1 = 3$ , and  $T_A = 1000$ . We also use kernel density estimation (KDE) to get the PDF curve. The result is shown in Fig. 3.

For virtual machines, as Eq. 3 shows, it is a product distribution of two distributions that belong to the host OS and guest OS respectively. Here we assume that  $p_1 = p'_1$ ,  $p_2 = p'_2$  and  $n_1 = n'_1$  to make it simpler to reveal the key relationships. We use a Monte Carlo Simulation to calculate the product distribution. The result is shown in Fig. 4.

Put the KDE curve together, as Fig. 5 shows, and the two curves show huge differences. The curve of the virtual machine situation has a lower peak at the higher time unit number, which is in line with our preliminary analysis. It is obvious that two-level scheduling will enlarge the range of possible running time. We apply Variance, Kurtosis, and Skewness to express this difference.

Table 1 shows the variance, kurtosis and skewness of simulated data for physical machines and virtual machines. All the running time data has been divided



**Fig. 3.** Frequency and KDE of numerical simulation: physical machine

by  $T_A$  to eliminate the difference of test code. The virtual machine has a higher variance, as its distribution curve is flatter. It also has lower kurtosis and skewness because its tail is short. Based on the fact above, we propose a composite index, which describes how likely is an operating system running under a VMM: Virtual Machine Index.

**Table 1.** Key Statistics of Distribution.

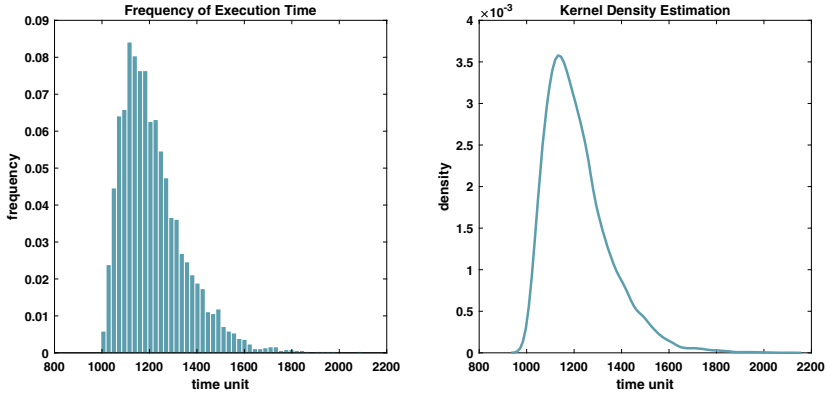
Type	Variance	Kurtosis	Skewness
Physical machine	0.018	5.60	1.30
Virtual Machine	0.052	5.09	1.15

The Virtual Machine Index (VMI) is defined as:

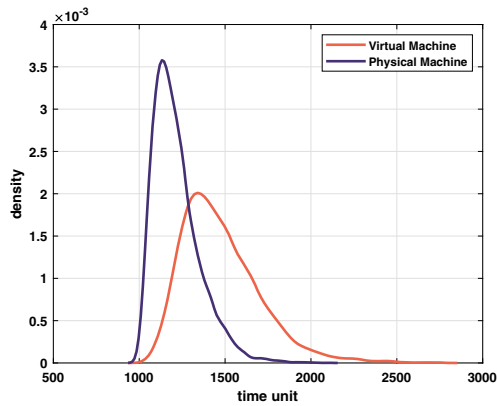
$$VMI(X) = \lg \frac{\text{Var}(x)}{\text{Kurtosis}(X) \cdot \text{Skewness}(X)} \quad (4)$$

Since VMI is usually negative, in this paper we use it's opposite, which is named as Physical Machine Index (PMI):

$$PMI(X) = \lg \frac{\text{Kurtosis}(X) \cdot \text{Skewness}(X)}{\text{Var}(X)} \quad (5)$$



**Fig. 4.** Frequency and KDE of numerical simulation: virtual machine

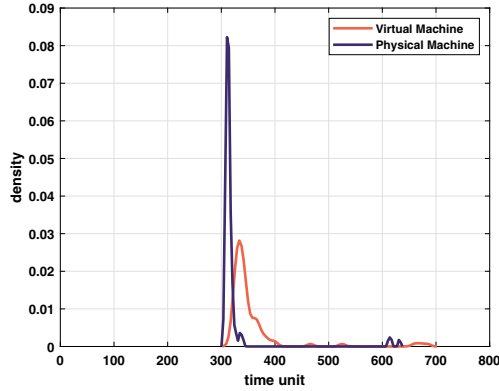


**Fig. 5.** KDE curve of virtual machine and physical machine

To verify the findings and get a reference value, we perform actual testing on physical machines and virtual machines. Physical machines have different operating systems (Windows and Linux), processor types, and running status (Controlled by opening different kinds of other applications). Virtual machines are running under different types of VMMs, mainly VMware WorkStation and KVM. Virtual machines on the cloud are also included.

The test code was written in JavaScript, mainly for two reasons: Firstly, JavaScript codes run in a single thread, it can eliminate the effect of parallel optimization and multi-core to the most extent. Secondly, JavaScript code embedded in a web page can run on the browser, which means it can be applied to detect virtual machines remotely. One of the test result groups on a physical machine and a virtual machine run on it is shown in Fig. 6. The results in the real test show similar patterns as simulation results.

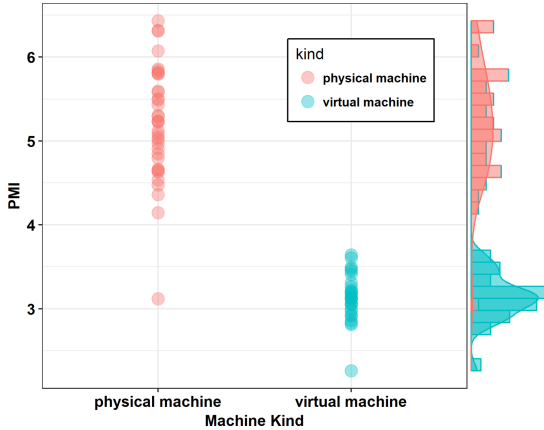




**Fig. 6.** KDE curve of virtual machine and physical machine: real test

The test results of 36 physical machine cases and 28 virtual machine cases are shown in Fig. 7. In this figure, the PMI of virtual machine groups and physical machine groups differ greatly, and a criterion could be easily set up to identify virtual machines and physical machines.

$$\begin{cases} PMI < 4 & \text{Virtual Machine} \\ PMI > 4 & \text{Physical Machine} \end{cases} \quad (6)$$



**Fig. 7.** Results of all the cases

Under this standard, the identification results are shown in Table 2, where the accuracy of identifying physical machines is 0.9722 and for virtual machines it is 1, which means all the virtual machines are successfully identified.

A physical machine was identified as a virtual machine maybe because it was an old computer or it was running some applications that change the environment of the OS fast.

**Table 2.** Identification Results.

Real Type	Identified as a PM	Identified as a VM	Accuracy
Physical Machine	35	1	0.9722
Virtual Machine	0	28	1

## 4 Conclusion and Future Work

The former studies on virtual machine detection strategies mainly focus on some variables in CPU registers or in the guest OS. However, these strategies can only be applied natively, and these variables can be modified by VMM easily. Some research care about remote detection methods, mainly through Network packages. This method demands many preliminary requirements, which make it less practical. Benchmark test is a new direction that is practical in remote scenarios, but it has more randomness, and how to reduce the effect of hardware difference is a problem.

We noticed the time difference between running the same program in physical machines and virtual machines, and explain that by introducing a probability-based thread scheduling model. This model focuses on MLFQ and RR scheduling algorithms, and includes a recurrence formula to calculate the number of threads any round. We also set a clear index system called physical machine index to identify virtual machines in practical scenarios. Results of the numeric simulation and the real tests show that this model can explain the main points of the existing difference, and gain a high accuracy in identifying virtual machines.

Some other factors that are not taken into consideration, however, may also affect the test accuracy. For example, a MITM attack. Any test data was collected on the user's computer and sent back to the server, which is of a high risk of being modified in this process. Moreover, the user could view the test code fragments in the browser and make some disruption. The test efficiency is also a point to consider in practical application since we use some CPU-bound logic, which may occupy the processor for a long period of time and impact the user's experience. On balance, we hope the testing strategy mentioned in this paper will be a prototype for more practical red pill schemes in the future.

## References

1. Alnaim, A.K., Alwakeel, A.M., Fernandez, E.B.: A pattern for an NFV virtual machine environment. In: 2019 IEEE International Systems Conference (SysCon), pp. 1–6. IEEE (2019)

2. Asvija, B., Eswari, R., Bijoy, M.: Security in hardware assisted virtualization for cloud computing-state of the art issues and challenges. *Comput. Netw.* **151**, 68–92 (2019)
3. Brengel, M., Backes, M., Rossow, C.: Detecting hardware-assisted virtualization. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) *DIMVA 2016*. LNCS, vol. 9721, pp. 207–227. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-40667-1\\_11](https://doi.org/10.1007/978-3-319-40667-1_11)
4. Favre, O., Tellenbach, B., Asenz, J.: Honey-copy: a concept and prototype of a generic honeypot system. In: *ICIMP 2017 the Twelfth International Conference on Internet Monitoring and Protection*, Venice, Italy, 25–29 July 2017, pp. 7–11. IARIA (2017)
5. Franklin, J., Luk, M., McCune, J.M., Seshadri, A., Perrig, A., Van Doorn, L.: Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Oper. Syst. Rev.* **42**(3), 83–92 (2008)
6. Ho, G., Boneh, D., Ballard, L., Provos, N.: Tick tick: building browser red pills from timing side channels. In: *8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14)* (2014)
7. Jämthagen, C., Hell, M., Smeets, B.: A technique for remote detection of certain virtual machine monitors. In: Chen, L., Yung, M., Zhu, L. (eds.) *INTRUST 2011*. LNCS, vol. 7222, pp. 129–137. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32298-3\\_9](https://doi.org/10.1007/978-3-642-32298-3_9)
8. Keith, A.: Detection in two easy steps. <http://x86vmm.blogspot.mx/2007/07/bluepill-detection-in-two-easy-steps.html>
9. King, S.T., Chen, P.M.: Subvirt: Implementing malware with virtual machines. In: *2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pp. 14–pp. IEEE (2006)
10. Klein, T.: Scoopyng-the vmware detection tool. <http://www.trapkit.de/research/vmm/scoopyng/index.html>
11. Ma, T., Pang, S., Zhang, W., Hao, S.: Virtual machine based on genetic algorithm used in time and power oriented cloud computing task scheduling. *Intell. Autom. Soft Comput.* **25**(3), 605–613 (2019)
12. Sierra-Arriaga, F., Branco, R., Lee, B.: Security issues and challenges for virtualization technologies. *ACM Comput. Surv. (CSUR)* **53**(2), 1–37 (2020)
13. Wang, Q., Zhu, F., Leng, Y., Ren, Y., Xia, J.: Ensuring readability of electronic records based on virtualization technology in cloud storage. *J. Internet Things* **1**(1), 33 (2019)
14. Zhang, Z., Cheng, Y., Gao, Y., Nepal, S., Liu, D., Zou, Y.: Detecting hardware-assisted virtualization with inconspicuous features. *IEEE Trans. Inf. Forensics Secur.* **16**, 16–27 (2020)