Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH **Open Access**

Volatile Kernel Rootkit hidden process detection in cloud computing

Suresh Kumar S[1*] and Sudalai Muthu T[1]

Abstract

The rootkit industry has advanced significantly in the last decade. Attackers want to leave a backdoor for quick reoccurring exploits rather than launching the traditional one-time worm/virus attacks. Meanwhile, as intrusion detection technologies improve, rootkits have grown in popularity. For the attackers to succeed, stealth becomes critical. The primary function of rootkits is to provide stealth. The modifications a rootkit makes conceal the presence of a rootkit. Determining the presence of mutation rootkits was quite challenging. Attackers can silently alter volatile (processes) and non-volatile (files) with the aid of rootkits without being noticed. We suggested the VKRHPDV (Volatile Kernel Rootkit Hidden Process Detection) framework to find the hidden techniques. This system includes process monitors, process comparison analysts, and contaminated process data gathering. Process monitoring is nothing more than clean process collection in the absence of rootkits, whereas pure process collection has been corrupted by rootkit injection. The process analyzer compares clean and tainted processes, some of which were concealed. VKRHPDV can identify process hiding behaviors in all datasets in the shortest period, according to the findings of an extensive performance analysis carried out on 64 rootkit datasets for each UNIX and Windows kernel in a cloud environment.

Keywords Cloud, Malware, Rootkits, Process, File

Introduction

The purpose of rootkits is to penetrate a computer system or network without authorization and take control of it. While operating, they want to remain undetected by the user and other security software. Rootkits [1] frequently target a system's "root" or administrative level, which gives them a wide range of rights and power. They can alter or replace essential system files, listen to system calls, and change how the operating system behaves. This enables them to avoid detection by standard security measures and antivirus software. It needs detection for target systems. There are different types of rootkit detection, like Signature-based detection [2], Heuristic-based detection, Memory analysis, Rootkit [3] scanning tools, System integrity checking, and Network monitoring. These types of rootkit detection detect only a limited number of rootkits, but it does not update the new mutation rootkit. This work includes Process comparison analysts, process monitors, and data collection on tainted processes. In the absence of rootkits, process monitoring is nothing more than clean process collection, but clean process collection has been tainted by rootkit injection. Process analyzers compare pure and tainted processes, some of which were hidden. According to the results of a complete performance investigation performed on 64 rootkit datasets for each UNIX and Windows kernel in a cloud environment, VKRHPDV can detect process-hiding behaviors in all datasets in a short period.

Related work

Rootkit [4]: This is associated with "criminals waiting to rob you while you are away in the upstairs room." Many specialists advise entirely wiping your hard drive and starting over because it is the hardest malware to

*Correspondence:
Suresh Kumar S
sureshkumarphd2018@gmail.com
[1] Department of Computer Science and Engineering, Hindustan Institute of Technology and Science, Chennai, India

detect and remove. Allowing other details computer viruses into your computer so that it can collect identification information from it without your understanding is recommended. Individuals trying to access the PC for financial gain create such malware. Rootkit [5, 6] identification in distributed computing management anticipates a crucial job. This inquiry is related to a few previous studies on distributed computing [7], structure, and acknowledgment systems in general.

These techniques are described in the most recent rootkit disclosures [8], which cover signature, conduct, cross-view, respectability, and equipment. The way to distinguish rootkits that is most frequently employed is the mark-based location method [9]. When antivirus software detects malware, it recognizes a "sign" specific to the pathogen's byte case and saves those antecedents in a DBMS. Location-based coding compares examples from the framework to database markers [10].

The system's two distinct views of the system are presented differently to detect contrasts in cross-view rootkit discovery. This process obtains an unusual state perspective of the framework from a vulnerable malware control area. Whatever the rootkit conceals will not be detected by the strange state. Because it presents a genuine perspective on the template rather than assuming that the framework reports are essential, this methodology recognizes the external view as the framework's verified perspective. The most excellent strategy for dealing with stealth malware is clean booting, which prevents the infection from hiding. Given that the OS and the malware are not functioning, it cannot protect records or techniques from an external source. For Windows 2000 systems, the fresh boot method uses the setup wizard to launch the framework into Disk OS mode and examine the archive structure. The framework must be established in Disk OS mode using external tools when Windows 8.1 is first booted.

The current view of memory or the file system is compared to a trusted standard obtained when the system was flawless. Any improvements could reveal a rootkit's capabilities on the system. Because external equipment is unlikely to compete for resources with the backdoor, equipment-based rootkit discovery was born as software identification does. External hardware, like external programming, reduces framework activity. The benefit of equipment rootkit discovery is that the rootkits cannot alter the equipment because it uses an external OS. This research shows a bridge clean boot-based discovery technique that detects some of the rootkit's persistent systems. The proposed system aims to see rootkit-hidden records.

Methods and materials

Rootkit applications available on the Amazon web services stage typically attempt to conceal some information while they are running. In any case, execution has only been seen rarely in current software. The system's boot-up time is reduced as a result of this. From this point forward, it is crucial to separate the protest hiding behavior, which is fundamentally classified as a rootkit behavior and significantly impacts the execution of Amazon web services cloud setup, especially in over-the-top load assignments. It is crucial to identify the covered documents to improve the heap characteristics and net execution of the current cloud configuration. The proposed rationale defines a transparent system for identifying non-unstable rootkits.

In light of the study, it is crucial to create and alternately examine the clean process list and contaminated process list to separate the covered documents automatically. It is sometimes challenging to obtain the appropriate segment document list since the presence of rootkits rapidly alters the framework's direction. Even with minimal structural information, guessing the optimum parcel record list is still possible. The gauge is consistently trusted to reflect reality and is never given access to rootkits. Accordingly, rootkits known as the tainted process list occasionally trade-off and control what the client sees.

P1 stands for the clean process list, and U1 for the contaminated process list due to the catalog. The clean process list P reflects reality and is unaffected by rootkit manipulation. The corrupted process list U indicates the user's perspective, which is open to rootkit attacks. U R1 (Time1) refers to the set of corrupted process collections of type R1 that are shown to the user at time Time1, and PR1 (Time1) represents the set of pure process lists of type R1 at time Time1.

Assume Entry1 is an entity (or process)

$$\mathrm{TimeT1, Entry1st., R1 = R1(Entry1)} \tag{1}$$

When object Entry1 is concealed, then

$$\mathrm{PR1(Time1)} \bigwedge \notin \mathrm{UR1(Time1)} \tag{2}$$

else If

$$\mathrm{P(pure\ process)R1\ (T1)\backslash U(corrupted\ process)R1\ (T1)} = \oint \tag{3}$$

RE stands for every form of system entity that could exist, including files, processes, ports, etc.

$$\forall \mathrm{R1} \in \mathrm{REntry\{T1\ for\ All\ Time\}} \tag{4}$$

About each kind of R1's efficient system, The results P and U can be obtained as follows to enable the detection

$$P = \cup R0 \in RPR1(T1)\{TimeT1\} \quad (5)$$

$$U = \cup R0 \in RUR1(T1)\{TimeT1\} \quad (6)$$

The problem is solved by employing cross-view-based identification. In this approach, separate lists are created and analyzed for clean and contaminated processes as part of cross-view-based detection.

Systems design

The proposed system VKRHPDV design, which is depicted in Fig. 1, is used to detect the existence of hidde processes.

User mode is also known as user space or Ring 3. It is a restricted execution mode where user applications and processes are run. Examples of user mode activities include the execution of applications, files being opened, network resources being accessed, and interaction with the user through the graphical user interface. Kernel mode is also known as privileged mode, supervisor mode, or Ring 0. Unrestricted access to hardware is granted to the kernel, allowing it to execute any instruction and access any memory location. Critical system operations, such as memory management, interrupt handling, task scheduling, and hardware device control, are performed in kernel mode.

The three crucial components of VKRHPDV are Process Monitor, Compare Process Analyzer, and Process Enumerator. System calls are received and interpreted by the OS libraries to perform process activities, and these system calls are utilized to maintain an excellent and up-to-date sequence collection. The process collection is then revived, stored in the index, and provided to the detection method as needed. The refreshed list can be retrieved by the userland program running on the Amazon Web Services Windows 2022 new host.

The initial process without the injected rootkit, which falls under the process monitor, forms the clean process collection. This process is then repeated with the injection of the rootkit, resulting in a group of tainted methods gathered in the process enumerator. In the comparing process analyzer, the contaminated process collection and clean process collection are compared, where some processes may be hidden. The detection of hidden processes occurs in the AWS Ubuntu Server 22.04 LTS (HVM) operating system with SSD volume type.

Designing the clean process list

Process Monitor is the first module in the Volatile Kernel Rootkit Hidden Process Detection View. In AWS, the Windows operating system is launched. Clean process collection is one of the operations performed in Process Monitor.

1. Initialize the clean process list when the Aamzon web service-Windows 2022 server instance has been initiated.
2. add a new procedure to the collection for every new process.
3. For each new process, update the clean process collection.

From the procedure, the accompanying activities were inferred:

No attack occurs while launching new AWS Windows instances because the rootkit is not injected. The process is collected and referred to as clean process collection. This process falls under the purview of the

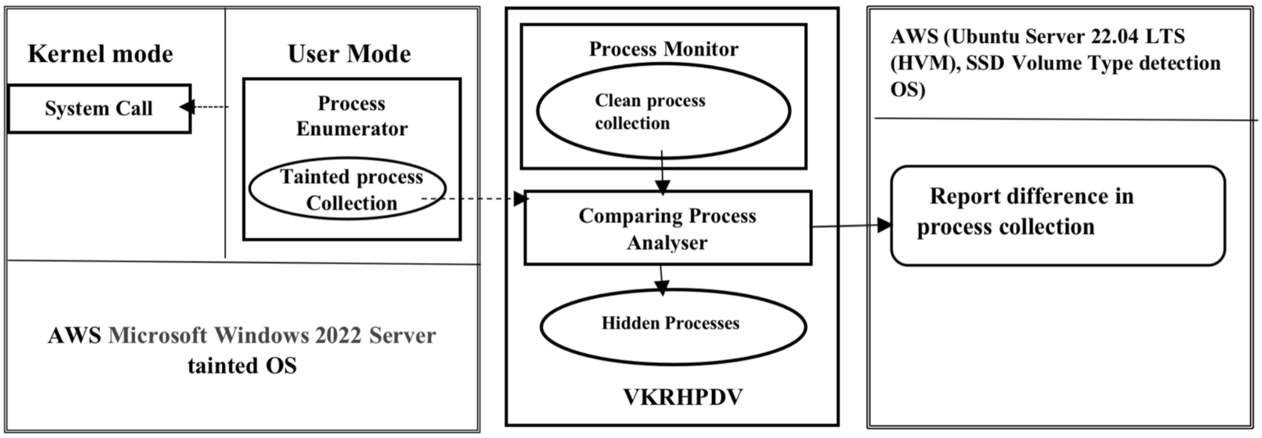
Kernel mode
System Call
User Mode
Process Enumerator
Tainted process Collection
AWS Microsoft Windows 2022 Server tainted OS
Process Monitor
Clean process collection
Comparing Process Analyser
Hidden Processes
VKRHPDV
AWS (Ubuntu Server 22.04 LTS (HVM), SSD Volume Type detection OS)
Report difference in process collection

Fig. 1 The Architecture of Volatile Kernel Rootkit Hidden Process Detection View (VKRHPDV)

Process Monitor. This method generates the pure process list T1, which contains dynamic processes using low-processing system call information. It is based on the knowledge that the operating system uses specific system calls (in this case, AWS-Windows 2022 Server) to carry out both new and legacy functions.

On Windows, for instance, several Win32 APIs, such as Build System and WinExec, are frequently used in the client space to create new methods. Thus, these abnormal condition APIs are called a collection of low-level frameworks. It is also possible to explicitly use system calls to construct processes. The process manages creation by responding to those system calls in any scenario. As a result, by collecting these associated system calls, a process can screen its beginning and end. Rootkits must never change the trusted view. Given this premiere, the growth of trusted opinion should be as minimal as possible given the normal situation.

The information used for reliable perspective is limited enough that rootkits cannot significantly impact or overtake it. From one perspective, it is challenging to create a procedure with only an operating system because these system calls are attacked and, in general, unrecognized, regardless of whether it is possible to develop a technique without declaring any abnormal state Windows APIs. This dramatically expands on current rootkit design ideas as a rootkit must execute particular Windows operating system components to evade these system calls. Of course, terminating a process is less demanding than creating one, and it may be possible to complete an operation without a regular cycle. False positives could happen if those system calls for process termination are followed through with. In any case, rootkits will benefit from using such frameworks because they cannot conceal any system.

Tainted process collection

The Tainted Process Collection is the second module in the Volatile Kernel Rootkit Hidden Process Detection View. In AWS, the Windows operating system is launched. Tainted process collection is one of the operations performed in Process Enumerator. To detect all rootkit-hidden records, create an infected process list and differentiate it from the clean process list. The corrupted process list contains processes that are unmistakable to system clients; therefore, it is critical that every covered report be avoided. On a fundamental level, there are three places to get the tainted process list system:

- The tainted operating system itself
- The corrupted operating system's kernel space
- The contaminated operating system's client space

The primary option is navigating the corrupted operating system's dynamic simulation list to determine the operating procedures. However, this list also includes operations that client-level rootkits and snare [10, 11] based kernel-level rootkits cover-up. Because these procedures are typically invisible to clients, there may be Zero negatives when comparing the polluted OS process list to the clean process list.

The catalog record rundown of the compromised operating system would include a few documents hidden by client data rootkits [12] and specific kernel-level rootkits, leading to Zero negatives for the second option. As a result, it is decided to create a process list of the corrupted operating system in the user space of the corrupted operating system. In contrast to the clean process collection, the process collection of a contaminated operating system does not expect to be unaffected by rootkits, and its development should be as abnormal as possible, with the end goal of completely exposing all rootkit controls.

Comparing the process analyser

Comparing process analyzer is used to compare clean process collection and tainted process collection. While comparing the process, if there is a difference, the process is hidden or not hidden. When instructed by the procedure, the compare process analyzer [13] adapts the contaminated process collection from the correspondence support in the corrupted operating system. It is decoded, and its validity is checked before the corrupted process list is compared to the unmistakable segment record shown. If the confirmation stops working, a notification message shows a potential attack on the contaminated operating system.

The breakdown report analyzer looks at the two document lists once the clean process collection and the tainted process collection are ready. The record analyzer searches the folder collection of corrupted Operating Systems for each document list in the spotless segment record list. If the report is also found in the corrupted OS's process collection, it continues with the accompanying document in the clean process collection until all records are dealt with. If any description from the pure process collection is missing from the catalog of tainted OS, it is considered a covered archive. The record analyzer then refers to the covered report and provides more information about it. When hidden processes are found, rootkits may have infiltrated that process.

The following algorithm-1compares the clean and tainted process lists to identify hidden processes:

```
1: function cmpCPL-PLT  string P1, K1;
       P1 = {Clean process collection}
       K1 = {Tainted process collection}
2:    while T do  f1 (create process) or System.exit();
       P1[j] = P1[j] + p1;
       U1 = U1[j] + p1;
3:        while T do
4:            if U1[j] != P1[j] then
       p1 not in U1
5:            else NULL;
6:    =0
```

Algorithm 1. cmpCPL-PLT

Algorithem :1

The rootkit was injected into the machine with the clean operating system. The term clean process collection was given to the process collected from this operating system, which is kept in the process monitor. The same machine was corrupted and became known as having a tainted operating system after the rootkit was introduced. The process was collected and updated by the impaired operating system. After an update, when the clean process collection and tainted process collection are compared, if there is a difference, the process should be hidden; otherwise, it should not be hidden.

The issue declaration states that the two perspectives under consideration, T1 and U1, must be created simultaneously. In any case, there could be a brief delay between stages three and four; as a result, this methodology examines two nonconcurrent sees developed on separate occasions. This may be fine on an inactive framework. Still, it may impact the identification result of a functioning system where processes are created regularly. Check to see if this will cause any zero positives or zero negatives.

The clean process collection at time1 is distinguished from the tainted group at time2. This recognition calculation deduces that time2 comes after (time1 time2). When a system starts, it is indicated by an A timep1; when it ends, it is stated by a T'p1. Within the day and age, the following things could happen between time1 and time2.

- If ∄p1 time1 < timep1 < time2 or time1 < time'p1 < time2 is made or ended somewhere in the scope of time1 and time2, then the time qualification can essentially be ignored because the system remains the same, and the outcome of the discovery is unaffected.
- If ∃p1 the procedure P1, which starts before time1 and ends somewhere between t1 and t2, happens with the true objective that time1 < timep1 < time2 or time1 < time'p1 < time2, Recognize that if p1 is actually not one of the covered documents, p1 ∈ T1 anyway p1 ∉ U may produce a false positive.
- If ∃p1, which is made and ended at between time1 and time2, has the main objective that time1 < timep1 < time2 < time'p1 < time2. Recognize that p1 ∉ T1 regardless of p1 ∉U, so the outcome is unaffected. In any case, p1 may have a zero negative if the record is covered.
- The procedure ∃p1, which is performed there between time1 and time2, has the ultimate goal of time1 < timep1 < time2 < time' p1 < time2. The calculation shows that the result is unaffected by p1 ∉ T1 regardless of p1 ∉U. However, a false negative is possible if p1 is a covered document.

Based on the above examination, the identification may result in zero positives or zero negatives. When the following scenario happens, the trusted in view (clean fragment file collection) would be renewed and would appear differently in comparison to another most tainted process list. This refreshing and contrasting would continue until the system became consistent, as it was in the initial condition. The likelihood of false positive focuses in the second scenario is low because the time interval between time1 and time2 is close to zero, and there is no reason to think that false negatives will be a problem. For starters, potential results for the last two conditions are limited because the break between time1 and time2 is practically nothing. As shown in the third condition, a brief strategy does not need to be concealed: the client will most likely not see whether it is covered. According to the definition, this is not a rootkit procedure in this case. As a result, zero negatives, in this case, will be insignificant before long.

Implementation

A root account must be created in the Amazon Web Services Management Console to activate Amazon Web Services. Then, choose "Launch Instance" on the Amazon Elastic Compute Cloud Dashboard to build and customize your virtual computer. You can configure your instance features in this wizard. Initializing your instance could take some time. SSH, PUTTY, and RDP are all options for connecting AWS instances. By choosing the Elastic Compute Cloud instance, clicking "Actions," then "Instance State," and lastly, "Terminate."

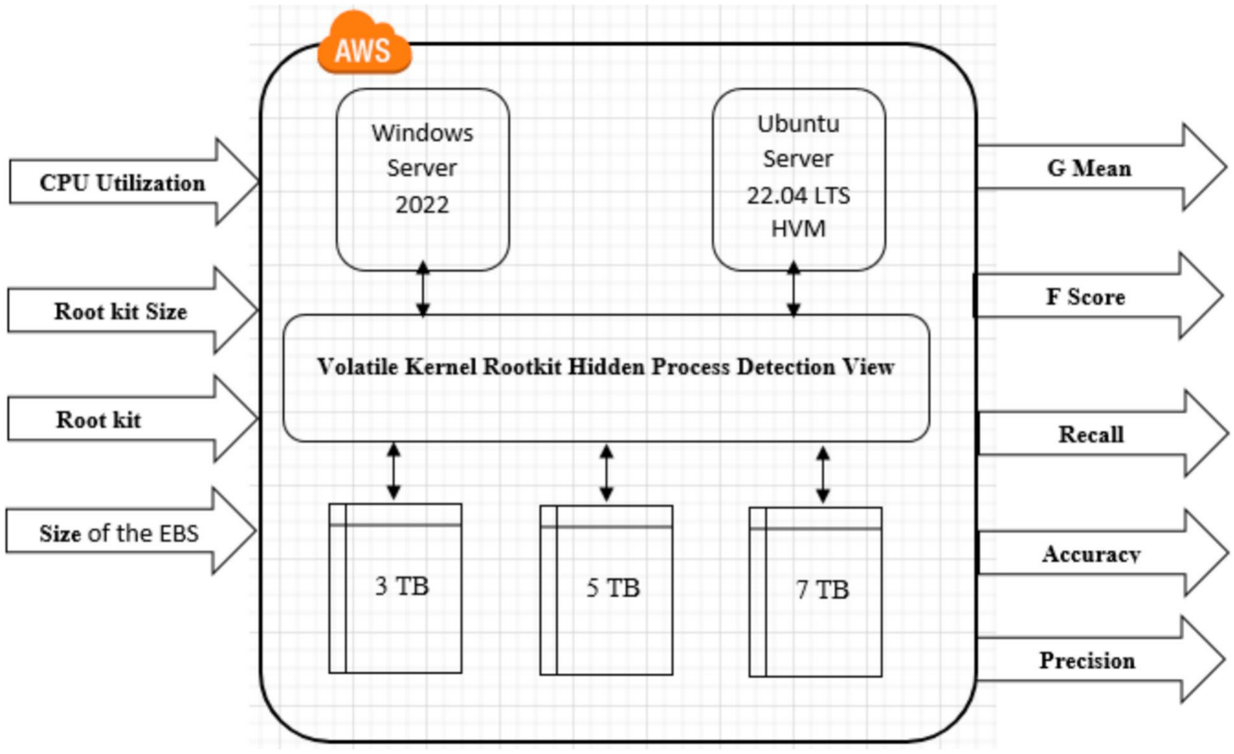

Fig. 2 Deployment of Volatile Kernel Rootkit Hidden Process Detection View (VKRHPDV)

Figure 2 demonstrates the implementation process. In this trial, rootkit tests for VKRHPDV are being run in a cloud environment provided by Amazon Web Services. To begin, run the VKRHPDV on an Amazon-provided (first event tainted OS) Microsoft Windows Server 2022 Full Locale English AMI with an Intel Xeon Family 3.70 GHz CPU and 16GB RAM. With administration pack First, the framework is a standard foundation. Second, use another example AWS Ubuntu Server 22.04 LTS (HVM) with an Intel Xeon Family 3.70 GHz CPU and 16GB RAM to run the VKRHPDV.

VKRHPDV contains three essential components: Process Monitor, Comparing Process Analyzer, and the process collection of spoiled OS (process Enumerator). The OS libraries use these template calls to maintain a perfect, fresh segment collection by obtaining and decoding them when performing process actions. When the approach seems novel, the process rundown will be renewed and saved in the repository. The created list is evaluated using the corrupted OS's archive analyzer. The procedure screen maintains the dynamic processes and the apparent segment record list. How the process screen is configured depends on the operating system to handle system calls related to operations. It screens process workouts comprising the procedure creation and the end to keep the archive visible.

The rootkit was first presented in Windows, and its popularity will keep rising when it is made available on the internet. While detached, the rootkit's behavior is minimal. After the window is closed, the limit (volume) envelope is mounted to Ubuntu. The lead will be distinguished when setting up into another OS. Based on these qualifications, distinct proof of covered documents is created, a process known as VKRHPDV discovery. A clean boot is an identification technique that keeps malicious processes from hiding. Because the rootkit operates while the operating system is in use, it protects archives or operations from being accessed from external sources. For the Windows 2000 system, the ideal boot procedure, for instance, tests the file structure while booting the system into disc operating system mode from the boot menu. This assignment turns off a few rootkit features, which suggests that the certified positive rate should rise from 68.421% while the false positive rate stays at 0%.

Proposed algorithm

The three essential components of VKRHPDV are Process Monitor, Comparing Process Analyzer, and the tainted process list (process Enumerator). System calls are acquired and interpreted by the OS libraries to execute procedures, and these requirements and support are utilized to maintain a pristine collection of processes. Once the method is completed, the process list is updated and stored in the index. Once the technique is operational, the process list will be updated and stored in the index. This index is utilized to construct the collection of contaminated OS processes, aiming to maintain maximum irregularity within the given conditions to represent the view observed by consumers accurately. The resulting group is then analyzed using the tainted OS report analyzer, which establishes process relationships. Maintaining these dynamic procedures and the distinct process collection is entrusted to the process display. When configured, the operating system handles system calls related to methods. The process display monitors various process activities, such as initiating and completing procedures, to maintain a visible record. As a result, regular feedback is provided. The implementation of this model, as depicted in Fig. 2 VKRHPDV, elucidates the process. The model is used for performance analysis and seamlessly integrated into the Amazon Web Services cloud environment. Accuracy, P precision, recall, F score, and G mean are accuracy parameters. In terms of efficiency, this model surpasses its predecessors.

Begin the EC2 Windows example setup. Create a procedure in VKRHPDV that looks like a clean segment document and a tainted process list. Display the rootkit in the tainted OS's index rundown. One by one, annex the 3TB/5TB/ 7TB volumes to the Windows model. The Windows envelope is moved to 3TB/ 5TB /7TB volume one by one, and then the Windows precedent is closed, limiting the 3TB/5TB/7TB volume from Windows cases one by one. Start the EC2 Ubuntu instance. Join the three tb/5tb/7tb volume to the Ubuntu model individually. Install Ubuntu on the three tb/5tb/7tb volume. The clean and tainted processes are examined, and what makes a difference is discovered. If there is a distinction, it is a direct result of the covered documents.

```
1: function VKRHPDVforWindows($EBS$ ebs, $CPUuti$ u1, $RootkitSize$ rs1, $Rootkits$ rs)  windowsInt; int z = 74;
2:    while $N \neq z$ do
        T1 = {Clean process collection}
        U1 = {Tainted process collection}
3:        if ($EBSis3TB$) —— ($EBSis5TB$) —— ($EBSis7TB$) then
        windowsInt = windowsInt + $EBS(3TBor5TBor7TB)size$;
        windowsInt = U1;
        cmpCPL-PLT
4:            if (($EBSsize3TB$) —— ($EBSsize5TB$) —— ($EBSsize7TB$))
   then
        windowsInt = windowsInt - EBS;
5:        if $T1$ not match $UI$ then  // Locate the concealed Process;
6:        else NULL;
7:    =0
```

Algorithm 2 a. a Function VKRHPD for Windows

Algorithem-2 a

An AWS account was created, and Windows instances were launched into the AWS account. For testing purposes, seventy-four rootkits were used. The clean process collection and tainted process collection were collected. One by one, 3TB/5TB/7TB were attached to the Windows instances. First, 3TB was connected, and the Volatile Kernel Rootkit Hidden Process Detection View was executed. The clean process collection and tainted process collection were compared. The process was hidden if there was any difference; otherwise, it was not hidden. The time taken to detect this was known as the Rootkit detection time, and then 3TB was detached. Next, 5TB was attached, and the detection time was calculated before detaching. Finally, 7TB was connected, the detection time was calculated, and the detection time varied based on the storage capacity (3TB/5TB/7TB). If the storage was minimum, the detection time was also minimum; if it was high, the detection time was also increased. The calculated detection time results are shown in Table 3.

```
1: function VKRHPDVforUbuntu($EBS$ ebs, $CPUuti$ u1, $RootkitSize$ rs1, $Rootkits$ rs)  ubuntuInt;
      int z = 74;
2:    while $N \neq z$ do
        T1 = {Clean process collection}
        U1 = {Tainted process collection}
3:        if ($EBSis3TB$) —— ($EBSis5TB$) —— ($EBSis7TB$) then
        ubuntuInt = ubuntuInt + $EBS(3TBor5TBor7TB)size$;
        ubuntuInt = U1;
        cmpCPL-PLT
4:            if (($EBSsize3TB$) —— ($EBSsize5TB$) —— ($EBSsize7TB$))
   then
        ubuntuInt = ubuntusInt - $EBS$;
5:        if $T1$ not match $UI$ then  // Locate the concealed Process;
6:        else NULL;
7:    =0
```

Algorithm 2 b. b VKRHPD for Ubuntu

Algorithem-2 b

An AWS account was created, and Ubuntu instances were launched into the AWS account. For testing purposes, seventy-four rootkits were used. The clean process collection and tainted process collection were collected. One by one, 3TB/5TB/7TB were attached to the Ubuntu instances. First, 3TB was connected, and the Volatile Kernel Rootkit Hidden Process Detection View was executed. The clean process collection and tainted process collection were compared. The process was hidden if there was any difference; otherwise, it was not hidden. The time taken to detect this was known as the Rootkit detection time, and then 3TB was detached. Next,5TB was attached, and the detection time was calculated before detaching. Finally, 7TB was connected, the detection time was calculated, and the detection time varied based on the storage capacity (3TB/5TB/7TB). If the storage was minimum, the detection time was also minimum;

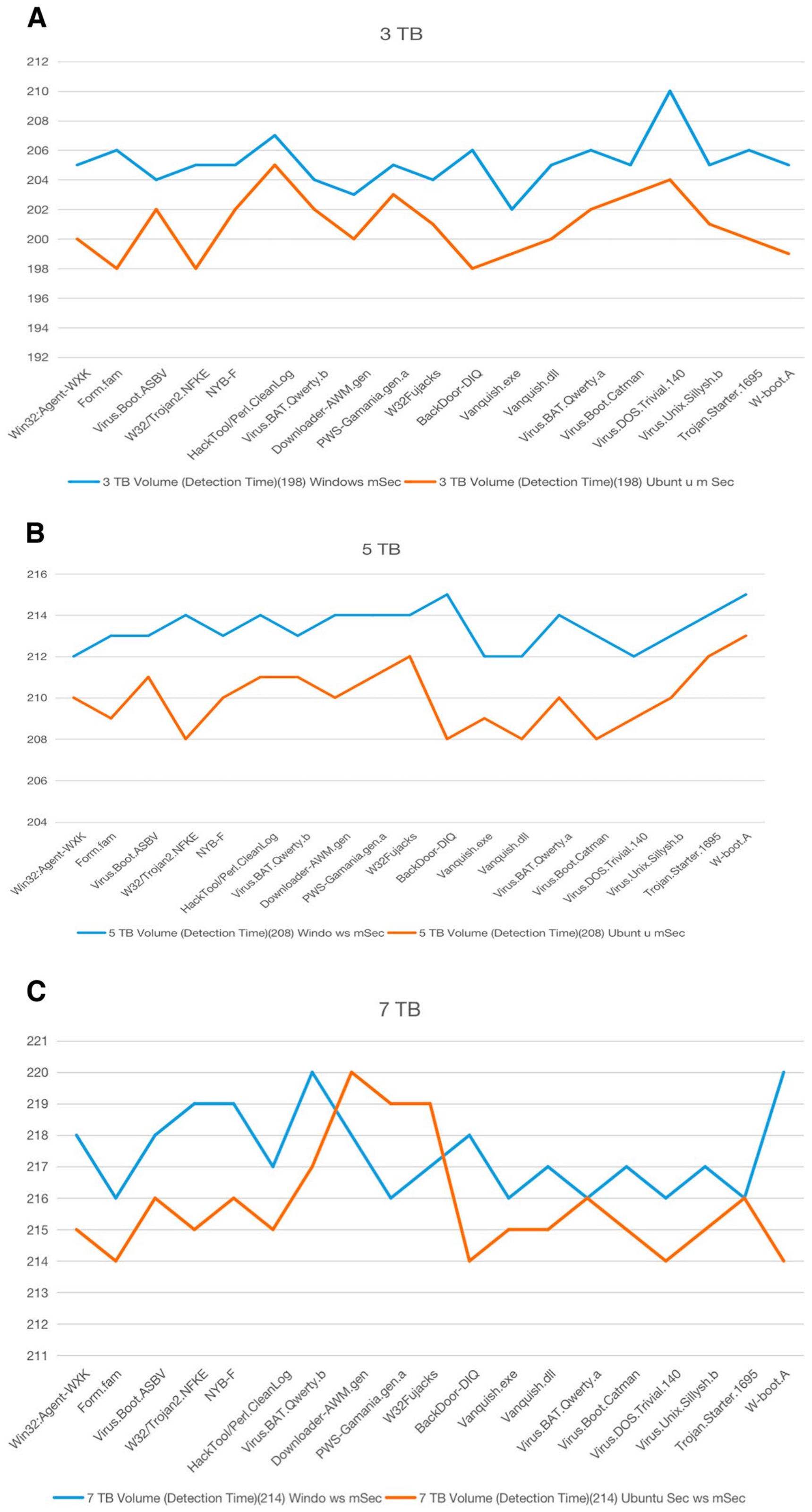

Fig. 3 Rootkit detection time for Windows and Ubuntu. **a** Detection Time of 3 TB. **b** Detection Time of 5 TB. **c** Detection Time of 7 TB

Table 1 Parameter information

S.No	Rootkit s name	Rootkits size	Hidden Operation	process
1	Win32:Agent-WXK	6.84 kb	GoogleCrashHandler64.exe	26
2	Form.fam	676 kb	LiteAgent.exe	26
3	Virus.Boot.ASBV	29 kb	wininit.exe	27
4	W32/Trojan2.NFKE	82.5 kb	inetinfo.exe	29
5	NYB	791 kb	chrome.exe	37
6	HackTool/Perl.CleanLog	22.86kb	lsass.exe	29
7	Rootkit.Win32.Agent.enz	79 b	Svchost.exe	22
8	Rootkit.Win32.Agent.fht	8 kb	dllhost.exe	29
9	Rootkit.Win32.Agent.xp	122 kb	rdpclip.exe	29
10	Rootkit.Win32.Qandr.ac	1.27 mb	Taskhost.exe	25
11	Rootkit.Win32.Qandr.ak	88 kb	LogonUI.exe	24
12	bakuryu	46 kb	taskhostex.exe	23
13	shell.jpg	27 kb	WUDFHost.exe	29
14	f6e671d8630df5d8045ff4243da94f74	1 kb	MpUXSrv.exe	26
15	afe8df184dccf6db48cf27916d0d0da6	5 kb	Svchost.exe	29
16	6eddd98e0463acaa3aa0eeab26b1d3c9	142 b	Taskhost.exe	24
17	80da4801d2b70d7044e9d660a05c676	109 b	Dllhost.exe	25
18	4356aded80ee30d1f85321ecc28694b3	519 b	Scchost.exe	25
19	e08de794d84c472b1fd9a862bd729556	519 b	Vmtoolsd.exe	27

Table 2 Ambiguity matrix of VKRHPDV examination

		Predicted	
		-	+
Actual	-	6	0
	+	55	13

if it was high, the detection time was also increased. The calculated detection time results are shown in Table 3.

In the Ubuntu instance, the detection time was significantly less when compared with the Windows instance. The compared detection times of Ubuntu and Windows instances are shown in Fig. 3 (a, b, c).

Performance analysis

VKRHPDV was tested on a model set consisting of nineteen rootkit tests, most freely available on the internet [14], to determine the feasibility of an outcross view and clean boot-based rootkit identification system. Table 1 [15] summaries the characteristics of the data parameter rootkit tests based on if they are a portion of a user space string, as well as which systems (catching and DKOM) they change during the cover process.

The experiment is run on the perfect boot system in Win 2022 against 74 rootkits, with each test failing to delineate the cloud. The rootkit is presented for each test and run against a fresh boot system. This system produces no false positives (zero false positives, 100% certified negatives). We anticipate that the ideal Windows instance system will remain pristine.

In this research, 74 rootkits were tested for rootkit detection. Six rootkits are in the validation set, while the other 68 are invalid. True positive 13 rootkits of the 68 rootkits(55 are added with 13) are distinguished, while the additional 55 are not. True positive 13 rootkits are added with false negative six rootkits, and together, 19 rootkits were tested and listed above in Table 2. This result is 23.63% obvious positive and 76.36% false negative. An erroneous rootkit installation setup caused the false negative. Assailants learn how to install the rootkit perfectly, and the false negative rate decreases. The limitation of rootkit-covered records is illustrated in Table 1. (legal and ill-legal rootkits).

A few other rootkits conceal transparent files, while others hide files and folders. Before testing, virus total examines each rootkit [16] and assigns a specific name from K7, Symantec, McAfee, and others. Table 1 shows that the first six rootkits are in the confirmation set, while the remaining rootkits from 7 to 19 are not. Agent WXK, Form.fam, Virus.Boot.ASBV, W32/Trojan2.NFKE,NYB, and HackTool/Perl are six Win32 rootkits. CleanLog encloses all of the files, and the rootkits are validly set.

In the analyses, VKRHPDV has viably recognized covered records for all the models. The acknowledgment results are shortened in Table 3. VKRHPDV can distinguish hid documents if a rootkit test can viably cover

Table 3 True positive rootkit detection time

Rootkit Name	3 TB Volume (Detection Time)(198)		5 TB Volume (Detection Time)(208)		7 TB Volume (Detection Time) (214)	
	Windows mSec	Ubuntu m Sec	Windows mSec	Ubuntu mSec	Windows mSec	Ubuntu Sec
Win32:Agent-WXK	205	200	212	210	218	215
Form.fam	206	198	213	209	216	214
Virus.Boot.ASBV	204	202	213	211	218	216
W32/Trojan2.NFKE	205	198	214	208	219	215
NYB-F	205	202	213	210	219	216
HackTool/Perl.CleanLog	207	205	214	211	217	215
Virus.BAT.Qwerty.b	204	202	213	211	220	217
Downloader-AWM.gen	203	200	214	210	218	220
PWS-Gamania.gen.a	205	203	214	211	216	219
W32Fujacks	204	201	214	212	217	219
BackDoor-DIQ	206	198	215	208	218	214
Vanquish.exe	202	199	212	209	216	215
Vanquish.dll	205	200	212	208	217	215
Virus.BAT.Qwerty.a	206	202	214	210	216	216
Virus.Boot.Catman	205	203	213	208	217	215
Virus.DOS.Trivial.140	210	204	212	209	216	214
Virus.Unix.Sillysh.b	205	201	213	210	217	215
Trojan.Starter.1695	206	200	214	212	216	216
W-boot.A	205	199	215	213	220	214

them. This examination assumes that the limit volume remains reliable; thus, the invention time will be shorter. The storage is 3 TB, 5 TB, and 7 TB. In this case, the 3 TB volume exceeded the other two volumes regarding area time.

In Fig. 3 (a, b, c) illustrate how as volume size grows, so does time consumption. This system has high accuracy and a short rootkit [17, 18] recognition time compared to other rootkit methodologies, such as the point tactic and the device location procedure. The revelation time will be reduced if this examination's limit volume remains constant. The storage capacities are 3 TB, 5 TB, and 7 TB. The discovery system recognizes rootkit [19] shapes by creating and differentiating two unmistakable points of view. The advancement of the tainted process list is done in the customer measurement of the infected OS [20]. On the other hand, as in a contaminated Operating System, the advancement of the clean process list and the relationship of the two points of view are carried out (before the present rootkit). On the tainted OS [21], there are a manageable number of processes running. Our view examination calculation provides a small workload with the infected OS [22].

VKRHPDV dynamically maintains an archive list to generate the clean process list by retrieving and interpreting specific framework-specific system calls. The additional work may add to the runtime overhead.

Specification of detection

This template has five performance metrics in this measure. The first metric is the detection accuracy of the rootkit, which is written as

$$\text{Accuracy} = \frac{\text{Rootkit Tested} - \text{True Positives} - \text{False Negatives}}{\text{Rootkit Tested}} \tag{7}$$

$$\text{Precision} = \frac{\textit{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{8}$$

$$\text{Recall} = \frac{\textit{True Positives}}{\textit{True Positives} + \textit{False Negatives}} \tag{9}$$

$$\text{Fscore} = 2\text{x}\left(\frac{\textit{Precision Recall}}{\textit{Precision} + \textit{Recall}}\right) \tag{10}$$

$$\text{Gmean} = \sqrt{\text{Precision} * \text{Recall}} \tag{11}$$

Precision measures how many positive classifications are correct, i.e., the probability that a detected anomaly variance has been correctly classified. In contrast, accuracy is the degree to which the detector correctly classifies any newly tested data sample. The recall metric assesses the detector's capability to recognize variances accurately or the probability that an abnormal model

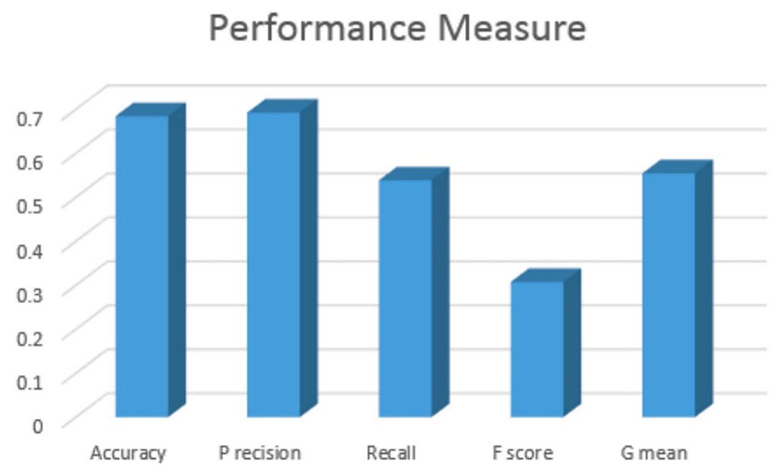

Fig. 4 Performance Measure

will be appropriately identified. The final two metrics, the harmonic mean (F score) and geometric mean (G mean) give a more comprehensive picture of a detector's performance by partially accounting for all of the results (Fig. 4).

Conclusions and future work

This paper uses an original thought of the Volatile Kernel rootkit hidden process detection View (VKRHPDV) of a procedure to rootkit [23] recognition. This procedure's accuracy is excellent, and the rootkit [24, 25] ecognition time is fast compared to other rootkit finding techniques like mark-based systems and equipment-based recognition techniques. The discovery time will be reduced if the limit volume in this examination remains constant. The storage is 3TB, 5TB, and 7 TB. In this case, the acknowledgment time in the 3 TB volume is shorter than in the other two volumes. Despite maintaining a 0% false positive rate, the cross-view fresh boot procedure detects 68.421% of rootkit attempts on Windows Server 2022. The not recommended rootkit foundation does not cover records and envelopes. Furthermore, rootkits that distinguish Amazon Web service cloud condition have a high false negative rate. Because the attacker will present the rootkit adequately on the cloud case, the false negative rate should decrease in this action. Future work will be based on distinguishing open ports and memory levels in cloud conditions.

Abbreviations

VKRHPDV	Volatile Kernel Rootkit Hidden Process Detection View
OS	Operating System
Win	Windows 8.1
TB	Terabyte
DKOM	Direct kernel object manipulation
Ubu	Ubuntu
SSH	Secure shell
RDP	Remote Desktop Protocol

Acknowledgements
Our Research supervisor Dr SudalaiMuthu T for his encouragement and guiding us throughout the research work and to our parents and wife.

Authors' contributions
S. Suresh Kumar is a Research scholar at the Hindustan Institute of Technology in Chennai, India. He obtained his Bachelor of Engineering in Computer Science and Engineering from the University of Madras, Chennai and his Master of Engineering in Computer Science and Engineering from Anna University, Chennai. His current research is focused on cloud computing security. He published many manuscript.

Funding
Not applicable.

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 26 November 2022 Accepted: 12 November 2023
Published online: 25 November 2023

References
1. Tian D, Ying Q, Jia X, Ma R, Hu C, Liu W (2021) MDCHD: a novel malware detection method in cloud using hardware trace and deep learning. Computer Networks 198:108394. https://doi.org/10.1016/j.comnet.2021.108394. (ISSN 1389-1286)
2. MoonLeeHeoKimPaekKang HHIKYBB (2017) Detecting and preventing kernel rootkit attacks with bus snooping. IEEE Transactions on Dependable and Secure Computing 14(2):145–157. https://doi.org/10.1109/TDSC.2015.2443803
3. Zhou H, Fei C, Ni L, Wu B, Li G, Han K (2022) "Detecting Kernel Rootkits in a Virtualized Infrastructure with Low-Level Architectural Features," 2022

IEEE 5th International Conference on Electronics and Communication Engineering (ICECE), Xi'an, China. pp 244–247. https://doi.org/10.1109/ICECE56287.2022.10048623

4. Krishnamurthy P, Salehghaffari H, Duraisamy S, Karri R, Khorrami F (2019) "Stealthy Rootkits in Smart Grid Controllers," 2019 IEEE 37th International Conference on Computer Design (ICCD), Abu Dhabi, United Arab Emirates. pp 20–28. https://doi.org/10.1109/ICCD46524.2019.00012
5. Xing X, Jin X, Elahi H, Jiang H, Wang G (2022) A malware detection approach using autoencoder in deep learning. IEEE Access 10:25696–25706. https://doi.org/10.1109/ACCESS.2022.3155695
6. I. Kuzminykh and M. Yevdokymenko, "Analysis of Security of Rootkit Detection Methods," 2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT), Kyiv, Ukraine, 2019, pp. 196-199, https://doi.org/10.1109/ATIT49449.2019.9030428
7. Mohammadhadi Alaeiyan, Saeed Parsa, Mauro Conti, "Analysis and classification of context-based malware behavior",Computer Communications,volume 136, February 2019, Pages 76-90, 10.1016/ j.co m c o m . 2019 .01.003.
8. Xiao J, Lu L, Wang H, Zhu X (2016) "HyperLink: Virtual Machine Introspection and Memory Forensic Analysis without Kernel Source Code," 2016 IEEE International Conference on Autonomic Computing (ICAC), Wuerzburg, Germany. pp 127–136. https://doi.org/10.1109/ICAC.2016.46
9. S. Kumar Verma, N. Anjum, A. Sharma and A. Mishra, "iSIMP with Integrity Validation using MD5 Hash," *2021 International Conference on Computational Performance Evaluation (ComPE)*, Shillong, India, 2021, pp. 094-097, https://doi.org/10.1109/ComPE53109.2021.9752433.
10. Alshamrani SS. Analysis of MachineLearning Based Technique for Malware Identification and Classification of Portable Document FormatFiles, Hindawi Security and Communication Networks Volume 2022, Article ID 7611741, 10 pages https://doi.org/10.1155/2022/7611741.
11. Donghai Tian, Rui Ma , Xiaoqi Jia, and Changzhen Hu, "A Kernel Rootkit Detection Approach based on Virtualization and Machine Learning" *IEEE Access* PP (99):1-1 july, 2019.
12. Chin-Ling Chen, Supaporn Punya, "An enhanced WPA2/PSK for preventing authentication cracking", The International Journal of Informatics and Communication Technology (IJ-ICT), Vol.10, No.2, August 2021, pp. 85-92,DOI: https://doi.org/10.11591/ijict.v10i2.pp85-92.
13. Sanjay Sharma, C. Ramakrishna and Sanjay K. Sahay, "Detection of Advanced Malware by Machine Learning Techniques" Access AISC, Volume 742, 2019.
14. Panker T, Nissim N (2021) Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in Linux cloud environments. Knowl. Based Syst. 226:107095
15. Lin Y, Huang S, Hong M, Chen S, Li X, Lin D, "MD5 Encryption Algorithm Enhanced Competitive Swarm Optimizer for Feature Selection,", (2019) IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). Xiamen, China 2019:1250–1254. https://doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00178
16. Wang Q, Qian Q (2022) Malicious code classification based on opcode sequences and textCNN network. J Inf Secur Appl 67:103151
17. Diaz JA, Bandala A (2021) Portable Executable Malware Classifier Using Long Short Term Memory and Sophos-ReversingLabs 20 Million Dataset", *In Proceedings of the TENCON 2021—2021 IEEE Region 10 Conference (TENCON)*, Auckland, New Zealand, 7–10 December. pp 881–884
18. Ullah A, Laassar I, Şahin CB, Dinle OB, Aznaoui H, "Cloud and internet-of-things secure integration along with security concerns", International Journal of Informatics and Communication Technology, Vol. 12, No. 1, https://doi.org/10.11591/ijict.v12i1.pp62-71.
19. J. Zhang, F. Zou and J. Zhu, "Android Malware Detection Based on Deep Learning," 2018 IEEE 4th International Conference on Computer and Communications (ICCC), Chengdu, China, 2018, pp. 2190-2194, doi: https://doi.org/10.1109/CompComm.2018.8781037.
20. Rezende E, Ruppert G, Carvalho T, Ramos F, de Geus P (2017) "Malicious Software Classification using Transfer Learning of ResNet-50 Deep Neural Network.", *In Proceedings of the 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Cancun, Mexico, 18–21 December. pp 1011–1014
21. J. Zhao, S. Zhang, B. Liu and B. Cui, "Malware Detection Using Machine Learning Based on the Combination of Dynamic and Static Features," 2018 27th International Conference on Computer Communication and Networks (ICCCN), Hangzhou, China, 2018, pp. 1-6, https://doi.org/10.1109/ICCCN.2018.8487459.
22. J. Zhao, S. Zhang, B. Liu and B. Cui, "Malware Detection Using Machine Learning Based on the Combination of Dynamic and Static Features," 2018 27th International Conference on Computer Communication and Networks (ICCCN), Hangzhou, China, 2018, pp. 1-6, doi: https://doi.org/10.1109/ICCCN.2018.8487459.
23. Vinayakumar R, Alazab M, Soman KP, Poornachandran P, Venkatraman S (2019) Robust intelligent malware detection using deep learning. IEEE Access 7:46717–46738. https://doi.org/10.1109/ACCESS.2019.2906934
24. Nadim M, Akopian D, Lee W (2021) "A Review on Learning-based Detection Approaches of the Kernel-level Rootkit," 2021 International Conference on Engineering and Emerging Technologies (ICEET), Istanbul, Turkey. pp 1–6. https://doi.org/10.1109/ICEET53442.2021.9659710
25. Win TY, Tianfield H, Mair Q (2015) "Detection of Malware and Kernel-Level Rootkits in Cloud Computing Environments," 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, New York, NY, USA. pp 295–300. https://doi.org/10.1109/CSCloud.2015.54

Publisher's Note