A framework for detecting malware in Cloud by identifying symptoms

Keith Harrison[1], Behzad Bordbar[1,2]
Hewlett-Packard Laboratories1,
Bristol, BS34 8QZ, UK
{keith.harrison, syed-taha-tariq.ali,
cid,andy.norman}@hp.com

Syed T.T. Ali[1], Chris I.Dalton[1], Andrew Norman[1]
School of Computer Science2,
University of Birmingham,
B15 2TT, UK
B.Bordbar@cs.bham.ac.uk

***Abstract*—**Security is seen as one of the major challenges of the Cloud computing. Recent malware are not only becoming more sophisticated, but has also demonstrated a trend to make use of components, which can easily be distributed through the Internet to develop newer and better malware. As a result, the key problem facing Cloud security is to cope with identifying diverse set of malwares.

This paper presents a method of detecting malware by identifying the symptoms of malicious behaviour as opposed to looking for the malware itself. This can be compared to the use of symptoms in human pathology, in which study of symptoms direct physicians to diagnosis of a disease or possible causes of illnesses. The main advantage of shifting the attention to the symptoms is that a wide range of malicious behaviour can result in the same set of symptoms. We propose the creation of Forensic Virtual Machines (FVM), which are mini Virtual Machines (VM) that can monitor other VMs to discover the symptoms. In this paper, we shall present a framework to support the FVMs so that they collaborate with each other in identifying symptoms by exchanging messages via secure channels. The FVMs report to a Command & Control module that collects and correlates the information so that suitable remedial actions can take place in real-time. The Command & Control can be compared to the physician who infers possibility of an illness from the occurring symptoms. In addition, as FVMs make use of the computational resources of the system we will present an algorithm for sharing of the FVMs so that they can be guided to search for the symptoms in the VMs with higher priority.

Keywords- Cloud; security; virtualisation

I. INTRODUCTION

Cloud Computing is increasingly being promoted as a business model comprising of services that are marketed and delivered in a mode similar to traditional utilities of electricity, gas and water. In order for the Cloud environment to be profitable, there is temptation to homogenize the applications and Operating Systems used. But as the Cloud becomes more homogenous, it will provide bigger and richer targets for attackers; places where the attacker may be confident to find lucrative information or where disruption will have the greatest impact. As a result, ensuring security of the cloud is seen as a major Engineering challenge [1].

The malware's landscape is rapidly changing. Malware is getting ever more sophisticated. They are getting more competent at mutating the target environment in order to avoid detection [2-4]. The use of polymorphism and metamorphism has become a common practice and Root-kit technology is routinely being used [5-7]. Recent malwares have also demonstrated a trend to make use of components for their construction [3, 8]. These components can easily be distributed through the internet and could be easily used to develop newer and better malware. It is possible to combine variants of such components, which perform similar functionalities, to produce even more diverse variants of malicious code that can evade detection. As a result, a key challenge of Cloud security is to cope with identifying diverse set of malwares.

The key idea underlying this paper is to focus on identifying the symptoms of malicious behavior as oppose to directly looking for the malware within a Cloud. For example, a wide range of malwares disable the defences of the system by stopping the Antivirus software. Absence of Antivirus software from the Process Table of a system can be seen as a symptom that points to the possibility of malicious behavior. Of course, it is possible that the Antivirus has been stopped for various legitimate reasons. The key point is that, appearance of the symptoms can be a reason for further investigation. In particular, observing more than one symptom can convince us of the higher possibility of an undesirable behavior. This can be compared to a patient who has more than one symptom: headache, fever and etc. The main advantage of shifting the attention to the symptoms is that a wide range of

malicious behavior can result in the same set of symptoms.

The method suggested in this paper relies on the Virtualization [9] which is widely used within the Cloud. We propose creation of Forensic Virtual Machines (FVM), which are mini Virtual Machines (VM) that can monitor other VMs to discover the symptoms in real-time via Virtual Machine Introspection [10]. Each FVM is specialized to look for a unique type of symptom. The FVMs are small, so that they can be checked manually to ensure their integrity. In addition, FVMs exchange messages via secure multicast channels to share information about discovering of symptoms within VMs. The discovery of symptoms within a VM is a collaborative effort; identifying a symptom would result in a chain of activities to direct other FVMs to the VM to inspect for further symptoms. This is because the more symptoms that are detected the higher is the chance of finding malicious behavior in the VM. The FVMs report to a Command & Control module that collects and compiles the information and analyzes them. The Command & Control module can use the virtualisation mechanism to "freeze" the VM by denying it any CPU cycles, as a result to stop the malicious activity. The memory will remain frozen until it can be forensically examined or copied for further analysis.

The paper is organised as follows. Section II presents the preliminary material used in the rest of the paper. In section III we shall describe the newly emerging trend of Component based Malware, which motivates our idea of looking for the symptoms rather than the malware itself. Section IV describes the new trend of Component-based malware and the challenges that it poses to security. Section V formulates the problem addressed in the paper followed by Section VI, which presents our sketch of the solution. In Section VII we describe the Symptoms and presents samples of symptoms appearing in a number of high-profile malware. Design principles behind FVM are explained in Section VIII. This section also illustrates implementation of an example FVM. In Section VIII we focus on Mobility algorithms and describe a simulator that we have developed for the analysis of the mobility algorithms. The paper ends with a brief conclusion.

II. PRELIMINARIES

A. *Virtualization*

Virtualisation is "A framework or methodology of dividing the resources of a computer hardware into multiple execution environments..." [9]. Virtualisation relies on Virtual Machines, software that emulates or simulates the capabilities of the hardware. It is capable of running a complete operating system along with any applications that runs on top of that OS [14].

SYMPTOMS INDICATING MALICIOUS BEHAVIOUR

Definition: A symptom is an abstraction of an observable (via VMI) characteristic, which can be linked to malicious behaviour, so that appearance of a symptom indicates possibility of a malicious behaviour.

Our approach relies on the FVMs to search for the symptoms within the VMs. In what follows we shall explain examples of the symptoms. Fig. 1 depicts a high level view of Xen [9], which is an open source virtualization software based on paravirtualization technology. In this architecture, the Virtual Machine Monitor (VMM) is an abstraction of the underlying physical hardware and provides hardware access for the different virtual machines. Xen includes a special VM called Domain 0 (Dom0). Only Domain 0 can access the control interface of the VMM, through which other VMs can be created, destroyed, and managed. This powerful VM is used to create other Virtual Machines that can access the hardware through secure interfaces provided by Xen. In addition it is possible to create other virtual machines that can access the physical resources provided by Domain 0's control and management interface in Xen.

Virtual Machines are heavily used within the Cloud. In addition to the advantage of running multiple Operating Systems simultaneously, Virtualisation reduces the cost of infrastructure implementation and the associated cost of maintenance by optimising the utilisation of the resources. A user can ask for new VMs when extra resources are required and decommission some of the VMs, when they are no longer required. In addition, Virtualisation provides a powerful technique for securing the VMs, which is commonly known as Virtual Machine Introspection.

A. *Virtual Machine Introspection*

Virtual Machine Introspection (VMI) can be defined as a virtualisation based technique that enables one guest VM to monitor, analyse and modify the state of another guest VM by observing its virtual memory pages. Such introspection can be carried out by a VMM that hosts the VM or another VM which has been granted special privileges by the VMM. VMI will allow product developers and researchers to move the security related software out of a probable target host/VM and take advantage of the hosts lack of awareness to detect any malicious events or code that is being executed in

runtime. One of the early methods of introspecting a Virtual Machine from an external VM is by Garfinkel and Rosenblum [10]. They used VMI to develop an Intrusion Detection System (IDS), called Livewire, for a customized version of VMWare Workstation for Linux. VMI techniques have also been used in Digital Forensics [13] and [15]. Hyperspector [16] implemented another Intrusion Detection System for distributed computer systems using VMI to isolate the IDS from the servers that they monitor. These isolated IDSes are located inside distinct VMs which are termed as IDS VM. There are also commercial products built using VMI technology [17] .

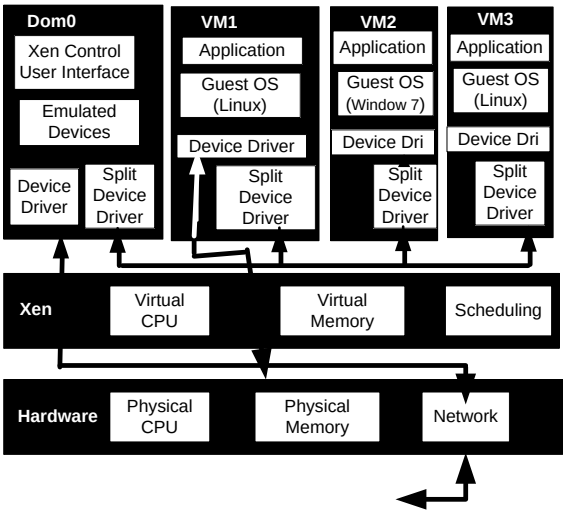

Figure 1: High-level view of Xen (see [9] for details)

IV. COMPONENT BASED NATURE OF MALWARE

There is a financial incentive for malware authors to write "good" software [2-4]. Writing malware, particularly malware that is designed to evade detection for a long time, is hard. There are many problems to be solved: how to gain entry to a machine, how to install itself, how to evade detection, how to prevent the infected machine informing the owner, how to propagate, how to make analysis difficult and so on. Solving these for the first time would indeed be a daunting task. The code would be "flaky" and easily detected. Fortunately, for the attacker, and unfortunately for the defender, there is the web. Malware writers publish solutions (even code) to solve these problems. As a result, it is common to come across variants of the same script within various malware products [18]. Toolkits and crimeware have been written that allow the management of the Botnets from user-friendly interfaces [8]. Radianti [3] compares the malware market for the Skilled-Hacker, who are interested in "big" 0-day scenarios, and Script-kiddies who are not skilled and only understand the effects and side effects of executing malicious code. In the past component-based malware was often used by the Script-kiddies; there is clear evidence that Skilled hackers are also increasingly relying on malware components. As new and more sophisticated solutions become known, they are being made available to even the novice attackers.

V. DESCRIPTION OF THE PROBLEM

Identifying malicious behaviour via VMI, which involves automated inspection of a VM via another VM, must address the following two challenges.

1) Diversity of Malware: Because of the polymorphic nature of the malware, the inspection of a virtual machine must identify potentially huge number of the variants of the same malware which perform identical functionality. One of the reasons for such diversity is the Component-based nature of modern malware. Indeed, the malware is becoming component-based. Modern malware reuse snippets of malicious code to reduce the chance of bugs and to produce better quality malware. Because of the large number of combinations, it is not simply possible to create Virtual Machine Inspectors which can identify all such possibilities.

2) Efficient and scalable management of computational resources required for Introspection: VMI requires computational resources, which could otherwise be allocated to the clients; indeed Computation is one of the key commodities provided by the Cloud. Producing a large number of virtual machines to monitor a guest Virtual machine can result in a waste of the valuable computational resources. Because Cloud systems are expected to be very large, any practical method of inspecting the host VMs must be scalable.

In addition to the above main challenges, any solution that enhances and extends the system must not introduce new attack vectors. It is crucial that the security experts and mangers who work with the system can inspect any enhancement so that to become convinced of its integrity.

VI. SKETCH OF THE SOLUTION

Modern malware, such as Rootkits, rely on being able to modify their environment to remain undetected. They are designed to prevent antivirus products from being able to report their existence. However, it is very difficult to remain invisible to someone viewing from "outside" when VMI is used. Relying on VMI, the external viewer can observe

- changing state of a VMs memory,

- processes that take inordinately long times to initialize,
- snippets of program code that has been obfuscated,
- snippets of code containing known crypto algorithms,
- system code has been replaced,….

We refer to the above, which are indicators of possible malicious behavior, as Symptoms. Symptoms are not malicious on their own; each symptom can be seen as a byproduct of one or more malicious activity. In reality a symptom can be caused by a number of malicious behaviors or an innocent system activity. One can draw an analogy between the symptoms in this context and symptoms associated with disease in human body. An occurrence of a symptom such as headache can be a sign of a number of illnesses. We may not know the cause of the headache but it is likely to prompt us to action such as visiting a doctor who may inspect for other symptoms or arrange further examination to make a diagnosis. In particular, appearance of more than one symptom can not only narrow down to specific group of malware, but also can alert us to take action rapidly, as the chance of malicious behaviour increases. Figure 2 depicts outline of the approach suggested in this paper. It shows a number of small independent VMs, called Forensic Virtual Machines (FVMs), which have been given the capability to inspect the memory pages of specific Customer Virtual Machines. Once a symptom has been detected, then the FVM reports its findings to other FVMs. In such cases, other FVMs will be prompted to inspect the VM for the additional symptoms. In addition, when a symptom is discovered, this fact is reported, via Dom0, to a Command & Control centre. The Command and Control Centre correlates this information with information from other sources to identify an appropriate mitigation. For instance, the Command & Control, through the Dom0 and hypervisor, can "freeze" the Customers VM by denying it any CPU cycles as a result to stop the malicious activity. The memory will remain frozen until it can be forensically examined or copied for further analysis.

FVMs make use of the computational resources that could otherwise be allotted to the Cutomer's VMs. As a result, management of the efficient allocation of the resources to the FVMs is crucial. In particular, creating and deploying an FVM is computationally intensive. In addition, permanent monitoring of an FVM is costly and wasteful, as the symptoms are expected to appear sparsely. We have designed the FVMs so that they regularly change their target Customer's VM. To achieve this, a distributed algorithm is created to allow the FVM schedule moving its searching process from one Customer's VM to another. We refer to such algorithms as *mobility* algorithms.

This paper addresses the two challenges posed in section IV as follows. *To deal with the Diversity of Malware, we propose looking for the symptoms of the malicious behaviour as oppose to looking for the symptoms themselves.* The infrastructure suggested will infer the possibility of malicious behaviour from the list of identified symptoms. *Efficient and scalable management of computational resources required for Introspection is achieved by using mobility algorithms that share the FVMs between the resources.*

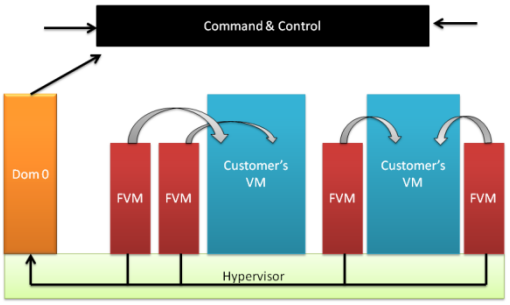

Figure 2: FVMs inspecting VMs

The infrastructure suggested in Figure 2 is a typical autonomic system, which adapts itself to the changes in its environment. The focus of this paper is not to report on the whole infrastructure, instead we are interested in dealing with the challenges described in Section IV, which are essential for the creation of this infrastructure. As a result, in this paper we shall report on our progress in three directions. Firstly, we will report on our finding and classifications of symptoms. In Section VI we will describe six examples of symptoms. We have created FVMs for the detection of some of the symptoms described. Secondly, in section VII we will describe our design of the FVMs, their lifecycle and an example of one of the FVMs developed. We have also developed a number of mobility algorithms. Mobility algorithms are distributed and complex. As a result, we have developed a simulator for studying the behaviour of the Mobility algorithms. Thirdly, in section VIII, we will describe a mobility algorithm, our simulator and its use in analysing the presented algorithm.

VII. SYMPTOMS INDICATING MALICIOUS BEHAVIOUR

Definition: A symptom is an abstraction of an observable (via VMI) characteristic, which can be linked to malicious

behaviour, so that appearance of a symptom indicates possibility of a malicious behaviour.

Our approach relies on the FVMs to search for the symptoms within the VMs. In what follows we shall explain examples of the symptoms.

A. Examples of Symptoms

Missing processes: A key strategy of malware is to remain hidden for as long as possible. A common technique used by malware is to stop crucial processes which might help in its detection. For example, in the Conficker C, 23 processes are immediately aborted whenever they are discovered running on the victim host, among them *sysclean*, *tcpview*, *wireshark*, *confik* and *autorun*, see page 12 of [11] for a list. Stopping such processes helps the malware to remain hidden. It is possible to develop FVMs to inspect the process tables and alert if a process is missing.

Modification of in-memory code: To remain undetected, a malware must make sure that the system continues with its normal course of behavior. A common technique is to inject code into the in-memory code. Doing so, the victim's DLL will remain unchanged. For example, depending on the security product installed, Stuxnet [12] injects itself into privileged processes. In one case the malicious code was injected into *winlogon*, so that it can run when a user logs in. Other applications that use the DLL will remain unaffected. Subsequently, detection mechanisms are not alerted towards the malicious behavior. Injection of such code may leave fingerprints that can be detected by an FVM. For example, we may be able to hash a portion of the in-memory version of the DLL for the authentication.

Tampering with the Registry keys: Some malwares modify the state of the victim's system by altering the Registry keys. It is expected that a Cloud to follow a homogenous structure consisting of a few similar configurations. Small FVMs can check the registry to identify any suspicious alterations in the values expected within the hierarchy of directories associated to a registry Key. In some cases, the malware attempts to obfuscate the registry cases. For example, Conficker [11] adds strings such as *app*, *audio*, *image*,... to the registry to obfuscate registry configuration changes in *svchost* and *netsvc*. FVMs can inspect the VM's registry to spot unexpected values.

Checking for symptoms at the startup: The Zeus malware [8] appends the path `C:/Windows/System32/sdra64.exe` to `HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/WindowsNT/CurrentVersion/Winlogon/Userinit` registry key. This entry enables the Zeus malware to initiate its installation process again during Windows startup. Inspection for such symptoms can be carried out only at the startup time. Another sign of malicious behaviour is when a process is initialised for too long [11].

Modifying the time attributes of a file: Zeus [8] modifies the time of the creation of some of the malicious exe files to the time of the installation of the operating systems. This is done to trick the human inspector by implying that the file has been there from the beginning. This is a symptom that can be detected easily by a simple search.

Identifying suspicious snippets of code: Obfuscation and use of crypto algorithms is very popular with malware writers. Snippets of program code that has been obfuscated or the code containing known crypto algorithms can be a sign of malicious behavior. For example, three variants of Conficker (A,B and C) make use of RC4, RSA, and MD-6 and keep updating the implementations, see [11] for details. Sometimes malware carry keys used for encryption. In fact these keys have been successfully used to search the memory for the sign of an infected machine, as the key for Conficker variant A, B and C are known. It is possible to develop FVMs for look at high entropy pieces of bytes to identify keys [19]. It is reasonable to be suspicious of the DLLs which do not require any encryptions and yet carrying high-entropy pieces of code.

VIII. Symptom detection via FVMs

We conduct the process of Virtual Machine Introspection through a number of Forensic Virtual Machines (FVM). As depicted in Figure 2, an FVM is a virtual machine that is configured by the user or administrator to observe and examine the state of a Customers' VM to detect presence of a symptom. Not only integrity of an FVM is of crucial importance, but also convincing the clients that an FVM is not conducting undesirable activities is essential. As a result, we follow the following four guidelines for the FVM:

A. Guideline for designing FVMs

1) *FVM only reads:* Virtualisation allows both reading and writing into a VM. As a design principle, our FVMs never alter states of a VM. This ensures the integrity of the operation within the VM and also allows searching for malicious behaviour while remaining hidden from hackers.

2) *FVMs are small; one symptom per FVM:* We design the FVMs to be small so that the clients can manually inspect their code and make sure of their integrity. In addition, although it is possible to create super size FVMs, in the interest of clarity, each FVM is designed to deal with identifying a single symptom. Creating small FVMs is a key step towards ensuring that our suggested symptom detection scheme is not introducing an easy attack vector into the overall system.

3) *FVMs inspect one VM at a time:* To avoid any possibility of leakage of information, an FVM will inspect only one VM at a time and will flush its memory before leaving to inspect another VM.

4) *Secure communication:* FVMs communicate with each other and the management system via sending messages through a secure multicast channel.

B. Sketch of implementation for an FVM

In this section we shall present an example of implementation sketch for an FVM that we have developed. Suppose that we are interested in establishing whether the address space of an Internet Explorer or Firefox process running within a virtualized guest OS (the target) contains a particular text string or section of (malicious) machine code. To search for something in the address space of a particular application process within the target guest VM, the FVM would proceed roughly as follows:

1) Locate the offset of the target guest kernel task structures. For both Windows and Linux, the guest (kernel) virtual address of these structures is either a well-known value or easily determinable. A guest operating system (Windows or Linux for example) maintains such internal structures that describe the application processes or tasks currently instantiated on the system. Included in the task structure for a particular application process is a pointer to region that contains the page tables that should be loaded when that process is running. Also included in that task structure is a list of areas of the application's virtual address space that it is actual using. Even on 32bit systems, the virtual address space of a process is very large (usually 4GB) and in reality the application will generally use a small fraction of that space. Together, the page tables and virtual address space region structures allow the determination of the actual system physical memory being used by that application process.

2) Convert the known target guest kernel virtual address of the task structures into a machine physical address so that same physical page can be mapped into the FVM and the contents examined, and the hence details found of the specific process we are interested in inspecting.

Taking Linux as an example, the page tables of ANY application process running within the OS contain a mapping for kernel virtual addresses of that OS as well as the application addresses (paging protection mechanisms prevent application processes actually accessing kernel memory).

On x86 based platforms, the root location of the set of page tables from the currently running guest process can be found by examining a particular system register (CR3). This register machine state is available to the FVM from the Hypervisor (Xen). Given that register value (which is actually a physical memory address), the FVM maps the physical page containing that physical address into its own address space. From that it can traverse the page tables being used by the target guest OS (mapping additional physical pages from the guest as required) until it finds the physical page corresponding to the virtual address of the guest OS task structures.

3) From the now located target guest OS task structures, it finds the actual page tables and memory regions being used by the specific process of interest. By proceeding along similar lines as step (2) including page table traversing, the FVM can now map the physical memory actually in use by the application process of interest into its own address space and inspect the contents.

In our current implementation, we assume that the target VM is running on top of a Xen hypervisor using the library XenAccess, which is an API to allow both Xen's priviliged VM, Dom0, and other suitably authorised VMs to read a target VM's memory. In addition, the library, XenAccess, provides an API which, along with other functionality, allows the calling process to map the contents of selected memory pages from a target VM into its memory space. We have also developed a library for conducting common task such as searching of a memory space.

C. Formalising the Forensic Virtual Machines

Consider a set of virtual machines $V = \{v_1, v_2, \cdots v_N\}$. To discover an attack we inspect the VMs for discovering the symptoms associated to it. Normally, an attack can be related to more than one symptom. In addition, appearance of a symptom ONLY points into the likelihood of the malicious behaviour. We formalise this with introducing the concept of *Configurations*. Suppose that $C = \{c_1, c_2, \cdots c_K\}$ denotes the set of all Configurations. For example, a Configuration c_1 might be

detected by identifying symptoms s_1, s_3 and s_6. Some of the Configurations are more important than the others. As a result, we attribute a value between 1 and 10 to each Configuration c_i, denoted by $val(c_i)$. The values of Configurations are determined by the security experts. For example, if the symptoms s_1, s_3 and s_6 appear together in an attack with serious consequences for the system, the experts assign higher values. In contrast, the symptoms which are signs of attacks that are either old or can be dealt with via existing antivirus products, the value for the Configuration would be low.

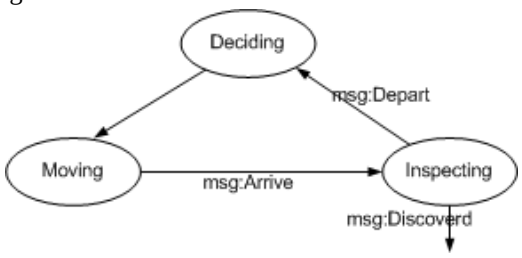

Figure 3: Life Cycle of an FVM

We make use of the Greek letters φ, φ_1, φ_2,... for referring to the FVMs. Each FVM is responsible for detecting a unique symptom which we refer to its *own symptom*. Each FVM deals with the symptoms within a given Neighbourhood $N(\varphi)$, which is a subset of all VMs.

Each FVM uses the messages received from other FVMs to form a picture of its surrounding world. Part of this involves establishing the symptoms which have been discovered by other FVMs. This is done with the help of a number of variables to record if the symptom s_i is discovered in the virtual machine v_j. In addition, in our current implementation, an FVM is only interested in the VMs within its own Neighborhood, the values for the coordinates corresponding to its Neighborhood is updated. Each FVM keeps a record of the last time that a VM is visited with the help of the local variable ***lastVisited***. To be precise, ***lastVisited*** assigning the local time of the last visit to a virtual machine in $N(\varphi)$ via an FVM of type φ. This can be the φ itself or any other FVM which inspects the same type of symptom φ. ***lastVisited*** is updated by the messages arriving at the FVM.

To keep a balanced number of the FVMs visiting the VMs, each FVM, φ is also interested to know the number of other FVMs of *its own kind* which are visiting the virtual machines in its neighborhood.

We impose an upper bound on the amount of time that an FVM can inspect a VM. For each symptom there is an assigned time interval, which the FVM can spend a random amount of time from that interval in each virtual machine. We refer to that time as Permissible Time to Stay (PT2S).

D. Life cycle of an FVM

Each FVM follows the life cycle described in Figure 3. Most of the time an FVM inspects (*Inspecting*) a virtual machine. If the discovery is successful, a *msg:Discovered*, which is of the form $< Disc, s_i, v_j >$, is sent. The FVMs also report the absence of the symptoms to clarify if the symptom has disappeared. Otherwise, at the end of the Permissible Time to Stay in the VM, it sends a *msg:Depart* and changes its state to Deciding during which on the basis of the information provided via the messages chooses the next VM to be inspected. As soon as the Deciding takes place, the FVM moves to Moving step. Then a message *msg:Arrive* is sent and the inspection is carried out on the new VM. *msg:Depart* and *msg:Arrive* have the format $< Dept, s_i, v_j >$ and $< Arriv, s_i, v_j >$.

IX. MOBILITY ALGORITHM

One of the key challenges of security of the Cloud is to cope with a large infrastructure. It is not possible to deploy many FVMs for each VM. This would be a drain on the computational resources as the FVMs will be wasting computational resources continuously, while the symptoms may appear sparsely. Instead, we introduce the concept of *Mobility* algorithms. An FVM, which is programmed to look for a symptom, regularly changes its target virtual machine. For example, if an FVM is designed to inspect the Process Tables to identify processes that take inordinately long times to initialize, it will move from one VM to another. In doing so, it will inspect one virtual machine, flush its memory and start inspecting a new virtual machine. Each FVM carries a copy of a Distributed Algorithm which identifies its next target machine. It is crucial to use a Distributed Algorithm to avoid any bottlenecks.

A. Guideline for designing Mobility algorithms

In designing Mobility algorithms, we have taken a number of issues into consideration.

1) All VMs must be visited. It is important to avoid leaving a VM uninspected for a long time. In addition, we may decide to visit VMs belonging to a group of premium customers or VMs carrying crucial duties to be visited more often.

2) The algorithms must make sure that urgency of visiting a VM increases when more symptoms are detected. This

would be similar to the cases that a patient is showing multiple disease symptoms.

3) Movement of the VM must not follow a predetermined pattern. Any predictable patterns of movement would assist the malicious activities to be hidden.

4) Simultaneous inspection of VMs by multiple FVMs. Having multiple FVMs increases the defences and improves the coverage. For example, if the FVMs scan the memory, having multiple independent FVMs will increase the chance of detecting the malicious behaviour in real-time.

B. A Mobility algorithm

In this section, we shall describe the Mobility algorithms that we have developed. As described in Fig. 3, while in Deciding state, an FVM chooses the best possible VM to move to. Each FVM will carry a copy of the following algorithm which is executed in *Deciding* state to identify the target virtual machine.

Algorithm 1: Identifying the target VM in *Deciding* state

INPUT:

$NEIB = \{v_j\}_{j=1}^{n}$ // Neighbourhood of the FVM

$\{Max(v_j)\}_{j=1}^{n}$ // Maximum number of FVM

$C = \{c_i\}_{i=1}^{K}$ // list of Configurations

$\{Disc(c_i, v)\}$ // number of symptoms discovered

$\{numFVM(v)\}$ // number of FVM in $v \in NEIB$

$\{val(c_i)\}$ // value assigned to c_i

$\{lastVisited(v_j)\}$ // last time v_j visited

λ // loneliness factor

$1 \leq \mu \leq 100$ // percentage of top VMs considered

OUTPUT:

$v \in NEIB$ // target VM for the next move

START

1. *A:=NEIB* // A is the list of potential targets
2. *For v in A* //discard VMs with too many FVMs
   if (numFVM(v) ≥ Max(v))Then A= A\{v}
3. *If A is empty choose a random v and go to END*
4. *For v in A calculate the F-value as follows:*

$$F(v) = \sum_{i=1}^{K} \frac{Disc(c_i, v)}{size(c_i)} val(c_i) + \lambda[\boldsymbol{CurrentTime} - \boldsymbol{lastVisited}(v)]$$

5. *Create $B \subseteq A$ of the VMs with top μ percent of the F-value.*
6. *return a random value from B*

END

The process of selection is carried out from a subset of Virtual Machines called Neighbourhood (*NEIB*). In fact, the FVM which implements this algorithm only keeps the data related to the VMs in its own Neighbourhood. We set an upper bound $Max(v_j)$ on the number of the FVMs associated to each VM v_j. This is to avoid allocation of all resources to a few virtual machines and starving the remaining VMs. It is important to prioritize inspection of the scenarios which lead to discovery of a combination of the symptoms that may unveil a crucial malicious behaviour associated to high value of Configuration (see VII.D for the definition). As a result, to each Configuration a value $\{val(c_i)\}$ is assigned.

In this algorithm, we give priority to the discovery of the Configurations with the shortest remaining steps to be completed. Assume that a Configuration C1 requires inspection of symptoms s_1, s_2, s_3 and s_4. If on the VM v_1 symptoms s_1, s_2 and s_3 are discovered, while on VM v_2, only the symptom s_1 is discovered. It makes sense to complete detection of the symptom s_4 on v_1 as oppose to the detection of the same symptom on v2. So moving the FVM to v_1 is better than moving to v_2. As a result, the double-array variable $\{Disc(c_i, v)\}$ is included to keep the record of the number of number of symptoms of c_i discovered in FVM v.

We wish to avoid *lonely* VMs. A lonely VM is a VM that has not been inspected for a long time by FVMs of a given type. Such VMs can be a prime target for a malicious behaviour. The variable λ is defined to adjust the importance of loneliness, as we will explain later.

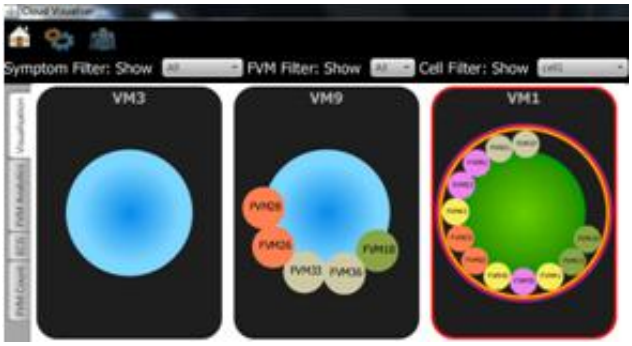

Figure 4: FVMs are swamping VM1 (RHS of picture)

Finally, allocation of the FVMs to a VM must not follow a fully deterministic pattern. Any predictable behaviour is easier to be exploited. The algorithm makes random moves. The target VM is chosen from the top μ percent of the VMs with top value of the function F, as we will explain below.

The algorithm starts by assigning all elements of the neighbourhood into a set variable *A*. In (2) we discard all

FVMs which have too many FVMs. This is done by comparing the value of *numFVM(v)* and *Max(v)*, where the variable *numFVM(v)* keeps the current number of FVMs in the VM v and *Max(v)* is the maximum permissible number. The step (3) will never execute in a realistic scenario as it represents surplus of the FVMs; step (3) added to ensure the algorithm terminates in all cases.

To find the most suitable VM in the Neighbourhood a valuation function F is presented, in which $Disc(c_i, v)$ is the number of symptoms of ci which are discovered within the virtual machine *v*, while $size(c_i)$ is the number of the symptoms in c_i. In effect, we are scaling the importance of a Configuration, i.e. $val(c_i)$, with the ratio of the symptoms which are discovered. Adding such values gives an importance to the configurations with higher values and in special those which we are about to discover all their symptoms. The value *CurrentTime - lastVisited(v)*, which is the time since last visit by an FVM, denotes how "lonely" the virtual machine *v* has been. λ is a scaling factor to adjust the importance of the loneliness factor. If this value is set to a high value, the valuation function F will give more importance to loneliness. Then we can assume that we have ordered all VMs that can be inspected according to the value of F. Then we choose a virtual machine among the VMs in the top μ percent values of F(v) to create the set B. To ensure a random selection of the destination, we randomly choose one of the VM with high *F*-value.

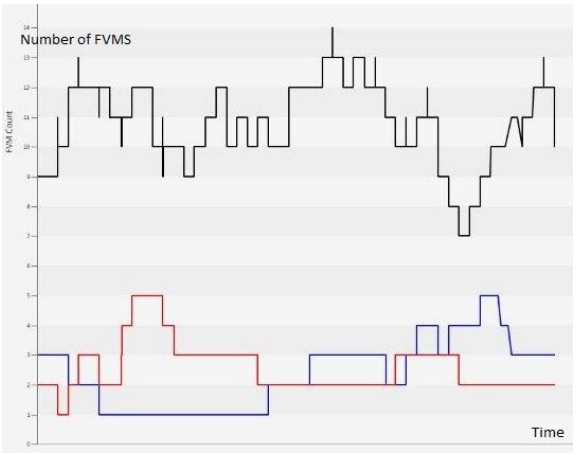

Figure 5: Graphical representation of Swamp

C. A Simulator to study Mobility algorithm

Mobility algorithms are complex. Since the algorithm is highly distributed ensuring stability of the system, i.e. the FVMs of different types eventually visit all the VMs to discover different symptoms, is highly non-trivial. It is also important to discover guidelines for setting parameters such as loneliness, size of the neighbourhood, maximum number FVMs etc. In addition, it is crucial to be aware of the effect of changes in the parameters. As a result, to evaluate the algorithm we have developed a simulator.

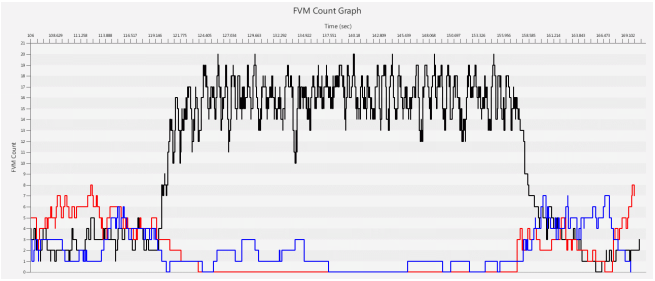

Figure 6: number of FVMs visiting VMs (λ= 0.01)

The simulator is written in Scala with a front end written in JavaFX. Scala being an actor-based language has allowed us to create the VMs and FVMs as autonomous distributed agents. As a result, the simulator mimics the real world. Currently, we run the simulator within a single machine, but the system is designed so that it is possible to distribute the simulator across multiple machines for conducting distributed simulations. Virtual machines are actors that contain a dictionary giving the status of symptoms present in the VM. The status of these symptoms is altered in real time to simulate attacks. FVMs are also actors that mimic the real time behaviours of the real FVMs, they move their attention according to their in-built Mobility algorithms. To simulate the VMI, the FVMs send messages to the VM actor asking whether the symptom is present or not. The FVMs announce their discoveries on to an internal bus where other FVMs, the graphics subsystem and the command and control system can listen. We have used the simulator to study various algorithms and the effect of assigning different parameters within an algorithm. The simulator is equipped with a user interface written in JavaFX. In Figure 4 we can see that when one symptom is discovered other FVM join the hunt to discover if other symptoms exist. We refer to this phenomenon as swamping. In addition to the animated user interface, the simulator allows capturing of the data produced in form of various graphs. For example, given a set of parameters we can study how often FVMs of a given type visit the VMs. Figure 5 shows the number of FVMs in three VMs when one of the VMs (depicted by black line and a higher graph) is attacked. The swamping effect is clearly visible while other two VMs are also visited.

We have also used the simulator to study the effect of changes in the parameters. Figures 6 and 7 show the number of FVMs present on the same VM when λ was set to 0.01 and 10 respectively. In Figure 6, when under attack, all resources are diverted to the VM which is being attached. As a result, since more resources are allotted, it is expected that the symptoms be discovered faster. But this increase comes at a cost of almost complete loss of coverage received by the other FVMs. On the other hand if we set λ to a large value the FVMs tend to distribute themselves uniformly across all the VMs. The optimum value of λ will eventually be a trade-off between the required coverage and level of protection received by either the attacked FVM or by the whole system and it could vary with different types of attack.

To summarize, our study of the algorithms via the simulator resulted in identifying suitable values for the parameters so that the mobility algorithm will tend to distribute the FVMs uniformly across the system. The algorithm is configurable and it could be configured to respond rapidly to emerging threats by dispatching FVMs, in a short time window, towards a VM behaving suspiciously. This would lead to an early detection of a malware/attack in a cloud. We believe that knowing the cause accurately and detecting it at an early stage will be the key for taking the right actions to mitigate the potential threat to the entire cloud. It is also equally important that the other VMs in the system are not left lonely at any stage. We achieve this by setting the value of λ to an optimum for our simulation.

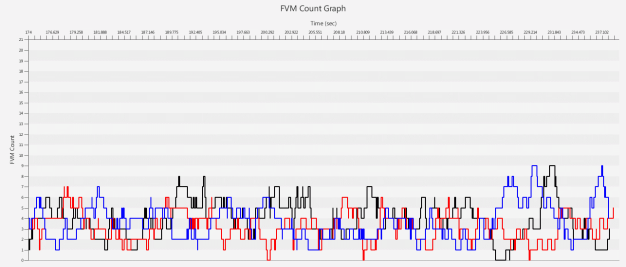

Figure 7: number of FVMs visiting VMs (λ= 10.0)

X. CONCLUSION

This paper presents a method of detecting the symptoms of malicious behaviour in Cloud using Virtual Machine Introspection. Samples of symptoms from real-world high-profiles attacks are presented. The process of inspection of a VM, which involves mapping of the memory pages of the machine physical address of the VM to the FVMs is described. In addition, a sample of a Mobility algorithm, a Distributed Algorithm, which allows collaboration of the FVMs in identifying multiple symptoms is explained. Finally, the paper reports on the simulator that we have developed to study the mobility algorithms.

REFERENCES

[1] T. Mather, S. Kumaraswamy, and S. Latif, Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance: O'Reilly Media, Inc. , 2009.

[2] J. Franklin, V. Paxson, A. Perrig, and S. Savage, "An inquiry into the nature and causes of the wealth of internet miscreants," presented at the Proceedings of the 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA, 2007.

[3] J. Radianti, "Eliciting Information on the Vulnerability Black Market from Interviews," in Emerging Security Information Systems and Technologies (SECURWARE), pp. 154-159, 2010.

[4] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou, "Studying malicious websites and the underground economy on the Chinese web," presented at the on the Chinese web. Workshop on the Economics of Information Security (WEIS), 2007.

[5] J. Baltazar. (2009). The Real Face of KOOBFACE: The Largest Web 2.0 Botnet Explained. Technical report from Trend Micro.

[6] P. Szor, The Art of Computer Virus Research and Defense: Symantec Press, 2005.

[7] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: a new approach to binary code obfuscation," presented at the 17th ACM conference on Computer and communications security, 2010.

[8] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang, "On the analysis of the Zeus botnet crimeware toolkit," in Privacy Security and Trust (PST), 2010 Eighth Annual International Conference on, 2010, pp. 31-38.

[9] D. E. Williams and J. R. Garcia, Virtualization with Xen: including XenEnterprise, XenServer, and XenExpress Syngress, 2007.

[10] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," presented at the In Proc. Network and Distributed Systems Security Symposium},, 2003.

[11] P. Porras, H. Saidi, and V. Yegneswaran, "Conifcker C Analysis," 2009.

[12] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier, Version 1.4 (February 2011)," 2011.

[13] K. Nance, M. Bishop, and B. Hay, "Investigating the Implications of Virtual Machine Introspection for Digital Forensics," presented at the International Conference on Availability, Reliability and Security (ARES 09), 2009.

[14] R. P. Goldberg, "Survey of Virtual Machine Research June 1974," IEEE Computer Magazine, pp. 34-45, 1974.

[15] B. Dolan-Gabitt, B. D. Payne, and W. Lee, "Leveraging Forensic Tools for Virtual Machine Introspection. Technical Report. Georgia Institute of Technology, GT-CS-11-05 (www.bryanpayne.org/research/papers/GT-CS-11-05.pdf)," 2011.

[16] M. Hind and J. Vitek, "Hyperspector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection," presented at the First ACM/USENIX International Conference on Virtual Execution Environments (VEE '05), 2005.

[17] Hypertection. Hypervisor-Based Antivirus. Hypertection Team. Web. www.hypertection.com [accessed 13 Sept. 2011].

[18] Y. Sheng, Z. Shijie, L. Leyuan, Y. Rui, and L. Jiaqing, "Malware variants identification based on byte frequency," in Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on, 2010, pp. 32-35.

[19] A. Shamir and N. V. Someren, "Playing Hide and Seek With Stored Keys," presented at the Third International Conference on Financial Cryptography, 1999.