Service Oriented Computing and Applications (2019) 13:17–29
https://doi.org/10.1007/s11761-018-0252-2

SPECIAL ISSUE PAPER

Detecting indicators of deception in emulated monitoring systems

Kon Papazis[1] · Naveen Chilamkurti[1]

Received: 23 August 2018 / Revised: 24 November 2018 / Accepted: 20 December 2018 / Published online: 2 January 2019

Abstract
There has been a proliferation of cyber attacks in the form of malware manifestations, Botnet attacks and intruder access to unauthorized systems due to a larger attack surface available to threat actors. Security researchers leverage computer systems to monitor and analyze security threats in order to secure their data. Some of the security tools employed by security analysts are Honeypots, virtual machines, sandboxes and debuggers referred to as emulated monitoring systems (EMS). However, threat actors are working hard at reducing the efficacy of EMS by exploiting the inherent limitations of these security tools. They have employed various detection techniques to reveal EMS artifacts referred to as indicators of deception. In this paper, we investigate the level of EMS evasive measures and provide a taxonomy on the indictors of deception in EMS to gain an insight into the broad range of detection vectors available to threat actors. This would enhance EMS as a formidable weapon in the continuing struggle against threat actors, resulting in an improved detection of advanced malware samples and higher detection of intrusions.

Keywords Debugger · EMS · Indicators of deception · Honeypot · Sandbox and virtual machine

1 Introduction

Computer security systems are constantly under pressure to defend and protect the computer network from threat actors and malware proliferation. At the network level, they have at their disposal, security tools such as firewalls and IDS to assist in their endeavor to safeguard their data. At the host level, analytical techniques that perform static analysis and dynamic analysis are employed to determine the characteristics and behavior of malware [1]. These analysis techniques leverage Honeypots, virtual machines, sandboxes and debuggers that are collectively referred to as emulated monitoring systems (EMS).

A Honeypot main function is to deploy it in the computer network to act as a decoy to divert attacks away from valuable production systems. It deceives the intruder into attacking the vulnerable Honeypot target in order to gather methods and tools utilized by the intruder [2]. On the other hand, virtual machines, debuggers and sandboxes serve the purpose to monitor and analyze suspicious files. To circumvent these security tools, threat actors are developing new detection techniques to further expose indicators of deception in EMS in order to avoid analysis of their tools and techniques [3–5]. They are leveraging underlying EMS artifacts to expose the analysis tools and applying evasive techniques to bypass analysis. They are leveraging underlying EMS artifacts to expose the analysis tools and applying evasive techniques to bypass analysis. This is evident by the sharp rise of malware evasive techniques employed by malware authors in a 2017 Report on exploit kits by Minerva that found over 60% of exploit kits employed some sort of evasive technique [6]. This presents a challenge to security researchers to mitigate the detection surface of EMS and introduce additional obfuscation vectors to camouflage them.

✉ Naveen Chilamkurti
n.chilamkurti@latrobe.edu.au

Kon Papazis
k.papazis@latrobe.edu.au

1 La Trobe University, Victoria 3086, Australia

1.1 Motivation

EMS is a key component in the defense of computer systems and one of the fundamental properties that ensure its effectiveness is its deceptive ability to avoid detection from threat actors. The motivation of this study is to explore the magnitude of evasive techniques across all security tools applied to EMS by performing a comprehensive review of the literature on this topic. This includes the range of countermeasures that might render the detection techniques ineffective. A systematic taxonomy is required to address the depth of evasive techniques that exist in order to reduce the detection surface of indicators of deception exhibited by EMS, opening up the way for security researchers to increase the effectiveness of the EMS tools to combat the increasing cyber threat in the community.

A few recent papers have surveyed specific EMS tools that describe techniques to detect Honeypots and introspection tools [7] and sandbox environments [8]. However, these papers are limited in scope and do not provide the depth and diversity in relation to detection of indicators of deception in EMS tools. They do not take into account the countermeasures available to the security community to hinder the detection techniques used by threat actors. The focus of this paper is to explore, in depth, the range of detection techniques available in exposing EMS security tools and related countermeasures. Our contributions to this paper are the following:

- We present an in-depth taxonomy of various detection techniques available to threat actors across all security tools applied to emulated monitoring systems. Our study is broader in scope and detailed than other literatures.
- We provide a classification for detection techniques that identify indicators of deception in EMS in the context of network- and host-based detection techniques.
- Encompass preventative measures to conceal indicators of deception in EMS.

The remainder of this paper is organized into the following sections: Sect. 2 focuses on Honeypot Technologies. Section 3 discusses the concept of virtual machines and virtual environments. Section 4 gives an insight into debuggers. Section 5 describes the array of IoD in Honeypots. Section 6 presents IoD vectors in virtual environments. Section 7 describes detecting IoD in debuggers. Section 8 provides the type of countermeasures to evade detection techniques in EMS. Section 9 describes related work in this topic. Section 10 presents the conclusion.

2 Honeypot Technologies

Honeypots were introduced as an offensive mechanism to learn of new techniques, tools and motives perpetuated by the intruders [2]. It is a security resource with the main purpose being to be explored, attacked and be compromised. It is an expansion from work done on the deception toolkit [9]. A Honeypot diverts attacker attention away from production systems by manifesting itself as a decoy exhibiting various vulnerabilities. This security flaw in the fake system allows an intruder to gain unauthorized access to the system where the malicious activity can be monitored and logged, which allows capturing new hacker tools, learning new malware threats, building profiles of attacker methods and identifying new system vulnerabilities. Honeypots have been adapted to function in specific security domains such as employing client Honeypots to search for potential malicious servers [10] or acting as a vulnerable server to gather malware samples.

2.1 Types of Honeypots

Honeypots are categorized based on several factors such as the environment they operate and are identified as either production Honeypots or research Honeypots [11]. Level of interactivity between the intruder and the Honeypot service is an attribute that characterizes Honeypots into three main classes: low-interaction Honeypots, medium-interaction Honeypots and high-interaction Honeypots. Higher level of interaction with the intruder results in additional risk placed on the Honeypot. There is further distinction of Honeypot Technologies based on their activity level and functionality. The classic server Honeypot model [12] inherits vulnerabilities and then passively waits for attacks, while the client Honeypots model [13] searches the Internet to interact with servers in order to identify any benign malicious servers and separate them from benign servers.

2.1.1 Production Honeypots

Real production systems implement this type of Honeypot to augment traditional security services. They have the ability to delay and stop possible attacks by diverting or frustrating the intruder away from valuable production systems.

2.1.2 Research Honeypots

The purpose of these Honeypots is to find information and expose the methods, motives and tools used by the intruders. They are an invaluable tool to researchers in order to study advanced threats and capture unknown tools and techniques from the intruders.

2.1.3 Low-interaction Honeypots (LI Honeypots)

LI Honeypots provides emulated services of a real OS running on a host with a limited capacity to capture important intruder data. Emulating services has the benefit of mitigating the risk of exposing the rest of the production system to further attacks. However, their limitation is that they are not suitable for interactive attacks as they only emulate real services.

2.1.4 Medium-interaction Honeypots (MI Honeypots)

MI Honeypots are more advanced and allow a greater degree of interaction with the intruder. The introduction of added complexity allows capturing additional advanced attacks. They provide the benefits of HI Systems without the complexity and security concerns. However, their deception is trivially exposed by the fingerprinting technique.

2.1.5 High-interaction Honeypots (HI Honeypots)

HI Honeypots allow the highest level of interaction as they employ a fully functional system and not an emulator. This allows a greater spectrum of intruder data to be captured and makes it more difficult to expose its deception. However, they are more complex to setup and pose an increasing risk to the production network.

2.1.6 Server Honeypot

Server Honeypots acts as a commodity server that exhibits vulnerable services in order to attract unauthorized use of network services, acting as a form of early warning to the rest of the production system. It is reactive in nature as it does not actively seek malicious actors but responds to actors that interact with it.

2.1.7 Client Honeypot

Client Honeypots search the Internet for malicious servers and interact with the suspect server to elicit a response. This helps determine whether the visited server performed an attack such as a drive-by download on the client via the web browser. Some of the malicious activity monitored includes modification to the file system and processes invocation or windows registry modification on Windows-based systems.

3 Virtual environments

Virtualization is a technique of abstracting physical resources into a logical representation that increases utilization of computer system resources [14]. This abstraction allows emulation of hardware resources that are applied to a virtual environment that is comprised of a virtual machine monitor (VMM) aka hypervisor and a virtual machine. A VMM is a system software that forms a level of indirection by decoupling the software from the underlying hardware. It monitors and controls virtual environments by intercepting system calls from the guest OS and interacting with the instruction set architecture on behalf of the guest OS. A virtual machine (VM) is defined according to Goldberg [15] as a software duplicate of a real computer system in which a principal subset of the virtual processor's instructions execute on the host processor in native mode.

3.1 Server-level virtualization

3.1.1 Full virtualization

This technology provides complete emulation from the underlying hardware by using binary translation that converts blocks of instructions from the guest into native instruction on the host machine. This allows the VMM to run in ring 0 (privileged level), while ring deprivileging is performed on the guest OS by moving it to a lower privilege level of ring 1. The benefit is that it allows the guest OS to run in the virtual environment without any modification; however, it performs slower due to the use of binary translation.

3.1.2 Para virtualization

This technology modifies of certain parts of the kernel occurs by replacing non-virtualized operations such as memory management and interrupt handling in the kernel with hypercalls [16]. The privilege level of the guest OS is set to ring 1 to prevent guest OS direct access to privileged machine instructions [17]. The benefit lowers overhead due to reducing unnecessary traps to the VMM; however, it does not work on unmodified guest OSs like Windows.

3.1.3 Hardware-assisted virtualization

Intel VT and AMD-v virtualization extensions were introduced to support x86 architecture. The extensions leverage virtualization features inherent in next-generation CPUs to reduce the overhead inherent in trap and emulate software virtualization techniques [18]. These enhancements allow the VMM to operate at a higher privilege level ring 1, known as root mode, while guest OS runs at privilege level 0. This allowed virtualizing the guest OS without modifying the guest kernel.

3.2 Application-/OS-level virtualization

3.2.1 Container virtualization

Container virtualization is a compact OS that shares the kernel with the host OS and avoids the need to emulate instructions [19]. This allows it to run multiple processes in an isolated environment at the OS level, reducing resource consumption in CPU, memory and disk. It provides savings in resource consumption without the overhead, allowing more virtualized instances to be activated. However, they are limited in running the same OS as the host.

3.2.2 Sandbox virtualization

A sandbox creates a virtual space for an application to run without affecting other applications or the OS. Any changes made by the application are discarded on termination of the application to prevent contamination of program behavior on other system resources. This allows applications such as web browsers to run in an isolated virtual environment to prevent unwanted file and registry changes. Another approach is to run a sandbox in a VM that is configured to scan and analyze potentially suspicious files for malicious behavior. The benefit is the ability to detect stealth and zero-day attacks without prior knowledge.

4 Debuggers

A debugger is a monitoring tool that allows a program to be inspected during execution by interacting with the program flow to find the anomaly. Software developers employ debuggers to locate bugs that cause the program to malfunction by tracing through the execution of the program. Debugging tools are leveraged to find possible vulnerabilities or maliciousness in program code. There are two types of debugging modes that are available to the analyst. Source-level debugging is performed on the programs source code to allow setting of breakpoints, while assembly-level debugging is performed on the assembly code when source code is unavailable [20].

A debugger is able to run in kernel mode or user mode. Kernel mode debugging allows the debugger to access kernel mode space and core kernel components such as kernel drivers. To enable kernel debugging requires that one system is to be designated the host, while the other system is designated as the target. The host performs the debugging, while the target executes the code. In user mode debugging, one system is required to run the debugger, while the code is executed within the system. This allows the debugger to access user mode space to investigate user applications such as suspicious files. The benefit of using debuggers is that the analyst is able to set breakpoints to check the functionality of certain parts of the program. The debugger provides a means to change the flow of the program to find alternative avenues of execution, which is an important aspect in malware analysis.

5 Detecting indicators of deception in Honeypots

Honeypots have been operating in the IT Security landscape as a part of the perimeter defense available to an organization. Threat actors have developed techniques to detect their presence such as lack of system artifacts or behavioral traits in the Honeypot environment. We will introduce a taxonomy of Honeypot detection vectors available to threat actors to better understand the scope of exposed indicators of deception in Honeypots.

5.1 Honeypot network indicators of deception

5.1.1 Network anomalies

One way to identify deceptive Honeypots is by exploiting the virtual network layer implemented in Honeypots using tools such as NMAP port scanner. Sysman et al. [21] leveraged NMAP tool to fingerprint Dionaea Honeypot by constructing the following command: *$ nmap –sS –sV 192.168.0.1* to elicit response that identified the Honeypot. Valli [22] leveraged NMAP by probing the target using several scan modes and aggregating the response value to determine authenticity of target. A high value would validate the fake signature which exposes the Honeypot.

Link latency measurements developed by Fu et al. [23, 24] expose virtual Honeypots by the distinct temporal signatures that they exhibit compared to real machines. Virtual Honeypots emulate temporal behavior with a lower fidelity than real machines. Neyman–Pearson theory was leveraged to build a classification model from preprocessed link training data to derive link latency and identify target as a Honeypot with a high detection rate. Honeyd is exposed to link latency due to possessing a low default timer interrupt in kernel parameter HZ, which translates to a link latency of 1 ms. The link latency is derived by measuring the RTT of the packet sent and comparing it with the Honeyd profile to identify the virtual link. A timing analysis technique using ICMP [25] was leveraged to expose Honeypots by sending ICMP echo requests and extracting the responses into 49 features, which are fed into an SVM, resulting in high detection accuracy of target node. One limitation with this technique is identifying the correct threshold value to separate the two groups. Defibaugh-Chevaz exploits ICMP [26] by sending a stream of echo requests to suspect node and measuring the cumulative average response from the timing information in data link headers. A clustering algorithm is applied to the cumulative average against a threshold value to identify the Honeypot. The limitation is that the threshold value requires manual configuration based on network characteristics.

5.2 Honeypot characteristic indicators of deception

5.2.1 Service exercising anomalies

Service exercising exploits incomplete functionality in Honeypot services [25, 26] by exercising a service such a HTTP on native machine and comparing the feature set to the target. Any features that are not available on the target will lead to classification as suspect Honeypot. Service exercising is leveraged to find fake Honeypot proxies [27] by testing valid connection to open proxy servers using fake mail server. The target server is classified as a Honeypot if a connection is granted to the fake mail server. Botmasters leverage service exercising to expose Honeypots that act as infected hosts due to the liability constraints [28, 29] that prevent Honeypots from attacking other computers without legal ramifications. Commands are issued to the infected host to attack the remote sensor controlled by the Botmaster. The disguised sensor checks any invalid commands from the infected host or identifies the infected host as a Honeypot. This technique is able to expose hijacked bot controllers by issuing attack commands to several bots via the bot controller, if the sensor has not received the commands than the bot controller has been compromised. However, this technique is ineffective if the bot controller is monitored by security analysts.

5.2.2 Honeypot server anomalies

High-interaction Honeypot servers such as Sebek are not exempt from being detected and disabled [30]. Some of the artifacts that expose Sebek are accessing specific data structures in memory, checking proximity of Sebek read function, looking for the magic number, source port and destination port variables as well as finding location of the hidden Sebek kernel module. Disabling the kernel module is performed by replacing the modified system call table with the original, which will disable any logging to the Sebek Server. Fourier transform [31] is leveraged by measuring distribution of execution times in *read*() system call to exhibit discrepancies between Sebek and a real system. Certain types of Honeypots [21] exhibit indicators of deception that identify them. Artillery Honeypot returns random data to client upon connection and proceeds to disconnect and block the client IP. BearTrap displays a default banner that gives its presence away and returns 530 to system commands. Nova exposes an open port and connection to an inactive NetBIOS service. Artifacts specific to Dionaea are certificate issuer displaying "*dionaea.carnivore.it*", same user login to ftp with two different passwords and certain commands not implemented. Glastopf displays a random generated template webpage to identify it. KFSensor https port is opened, but the service is not implemented and has a limit of forty concurrent connections before it performs blocking techniques. Inherent limitations of Honeyd [21] are that it has a distinct response to a specifically crafted GET request, FTP service does not support the delete command, while the SSH server is non-functional. Kippo MI SSH Honeypot [21] does not implement the full range of commands compared to a vanilla SSH server and it performs an unusual key exchange by sending a larger message size to the client.

5.2.3 Honeypot client anomalies

Kapravelos et al. [32] perform detection of Honeyclients by checking for specific artifacts such as file system binaries, related kernel modules and running Honeyclient processes. A malicious Web site might defer attack if referrer header of HTTP request does not match expected header or it waits for events such as time-out or user interaction. Reflective DLL injection allows malicious Web site to load remote library into memory of client without being exposed in process list. Browser cache poisoning attack modifies client JavaScript files in IE browser cache. A client browser that visits webpage that is cached locally will be redirected to a malicious Web site from the poisoned JavaScript file. Mitigation of JavaScript poisoning is possible by deleting browser cached content or prevent existing cache files being modified (Table 1).

6 Detecting indicators of deception in virtual environments

Security analysts perform dynamic malware analysis and monitor intruder artifacts leveraging virtual environments and sandboxes. The virtual machine monitor does not provide a totally transparent virtual environment and this allows threat actors to exploit this loophole and evade detection. According to Popek and Goldberg [35], a VMM should possess the equivalence property that program behavior in VMM exhibits near identical in native machine. The reality is that the equivalence property does not hold for the current crop of VMMs. This knowledge empowers threat actors to search for artifacts to expose the virtual environment and modify or halt their attack. Following are some evasive detection techniques that threat actors such as malware writers and intruders utilize to detect the target system is a virtual environment.

6.1 Virtual network indicators of deception

6.1.1 Network anomalies

VirtualBox and VMware type two hypervisors exhibit irregular behaviors that identify them using a passive remote detection technique [36]. There are anomalies in the control flags of TCP/IP packet headers when running in a NAT network that causes the guest OS TTL value to be set to the host OS. A VM running a different guest OS than the host is exposed from the values in the IP ID field. Hypervisor introspection [37] leverages timing side-channel attacks to infer information about a VMM. This information leakage from

Table 1 List of detection vectors in Honeypot environments

Detection vector	Target	Description	Ref.
Fingerprinting Honeypots	Several Honeypots	Disclose connection characteristics of Artillery and KFSensor exposing non-function port service	[21]
NMAP	TCP/IP	Craft several NMAP scan modes and aggregate response from target; a high value indicates Honeypot	[22]
Link latency	Network	Measure link latency of network and compare with baseline real network using Neyman–Pearson theory	[23]
Link latency	Network	Exploit Honeyd timing anomalies and measure RTT to build profile using *t* test to identify virtual link	[24]
Service exercising	Services	Compare functionality in full feature service on real host and limited feature service on Honeypot	[25]
Timing analysis	Network	Send stream of ICMP echo requests to suspect host and measuring delay between request and response	[26]
Honeypot Hunter	Honeypot	Test whether the target is a proxy Honeypot by checking connection to fake mail server	[27]
Liability constraints	Honeypot	Use remote sensor to communicate and test validity of infected victim based on Honeypot liability constraints	[28, 29]
Kebes	Sebek	Introduce Kebes tool to detect and circumvent Sebek Honeypot by traversing memory to find kernel module	[30]
Environment artifacts	UML, VMware	Use system tools and search certain system locations for the presence of virtual Honeypots in UML and VMware	[31]
Honeyclient detection	Honeypot	Search for artifacts such as file system binaries specific to Honeyclient on target host	[32]
Environment artifacts	UML, VMware	Identify UML and VMware characteristics by querying CPUID and searching virtual artifact	[33]
IP packet fragmentation	TCP/IP	Craft fragmented packet to illicit an incorrect response from Honeyd that would be dropped by real host	[34]

VMM to guest VM is most vulnerable when a VMM performs a monitoring activity of the guest VM during suspend or paused state. A suite of browser-based red pills were constructed [38] as JavaScript code snippets that are run within a web browser that leverages the JavaScript Framework to expose timing variations between virtual environment and a native environment. Procedures are performed on the target to produce the average timing ratio by comparing differential execution time to a baseline value. The virtual environment is identified if there is a strong statistical confidence in the distribution of the average timing ratio of target.

6.2 Virtual machine indicators of deception

6.2.1 Environmental anomalies

Host-related virtual environment indicators of deception [39] include OS artifacts in system processes, memory, virtual hardware and specific processor instructions and capabilities. VM tools might insert identification markers in running processes, services, files, directories and registry keys. The IDT is located in a different address space in memory within a VM. There are VM-specific virtualized hardware artifacts such as NIC cards, USB controllers, video and audio adapters to identify a virtual environment. Miramirkhani et al. [40] introduce a probing tool to detect virtual artifacts by exploiting wear and tear artifacts that are inherent in normal systems and absent from virtual environments. The artifacts are categorized into system artifacts such as number of running processes, disk such as temporary files, network such as cached certificate revocation list, registry artifacts such as size of entry and browser artifacts such as saved bookmarks. A statistical model was developed using SVM classifier to identify artifacts belonging to a virtual environment. However, this tool is only implemented in a Windows OS environment.

Barham et al. [41] applied direct kernel structure manipulation (DKSM) to attack and subvert VMM introspection tools by manipulating specific fields of the kernel data structures on the target such as direct mode, shadow mode and return mode. A direct attack mode is manifested on the target VMM by applying semantic manipulation to directly modify kernel code. The ability to manipulate the syntactic and semantic representation of the kernel data structure would cause inconsistencies between internal and external views in the context of the VMM. This would lead to misleading and unreliable results presented to the introspection tool. However, this technique requires root access and hijacking of kernel code to directly manipulate kernel data structures.

Some of the artifacts that can expose VMware [33] are the type of virtual devices installed such as video card, NIC, hard disks, CD drive and SCSI controller. Further evidence exposing VMware is by analyzing the network card. There are various indicators of deception exhibited by VMware-based Honeypots [31]. Some artifacts that identify VMware virtual environment are the default hardware signatures associated with devices such as video card, network card, IDE and SCSI controller. A similar technique identifies VMware by producing an illegal instruction to generate an exception and the increased time to resolve the exception by VMware as opposed to a real system would give away its presence.

6.2.2 Host timing anomalies

Three timing-based mechanisms [42] were introduced to extract anomalies resident in HVM systems without using escalated privileges. RDTSC instruction measured elapsed CPU cycles between code execution by reading time stamp counter pre- and post-code execution. One method exploits VM exit overhead by executing CPUID guest privileged instruction and comparing it to a baseline threshold value to detect virtual environment. The second method exploits TLB cache evictions that are caused by VM exits to measure the difference in access times and compare the minimal access time ratio against a threshold value. The final method leverages parallel computing to measure the duration of the CPUID instruction without excessive use of the CPUID instruction that might provoke suspicion. The techniques do suffer from obtaining a stable threshold value that is constrained by specific hardware and virtualized environments. Local in-VM timing measurements [37] technique measured process scheduling and I/O read operations by hooking into the kernel function entry point and measuring time between events. A VM is identified if the duration between the current time and the last read or schedule event time stamp was higher than a specified threshold.

VM signature technique leverages $\times 86$ data structures such as local descriptor table [43] to detect VMs. The values of the LDT are readable from an unprivileged process such as a VM, which could be used to infer a virtual environment if the values differ from a native host. However, it is observed that running this technique on fully emulated systems produces inaccurate results in identifying the target system as a VM. Franklin et al. [44] perform fuzzy benchmarking on remote host to determine whether it exhibits VM characteristics. Fuzzy benchmark tool executes on target host and connects to a verifier that checks for timing dissimilarities. Sufficient overhead is generated on target host by executing several sensitive instructions that cause expensive TLB flushes. The verifier checks that execution time is greater than a predefined threshold value to identify target as VM. A caveat is that it is dependent on an accurate threshold value that is obtained using empirical testing. It also requires root access on target guest OS to disable interrupts in order to acquire accurate timing measurements.

6.2.3 CPU emulator anomalies

An automated technique for generating a suite of red pills is realized [45] to detect the presence of CPU emulators such as QEMU and BOCHS. The same sequence of instructions is generated to induce faults that result in behavioral differences in each execution environment. Observing the posterior CPU state provides a way to identify the emulated environment. However, this technique requires extensive testing to determine the set of instructions that are viable as candidate red pills. CPU characteristics that detect emulators such as QEMU' are explored [46] by exploiting design anomalies specific to Intel IA-32 architecture. Executing an instruction to induce a CPU bug would identify an emulated environment as QEMU by the type of exception returned. A similar technique is applied in accessing model-specific registers that do not signal an exception in QEMU, while setting certain flags in EFLAGS register would fail to raise exception in QEMU.

6.2.4 Security appliance anomalies

User mode Linux [33] is a security appliance that is identifiable due to characteristics such as the use of virtual devices displayed in*/dev/ubd/* directory as well as querying CPUID information related to ×86 CPU located in*/proc* tree. You can trivially fingerprint chroot by running *ls –lia* in the root directory of Linux and observing that the inode values are unrelated to a non-chroot environment. Exposure to fingerprinting UML is trivially demonstrated by Holz et al. [31] in running a simple Linux command *dmesg* to display UML artifacts. There are other locations in */etc/fstab* and */dev/ubd* directory that would betray the presence of UML. Linux LXC Containers possess certain characteristics that identify their presence [47] such as leveraging namespaces by returning the number of processes from PS command and comparing it to the sysinfo output value. A value returned by PS that is smaller than sysinfo implies hidden processes, thus exposing the Container. Another method is leveraging user permissions by running commands as root user that are denied within a Container environment; however, this method requires root privileged access.

6.3 Sandbox indicators of deception

SandPrint is a tool [48] that exposes online sandboxes using unsupervised learning by finding inherent patterns in their reports common to sandboxes. A distance function was leveraged to differentiate two reports, while agglomerative single link clustering assisted with comparing and grouping similar reports. Some of the features identified included OS installation date, product ID, file access history and empty clipboard. Ferrand [49] exposed Cuckoo sandbox by running code to identify hooking function used by Cuckoo that checks the first opcode instruction. Looking for Cuckoo-specific folders such as configuration files and startup scripts would expose Cuckoo to the attacker. Detecting the communication pipe between Cuckoo and the host system will give away its presence. Chailytko [50] leverages the stack to provide a means to identify Cuckoo hooked functions by populating the functions reserved space to elicit a crash. Malware could leverage sleep function to evade Cuckoo by inducing malware to sleep for an infinite time and sleeping for a certain duration outside the analysis time limit (Tables 2, 3).

6.4 Malware evasive techniques

Malware authors have continued to expose indicators of deception in EMS to evade analysis. They have armed malware with anti-emulation, anti-virtualization, anti-debugger techniques, packers and encryptors to circumvent analysis of the behavior of the malware sample. Some malware families such as Zeus and Spyrat have employed techniques to hide their nefarious behaviors from EMS [51]. We characterize the pervasiveness of malware evasion methods that are available to malware authors and the EMS target vector that could assist malware samples to evade analysis (Table 4).

7 Detecting indicators of deception in debuggers

Debuggers Holz et al. [31] explored detecting debugger security appliance by running tools such as *ptrace*() system call for Linux and running *IsDebuggerPresent*() in Windows environment. Calling *ptrace* with specific parameters would determine whether a process is being debugged by *ptrace* and thus result in failure to run another instance of *ptrace*(), revealing its presence. Further detection of debug tools is evident by accessing EFLAGS register and checking whether TRAP bit is set to one via the *pushf* instruction. Chen et al. [52] leveraged specific Windows API function calls such as *IsDebuggerPresent*() to detect the presence of debugger. Timing behavior of a debugger is leveraged to identify its presence by comparing distinct timing differences between an execution in debugger environment and native execution.

There are several ways to detect the presence of a debugger like accessing the Process Control Block in Windows OS to check whether the BeingDebugged field in PEB is set. Holz et al. [31] explored detecting debugger security appliance by running tools such as *ptrace*() system call for Linux and running *IsDebuggerPresent*() in Windows environment. Calling ptrace with specific parameters would determine whether a process is being debugged by ptrace and thus result in failure to run another instance of *ptrace*(), revealing its presence. Further detection of debug tools is evident by accessing EFLAGS register and checking whether TRAP bit is set to one via the pushf instruction. Chen et al. [52] leveraged specific Windows API function calls such as *IsDebuggerPresent*() to detect the presence of debugger. Timing behavior of a debugger is leveraged to identify its presence by comparing distinct timing differences between an execution in debugger environment and native execution.

8 Countermeasures to evasion in EMS

The security community has continuously pursued ways to reduce indicators of deception in emulated monitoring systems. A proven methodology is to examine various detection techniques and tools utilized by threat actors to patch EMS, improving the efficacy of these monitoring tools. The security community needs to seek alternative approaches to hide the indicators of deception in emulated monitoring systems from threat actors.

8.1 Mitigating techniques

An approach to mitigate indicators of deception is by configuring production systems to imitate the behavior of EMS such as debuggers and virtual machines [52]. This would disguise the production systems by fooling threat actors into

Table 2 List of detection vectors in virtual environments

Detection vector	System	Description	Ref
TCP/IP anomalies	Network	TCP/IP anomalies exist in VirtualBox and VMware NAT environment specific to IP ID value	[36]
Local timing delays	Host	Local in-VM timing measurements identify VM suspend state due to long pause from measuring current time and last time stamp	[37]
Browser red pill	Host	Runs within a web browser that leverages JavaScript framework to expose timing variations between VM and native host	[38]
IDT location	Host	IDT memory address location of VM stored in upper memory	[40]
DKSM	Host	Subvert VMM introspection tools by manipulating certain kernel data structures to provide unreliable results to introspection tools	[41]
Timing mechanism	Host	Leverage rdtsc and cpuid instructions timing anomalies in execution access times and TLB eviction process to detect VE	[42]
LDT readable values	Host	Access and read LDT values in VM that differ from that in native	[43]
Fuzzy benchmarking	Network	Generate sufficient overhead to perform timing measurements on remote host and comparing it to a predefined value to identify VM	[44]
Red pills	Host	Red pills are generated to induce faults in CPU emulators that help to distinguish VM from normal machine	[45]
CPU characteristics	Host	Leverage CPU bugs such as using MSR to generate exception in native, while QEMU emulator does not raise an exception	[46]
Cuckoo sandbox detection	Host	Detection methods include searching for Cuckoo-specific folders and detecting communication pipe in Cuckoo guest system	[49]

believing that it is a monitoring system. Imitating EMS is possible by modifying driver information in Windows registry, simulating VM hooks by intercepting system calls to specific Windows API functions. Modify behavior of Windows API debug checking functions to imitate the presence of WinDbg debugger. Present an empty windows application that inherits an analysis application such as OllyDbg to fool the malware sample that it is inside an analysis environment.

Threat modeling is applied to several Honeypot security appliances such as Honeyd, Dionaea, Kippo and Glastopf [53] to reduce their exposure from successful fingerprinting results. Fingerprinting mitigation techniques involve modifying the time stamp of Honeyd to resemble a real IIS web server, and modify Honeyd HTTP replies to match those of real Apache web server. Dionaea Honeypot is enhanced by configuring FTP banner to mimic banner of real FTP server, allow modification of configuration file to display realistic workgroup names via the Samba service and modify the MSSQL service script. Kippo ping function succumbs to fingerprinting attack that is rectified by generating correct replies outside of network group. Glastopf LFI vulnerability is resolved by changing folder access response to permission denied error. These enhancements would reduce rate of successfully identifying specific Honeypots.

Some techniques to counter evasive malware tactics [8] are to modify sleep duration to wake up malware before time-out of analysis in sandbox. Increasing malware analysis period would reduce the false negatives by catching malicious activity outside of the default analysis period. Improving simulated mouse actions to resemble that of human-based mouse actions would also reduce the evasive rate of malware. Traditional method to perform dynamic analysis is by installing monitoring agents within the EMS environment. Virtual machine introspection is a stealthier monitoring vector to mitigate the artifacts exposed by traditional monitoring agents. VMI observes the hardware state and events in the VMM layer and extracts information from these variables to find the software state [54]. The VMM leverages VMI by

Table 3 List of fingerprinting tools to detect virtual environments

Fingerprint tool	OS	Description	Privilege
devmgmt.msc	Windows	Applet that displays hardware components attached to the computer system	User
dmesg	Linux	Displays ring buffer messages regarding kernel operation	User
dmidecode	Linux windows	Displays DMI table contents such as system hardware description	Superuser
ethtool	Linux	Display network device attributes	User
facter	Linux	Command line tool to gather system information	Superuser
hostnamectl	Linux	Query the system hostname and related settings	User
hwinfo	Linux	Generate overview of hardware on system	Superuser
imvirt	Linux	Detects whether it is running in virtualized environment by looking at boot message and reading DMI	Superuser
inxi	Linux	Displays system hardware such as CPU and drivers	Superuser
lscpu	Linux	Displays information on CPU architecture	User
lshw	Linux	Displays hardware information such as product information	User
lsmod	Linux	Displays list of modules loaded into the kernel	User
lspci	Linux	Utility to display devices connected to PCI bus	User
msinfo32.exe	Windows	Displays information about the OS and hardware characteristics such as motherboard, graphics adapter	User
pafish	Windows	Performs anti-debugger anti-VM and anti-malware detection	User
/proc	Linux	Displays runtime system information such as kernel, hardware configuration and mounted devices	Superuser
systemctl	Linux	Displays list of current running units	User
systemd-detect-virt	Linux	Detects and identifies virtualization technology and distinguish full virtualization from containerization	User
virt-what	Linux	Displays if run in VM derived from a set of heuristics	Superuser

observing the CPU and memory state of the VM to monitor sensitive areas for suspicious activity and respond by performing actions such as stopping unauthorized services from running or suspending and shutting down the VM state.

9 Related work

Uitto et al. [7] conducted a brief survey of anti-honeypot detection and anti-introspection methods employed by malware. Network, application and service fingerprinting-based attacks were prevalent in LI and MI Honeypots, while HI Honeypots were susceptible to system fingerprinting. They expanded on detection vectors introduced by Chen et al. [52] to include a two tier system composed of top tier categories such as temporal, operational, hardware and environment. However, this study is limited in scope and only focuses on a subset of detection techniques applicable to Honeypots and virtual environments, not providing an extensive taxonomy of the breadth and depth of detection techniques available to threat actors.

VMM transparency problem [55] shows the disparities between virtual machines and native hardware. Virtualization-induced anomalies are leveraged to identify discrepancies inherent in virtualized hardware. The anomalies care categorized into logical discrepancies that attribute to semantic differences in the interface between the opposing logical architectures, and resource discrepancies focus on irregularities in physical resources such as CPU cycles and timing discrepancies that manifest themselves when performing two operations and measuring the variance in latency between the operations. This work confines the detection surface to host-based semantic anomalies related to VMM.

A taxonomy of anti-virtualization and anti-debugging techniques [52] is presented and enumerated based on system abstraction in hardware, environment, application and behavior groups. VM devices contain identifiable attributes such as network device prefix, VM hooks in execution environment, specific tools located in known system locations that are identifiable through enumeration of registry, file system and running processes. Remote fingerprinting of a VM

Table 4 List of malware evasive methods

Evasive tactic	Target
Checking initial state and values of CPU Registers to verify type of environment running	CPU
Checking predicted value of certain registers such as EAX on some OS	CPU
Value of ECX register determines whether the value is non-conventional	CPU
Checking the range of the stack against running processes to identify emulation	Memory
Checking predicted memory regions of common DLL's by accessing the EDX register	Memory
Performing junk API calls with prohibited parameters to subvert some emulators	Kernel
Spyrat is able to check value of EBX register of 564d5868 h, which translates to ASCII code "VMXh" to identify VMware using port 5658 h for communication	CPU
Checking VirtualBox environment for specific processes such as VboxServices.exe	Environment
Checking for online malware analysis sandboxes by reading registry key with product ID	CPU
Checking whether debug library dbghelp.dll in Windows environment is loaded into memory	Kernel
Malware performs stalling technique by consuming CPU cycles disguised as benign tasks to cause time-out due to limited analysis time	CPU
Malware performs environment check for artifacts such as registry keys OS version, apps and files	Environment
Malware employs sleep technique that activates sleep calls or triggers to be activated at later time	Environment
msinfo32.exe	Environment
Interaction techniques where malware lies dormant until human interaction is initiated	Environment
Human scrolling to a specified place in a file	Environment
Wait until a required number clicks have occurred to activate	Environment
Mouse speed looks for abnormal mouse movement such as faster scroll speeds	Environment
Lack of user interaction such as mouse movement or scrolling	Environment

is achieved by running a remote timing test that exploits clock skew behavior. A random indicator derived from mean squared error is compared with a baseline value that helps to identify a VM due to exhibiting a larger MSE value than a native machine. This taxonomy restricts its detection surface to host system VM and debugger detection techniques.

A collection of malware evasive measures and counter measures was outlined to detect the presence of sandbox [8]. Some of the measures related to configuration settings include sleep tactics to run malicious task after time-out of automated analysis has occurred. Apply fast flux to conceal malware and phishing sites using DGA. Evasive methods based on human interaction include lack of consistent mouse movement, lack of scrolling, speed of mouse movements and scrolling an RTF document to match that of a real user. Environmental evasive techniques include searching for sandbox artifacts in registry and check number of CPU cores available, device information, MAC address vendor and file system information. The main focus of this work is on evasive techniques employed by malware related to sandbox environments.

A variety of attack vectors are presented to detect certain VM emulators [56]. CPUID instruction is leveraged to trigger hypervisor interception that causes TLB to be flushed providing a means to identify access time belonging to hypervisor. VM emulators relocate certain data structures in memory that allows them to be detected by checking value of LDT in Windows. VirtualPC is detected by running undefined or illegal opcode instructions that fails to raise an exception. Several methods to detect BOCHS are available such as using INVD and WBINVD instructions to cause TLB to be flushed by entering page mode. Executing CPUID instruction to check for AMD processor ID will return incorrect value in BOCHS. QEMU is detectable by using CPUID instruction to retrieve incorrect processor name. The CMPXCHG8B instruction is leveraged by executing instruction on read-only page that fails to generate a page fault in QEMU. This work only focuses on certain types of VM emulators.

10 Conclusion

This paper provides an exhaustive study on the techniques and tools available to threat actors to identify indicators of deception that are inherent in emulated monitoring systems. It gives an insight into the plethora of choices available to threat actors to augment evasive maneuvers if suspected of inhabiting an emulated monitoring system. We show that these security tools inherit design features that expose them by leveraging anomalies found in network, environment, hardware, service and application behaviors. An interesting observation from this taxonomy is the small amount of research that has been invested on countermeasures available to security researchers to mitigate detection evasion and hardening of the computer systems. The rich set of emulated monitoring system evasive techniques and trivial amount of countermeasures presented in this taxonomy shows how large the gap is to harden these security tools in order to maintain their effectiveness against threat actors. To mitigate and reduce the scope of these evasive techniques requires a partnership of collaboration within the security community.

Further research is required by the security community to harden and obfuscate EMS to more closely mimic real systems. Thus, reducing the scope of these evasive techniques requires new research of EMS obfuscation and a partnership of collaboration within the security community. Future work is to develop a framework of stealth strategies to reduce the indicators of deception currently present in various emulated monitoring system tools, without significant performance degradation.

Acknowledgements The research work reported here was made possible by the Defence Science Institute Grant G22015SChilamkurtiLaT023, an initiative of the State Government of Victoria.

References

1. Gandotra E (2014) Malware analysis and classification: a survey. J Inf Secur 5:56–64
2. Spitzner L (2002) Honeypots: tracking hackers. Addison-Wesley Longman Publishing Co. Inc., Boston
3. Omella AA (2006) Methods for virtual machine detection. http://www.s21sec.com/descargas/vmware-eng.pdf. Accessed May 2017
4. Marpaung JA, Sain M, Lee H-J (2012) Survey on malware evasion techniques: State of the art and challenges. In: 2012 14th international conference on advanced communication technology (ICACT). IEEE
5. Kolbitsch C, Kirda E, Kruegel C (2011) The power of procrastination: detection and mitigation of execution-stalling malicious code. In Proceedings of the 18th ACM conference on computer and communications security. ACM
6. Minerva-labs (2018) *minerva labs research report: 2017 year in review*. https://l.minerva-labs.com/hubfs/Minerva%202017%20Yearly%20Report_FINAL.pdf. Accessed 25 Nov 2018
7. Uitto J, et al. (2017) A survey on anti-honeypot and anti-introspection methods. In: World conference on information systems and technologies. Springer
8. Keragala D (2016) Detecting malware and sandbox evasion techniques. SANS Institute InfoSec reading room. https://www.sans.org/reading-room/whitepapers/forensics/detecting-malware-sandbox-evasion-techniques-36667. Accessed Dec 2018
9. Cohen F (1998) The deception toolkit. http://all.net/dtk.html. Accessed May 2017
10. Symantec (2008) A guide to different kinds of honeypots. https://www.symantec.com/connect/articles/guide-different-kinds-honeypots. Accessed May 2017
11. Akkaya D,. Thalgott F (2010). Honeypots in network security. http://www.divaportal.org/smash/get/diva2:327476/fulltext01. Accessed May 2017
12. Gorzelak K, et al. (2011) Proactive detection of network security incidents. In: Belasovs A (ed) ENISA report. http://www.enisa.europa.eu. Accessed June 2017
13. Riden J (2008) Server honeypots vs client honeypots. https://www.honeynet.org/node/158. Accessed June 2017
14. Campbell S, Jeronimo M (2006) An introduction to virtualization. Published in “Applied Virtualization”, Intel, pp 1–15
15. Goldberg RP (1972) Architectural principles for virtual computer systems. Ph.D thesis, Harvard University
16. Marshall D (2007) Understanding full virtualization, paravirtualization, and hardware assist. VMWare White Paper. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf. Accessed June 2017
17. Barham P, et al. (2003) Xen and the art of virtualization. In: ACM SIGOPS operating systems review. ACM
18. Rodríguez-Haro F et al (2012) A summary of virtualization techniques. Proc Technol 3:267–272
19. Morabito R, Kjällman J, Komu M (2015) Hypervisors vs. lightweight virtualization: a performance comparison. In 2015 IEEE international conference on cloud engineering (IC2E). IEEE
20. Sikorski M, Honig A (2012) Practical malware analysis: the hands-on guide to dissecting malicious software. No starch press, San Francisco
21. Sysman D, Evron G, Sher I (2015) Breaking honeypots for fun and profit. Talk at Blackhat, vol 8
22. Valli C (2003) Honeyd-A OS fingerprinting artifice. In: Proceedings of Australian computer, network and information forensics conference
23. Fu X, et al. (2006) On recognizing virtual honeypots and countermeasures. In: 2nd IEEE international symposium on dependable, autonomic and secure computing. IEEE
24. Fu X et al (2005) Camouflaging virtual honeypots. Texas A&M University, College Station
25. Mukkamala S, et al. (2007) Detection of virtual environments and low interaction honeypots. In: Information assurance and security workshop, 2007. IAW’07. IEEE SMC. IEEE
26. Defibaugh-Chavez P, et al. (2006) Network based detection of virtual environments and low interaction honeypots. In Proceedings of the 2006 IEEE SMC, workshop on information assurance
27. Krawetz N (2004) Anti-honeypot technology. IEEE Secur Priv 2(1):76–79
28. Zou CC, Cunningham R (2006) Honeypot-aware advanced botnet construction and maintenance. In: International conference on dependable systems and networks, 2006. DSN 2006. IEEE
29. Wang P et al (2010) Honeypot detection in advanced botnet attacks. Int J Inf Comput Secur 4(1):30–51
30. Dornseif M, Holz T, Klein CN (2004) Nosebreak-attacking honeynets. In Information assurance workshop, 2004. Proceedings from the fifth annual IEEE SMC. IEEE
31. Holz T, Raynal F (2005) Detecting honeypots and other suspicious environments. In Information assurance workshop, 2005. IAW’05. Proceedings from the sixth annual IEEE SMC. IEEE
32. Kapravelos A, et al. (2011) Escape from monkey island: evading high-interaction honeyclients. Detection of intrusions and malware, and vulnerability assessment, pp 124–143
33. Innes S, Valli C (2006) Honeypots: how do you know when you are inside one? In: Australian digital forensics conference
34. Oberheide J, Karir M (2006) Honeyd detection via packet fragmentation. Ann Arbor 1001:48104
35. Popek GJ, Goldberg RP (1974) Formal requirements for virtualizable third generation architectures. Commun ACM 17(7):412–421
36. Jämthagen C, Hell M, Smeets B (2011) A technique for remote detection of certain virtual machine monitors. In International conference on trusted systems. Springer
37. Wang G, et al. (2015) Hypervisor Introspection: a technique for evading passive virtual machine monitoring. In WOOT
38. Ho G, et al. (2014) Tick tock: building browser red pills from timing side channels. In Proceedings of the USENIX workshop on offensive technologies
39. Liston T, Skoudis E (2006) On the cutting edge: Thwarting virtual machine detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf. Accessed July 2017
40. Miramirkhani N, et al. (2017) Spotless sandboxes: evading malware analysis systems using wear-and-tear artifacts. In IEEE symposium on security and privacy
41. Bahram S, et al. (2010) Dksm: subverting virtual machine introspection for fun and profit. In: 2010 29th IEEE symposium on reliable distributed systems. IEEE
42. Brengel M, Backes M, Rossow C (2016) Detecting hardware-assisted virtualization. In: Detection of intrusions and malware, and vulnerability assessment. Springer, pp 207–227
43. Quist D, Smith V, Computing O (2006) Detecting the presence of virtual machines using the local data table. Offensive

Computing 2006. http://www.offensivecomputing.net/files/active/0/vm.pdf. Accessed July 2017
44. Franklin J et al (2008) Remote detection of virtual machine monitors with fuzzy benchmarking. ACM SIGOPS Oper Syst Rev 42(3):83–92
45. Paleari R, et al (2009). A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In: USENIX workshop on offensive technologies (WOOT)
46. Raffetseder T, Kruegel C, Kirda E (2007) Detecting system emulators. In: International conference on information security. Springer
47. Kedrowitsch A, et al. (2017) A first look: using linux containers for deceptive honeypots. In: Proceedings of the 2017 workshop on automated decision making for active cyber defense. ACM
48. Yokoyama A, et al. (2016) SandPrint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In International symposium on research in attacks, intrusions, and defenses. Springer
49. Ferrand O (2015) How to detect the cuckoo sandbox and to strengthen it? J Comput Virol Hack Tech 11(1):51–58
50. Alexander Chailytko SS (2016) Defeating sandbox evasion: how to increase the successful emulation rate in your virtual environment. https://blog.checkpoint.com/wp-content/uploads/2016/10/DefeatingSandBoxEvasion-VB2016_CheckPoint.pdf. Accessed Sep 2018
51. Issa A (2012) Anti-virtual machines and emulations. J Comput Virol 8(4):141–149
52. Chen X, et al. (2008) Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: IEEE international conference on dependable systems and networks with fTCS and DCC, 2008. DSN 2008. IEEE
53. Dahbul R, Lim C, Purnama J (2017) Enhancing honeypot deception capability through network service fingerprinting. In: Journal of physics: conference series. IOP Publishing
54. Garfinkel T, Rosenblum M (2003) A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of network and distributed systems security symposium
55. Garfinkel T, et al. (2007) Compatibility is not transparency: VMM detection myths and realities. In: Proceedings of the 11th workshop on hot topics in operating systems (HotOS-XI)
56. Ferrie P (2017) Attacks on more virtual machine emulators. Symantec technology exchange 2007. http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf. Accessed Dec 2017