A Scalable High Fidelity Decoy Framework against Sophisticated Cyber Attacks

Jianhua Sun
College of William and Mary
jsun01@email.wm.edu

Songsong Liu
George Mason University
sliu23@gmu.edu

Kun Sun
George Mason University
ksun3@gmu.edu

ABSTRACT

Recent years have witnessed a surging trend of leveraging *deception* technique to detect and defeat sophisticated cyber attacks such as the advanced persistent threat. Deception typically employs a decoy network to entrap the attackers and divert the firepower away from the real protected assets. Unfortunately, existing decoy systems failed to achieve a balanced tradeoff between the decoy fidelity and scalability, which potentially undermines the effectiveness of attacker deception. In this paper, we propose a hybrid decoy architecture that separates lightweight front-end decoys from high-fidelity back-end decoy servers. To enhance the deception effectiveness, we introduce dynamics into the decoy system design to make the decoy a *moving target*, where the front-end decoys constrain attackers by transparently intercepting and forwarding the malicious commands to the heterogeneous back-end decoys for real execution. We implement two prototypes of the hybrid decoy architecture based on Linux Bash shell and Windows PowerShell. The experimental results demonstrate that our system can effectively misdirect and disinform attackers with small network and system overhead.

CCS CONCEPTS

• **Security and privacy** → *Intrusion detection systems*; Malware and its mitigation.

KEYWORDS

cyber deception; decoy network; moving target defense

ACM Reference Format:
Jianhua Sun, Songsong Liu, and Kun Sun. 2019. A Scalable High Fidelity Decoy Framework against Sophisticated Cyber Attacks. In *6th ACM Workshop on Moving Target Defense (MTD'19), November 11, 2019, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3338468.3356826

1 INTRODUCTION

Recent years have witnessed the increase of ever-sophisticated cyber threats against business and government organizations. In advanced persistent threat (APT), well-resourced attackers can bypass existing preventive security measures (e.g., intrusion detection systems and firewalls) through exploiting zero-day vulnerabilities or well-planned social engineering campaigns [37]. Consequently, there has been a surging trend of leveraging *deception* technique to detect and defeat such advanced cyber threats. By meticulously deploying a myriad of *decoys* (or *traps*) across the target network, deception can effectively misdirect the attackers to camouflage real protected servers [7, 13, 32]. One advantage of deception compared to conventional IDS systems [38] is it can accurately expose unknown stealthy attacks with nearly zero false positive rate, considering that legitimate users usually do not have the intention to access those decoys. Moreover, decoys can significantly accelerate attack information gathering [9, 10, 22] with a dedicated trap environment, which often disinforms attackers with falsified data such as fake document, password, and encryption key [11, 35, 39].

Fundamentally, a decoy network may effectively trap attackers into believing that they have succeeded in penetrating the real system, while they actually only penetrate one or more of the decoy mirage nodes. Afterwards, strategic disinformation can be served to the trapped attackers and make them believe what the defenders want them to believe. Furthermore, even if the real system has been compromised by the attacker among a number of decoy systems, the decoy systems can confuse the attacker with multiple fake but believable choices, thus rendering it difficult for attackers to make correct decisions. However, the attackers may exploit various decoy evasion techniques to distinguish decoys from the real systems. In general, the decoy evasion techniques can be classified into two major categories: *pre-exploitation techniques* and *post-exploitation techniques.* In the pre-exploitation phase, before compromising the targeted system with exploits, the attackers mainly rely on network reconnaissance techniques, which conduct either fingerprint-based or timing-based network traffic analysis [18, 21], to remotely identify decoy systems. The pre-exploitation techniques are effective at identifying low or medium interaction decoys, but they have difficulties in identifying high interaction decoys that run the same software stacks as the real system. As a defender, we are facing one challenge to provide a number of high interaction decoys distributed in different subnets with limited system resources and strict access control constraints.

In the post-exploitation phase, after breaking into the targeted system and gaining system resource access privileges (e.g., user privilege or root privilege), the attackers can further identify decoys by using decoy evasion techniques, which are generally based on two critical design and implementation gaps between decoys and real systems. First, there is no believable user activities and user interactions on the decoys. Therefore, the attackers can easily detect decoys by observing system artifacts such as file systems, running processes, and OS patches [5, 16]. Second, there is no believable network activities on the decoys. By eavesdropping and inspecting network traffics from the compromised computer, the attackers may figure out that there is no real user triggered or system triggered network packets on the decoys [25, 31]. In this work, we propose to design and develop a scalable high fidelity decoy framework that integrates believable user activities and network activities to defeat the decoy evasion techniques in both the pre-exploitation and post-exploitation phases.

To achieve both high fidelity and good scalability, we propose a hybrid decoy architecture that separates lightweight front-end decoy proxies from high-fidelity back-end decoy servers. It is based on one key observation that no matter what evasion approaches the attackers may utilize to identify a decoy, when we have full control of the malicious processes on the front-end decoy, we are able to feedback believable fake information prepared by the back-end decoy server to the attackers. Since the size of front-end decoy proxies are small, we can afford to create hundreds (or even thousands) of lightweight proxies. Further, due to the small size of decoy proxies, it is easy to integrate them into existing networking systems. The back-end decoys could be dynamically created or added when multiple front-end decoys are under attacks around the same time. Specifically, the front-end decoy will intercept the attacker's malicious commands and forward them to the back-end decoy for real execution. Then, the front-end decoy is responsible for preparing the responses based on the command execution results received from the back-end decoy. For instance, if the attacker has created a reverse shell using tools such as Metasploit's Meterpreter and runs the *ps -ef* command, instead of running it directly on the front end decoy, we intercept this command and run it on the back-end decoy and then inject the results back to the malicious process.

To enhance the deception effectiveness, we introduce dynamics into the decoy system design to make the decoy a *moving target* from the attacker's perspective. Specifically, the front-end decoys are uniform and lightweight; while the back-end decoys are versatile and believable with various practical OSes, applications and decoy data. Whenever a front-end decoy node is accessed or even compromised, we can then dynamically tunnel the remote reconnaissance traffic and the intercepted attack C&C commands to different back-end decoys. Furthermore, we can even dynamically shift the network attack surface by exposing different OS and service vulnerabilities through the back-end decoys. Therefore, the attackers are forced to perceive different views of the decoy network at different times, such as the network addresses, system configurations and service versions. In this way, our decoy system significantly enlarges the network attack surface and increases the difficulty of attacker reconnaissance.

To demonstrate the feasibility and effectiveness of our decoy system in attacker deception, we implement two prototypes of our hybrid decoy architecture based on Linux Bash shell and Windows PowerShell. The Linux prototype leverages state-of-the-art OS virtualization technique to deploy OS containers as decoy sandboxes; while the Windows prototype uses virtual machines as decoys. Both prototypes support transparent tunnelling and offloading of attacker commands. The experimental results show that our system is effective in attacker deception through attacker misdirection and disinformation. Our system is also scalable in that it can consolidate a large number of decoy nodes with limited system resources. Moreover, the performance overhead introduced by the implemented attack C&C interception/redirection scheme is fairly small.

In summary, we make the following contributions:

- We propose a scalable decoy system for constructing high-fidelity decoy networks to mitigate remote malicious reconnaissance in the pre-exploitation phase and insider threats in the post-exploitation phase.
- Our design is featured with a hybrid architecture that separates lightweight front-end decoy proxies from high-fidelity back-end decoy servers such that a large number of believable decoys can be consolidated using constrained system resources.
- We implement two prototypes of our hybrid decoy architecture based on Linux Bash shell and Windows PowerShell. The experimental results demonstrate that the network and system overhead is small and our system can effectively misdirect and disinform attackers.

The remainder of this paper is organized as follows. Section 2 presents the threat model and assumptions. Section 3 describes the system design in detail. The prototype implementations are discussed in Section 4 and evaluated in Section 5. Discussion and related work are presented in Section 6 and 7, respectively. Section 8 concludes the paper.

2 THREAT MODEL

Nowadays all so-called "high fidelity" decoys and honeypot systems can be easily identified by attackers using the increasing number of decoy evasion techniques [5, 16, 25, 31]. Attackers are well motivated to detect decoys in both the pre-exploitation phase and the post-exploitation phase. First, in the pre-exploitation phase, to protect their valuable zero-day exploits from being analyzed, the attackers are inclined to commit attacks after confirming that the target system is not a decoy. At this phase, the attackers mainly rely on network reconnaissance to recognize a remote decoy system. Second, after breaking into the target system in the post-exploitation phase, attackers still want to check if they are trapped into a high fidelity decoy in order to protect their attacking strategies, malicious software toolkit, attacker identity, and other compromised servers. At this phase, the attacker may utilize various decoy evasion techniques via either downloading and running a decoy detector tool or running shell commands through an interactive command shell.

3 SYSTEM DESIGN

3.1 Overview

As shown in Figure 1, we develop a scalable high fidelity decoy architecture to generate a large number of high interaction decoys with limited computer resources. It consists of a large number of small-sized front-end decoys in different subnetworks and a small number of high interaction back-end decoys. The front-end decoys are uniform and lightweight; while the back-end decoys are versatile and believable with various practical OSes, applications and decoy data. When one front-end decoy receives a network service request from an adversary, it forwards the request to the back-end decoy, which will generate believable responses and forward them

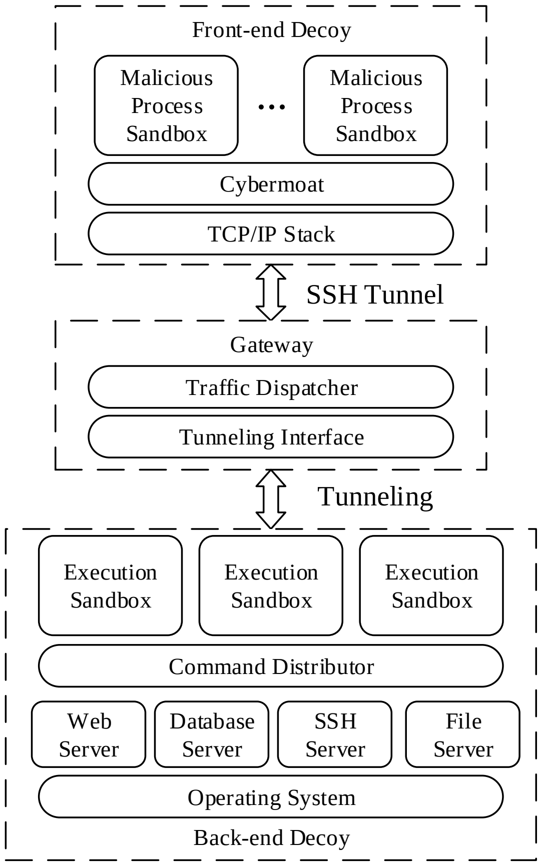

Figure 1: Hybrid Decoy Architecture

to the front-end decoy. This hybrid decoy architecture supports the separation of the front-end decoy and the back-end decoy in different networks to enable better scalability. We call the network hosting the front-end decoy as front-end network and the network hosting the back-end decoy as back-end network.

Specifically, the front-end decoy intercepts all malicious commands initiated from the attacker and then forwards them to be executed in an execution sandbox on the back-end decoy. The believable command results are generated on the back-end decoy. The main responsibility of the Cybermoat component is to package the command results and send them back to the attackers. Since the Cybermoat executes outside the sandbox, even if the attackers may obtain the root privileges in the sandbox, the Cybermoat can still control all views of the attackers about the system. The front-end decoy can forward the commands via an SSH tunnel to the gateway, which can further tunnel the commands to one back-end decoy. Since it does not need to maintain the network state information on the back-end decoy, we can run the attacker's commands in any available execution sandbox, which is controlled by the command distributor. By dynamically tunnelling the remote reconnaissance traffic and the intercepted attacker C&C commands to different back-end decoys, the decoy network is turned into a *moving target* from the attacker's perspective. Furthermore, we can even dynamically shift the network attack surface by exposing different OS and service vulnerabilities through the back-end decoys. Therefore, the attackers are forced to perceive different views of the decoy network at different times, such as the network addresses, system configurations and service versions. In this way, our decoy system significantly increases the difficulty of attack reconnaissance.

3.2 Front-end Decoy

The frond-end decoy is able to run a malicious shell in one sandbox, and it can intercept all commands initiated by the attackers. Then, it forwards those commands to the back-end decoy for real execution. After receiving the command execution results from the back-end decoy, it will generate believable network response packets that combine the packet payload received from the back-end decoy and the packet header of its own.

In the front-end decoy, the Cybermoat component is responsible for three main tasks. Firstly, it will create one sandbox to intercept all malicious commands initiated from the attackers. Secondly, it will forward the commands to a gateway via an SSH tunneling. The gateway will further forward the commands to the back-end decoy for real execution. Thirdly, after receiving the command execution results, the Cybermoat will add the packet header and send the entire network packets to the attacker.

The most critical task is to intercept the malicious commands and inject the responses received from the back-end decoy. In the post-exploitation phase, attackers may establish a command and control (C&C) channel to communicate with the compromised system. For example, the controller of a botnet can direct the malicious activities (e.g., sending spam email) on compromised computers through C&C channels formed by standards-based network protocols such ssh (22), http/https (80, 443), ftp (21), database (e.g., MySQL 3306), or a customized protocol. Since all the network activities are under full control and the malicious network traffic can be easily identified, defenders may detect and identify any specific command and control channel established by the adversary. After identifying such interactive C&C channel, we intercept the commands received from the adversary, send the commands to be executed on the back-end decoy, and then feedback the execution results to the adversary via the C&C channel. Since there are so many ways an adversary can establish command and control with various levels of covertness [1], we focus on the most popular C&C protocols.

In targeted attacks, the attacker usually forks a terminal with root privilege to facilitate its attacks. Moreover, the attacker may use the terminal to detect if it compromises a real system or just a decoy by looking into a number of system artifacts, including file systems, system process, and network interface. We focus on developing a mirage around a malicious shell to fool the attackers. There are two popular types of shells: bind shell and reverse shell. A bind shell is the kind that opens up a new service on the compromised machine and requires the attacker to connect to it in order to get a session. A reverse shell requires the attacker to first set up a listener on its machine, and when the compromised machine acts as a client connecting to that listener, the attacker will receive the shell. The reverse shell is needed when the compromised machine is behind a different private network or the compromised machine's firewall blocks incoming connection attempts to the bind shell. There are more than 160 different reverse shells in the Metasploit Framework [3], and we choose to work on two reverse shells, namely, *windows/meterpreter/reverse* (the most commonly used reverse shell in Windows) and *linux/x86/meterpreter/reverse_tcp* (the most commonly used reverse shell in Linux).

3.3 Back-end Decoy

Back-end decoys accept packets coming from the gateway and behave as if they are hosts in the front-end network. To reflect the real-system characteristics as closely as possible, we propose to construct back-end decoy by (1) creating usage models to emulate the legitimate operations or (2) following the behaviors of legitimate machines in near-real-time. To achieve resource efficiency, we could generate the back-end decoy servers on demand, along with a pool of pre-reserved servers. Note that how to generate believable fake data for replacing the sensitive data is out the scope of this work.

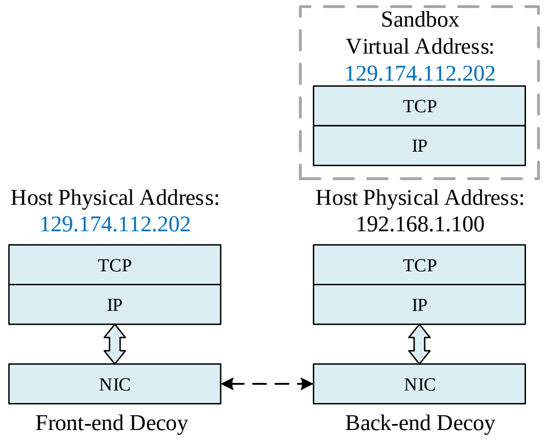

Figure 2: Physical/Virtual IP Mapping

Besides checking into a number of system artifacts, including file systems, system process, and network interface [31], attackers may utilize the hybrid decoy architecture to identify decoys. Particularly, attackers may inject network level information (e.g., IP address) into the application payload on the back-end decoy and compare it with the network packets received from the front-end decoy. Therefore, we must ensure the consistency between the two network stacks on the front-end decoy and the back-end decoy. As Figure 2 shows, we map the virtual IP address of the sandbox running on the back-end decoy to the physical IP address of the front-end decoy. In this example, the physical IP addresses of the front-end decoy and back-end decoy are 129.174.112.202 and 192.168.1.100, respectively. In the sandbox running on the back-end decoy, we set its virtual IP address the same as the front-end decoy's physical IP address, namely, 129.174.112.202. Since the sandbox has its own TCP/IP stack that is different from the back-end decoy's TCP/IP stack, we could maintain the consistency of other network state information.

3.4 Gateway

The gateway receives encapsulated packets from the redirectors on the front-end decoys in different networks, decapsulates the packets, and dispatches them to the intended back-end decoys. The gateway is similar to a transparent firewall on dispatching incoming packets from the redirectors to their respective back-end decoys based on the destination field in the packet header. In the reverse direction, the gateway accepts response traffic from the back-end decoys and resend all packets to the front-end decoys. If necessary, the gateway may curtail the interaction with the attackers to prevent a compromised back-end decoy from attacking other hosts on the Internet. If the continued interaction is allowed, the gateway will forward the packets back to their original front-end decoys which will then redirect the packets into the network, such that the packets appear to the remote attackers as originating from the front-end network.

The gateway can be implemented using a virtual machine that creates another point for packet logging, inspection, and filtering. We could adopt HonSSH [2] to create two separate SSH connections between the front-end decoy and the back-end decoy. We may also capture all connection attempts on gateway in order to detect and analyze colluding attacks that target at attacking multiple decoys simultaneously. For instance, a summary of attacker sessions can be captured in a text file.

4 PROTOTYPE IMPLEMENTATION

We implement two prototypes of our hybrid decoy architecture based on Linux Bash shell and Windows PowerShell. Both prototypes support transparent tunnelling and offloading of attacker commands.

4.1 Linux Bash Shell Prototype

We implement a virtualization-based Linux Bash shell prototype. Figure 3 shows the implementation architecture. The entire system is integrated on a single host machine running Ubuntu 16.04 with VirtualBox installed. The computer features an 12-core Intel(R) Xeon(R) E5-2620 CPU and 16 *GB* memory. Three virtual machines are created using VirtualBox, serving as attacker, a front-end decoy platform, and a back-end decoy server bed, respectively. Each VM is allocated one host CPU and 2 *GB* memory. The attacker VM is running Kali Linux and emulates remote attacker by using Metasploit to generate bind shell and reverse shell attack payload. The other VMs are installed Ubuntu 16.04. The front-end decoy VM uses LXC to create Linux containers as malicious process sandboxes; while the back-end decoy server VM generates Linux containers as execution sandboxes. Inside each malicious process sandbox, we deploy an instrumented Linux Bash binary, which transparently manipulates attacker commands whenever the attacker compromises the malicious process sandbox and boots up a bind or reverse shell. In contrast, the real attacker commands are redirected and executed in the sandboxes where a normal unmodified Bash is deployed. We can deploy both malicious process sandboxes and execution sandboxes as privileged or unprivileged containers. Using unprivileged containers provide better isolation since the attacker is given a "fake" root privilege with constrained capabilities. The Gateway is implemented using HonSSH to redirect attacker commands between the malicious process sandbox and execution sandbox through an ssh tunnel.

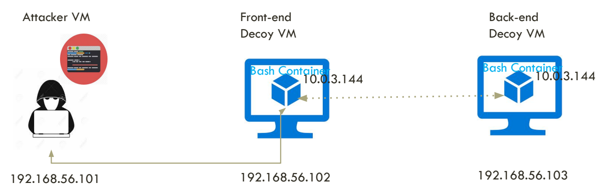

Figure 3: Linux Prototype Implementation Architecture

4.1.1 Transparent Command Interception and Redirection. To achieve transparent command interception and redirection, we develop an instrumented Linux Bash shell that loads two customized hook functions `preexec` and `precmd` and deploy it in each malicious process sandbox. The `preexec` function is executed after an interactive command has been entered and before being executed, with the interactive command as its argument. The `precmd` function is executed before each command prompt is printed.

Algorithm 1 Transparent Command Interception and Redirection

```
global $\_\_inside\_preexec \leftarrow 0$
global $\_\_inside\_precmd \leftarrow 0$
global $\_\_cmd\_redirected \leftarrow 0$
function PREEXEC( )
    if $\_\_inside\_preexec > 0$ then
        return
    end if
    $\_\_inside\_preexec \leftarrow 1$
    if ${BASH_COMMAND} == "precmd" then
        $\_\_cmd\_redirected \leftarrow 0$
    end if
    if $\_\_cmd\_redirected > 0$ then
        return 1        ▷ command already redirected, abort next command
    else
        return 0        ▷ user command not redirected yet
    end if
    $current\_cmd \leftarrow$ get_cmd_from_history()        ▷ extract the user command just entered
    tunnel_command($current\_cmd$)
    $\_\_cmd\_redirected \leftarrow 1$
    return 1
end function

function PRECMD( )
    if $\_\_inside\_precmd > 0$ then
        return
    end if
    $\_\_inside\_precmd \leftarrow 1$
    $\_\_cmd\_redirected \leftarrow 0$
end function
```

In particular, `preexec` is installed as part of the DEBUG trap using Bash `trap` builtin function. As shown in Algorithm 1, it extracts user commands entered interactively and transparently tunnels the commands to the back-end execution sandbox. To allow the invocation of `preexec` in subshells, the `functrace` option is enabled so that the DEBUG trap is inherited by shell functions executed in a subshell environment. To avoid local execution of user commands in the malicious process sandbox, the Bash `extdebug` option is set so that a non-zero return value of the `preexec` function sets a non-zero return value for the DEBUG trap, which causes the next simple command to be skipped. However, this strategy is ineffective for command pipeline and compound commands. For example, after user enters a compound command *pwd ; uname*, it would be redirected to the back-end execution sandbox. After the execution result is returned and printed, a manually-set non-zero return value of `preexec` causes the *pwd* command to be skipped in the local malicious process sandbox. However, the *uname* command still gets executed since Bash parses the compound command into separate simple commands and only the next simple command is skipped whenever the DEBUG trap returns. To resolve this problem, a global command redirection indicator is set when a compound command or pipeline starts being intercepted by `bash-preexec`, which indicates whether the compound command is in the process of redirection and all parsed simple commands should be skipped. After the back-end execution result is returned, the command redirection indicator is reset in the `precmd` hook function, which is pre-installed into the PROMPT_COMMAND environment variable.

4.1.2 Camouflaging Information Leakage Channel. Once a malicious process sandbox is compromised, the attacker will establish a control channel remotely by spinning up either a bind shell or reverse shell inside the sandbox. This allows the remote attacker to identify the IP address of the malicious process sandbox by inspecting the connection setup process. Unfortunately, the attacker can also examine the address configurations of the compromised sandbox using various commands to determine its real IP address. Since our hybrid decoy architecture intercepts and redirects these commands, the IP address of the back-end execution sandbox is revealed to the attacker. Therefore, to guarantee the fidelity of our decoy system, we need to camouflage such information leakage channel by maintaining identical address configurations between the front-end malicious process sandbox and the corresponding back-end execution sandbox.

Our solution is to deploy the sandboxes in an overlay network atop the decoy platform. Specifically, the container sandboxes in both front-end and decoy-end have identical IP addresses; while the underlying decoy VMs are located in a different underlay network. To maintain network connectivity between the front-end and back-end sandboxes, in the front-end decoy VM, we configure IP masquerading in the VM kernel such that the malicious process sandbox can access the back-end decoy VM. Similarly, in the back-end decoy VM, we enable port forwarding to the execution sandboxes. In this way, the malicious process sandbox can tunnel the intercepted commands to the execution sandbox by directly accessing the forwarded port in the back-end decoy VM.

4.2 Windows PowerShell Prototype

The Microsoft Windows PowerShell [4] is a task-based command-line shell and scripting language. It is pre-installed on all Windows system version starting from Windows 7 SP1 and Windows 2008 R2 in 2012. Including all the functions of command-line interpreter (cmd.exe), PowerShell also supports the Component Object Model (COM) and Windows Management Instrumentation (WMI), and is extendable to use .NET class. All these features make it play an important role in Windows system attacks [24].

Our PowerShell prototype is also implemented on the same host machine running Ubuntu 16.04 with VirtualBox installed. Compared to our Bash shell prototype implementation, we deploy the Windows 7 on both front-end and back-end decoy VMs. To present the difference in the deployment, the PowerShell prototype is running inside the Windows VM directly. In default, we provide the unprivileged PowerShell on the back-end decoy VM. The OpenSSH server 8.0 is running on the back-end decoy to receive the attacker's PowerShell commands. Other environment settings stay the same. We deploy the PowerShell 7 on the VM. To support the cross-platform feature, the newest Microsoft PowerShell project has been transplanted from old .NET Framework to .NET Core [4].

4.2.1 Command Interception and Redirection. To intercept and redirect the command of PowerShell, we modify the source code of PowerShell to hook the command execution function. The function `ExecuteCommand` is the core function for local command execution in the project Microsoft.PowerShell.ConsoleHost. All calls to execute a command line must be done with this function, which properly synchronizes access to the running pipeline between the main thread and the break handler thread. Every-time the console reads the commands, it will verify them and then pass the valid commands to this function. We substitute the local `ExecuteCommand` function with a internal SSH client, which is created from the third party .NET Core library Neon.SSH.NET 0.5.2. All the inputted commands in the front-end decoy VM would be sent to the back-end decoy VM via the SSH client. Then the SSH server on the back-end decoy VM would call its PowerShell to execute those commands and return the results to present on the front-end decoy VM.

4.2.2 System Consistency. To ensure the system consistency between front-end decoy and back-end decoy, two aspects must be taken into consideration: command typing feedback and collected information consistency. In our modified PowerShell, the internal SSH client would capture the whole inputted command line without any parsing in advance. It could prevent the information lost in the semantics level. Therefore, our modified PowerShell can execute command pipeline and compound commands correctly. The PowerShell console is able to help user complete the inputted commands and presents them in different colors based on the command types. We keeps the function `ReadLineWithTabCompletion` to ensure attackers can complete their input if they hit tab. The input command can still be color display. Therefore, the typing feedback on front-end decoy would be same with the original one. About the network configuration information, we continue to use the settings mentioned in Section 4.1.2 to ensure the consistency of IP address.

5 EVALUATION

We evaluate the deception effectiveness of our hybrid decoy system in attacker misdirection and disinformation and measure the performance of the implemented attack C&C interception/redirection scheme. For the Linux prototype, all experiments were performed on a desktop with Intel(R) Xeon(R) E5-2620 CPU and 16GB RAM running 64-bit Ubuntu 16.04. We use VirtualBox to create an attacker VM, a front-end decoy VM (i.e., victim VM) and a back-end decoy VM (i.e., decoy server VM), all running Ubuntu 16.04 with 1GB memory. We use LXC to deploy malicious process sandboxes and execution sandboxes in the victim VM and decoy server VM, respectively. Each malicious process sandbox is installed with the instrumented Bash shell; while the execution sandbox runs the unmodified Bash shell. To ensure its fidelity, we also deploy real services (e.g., ssh server, Apache HTTP server and MySQL database server) in the back-end execution sandboxes. To establish remote attack C&C channel, in the attacker VM we emulate attackers by using Metasploit to generate attack payload, which is manually injected into the malicious process sandbox to create bind shell and reverse shell.

5.1 Deception Effectiveness

To assess the effectiveness of our deception strategy, we maintain each malicious process sandbox minimal with only the instrumented Bash shell. In contrast, the back-end execution sandbox runs unmodified Bash as well as real services. After gaining access to the malicious process sandbox through the bind/reverse shell, the attacker can scrutinize the compromised decoy system. We tested various typical Bash and PowerShell commands. The execution results are listed in Table 1 and Table 2 based on whether the command is executed locally or redirected. Our decoy system can effectively deceive attackers into believing that they have compromised a valuable server. However, all the gathered information (e.g., system configuration, network addresses and running services) are provided purposely by the back-end execution sandbox.

5.2 Latency Overhead

Intercepting and redirecting attacker commands to the back-end execution sandbox introduces extra delay in command processing. To quantify the latency overhead, we measure the times of running remotely issued attacker commands in the malicious process sandbox and execution sandbox. The comparison of Bash shell prototype is illustrated in Figure 4. On average, our hybrid decoy design incurs additional $0.24 \sim 0.29s$ latency in command processing from the perspective of a remote attacker. We also use *sshping* to measure the interactive character echo latency through an SSH tunnel. We identify that the latency overhead is mainly attributed to the tunnel establishment process and the additional roundtrip between the front-end and back-end sandboxes. To set up the tunnel, it takes $\sim 0.1s$ to process the public key based authentication; while the average character echo latency is $\sim 0.5ms$.

In our PowerShell prototype, the average command execution overhead is $0.17s \sim 0.39s$, See Table 3. To set up the SSH tunnel between the front-end and back-end Windows VMs, it takes $\sim 2.2s$ to process the authentication, while the average character echo latency is $\sim 0.2ms$.

Table 1: Deception effectiveness as indicated by Bash shell command execution results.

Command	Description	Execution Results	
		Local in Malicious Process Sandbox	Redirected to Execution Sandbox
who	check users logged into the system	`decoy_client`	`decoy_server`
uname -o	print the operating system	`GNU/Linux`	`GNU/Linux`
pwd	print working directory	`/home/decoy_client`	`/home/decoy_server`
ps -ef	display active processes	`bash`	`bash, sshd, apache2, mysqld`
ip addr show eth0	display IP address	`10.0.3.144`	`10.0.3.144`

Table 2: Deception effectiveness as indicated by PowerShell command execution results.

Command	Description	Execution Results	
		Local in Malicious Process Sandbox	Redirected to Execution Sandbox
whoami	check users logged into the system	`decoy_client`	`decoy_server`
[Environment]::OSVersion	print the operating system	`Win32NT`	`Win32NT`
Get-Location	print working directory	`C:\Users\decoy_client`	`C:\Users\decoy_server`
Get-Process	display active processes	`powershell`	`powershell, sshd, apache2, mysqld`
ipconfig	display IP address	`10.0.3.144`	`10.0.3.144`

The timing difference may allow the attacker to identify the decoys from the real server when both a real server and a decoy are compromised. To mitigate such timing-based fingerprinting, we can deliberately introduce extra command processing latency in the compromised real server. For example, we can temporarily buffer malicious packets through network tarpitting.

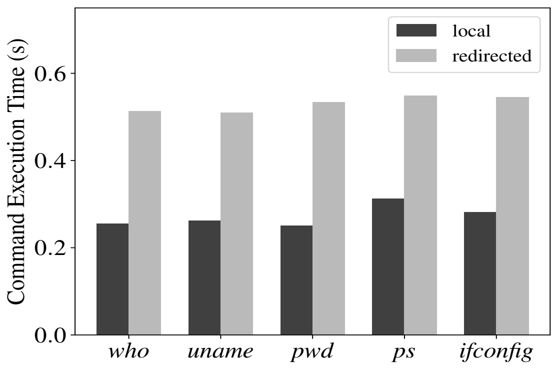

Figure 4: Latency Overhead of Command Redirection

Table 3: Latency Overhead of PowerShell Commands

Command	Increased Overhead (Second)
whoami	0.172
[Environment]::OSVersion	0.195
Get-Location	0.242
Get-Process	0.389
ipconfig	0.172

5.3 Decoy System Scalability

This section evaluates the scalability of our hybrid decoy system and demonstrates its practicality in real world deployment. We first evaluate its capability of massively consolidating a large number of decoys within constrained resource limit. To balance the tradeoff between decoy fidelity and scalability, we leverage lightweight Linux containers as front-end and back-end decoy sandboxes. In our experiment, we deploy LXC-based decoys that are installed with Ubuntu OS and various services (e.g., ssh, ftp and Apache http).

We first boot LXC-based decoys sequentially and measure the time it takes to launch each decoy. Figure 5 shows the times for continuously launching up to 200 decoy containers. On average it only takes 0.42 seconds to boot a decoy container even when a large number of decoys run simultaneously. Therefore, our decoy system design is resilient under the case of decoy crash, where we can quickly replace a compromised decoy with a pristine copy.

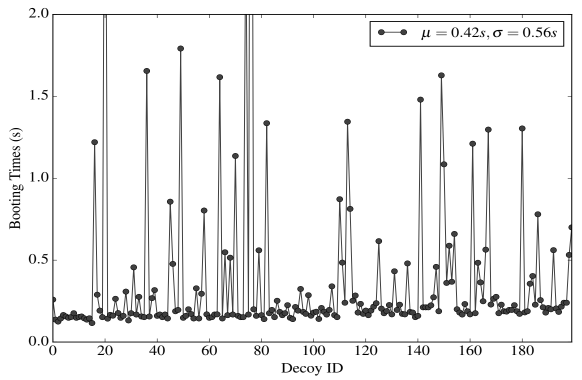

Figure 5: Decoy Sandbox Booting Times

We then evaluate container-based decoys by comparing its booting time, resource overhead and responsiveness with VM-based decoys. The responsiveness quantified by the round-trip time (RTT) of the ICMP echo request/reply. As summarized in Table 4, container based decoys incur substantially smaller resource consumption than VM based decoys (15% memory consumption and 19% CPU utilization), while achieving more efficient decoy booting and request processing. Our container-based decoy design can support dense aggregation of decoy nodes and in the meanwhile achieve the same level of deception fidelity (i.e., it can provide the same level of interactivity with remote scanners).

Table 4: System Overhead: Container Decoys versus VM Decoys.

Type	Boot Time	RES	VIRT	%CPU	%MEM	RTT
Container	1.67s	3.5M	15.8M	20.5	0.2	4ms
VM	6.52s	20.6M	86.4M	109	1.3	32ms

To ensure deception fidelity, the back-end execution sandboxes can be configured with real services. Therefore, we also measure the extra memory overhead introduced by various real servers and the result is listed in Table 5. It consumes several hundred of kilobytes of memory for for the application process. For more complicated industry-scale services such as Apache HTTP server and MySQL database server, the related overhead is moderately acceptable.

Table 5: Memory Overhead of Decoys with Real Services.

Service Type	Memory Consumption (KB)
TCP echo	144
Telnet	324
ssh	820
Apache HTTP	4925
Nginx	3726
MySql	38820

6 DISCUSSION AND FUTURE WORK

Network tarpitting against timing-based fingerprinting. The timing difference caused by the C&C interception/redirection may allow the attacker to identify a decoy from the real server when multiple real/decoy servers are compromised. To mitigate such timing-based fingerprinting, we can reshape the malicious traffic by introducing extra delays in the kernel network stack of the compromised real server. For example, we can use Linux Traffic Control (`tc`) [8] to add constant packet processing delay to a interface.

Stateful versus stateless command execution. Depending on the sophistication of the attacker, our decoy design can be either stateful or stateless. For a simple remote scanner that performs random reconnaissance, the probing request could be dynamically redirected to a back-end execution sandbox. However, for sophisticated APT attackers with the capability of correlating multiple probing sessions and analyzing the compromised decoy node (e.g., the OS version, disk and file system content, and network address configurations), the command execution should be stateful. That is, there should be a consistent mapping between a front-end process sandbox and a back-end execution sandbox during the APT attack cycle.

Web Shell. The web shell is a web-based script. To exploit web server existed target machine, attacker may upload the web shell which is programmed in any language supported by the target server. Unlike standard Bash Shell and PowerShell, the uploaded web shell is customized based on attacker's requirements and target server's vulnerabilities. Web shell may call the system shell to execute the command or call the syscall directly. Our solution can deal with the former situation, since commands are passed to our modified system shell. In the future work, we will handle the latter one. There are two solutions, namely, script-based solution and library-based solution. In the script-based solution, we need to identify the web shell on the target server and then substitute its syscall function to redirect command. Its difficulty is how to interpret the polymorphic encoded web shell effectively in the real time. In the library-based solution, we propose to modify the script language libraries supported by target server to intercept all possible syscall requests. Therefore, web shell has to call our modified library when it is running on the target server.

Generating believable user interactions and network activities. As future work, we will develop a high fidelity back-end decoy server with believable user interactions and network activities to defeat the armored decoy evasion techniques in the post exploitation phase. When the attackers break into the decoy system, we can fool the attackers to believe that they have gained the user or root privilege, but it is still under the full control of the defenders. We propose two mechanisms, namely, simulation-based solutions and record and replay (RnR) based solutions, to enhance the high-interaction back-end decoy server. First, since the effort for providing virtual indistinguishable properties for decoy nodes could be very expensive, we propose to simulate the typical user activities on the back-end decoy server using a usage model profiled from past logging information collected from real servers. Similarly, we will develop network simulators to simulate the network activities on the back-end decoy server. Second, we propose to develop real-time record and replay techniques to record the user activity and network traffic on the real server and then replay them on the back-end decoy server. It can be implemented at various levels including network level RnR, user interface level RnR, and instruction level RnR.

Scaling up to large number of decoys automatically. Our prototype implementation deploys the front-end and back-end sandboxes in two virtual overlay networks with identical IP and MAC address configurations; while the underlying decoy VMs are located in a different underlay network. To maintain network connectivity between the front-end and back-end sandboxes, we manually forward the port in the back-end decoy VM to the execution sandboxes such that the malicious process sandbox can tunnel the intercepted commands to the execution sandbox by directly accessing the forwarded port in the back-end decoy VM. As future work, we will develop and implement a protocol that allows the front-end and back-end sandboxes to negotiate about the port to be forwarded for a connection. This allows automatic configuration of the forwarding rules with the cost of a one-time round-trip delay.

7 RELATED WORK

Our system exploits knowledge of sandbox, honeypot and decoy to provide host deception with believable system environment. Our system targets at protecting the decoy system from being identified by attackers using various decoy identification and evasion attacks.

7.1 Honeypots and Deception

Honeypots are effective on capturing massive attacks such as worms, viruses, or botnets and gaining high visibility into attacker activities [10, 14, 19, 26, 27, 43]. Any interaction with a honeypot is likely to be malicious. For instance, Niels [34] developed a virtual Honeypot framework called "Honeyd" that can simulate the TCP/IP stack of the target operating system. However, since there is no strict isolation between those simulated network stacks, when one stack is compromised, all the other stacks may be compromised too. Michael et al. [40] developed a virtual HoneyFarm called Potemkin that can support hundreds of VMs on one computer, using a dynamical physical resource binding technique to create a lightweight VM for each active IP address. Similarly, there are parallel efforts of designing hybrid honeypot systems (i.e., *honeyfarms*) [22, 36]. These systems lack visibility into the attack command and control, while our system focuses on achieving fine-grained attacker deception through manipulating the attack control channel transparently. Empirical studies on detection and identification of decoy nodes focus on two methods: fingerprinting-based solution [15, 21] and timing-based solution [18]. First, adversaries can identify implementation differences and gaps between the protocols simulated by decoys and the real protocol embodied by the real system, such as the differences in IP fragmentation and implementation of TCP. Second, decoy nodes can also be identified by their longer response times, due to lack of system resources.

The major differences between our decoy system and honeypot systems are:

- Honeypots are typically used as data collection points and are designed to study the strategies carried out by attackers. When honeypots are used for malware analysis, malware developer may use various anti-honeypot techniques to identify honeypot. Our decoy system is one deception and misinformation system that focuses on preventing targeted attacks and misleading the attackers with untruth.
- A honeypot running in virtual machines should not be identified by an attacker; otherwise, the attacker will easily modify its behavior to evade detection. In our system, both the real system and decoys run in virtual machine environments. An attacker gains little information even if it detects that the real system is running in one of the virtual machines without knowing the exact virtual machine.

Besides decoy systems and services, cyber deception can also disinform attackers with falsified information, e.g., false passwords [23], OS fingerprint [6], fake document [35, 39]. All these designs are complementary to our work, which can be employed to create more believable sandbox environment.

7.2 Sandbox Detection and Evasion

The dynamic malware analysis techniques rely on a malware analysis sandbox that is implemented as instrumented execution environment to run program and decide if code is malicious via observing its activity. The sandbox is becoming popular recently since it can handle zero-day threats and partially automate tasks done by human analysts and reverse engineers. However, it is not easy to build an effective sandbox, since attackers work hard on developing various sandbox detection techniques that may detect runtime or analysis environments such as CPU features, OS artifacts (files, processes, etc.), and thus evade the detection [5, 12, 17, 33, 42]. Moreover, the armored malware can avoid being analyzed by conducting timing-based or user activity-based evasions such as mouse movements. For example, it can wait to unpack itself until someone does something such as opening a pdf file and scrolling to the second page. It can also simply time out the heavy instrumented analysis that may last only for a short time (e.g., a few minutes) before any interesting behaviors are revealed.

One difference between decoy evasion and sandbox evasion is that the malware running in the sandbox can detect the sandbox by analyzing the behavior on the malicious application itself; while in decoy detection, the attacker focuses on the user behavior on other system processes, since they need to keep their attacking processes stealthy. In other words, the key issue behind the major constraint of current sandboxes are their absence of perception into the execution of a malware program, which is not a concern in decoy evasion. Another difference is that the decoy system may face advanced persistent threat (APT) that last for a long time (e.g., multiple months/years), while malware sandbox only needs to run the targeted malicious process for a short time (e.g., several seconds/minutes).

7.3 Minimalistic OSes

Other candidate approaches for building up lightweight sandboxes include microkernels and unikernels. Microkernels or minimalistic OSes [20, 41] aim to provide just the required functionality for an application. In general, a microkernel executing at the privileged level exports the near- minimum set of mechanisms. Traditional OS functions, such as network protocol stacks, file systems and device drivers, are removed from the microkernel and are instead run in the user mode. However, most microkernels cannot run in virtualized environments. Another candidate for building the lightweight proxy platform incorporates unikernels [29], which are specialized, single address space LibOS-style virtual machines. They are built by compiling high-level languages directly into specialized images running on a hypervisor. Examples of unikernels include ClickOS [30] and Mirage [28]. The design options of unikernels reduce the amount of deployed code, thus reducing the attack surface. Moreover, the small footprint of unikernel renders it competitive for deploying decoys in high density.

8 CONCLUSION

We propose a hybrid decoy system for constructing high-fidelity decoy networks to defeat both remote malicious reconnaissance in the pre-exploitation phase and insider threats in the post-exploitation phase. By separating lightweight front-end decoy proxies from high-fidelity back-end decoy servers, our system is scalable to consolidate a large number of believable decoys with constrained system resources. Furthermore, we dynamically intercept and offload malicious attacker commands to heterogeneous decoy servers to enhance deception effectiveness. Implementations and evaluations of the Linux Bash shell and Windows PowerShell based hybrid decoy prototypes demonstrate that it can effectively misdirect and disinform attackers with acceptable network and system overhead.

9 ACKNOWLEDGMENT

This work is partially supported by ONR grants N00014-16-1-3214, N00014-16-1-3216, and N00014-18-2893.

REFERENCES

[1] 2019. Command and Control Tactic. https://attack.mitre.org/wiki/Command_and_Control.

[2] 2019. HonSSH. https://github.com/tnich/honssh.

[3] 2019. Metasploit Meterpreter. https://github.com/rapid7/metasploit-framework/wiki/Meterpreter.

[4] 2019. PowerShell. https://github.com/PowerShell/PowerShell.

[5] 2019. Sandbox Evasion. http://unprotect.tdgt.org/index.php/Sandbox_Evasion.

[6] M. Albanese, E. Battista, and S. Jajodia. 2015. A deception based approach for defeating OS and service fingerprinting. In *Communications and Network Security (CNS), 2015 IEEE Conference on.*

[7] Allure Security. 2016. Decoys and the Security of Deception. https://www.alluresecurity.com/blog/use-your-illusion-decoys-the-security-of-deception/.

[8] Werner Almesberger et al. 1999. Linux network traffic controlâĂŤimplementation overview.

[9] Hassan Artail, Haïdar Safa, Malek Sraj, Iyad Kuwatly, and Zaid Al Masri. 2006. A hybrid honeypot framework for improving intrusion detection systems in protecting organizational networks. *Computers & Security* 25, 4 (2006), 274–288.

[10] M. Beham, M. Vlad, and H. P. Reiser. 2013. Intrusion detection and honeypots in nested virtualization environments. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[11] Brian M. Bowen, Shlomo Hershkop, Angelos D. Keromytis, and Salvatore J. Stolfo. 2009. Baiting Inside Attackers Using Decoy Documents. In *Security and Privacy in Communication Networks - 5th International ICST Conference, SecureComm 2009, Athens, Greece, September 14-18, 2009, Revised Selected Papers.* 51–70.

[12] Yuriy Bulygin. 2008. CPU side-channels vs. virtualization rootkits: the good, the bad, or the ugly *(ToorCon).*

[13] Carolyn Crandall. 2016. The ins and outs of deception for cyber security. http://www.networkworld.com/article/3019760/network-security/the-ins-and-outs-of-deception-for-cyber-security.html.

[14] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian B. Grizzard, John G. Levine, and Henry L. Owen. 2004. HoneyStat: Local Worm Detection Using Honeypots. In *Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004, Sophia Antipolis, France, September 15-17, 2004. Proceedings.* 39–58.

[15] P. Defibaugh-Chavez, R. Veeraghattam, M. Kannappa, S. Mukkamala, and A. H. Sung. 2006. Network Based Detection of Virtual Environments and Low Interaction Honeypots. In *2006 IEEE Information Assurance Workshop.*

[16] Dilshan Keragala. 2016. Detecting Malware and Sandbox Evasion Techniques. *SANS Institute InfoSec Reading Room* (2016).

[17] Peter Ferrie. 2008. Attacks on virtual machine emulators. https://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.

[18] Xinwen Fu, Wei Yu, Dan Cheng, Xuejun Tan, Kevin Streff, and Steve Graham. 2006. On Recognizing Virtual Honeypots and Countermeasures. In *Second International Symposium on Dependable Autonomic and Secure Computing (DASC 2006), 29 September - 1 October 2006, Indianapolis, Indiana, USA.* 211–218.

[19] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA.*

[20] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: A Highly Reliable, Self-repairing Operating System. *SIGOPS Oper. Syst. Rev.* 40, 3 (July 2006).

[21] T. Holz and F. Raynal. 2005. Detecting honeypots and other suspicious environments. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop.* 29–36.

[22] Xuxian Jiang, Dongyan Xu, and Yi-Min Wang. 2006. Collapsar: A VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *J. Parallel Distrib. Comput.* 66, 9 (2006), 1165–1180.

[23] Ari Juels and Ronald L. Rivest. 2013. Honeywords: Making Password-cracking Detectable. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13).*

[24] Ryan Kazanciyan and Matt Hastings. 2014. Investigating PowerShell Attacks. *Black Hat* (2014), 25.

[25] Christopher Kruegel. 2015. Evasive malware exposed and deconstructed. In *RSA Conference.* 12–20.

[26] I. Kuwatly, M. Sraj, Z. Al Masri, and H. Artail. 2004. A Dynamic Honeypot Design for Intrusion Detection. In *The IEEE/ACS International Conference on Pervasive Services.*

[27] Tamas K. Lengyel, Justin Neumann, Steve Maresca, Bryan D. Payne, and Aggelos Kiayias. 2012. Virtual Machine Introspection in a Hybrid Honeypot Architecture. In *5th Workshop on Cyber Security Experimentation and Test, CSET '12, Bellevue, WA, USA, August 6, 2012.*

[28] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13).*

[29] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: the rise of the virtual library operating system. *Commun. ACM* 57, 1 (2014), 61–69.

[30] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).*

[31] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. 2017. Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. In *2017 IEEE Symposium on Security and Privacy (SP).*

[32] Nitsan Saddan. 2016. Hacking Team and Defense through Deception. https://securityledger.com/2016/05/opinion-hacking-team-and-defense-through-deception/.

[33] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security (EuroSec '14).*

[34] Niels Provos and Thorsten Holz. 2008. *Virtual Honeypots - From Botnet Tracking to Intrusion Detection.* Addison-Wesley.

[35] Salvatore J. Stolfo, Brian M. Bowen, and Malek Ben Salem. 2011. Insider Threat Defense. In *Encyclopedia of Cryptography and Security, 2nd Ed.* 609–611.

[36] J. Sun, K. Sun, and Q. Li. 2017. CyberMoat: Camouflaging critical server infrastructures with large scale decoy farms. In *2017 IEEE Conference on Communications and Network Security (CNS).*

[37] Symantec. 2018. 2018 Internet Security Threat Report. https://www.symantec.com/security-center/threat-report.

[38] Emmanouil Vasilomanolakis, Shankar Karuppayah, Max Mühlhäuser, and Mathias Fischer. 2015. Taxonomy and survey of collaborative intrusion detection. *ACM Computing Surveys (CSUR)* (2015).

[39] Jonathan Voris, Jill Jermyn, Nathaniel Boggs, and Salvatore Stolfo. 2015. Fox in the Trap: Thwarting Masqueraders via Automated Decoy Document Deployment. In *Proceedings of the Eighth European Workshop on System Security (EuroSec '15).*

[40] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005.* 148–162.

[41] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and Performance in the Denali Isolation Kernel. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002).

[42] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012. Down to the Bare Metal: Using Processor Features for Binary Analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12).*

[43] Vinod Yegneswaran, Paul Barford, and Dave Plonka. 2004. *On the Design and Use of Internet Sinks for Network Abuse Monitoring.* Springer Berlin Heidelberg.