TOWARDS BUILDING A SCALABLE AND BELIEVABLE HYBRID HONEYPOT FRAMEWORK

by

Songsong Liu
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

______________________	Dr. Kun Sun, Dissertation Director
______________________	Dr. Songqing Chen, Committee Member
______________________	Dr. Jens-Peter Kaps, Committee Member
______________________	Dr. Emanuela Marasco, Committee Member
______________________	Dr. Deborah Goodings, Associate Dean
Date: ______________________	Summer Semester 2022 George Mason University Fairfax, VA

Towards Building a Scalable and Believable Hybrid Honeypot Framework

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Songsong Liu
Master of Engineering
Huazhong University of Science and Technology, 2015
Bachelor of Engineering
Wuhan University, 2012

Director: Dr. Kun Sun, Professor
Department of Information Sciences and Technology

Summer Semester 2022
George Mason University
Fairfax, VA

Copyright © 2022 by Songsong Liu
All Rights Reserved

Dedication

I dedicate this dissertation to my family and friends.

Acknowledgments

I would like to express my sincerest gratitude to my advisor, Dr. Kun Sun, for his continuous support and guidance during my Ph.D. study at George Mason University. I am grateful for his countless hours of reading, refining, and commenting on my research papers. Along with Dr. Kun Sun, I would like to thank my Ph.D. committee, Dr. Songqing Chen, Dr. Jens-Peter Kaps, and Dr. Emanuela Marasco for their insights and feedback.

This work consists of four research papers. I would like to extend my sincere thanks to other co-authors: Dr. Pengbin Feng, Shu Wang, Xu He, Dr. Jiahao Cao, Tommy Chin, and Dr. Qi Li. This work would not have been possible without their knowledge and support. I also want to thank all the formal and current lab members who make our lab a friendly place.

This work was supported in part by the Office of Naval Research grants N00014-16-1-3214, N00014-18-2893, and N00014-20-1-2407.

Table of Contents

Page

List of Tables viii
List of Figures ix
Abstract x

1 Introduction 1
  1.1 Context and Motivation 1
  1.2 Research Contribution 3
  1.3 Dissertation Organization 5

2 A Hybrid Webshell Honeypot Framework against Command Injection 6
  2.1 Introduction 6
  2.2 Background 9
    2.2.1 Command Injection 9
    2.2.2 Webshell 10
    2.2.3 PHP 10
  2.3 Threat Model and Assumption 11
  2.4 System Design 11
    2.4.1 Overview 11
    2.4.2 Data Exchange 14
    2.4.3 Trace Hiding 15
  2.5 Implementation 16
    2.5.1 Command Redirection 17
    2.5.2 File Synchronization 20
    2.5.3 Trace Hiding 21
  2.6 Evaluation 23
    2.6.1 Deception Capabilities 23
    2.6.2 Runtime Performance 25
  2.7 Discussion 28
  2.8 Related Work 29
    2.8.1 Hybrid Honeypot 29
    2.8.2 PHP Webshell 30

2.9 Chapter Summary 30
3 Attacks and Countermeasures on the Network Context of Distributed Honeypots 31
3.1 Introduction 31
3.2 Related Work 34
3.2.1 Distributed Honeypot 34
3.2.2 Anti-Honeypot Mechanism 35
3.3 Threat Model 36
3.4 Network Context Cross-Checking Attacks 37
3.4.1 Pre-exploitation Reconnaissance 38
3.4.2 Post-exploitation Reconnaissance 39
3.4.3 Artifacts Cross-Checking 40
3.4.4 Attack Effectiveness 41
3.5 HoneyPortal: The Countermeasure 44
3.5.1 System Design 44
3.5.2 Implementation of HoneyPortal 47
3.6 Evaluation 52
3.6.1 Experiment Testbed 52
3.6.2 Defense Effectiveness 53
3.6.3 Processing Latency 55
3.6.4 System Overhead 56
3.7 Discussion 57
3.8 Chapter Summary 58
4 Enhancing Malware Analysis Sandboxes with Emulated User Behavior 59
4.1 Introduction 59
4.2 Threat Model 62
4.3 System Design 63
4.3.1 System Overview 63
4.3.2 Computer Usage Collector 65
4.3.3 User Profile Generator 65
4.3.4 Artifacts Generator 66
4.3.5 Update Scheduler 72
4.4 Prototype Implementation 73
4.4.1 Computer Usage Collector 74
4.4.2 User Profile Generator 75
4.4.3 Artifacts Generator 78
4.4.4 Update Scheduler 80

4.5 Effectiveness Evaluation 81
4.5.1 Experiment Setup 82
4.5.2 Accumulated Artifacts 82
4.5.3 Variation of Artifacts Generation 85
4.5.4 Artifacts Validation in Third-Party Tool 86
4.6 Discussions 87
4.7 Related Work 91
4.7.1 Artifacts Identification-based Sandbox Evasion Techniques 91
4.7.2 Countering Sandbox Evasion 92
4.8 Chapter Summary 93
5 Enhancing the Fidelity of Honeypot with Real-Time User Behavior Emulation . 94
5.1 Introduction 94
5.2 Related Work 96
5.2.1 Honeypot Detection 96
5.2.2 User Emulation 97
5.3 Threat Model and Assumption 97
5.4 System Design 98
5.4.1 Overview of HoneyMustard 98
5.4.2 User Behavior Emulation 99
5.5 Implementation 104
5.5.1 User Operations Collection 104
5.5.2 Computer Vision-based Emulation 107
5.6 Evaluation 108
5.6.1 Deception Effectiveness 108
5.6.2 System Overhead 112
5.7 Discussion 112
5.8 Chapter Summary 113
6 Conclusions 114
6.1 Summary 114
6.2 Future Work 116

List of Tables

Table Page

2.1 PHP Built-in Program Execution Functions 17
2.2 Webshell Datasets 24
3.1 The Effectiveness of NC3 Attacks 42
4.1 System Artifacts and Corresponding Emulation Strategies 79
4.2 System Artifacts Comparison 84

List of Figures

Figure		Page
2.1	The Architecture of HoneyBog	12
2.2	The Interceptor for Command Redirection	19
2.3	Testbed for HoneyBog	24
2.4	Latency for Redirected Built-in Function	26
2.5	Latency for Redirected Commands (`system()`)	27
3.1	The Attack Procedure of NC3	38
3.2	The Architecture of HoneyPortal System	45
3.3	Two Options of the Redirection Channel	47
3.4	Implementation of the Front End	48
3.5	Implementation of the Virtual Machine-based Back End	49
3.6	Implementation of the Physical Machine-based Back End	51
3.7	Testbed for HoneyPortal	52
3.8	Network Recon Results	53
3.9	Link Latency (RTT) under Different Packet Sizes	55
3.10	Processing Latency Breakdown of HoneyPortal	56
4.1	The Architecture of UBER System	64
4.2	The Structure of User Behavior Profile	66
4.3	The Workflow of Event Selector & Event Executor	69
4.4	Implementation of UBER	73
4.5	Implementation of Computer Usage Collector	75
4.6	An Example of Configuration File	77
4.7	The Daily Variation of Artifacts Generation	85
4.8	The Predictions of Sandbox Deployed with UBER	87
5.1	The Overview of HoneyMustard	99
5.2	The Workflow of HoneyMustard	100
5.3	Implement of Action Emulator	107
5.4	Results of Emulated Activities Identification Survey	111

Abstract

TOWARDS BUILDING A SCALABLE AND BELIEVABLE HYBRID HONEYPOT FRAME-WORK

Songsong Liu, PhD

George Mason University, 2022

Dissertation Director: Dr. Kun Sun

A honeypot is an effective tool to learn new attacking vectors and strategies from attackers or malware, and it has been adopted and deployed in production systems. Meanwhile, the distributed hybrid honeypot system has emerged as a new trend to achieve better scalability by improving the deployment, management, and security of honeypot systems. In practice, attackers are always well-motivated to detect whether the victim machine is a honeypot. Lack of real network activities, no believable user activities, and network context inconsistency during the exploitation are the main indicators of the honeypot to attackers. As a response to these challenges, this dissertation explores how to construct distributed hybrid honeypot systems to improve scalability and believability.

In the first work, we develop a hybrid webshell honeypot framework called HoneyBog to monitor and analyze webshell-based command injection. It intercepts and redirects malicious injected commands from the front-end honeypot to the high-fidelity back-end honeypot for execution. HoneyBog can achieve two advantages by using the client-server honeypot architecture. First, since the webshell-based injected commands are transferred from the compromised web server to a remote constrained execution environment, we can prevent the attacker from launching further attacks in the protected network. Second,

it facilitates the centralized management of high-fidelity honeypots for remote honeypot service providers. Moreover, we increase the system fidelity of HoneyBog by synchronizing the website files between the front-end and back-end honeypots.

In the second work, we uncover that all existing distributed honeypot systems suffer from one type of anti-honeypot technique called network context cross-checking (NC3) that enables attackers to detect network context inconsistencies before and after breaking into a targeted system. We perform a systematic study of NC3 and identify nine types of network context artifacts that may be leveraged by attackers to identify distributed honeypot systems. As a countermeasure, we propose HoneyPortal, a stealthy traffic redirection framework to defend against the NC3 attack. The basic idea is to project a remote honeypot into the protected local network as a believable host machine.

In the third work, we design an emulation-based system called UBER to enhance malware analysis sandboxes. The core idea is to generate realistic system artifacts based on automatically derived user profile models. We solve two major challenges. First, we generate authentic system artifacts continuously to emulate the real-user behaviors. Second, we integrate the generated artifacts stealthily to hide the trace of the emulation.

In the fourth work, we propose HoneyMustard, a real-time application-level user behavior emulation framework to enhance the fidelity of the honeypot. HoneyMustard leverages computer vision techniques to emulate GUI-based user activities in the honeypot via a remote desktop connection, achieving both believable and stealthy design goals. The emulated user activities are generated from the collected user operations from real user activities or application user manuals, which ensures attackers can observe logical and believable activities at the application level. Since attackers can only observe a remote desktop connection during the emulation, HoneyMustard can conceal the emulator as a normal service to achieve real-time emulation without being detected.

Chapter 1: Introduction

1.1 Context and Motivation

In recent years, we have witnessed an increase in 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 or firewalls) by exploiting zero-day vulnerabilities or well-planned social engineering attacks. Consequently, there has been a surging trend of leveraging deception techniques to detect and defeat such advanced cyber threats. One popular solution is to deploy a series of honeypots (or traps) in the target network, which can effectively misdirect the attackers to camouflage legitimate hosts.

The honeypot technique is introduced by Lance Spitze [1] in 2002. He proposed a system that can lure attackers by possessing "valuable information". This scenario is similar to a bee attracted by the honeypot. Since any movement found in the honeypot is deemed to be malicious, the honeypot is a low-noise environment to learn new attacking vectors and strategies from attackers or malware. In the practical environment, the defenders must deal with two critical issues: scalability and believability.

Scalability

For defenders, the ideal honeypot system should be affordable and easily manageable. However, maintaining a honeypot system could be an overwhelming burden for small/medium companies and organizations, since it requires extra investments in hardware, software, and hiring professional operators. It becomes an obstacle to deploying honeypot systems widely. The distributed hybrid honeypot system can satisfy the requirement of scalability to provide the honeypot as a service [2], where customers can rent high-fidelity and customer-tailored

honeypots from trusted third-party service providers. It adopts a client-server architecture to achieve better scalability. Low-interaction honeypots are deployed as clients (i.e., the front ends) in the protected local network to redirect interesting attack traffic to the remote high-interaction honeypots (i.e., servers as the back end), which are maintained by honeypot service providers for centralized attack containment and investigation. It allows the front end to reply to initial simple requests of the attacker and traps the attacker in the back-end honeypot for more sophisticated requests. It can combine the benefits of the lightweight low-interaction honeypots and the highly believable high-interaction honeypots and achieve centralized management and control of the high-interaction honeypots. The distributed hybrid honeypot systems can be constructed at the network flow level to handle all types of incoming attacks or at the application level to handle the attack on a specific application. For now, the study on the latter one is less.

Believability

For attackers, identifying the honeypot is crucial to ensure the stealthiness and effectiveness of the attack. As the weapon for attackers to identify honeypots, anti-honeypot mechanisms can be divided into network-level fingerprinting and system-level fingerprinting [3]. Network-level honeypot fingerprinting focuses on detecting the discrepancy of network activities. System-level honeypot fingerprinting is based on the information collected from operating systems and applications. These anti-honeypot techniques focus on the single honeypot by identifying the fingerprinting inside it. The deployment of distributed hybrid honeypot system could also cause inconsistency in the network context. The network context includes the setup information of the hosts (e.g., host number, IP address, services, etc.) and the traffic among them in the network. These differences in network context can be exploited as a new anti-honeypot technique. Any difference in pre-exploitation and post-exploitation stages will indicate the existence of the honeypot system. All the existing distributed hybrid honeypot schemes are suffering this attack.

Compared to the real host, the honeypot lacks believable network activities and system

activities since there is no real user on it. By using "wear and tear" artifacts, the malware can identify the honeypot [4]. The sophisticated attackers can go much further. They can not only check those "wear and tear" artifacts but also real-time and logical user activities before conducting further attacks. The intuitive solution is to emulate the user activities and generate "wear and tear" artifacts in the honeypot. However, it is still an open problem to emulate user activities effectively.

1.2 Research Contribution

In this research, we focus on improving the scalability and believability of the honeypot systems. The contributions of this research are as follows:

First, we work on the application-level hybrid honeypot system. The application-level honeypot can help operators focus on specific types of attacks. With the distributed hybrid honeypot framework, it is easy to integrate the application-level honeypots into existing networking systems. We propose the first webshell honeypot system called HoneyBog that is an application-level hybrid honeypot against the webshell-based command injection. To achieve both high believability and good scalability, HoneyBog applied the application level distributed hybrid architecture that separates lightweight front-end proxies from believable back-end servers. It is based on one key observation that no matter what anti-honeypot approaches the attackers may utilize when we have full control of the malicious processes on the front-end honeypot, we are able to feedback believable fake information prepared by the back-end server to the attackers. We implement a prototype of HoneyBog using PHP and the Apache web server. Our experiments on 260 PHP webshells show that HoneyBog can effectively intercept and redirect injected commands with low overhead.

Second, we study the network context issue of the distributed hybrid honeypot systems. We summarized a generic network context-based attack to detect the distributed hybrid honeypot system, called *network consistency cross-checking attack* (NC3). It is based on one key observation that inside attackers may collect the artifacts related to the target machine and other hosts in both pre-exploitation and post-exploitation stages. Compare

to the existed single honeypot detection method searching the features of the honeypot, NC3 enables the attacker to cross-check the inconsistency of network context during the exploitation life-cycle to identify the honeypot system. We identify nine types of network context artifacts. By checking the existing distributed hybrid honeypot schemes, we found all of them can be detected with NC3. As a countermeasure, we propose HoneyPortal, a stealthy traffic redirection framework to defend NC3 attack. To maintain the consistency of network context, HoneyPortal redirects all the traffic from the front end to the back-end honeypot, while all the outbound traffic (i.e., layer 2 and above) of the honeypot is returned to the front end. We conduct experiments in a real testbed, and the experimental results show that HoneyPortal can effectively defeat NC3 attacks with low overhead.

Third, we work on increasing the believability of the sandbox (honeypot) to deceive malware. Armored malware adopts various anti-sandbox techniques to evade analysis, from simple environment-specific traits detection to complex real-user operation environment verification. Particularly, malware may identify sandbox environments by checking several system artifacts that are impacted by the accumulation of normal user activities, such as file accesses. To respond to this kind of detection, we propose a new framework called *User Behavior Emulator (UBER)* which can automatically generate realistic system artifacts via user behavior emulation for the sandbox environments. UBER automatically derives the user profile model from real user activities. Then it emulates the high-level user behavior in an isolated always-on system and stealthily merges this system into the malware analysis sandboxes. We implement a prototype of UBER with Python system-event monitor and automation control modules. Our experimental results demonstrate that UBER is capable of generating believable system artifacts and effectively mitigates the sandbox evasion techniques that exploit system fingerprinting.

Fourth, we work on increasing the believability of the honeypot to deceive sophisticated attackers. Unlike the malware that observes the static artifacts at a time point, the attacker can keep observing long enough to check whether there is a real user in it. They can observe the real-time user activities and verify whether these activities are logical. The real

host is expected to have not only the static usage artifacts but also the dynamic artifacts (i.e., continual user activity) during the observation period. To solve this problem, we propose HoneyMustard, a real-time application-level user behavior emulation framework. HoneyMustard leverages computer vision techniques to emulate GUI-based user activities in the honeypot via a remote desktop connection (e.g., VNC). We collect user operations from real user activities or application user manuals, which ensures operation sequences are logical at the application level. Using the remote desktop connection and GUI-based emulation ensures the real-time emulation without leaving any traces of the emulator inside the honeypot. We implement a prototype of HoneyMustard and evaluate its deception effectiveness and performance overhead. The experimental results show that our solution can effectively improve the believability of honeypots with low overhead.

1.3 Dissertation Organization

The rest of this dissertation is structured as follows: Chapter 2 presents a hybrid honeypot framework to defend the webshell-based command injections. Chapter 3 investigates how the network context can be leveraged by attackers to detect distributed honeypot systems. We also present a countermeasure to maintain the consistency of distributed honeypot systems. Chapter 4 presents an emulation-based anti-evasion framework, which can automatically generate realistic system artifacts to deceive malware in the sandbox. Chapter 5 presents a real-time application-level user behavior emulation framework, which can provide real-time and logical user operations to deceive the attackers in the honeypot. In Chapter 6, we summarize this dissertation and outline directions for future work.

Chapter 2: HoneyBog: A Hybrid Webshell Honeypot Framework against Command Injection

2.1 Introduction

Web servers and web applications are often weak points in an organization's infrastructure. After a web server is compromised, it may be used as a stepping stone into an organization's internal network, making it an appealing target for attackers. To maintain long-term and stealthy access to the remotely compromised web server, an attacker may choose to run a piece of code or a script on the server to create a webshell interface, which can be interacted with via a web browser on the attacker's local machine [5].

Command injection [6] allows attackers to inject arbitrary system commands into the victim operating system in which the attacker does not have direct access privilege. By executing unexpected and dangerous commands on the operating system, the attacker can launch additional attacks to compromise other computers in the internal network. The webshell-based command injection is one of the most prolific command injection methods. According to IBM Managed Security Services (MSS) data, more than 20% of command injection attacks are caused by webshell [7].

To defend the webshell-based command injection, the first front-line is the webshell script detection. Traditional detection methods [8] include network/web traffic anomaly detection and script signature-based detection. In recent years, several machine learning-based techniques have been developed to enhance these detection techniques [9–15]; however, the various webshells may still elude detection by adopting armored encryption and obfuscation methods [16].

As a complementary solution to the detection-based techniques, the honeypot mechanism can provide another layer of protection by setting up a decoy to lure attackers and

analyzing unauthorized or illicit use of computer resources [1]. By monitoring and analyzing suspicious activities in honeypot systems, the administrator has a better chance of understanding attack techniques and constraining attackers at an early stage. However, it lacks a honeypot system that is dedicated to the detection of webshell-based command injections.

We develop HoneyBog, the first webshell honeypot framework that provides an application-level high-fidelity honeypot environment against the webshell-based command injection attacks. HoneyBog is a hybrid honeypot adopting a client-server architecture. It has two advantages. First, the webshell-based injected commands are transferred from the compromised host to a remote constrained environment, preventing the attacker from launching additional attacks on other hosts in the protected network. Second, the hybrid architecture facilitates the centralized management of high-fidelity honeypots by a remote honeypot service provider.

HoneyBog consists of three major components, namely, *the front end*, *the back end*, and *the redirection tunnel*. The front end is deployed in the protected network and contains a web server that may provide an execution environment for the webshell. The front end intercepts all injected commands and redirects them to the back end via a redirection channel. The back end is a constrained environment in a remote network to execute the commands received from the front end and return the results to the front end. A redirection channel is established to securely connect the front end and the back end.

We resolve two challenges of designing the hybrid webshell honeypot, namely, intercepting all injected commands and improving the fidelity of HoneyBog. First, in order to intercept all injected commands from the webshell, we need to identify all command execution related functions and intercept their execution on the local host. The webshell is written in server-supported script languages, such as PHP, JSP, and Python. The execution of these interpreted languages relies on a separate program called the interpreter. To hook all the injected commands, we manually analyze all built-in functions to identify command execution related functions. Then we place our interceptor into the interpreter to ensure

that any invocations of these functions can be intercepted. Also, when these functions are invoked by the webshell, our interceptor prevents them from being executed locally. Instead, the interceptor reads their parameters and sends those parameters to the back end as redirected commands for execution.

Second, we improve the fidelity of HoneyBog to hinder the identification of the hybrid honeypot by attackers. Since the web server may be used as a file exchanger by the attacker, we design a bi-directional file synchronization mechanism between the front end and the back end. Since the attacker may upload malicious files to the front end via a legal upload channel (e.g., file upload interface) to launch further attacks, when the injected commands are executed on the back end to interact with the uploaded files, these files should be found on the back-end honeypot too. Therefore, it requires the uploaded files to be synchronized from the front end to the back-end honeypot. Meanwhile, if any modifications on the back-end website files should be visible to the attacker in the front-end web server, those files need to be synchronized from the back end to the front end. To achieve the bi-directional file synchronization, we deploy two file synchronizers in the front-end and back-end honeypots, respectively. Since the injected command cannot be executed in the front end, the attacker is unaware of the existence of a file synchronizer on the front end. Also, on the back-end honeypot, we use a normal SSH server with the default configuration to implement the functions of the command executor and the file synchronizer.

Since PHP is one of the most-used languages for websites and the main choice for creating the webshell [17], we implement a prototype of HoneyBog over the Apache web server and PHP to demonstrate the effectiveness of HoneyBog. We place both the front-end and back-end honeypots into virtual machines and use the SSH port forwarding mechanism to establish a redirection tunnel between the two virtual machines. Also, we build a testbed for evaluating the capability and performance of our HoneyBog system. We test 260 PHP webshells that can inject commands, and the experimental results show that HoneyBog can successfully intercept and redirect injected commands from the webshells with low overhead. Under 32 concurrent requests, the interceptor consumes only 23 MB of memory and uses

less than 1% of CPU.

In summary, we make the following contributions.

- We propose the first webshell honeypot system called HoneyBog that is an application-level hybrid honeypot against the webshell-based command injection. It provides a high-fidelity environment for monitoring and analyzing attacks caused by injected commands from webshell.

- We design an interceptor embedded into the interpreter to control the attack traffic in the hybrid honeypot framework by intercepting risky functions and redirecting the injected commands to the remote back-end honeypot for execution.

- We improve the stealthiness of the hybrid honeypot framework by synchronizing the website files between the front end and the back end honeypots.

- We implement a prototype of HoneyBog on the Apache server and PHP. The experimental results demonstrate that our system can effectively intercept and redirect injected commands and increase the whole system's fidelity with a low system overhead.

2.2 Background

2.2.1 Command Injection

Command injection (also known as shell injection) is an attack whose goal is the execution of arbitrary commands on the host operating system via a command shell environment. This attack is largely possible due to insufficient input validation. The attacker can exploit this kind of vulnerability to inject unsafe data into a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the vulnerable application's privileges. Often, the attacker can leverage this attack to compromise other parts of the hosting infrastructure, taking advantage of trust relationships to pivot the attack to other hosts within the organization. Unlike the code injection, this attack extends

the application's default functionality merely via executing system commands. Injecting code is not necessary.

2.2.2 Webshell

A webshell [5] is a web-based implementation of the shell concept, which is written in server-supported programming languages such as PHP, JSP, etc. It usually is placed on an openly accessible web server as a backdoor to allow the attacker to gain persistent access to the server's operating system. The attacker can exploit common web page vulnerabilities such as SQL injection, remote file inclusion (RFI), or even use cross-site scripting (XSS) as part of a social engineering attack in order to attain file upload capabilities and transfer the malicious payloads. After successful installation by the attacker, the webshell can be commonly utilized to exfiltrate sensitive data from the web server, launch further attacks on hosts inside the network without direct Internet access, deface the website, etc. The webshell is the second step of an attack (this is also referred to as post-exploitation).

2.2.3 PHP

PHP [18] is an interpreted language developed in the mid-1990s to serve dynamic web pages. Over years, it remains the most-used language for websites, powering about 80% of all websites [17]. PHP is composed of two separate pieces. The surface level is the PHP core. It handles communication with, and bindings to, the SAPI layer (Server Application Programming Interface). It provides the PHP built-in functions to be invoked in PHP scripts. As an interpreted language, PHP relies on a separate interpreter - Zend Engine - at the lowest level. The Zend engine handles parsing a human-readable script into machine-readable tokens (Zend Opcode), for execution within a process space. To achieve high feasibility, PHP allows developers to write extensions in C to add functionality to the PHP language or overwrite function pointers provided by the PHP core. PHP extensions can be compiled statically into PHP or loaded dynamically at runtime.

2.3 Threat Model and Assumption

This work focuses on the webshell-based command injection. We assume that the target web server runs on an uncompromised OS and only uses one full-featured interpreted language. We focus on PHP. The attacker intends to compromise a web server and penetrate its internal network to launch additional attacks.

During the pre-exploitation stage, the attacker collects relevant information to discover exploitable vulnerabilities via classic network reconnaissance. The attacker can only interact with the web server via HTTP/HTTPS connection in this stage. Only open ports of HTTP/HTTPS are accessible, while other ports and services are invisible to the attacker. The attacker cannot access or recon the internal local network of the target web server.

During the post-exploitation stage, the attacker can inject or upload the malicious payloads into the vulnerable web server to construct the webshell. Once the webshell is successfully constructed, the attacker can use the webshell to issue OS commands remotely and leverage other exploitation techniques to escalate privileges. These commands include the ability to add, delete and modify files as well as the ability to run shell commands, executables, or scripts.

We assume the honeypot operator can terminate the attack by cutting off the connection or pausing the honeypot, once the attacker obtains the root privilege. At this point, the honeypot operator has collected all necessary information related to attacks in the honeypot. The main goal of the honeypot operator, in this case, is to monitor and analyze the attacks on the honeypot system caused by webshell-based command injection. Lateral movement after the attacker fully controls the honeypot is beyond the scope of this work.

2.4 System Design

2.4.1 Overview

Figure 2.1 shows the architecture of HoneyBog, which consists of three major components: the front end, the back end, and the redirection tunnel. The front ends are lightweight

honeypots that can be deployed in various protected networks (i.e., *front-end network*), whereas the back ends are full-featured honeypots that are located in the honeypot service provider's controlled network (i.e., *back-end network*). When the front end suffers webshell-based command injection, it forwards those commands to the back end for execution via the redirection tunnel. The back end can provide a practical OS environment, various applications, and decoy data. The command execution results will return to the front end along with the redirection tunnel and then be received by the attacker. Within this real environment, the honeypot operator can better observe and analyze the attacker's behavior. This hybrid architecture provides good scalability to deploy the front end in various networks.

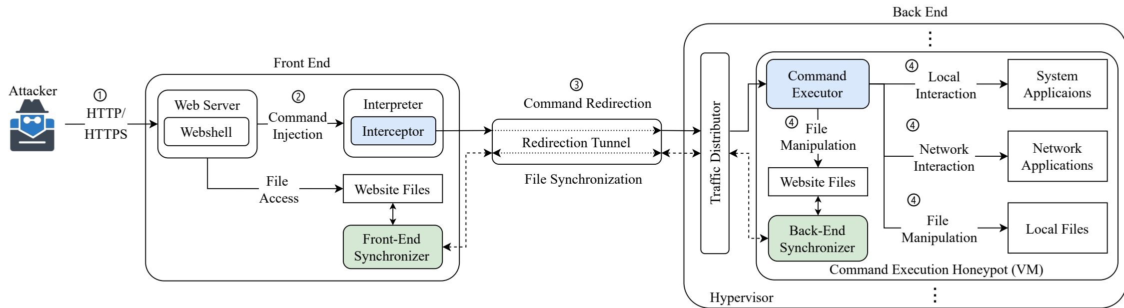


Figure 2.1: The Architecture of HoneyBog

Front End

To keep it lightweight, each front end only contains a web server and essential system supportive components as a honeypot. The deployment environment could be a virtual machine or container to facilitate the deployment. It can be deployed in any protected network with a basic configuration. Its web server uses a full-featured interpreted language library. The web applications running on it can be customized to meet the requirement of the honeypot

operator. Its web pages are accessible to the public and don't contain any sensitive information. When the attacker conducts the webshell-based command injection, our interceptor prevents those commands from executing locally and forwards them to the back end. This interceptor is responsible for the command redirection and file synchronization. The details will be explained in Section 2.4.2.

Back End

The back end supports multiple command execution honeypots (i.e., back-end honeypots) to provide centralized management. For the attacker, the back-end honeypot servers as a real execution environment. It deploys a full-featured OS inside the virtual machine or container. To increase the fidelity, decoy user files could also be deployed inside it.

All the OS commands will be executed in this constrained environment. The honeypot operator can observe the attacker's behavior through these executed OS commands. The honeypot operator terminates the attack after the attacker escalates privilege successfully, which can constrain the attacker from attacking the host machine and HoneyBog itself afterward. To maintain the status for the sophisticated attack, each front end has a corresponding and fixed back-end honeypot. The traffic distributor allocates the received command execution requests to the corresponding back-end honeypots. The execution results are returned to the front end via the same route.

Redirection Tunnel

Because the front end and the back end are deployed in different networks, the data exchange requires crossing multiple networks. As a result, a cross-network redirection tunnel is required. It could be a publicly accessible server or gateway to manage the data exchange or a series of configurable routing devices. The network routes and devices between the front-end network and the back-end network consist of the redirection tunnel.

2.4.2 Data Exchange

The data exchange includes two aspects: command redirection and file synchronization. As shown in Figure 2.1, after constructing a webshell on the target web server successfully, the attacker can interact with the webshell via HTTP/HTTPS at stage ①. Then the attacker may inject commands at stage ②. HoneyBog will redirect the injected commands from the front end to the back end at stage ③. At stage ④, the attacker can leverage the injected command to manipulate files and execute external programs at the back-end honeypot. The file synchronization is bi-directional. It occurs whenever the attacker uploads files to the web server in the front end or modifies the files inside the web directory in the back end.

Command Redirection

The interpreted language separates the program code from the low-level details of the underlying OS. It uses the interpreter to translate the script into executable machine code. In most cases, the interpreted language provides some system-related built-in functions to access OS-managed resources. For example, it allows the script to execute external programs. The interpreter invokes the system APIs for these functions and passes the inputted commands as arguments. The attacker can invoke the risky built-in functions to execute arbitrary system commands.

To redirect the injected command, we need an interceptor inside the interpreter of the front end. It intercepts those built-in functions before the interpreter invokes the local system APIs to execute their commands. These commands are redirected to the back-end honeypot. The redirection prevents the attacker from conducting webshell-based command injection in the front end and attacking other hosts in the front-end network from it.

In the back-end honeypot, the command executor executes these redirected commands with the same level of privilege as the web server. In this way, the attacker can invoke the system applications to collect system information and network applications to probe other hosts in the internal network or construct a new stable system shell. If a new stable system shell is established, the shell will connect to the attacker directly from the back end rather

than via the front end. This ensures that the attacker will not return from the back end to the front end. In addition, the attacker can also access local files and website files to conduct file manipulation.

File Synchronization

The web directory is the file exchange interface for the attacker. The attacker can utilize the webshell to upload files that may contain malicious payloads. By default, the upload files are placed in the web directory. The attacker may manipulate the uploaded files via the injected commands. Also, the website files allow public access. Once the attacker can manipulate the local files, it can move or copy them to the web directory as the website files for downloading.

According to our threat model, the attacker could access the uploaded file in the back-end honeypot, and download the back-end files from the front end. Therefore, we place synchronizers in both the front end and the back end to conduct the bidirectional file synchronization for the website files. The web server in the front end receives requests continuously and generates the access logs, while the back-end honeypot doesn't generate the same log. The synchronizer is also responsible for synchronizing the web server logs from the front end to the back end. This log synchronization is uni-directional. Both of these two synchronizations are incremental and in real-time.

2.4.3 Trace Hiding

To improve the fidelity, we need to hide the trace of HoneyBog. The trace hiding is divided into three dimensions: application, file, and system.

At the application level, we ensure the consistency of web server and interpreter information between the front end and back end. On both sides, we need to deploy the same version web server and interpreted language library. The interceptor has a close relationship - function hook, with the interpreter. The attacker could use the information disclosure functions to reveal its existence. Usually, the interpreted language has the information disclosure

feature. It provides a group of interfaces to observe information about the interpreter itself. The attacker could use these interfaces to collect information about the web server and interpreter in the front end, while the attacker could also use system commands to obtain the same information in the back end. Therefore, it is necessary to shield the interceptor from the attacker.

At the file level, we ensure the attacker can not notice the difference in the file system. As mentioned in the previous section, we synchronize the website files and web server access logs. However, the front-end honeypot is lightweight and only contains essential files. Those decoy user files are only in the back-end honeypot. The interpreted language has file manipulation built-in functions. Typically, due to the privilege limitation of the web server, these functions can only access the website files. In the front end, we strictly constrain the accessible files of the web server so that the attacker is only aware of the local files in the back-end honeypot.

At the system level, we ensure the attacker obtains the same system information from both the front end and back end. The interpreted language also provides the built-in functions to retrieve system information, such as disk size, hostname, user ID, etc. To resolve this issue, we need to adjust the system information of the front end. Since the front-end honeypot is lightweight, its hardware setup would differ from the back-end honeypot. Our interceptor will also intercept these built-in functions and feedback on the back-end system information. Meanwhile, programs (including our file synchronizer) in the front-end honeypot are not visible to the attacker, because the system commands can only be executed in the back-end honeypot.

2.5 Implementation

Our prototype is built on Apache 2.4.41 and PHP 7.4.3. We choose PHP due to its dominance among web applications. The front and back ends are both running Ubuntu 20.04 LTS with kernel 5.4.0. In this section, we detail the implementation in all three aspects: command redirection, file synchronization, and trace hiding.

2.5.1 Command Redirection

As mentioned in Section 2.4, HoneyBog has an interceptor in the front-end honeypot and a command executor in the back-end honeypot. We use the SSH tunnel to transmit the redirected commands. The SSH port forwarding technique can achieve cross-network data transmission. In the back-end hypervisor, we use its SSH server as the traffic distributor to allocate the redirected commands to the corresponding honeypot. In the back-end honeypot, we use its inside SSH server as the command executor. We keep using the default setup for the SSH server, thus it can be treated as a normal service. Hence, we mainly introduce the implementation of the interceptor in the front-end honeypot.

Command Injection Related Functions

Among these PHP built-in functions [19], there are three types of them that can be used to assist webshell-based command injection.

Table 2.1: PHP Built-in Program Execution Functions

Function	Description
`exec()`	Execute an external program.
`system()`	Execute an external program and display the output.
`shell_exec()`	Execute command via shell and return the complete output as a string.
Backtick Operator(``)	Use of the backtick operator is identical to `shell_exec()`.
`popen()`	Open process file pointer.
`proc_open()`	Execute a command and open file pointers for input/output.
`passthru()`	Execute an external program and display raw output.
`pcntl_exec()`*	Execute specified program in current process space.

* Function `pcntl_exec()` is usually disable in the default PHP configuration file for the web server (e.g., Apache). Sometimes, it may be enable mistakenly.

Type I functions are the program execution functions, see Table 2.1. In PHP scripts, these functions are used to execute commands or trigger external programs. Here the external programs could be the existing benign system programs or uploaded malicious payloads. When executing these functions, Zend engine directly invokes the program execution related system APIs, including `popen()`, `execl()`, `execv()`, `execve()`, etc. The attacker's inputted commands are passed as parameters into these system APIs. The attacker can invoke these functions to execute system commands directly.

Type II functions are the code execution functions, such as `eval()`, `preg_replace()`, `assert()`, etc. These functions treat the user inputted parameters as executable PHP codes. Because the inputted parameters are user-controllable data, the attacker can leverage this feature to craft self-defined input to modify the code to be executed, or inject arbitrary code that will be executed. By incorporating the Type I functions into the input, the attacker can still conduct command injection.

Type III functions are the callback functions, such as `array_filter()`, `array_map()`, `array_reduce()`, etc. These functions accept a string parameter that can be used to call a function of the attacker's choice. In this case, the attacker can choose the Type I functions as the callback to execute arbitrary commands.

Usually, Type II and III functions can be used as an obfuscation technique against webshell detection for command injection. The final step of the command execution still falls into Type I functions. Therefore, we only need to hook the Type I functions to intercept all the injected commands.

Interceptor

The interceptor for command execution functions should be exhaustively intercepting target functions and flexible in deployment. To achieve these two goals, we choose to implement the interceptor as a PHP extension instead of directly modifying the PHP library. PHP starts up just a little bit after the Apache server and comes to run the startup of its extensions. We load the interceptor at the beginning of the PHP extension module startup

(i.e., `MINT()`) to ensure our extension can handle all incoming requests.

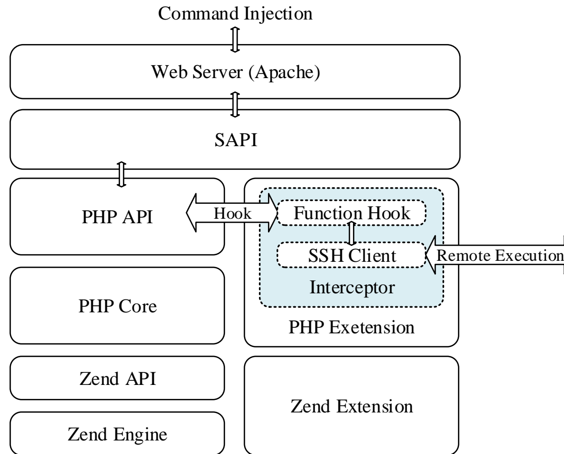


Figure 2.2: The Interceptor for Command Redirection

As shown in Figure 2.2, our interceptor consists of two main parts: function hook and SSH client. The function hook is responsible for intercepting Type I functions, while the SSH client is to execute the injected command remotely in the back-end honeypot. Our function hook searches the corresponding function name from the Zend compiler globals (`CG`) and redirects the execution flow to the extension before these functions are compiled into opcodes.

When the command injection request is received, the web server passes it to the SAPI (Server Application Programming Interface) layer. The SAPI layer connects the web server and PHP, so it provides the PHP built-in program execution functions to the web server. In normal mode, the PHP core will handle the invocation and forward it to the Zend engine. Here the function hook intercepts the invocation. It reads the parameters from these functions and then passes them to the embedded SSH client. The SSH client will send these parameters to the back end. The SSH server in the back-end honeypot executes these parameters as real commands. In this way, the invoked functions will not be executed locally.

The execution results of different Type I functions are displayed in different formats, while our back-end command executor doesn't distinguish the source of redirected commands and returns the execution results in the same format. To solve this problem, we fill the returned execution results into the function execution stream to make them revert to the default format processing. This ensures that each function can display results in the desired format.

2.5.2 File Synchronization

In the front-end honeypot, we deploy an independent synchronizer. In the back-end honeypot, the synchronizer is the same SSH server of command redirection that receives the synchronized files, allowing us to avoid introducing extra services and improve the stealthiness. As mentioned in the design, we only synchronize the website files and web server access logs to minimize the burden of transmission.

For the website files, we synchronize the file under the directory `/var/www` in bi-direction. To synchronize these files from the front end to the back end, we deploy the *rsync* 3.1.3 and *inotify* 3.14 together on the front-end honeypot. Here *rsync* is a file synchronization tool, which supports remote synchronization via SSH access. It minimizes the amount of data copied by only moving the portions of files that have changed. The *inotify* is a Linux kernel subsystem for monitoring file-system events. We use the *inotify* to monitor the website files in the front end. Once there is any modification on website files (e.g., the attacker uploads files), the *inotify* can notice the *rsync* to start the incremental synchronization. The *rsync* in the front-end honeypot is also responsible for synchronizing files from the back end to the front end through a periodic query each second via SSH. If the query detects any changes in its website files, *rsync* starts the incremental synchronization for the front end.

In some cases, the synchronizations in both two directions may occur at the same time to cause conflict. To solve this problem, we split the bi-directional synchronization into two processes. Its pseudo-code is shown in Algorithm 1. We prioritize the process of

Algorithm 1 Bi-directional File Synchronization

Input: dir_f: The front-end file directory
dir_b: The back-end file directory
t: The synchronization interval

```
1: global pid_b2f ← 0
2: function MAIN( )
3:     Start_Process(BackToFront(dir_b, dir_f, t))
4:     Start_Process(FrontToBack(dir_f, dir_b))

5: function BackToFront(dir_src, dir_dst, t)
6:     pid_b2f ← Get_Pid()
7:     while True do
8:         rsync(dir_src, dir_dst)
9:         Sleep(t)

10: function FrontToBack(dir_src, dir_dst)
11:     while inotifywait(dir_src) do
12:         Pause_Process(pid_b2f)
13:         rsync(dir_src, dir_dst)
14:         Continue_Process(pid_b2f)
```

synchronization from the front end to the back end (FrontToBack). When this synchronization starts, it will pause the synchronization from the back end to the front end (BackToFront) until this round task is completed. In this way, we ensure that only uni-directional synchronization is active at each time point.

For Apache web server, its synchronized logs locate in the directory `/var/log/apache2`. These logs are synchronized from the front end to the back end in uni-direction. Here we still use the *rsync* and *inotify* to conduct the incremental synchronization. The synchronization is triggered only when the front-end web server generates new logs.

2.5.3 Trace Hiding

In the implementation, we have two specific tasks: hiding the components of HoneyBog and making the hybrid architecture transparent to the attacker. For HoneyBog, its main

components are deployed in the front end, while only an SSH server is in the back end as a normal service module. For the hybrid architecture, our main goal is to eliminate inconsistencies in the collected information from both two sides.

At the application level, we use the same version of the Apache server and PHP library on both sides. PHP provides information disclosure functions to reveal the loaded extension information. The attacker can invoke these functions via code injection. Function `phpinfo()`, `extension_loaded()`, and `get_loaded_extension()` can list all the loaded extensions. For these functions, we hook them in our interceptor and modify their execution flow to skip our extension during PHP hash table enumerating. Then their execution results exclude our extension. Function `get_extension_funcs()` and `get_defined_functions()` can list all the available function for loaded extensions. For these functions, our extension doesn't contain any callable function for the PHP script, thus no function that belongs to our extension will present. Besides, PHP also provides function `dl()` to load arbitrary extension at run-time. The attacker could leverage this function to load an extension containing self-defined functions, which may access the PHP hash table directly. In other words, this could help the attacker bypass our interceptor and obtain information about our extension. For this reason, we forbid dynamic loading extension by disabling this function.

At the file level, PHP provides file access functions, which can be invoked by the attacker via webshell. The web server default user, `www-data`, can manipulate website files and read user-level system files. Since the files stored in the front-end and back-end honeypots are different, it may help the attacker find out components and the hybrid architecture of HoneyBog. To overcome this problem, we set up the Apache server's configuration file to constrain the user's accessible directory to the web directory `/var/www`. The attacker cannot read files outside of the web directory.

At the system level, we deal with the PHP built-in functions that can introspect system configuration. The functions related to the disk information include `disk_total_space()`, `disk_free_space()`, and `diskfreespace()`. The interceptor hooks these functions and queries the disk information from the back-end honeypot via SSH. The attacker only obtains

the disk information from the back-end honeypot. About other static system information (such as hostname, user ID), we configure them as the same in both the front end and back end.

2.6 Evaluation

Figure 2.3 shows our testbed for HoneyBog. All physical machines including the intermediate server use Ubuntu 20.04 LTS (kernel version 5.4.0) on Dell Precision 7810 desktops with Intel Xeon(R) E5-2620 CPU @ 2.40GHz and 16 GB memory. For the front-end and back-end honeypots, we use VMWare 16.1 to create VMs with 2 CPU cores and 2 GB memory. We use Cisco C921-4P as edge routers A and B to connect the front-end network and the back-end network to the Internet. The front-end honeypot is a VM connected to the front-end network under the bridge network model. By configuring NAT port mapping on router A, the attacker can access the front-end honeypot via port 5000 of router A. We configure the redirection channel between the front end and back end via the intermediate server using SSH port forwarding. We also configure the back-end honeypot to use the same IP address as the front-end honeypot.

2.6.1 Deception Capabilities

Effectiveness

To verify the deception effectiveness of HoneyBog, we test the real webshells on our test-bed. We collect PHP webshell samples from public Github repositories listed in Table 2.2. From these datasets, we pick out 939 webshells that are executable for our web server. Among them, there are 260 webshells directly supporting command injection or code injection. Other webshells only support file upload or information disclosure.

We only test webshells that can directly support command injection or code injection. The attacker can also invoke Type I functions via code injection to launch command injection. By testing all these 260 webshells, we found that HoneyBog can successfully intercept

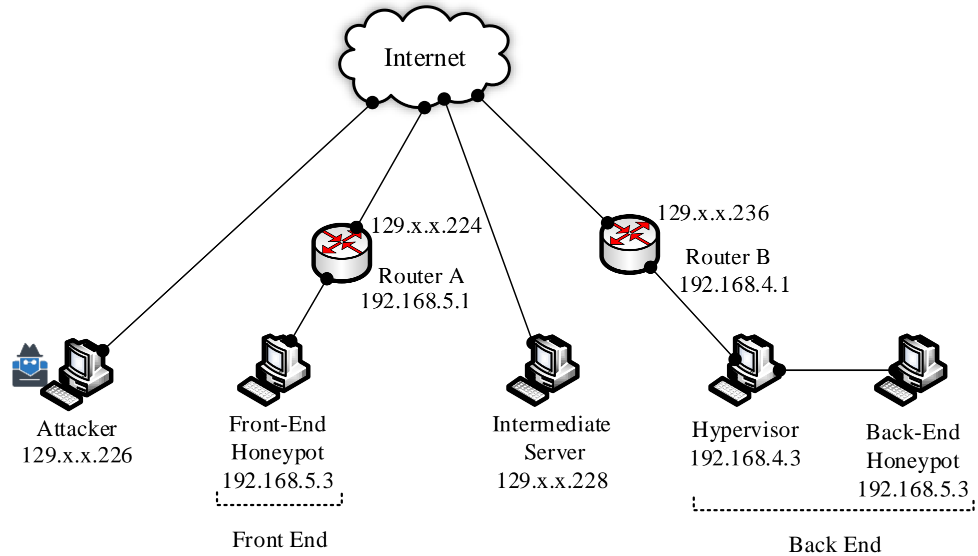


Figure 2.3: Testbed for HoneyBog

all the injected commands and redirect them into the back-end honeypot. For the file synchronization, we test the bi-directional synchronization separately. From the front end to the back end, we upload files via webshell in the front end. After that, we can successfully view upload files in the back-end honeypot. From the back end to the front end, we copy local files to the web directory of the back-end honeypot. We can also successfully access these local files from the front-end web server.

Table 2.2: Webshell Datasets

URL to the Webshell Datasets
https://github.com/BlackArch/webshells
https://github.com/JohnTroony/php-webshells
https://github.com/tennc/webshell
https://github.com/tanjiti/webshellSample
https://github.com/xl7dev/WebShell
https://github.com/ysrc/webshell-sample

Security Analysis

Because the front end redirects all the injected commands and synchronizes all the uploaded files to the back end, the attacker always executes system commands and triggers the malicious payloads in the back-end honeypot. For the attacker, the front end itself becomes an information exchange agent and is immune to command injection. In some cases, the attacker may exploit other vulnerabilities to construct a system shell in our front end. To solve this problem, we can use the solution proposed in [20], which redirects the injected command from the front-end system shell to the back end. The isolated back-end network ensures that the honeypot outbound network connection initiated by the attacker will not return to the front-end network. The attacker cannot use the honeypot to continue attacking the protected network. The security of the back-end honeypot relies on the adopted honeypot productions. In the back-end honeypot, the honeypot operator can terminate the attack, once the attacker obtains the root privilege. Thus, the attacker cannot manipulate the SSH server receiving the redirected commands to initiate the attack along with the redirection tunnel.

2.6.2 Runtime Performance

Redirection Latency

To quantify the redirection latency, we first use ApacheBench [21] to measure all of the redirected built-in functions related to command injection. To minimize the impact of result size, we use the simplified webshell, which only contains the tested functions and basic web page display elements. All these functions execute the command *who*. For each function, we send 100 requests in sequence. As shown in Figure 2.4, the average response time varies from $0.25s$ to $0.31s$ for different functions. The average local response time of them is around $0.03s$, thus the increased latency varies from $0.22s$ to $0.28s$. To figure out the dominant factor of the increased latency, we use the tool *ssh-ping* to measure the latency through the SSH tunnel, which is around $0.22s$. Therefore, most of the latency is caused by the redirection procedure, while the latency caused by the interception procedure

of the PHP extension is less than 0.06s.

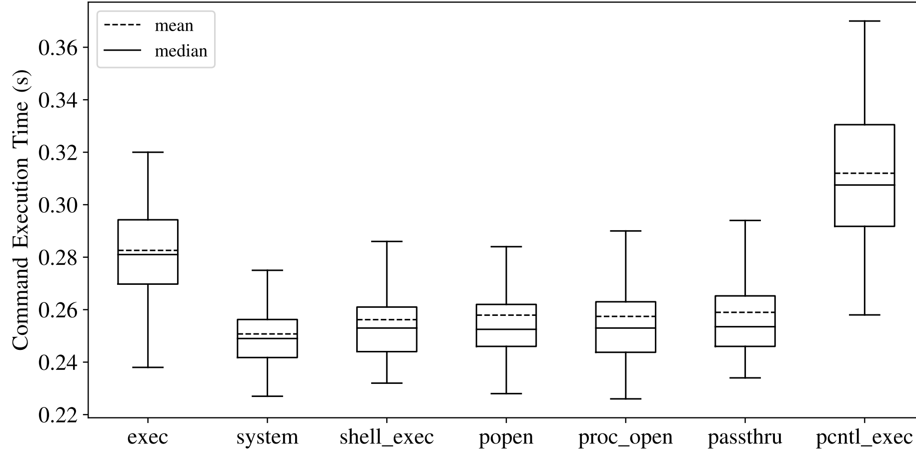


Figure 2.4: Latency for Redirected Built-in Function

Then we measure the response time of different commands for redirection. Here we test the function `system()` with five common system commands. The result is shown in Figure 2.5. The command *ps* has a higher average latency among these five commands because it produces more results. The latency also increases with the size of execution results.

From the measurement above, we can conclude that the response time is mainly affected by the redirection (distance and data size). According to Jakob Nielsen [22], 10s is the time that users consider the web page is over slow and decide to leave. We believe the attacker is deliberate, so it may be more patient than the average user. We still consider the latency should be under 1 minute; otherwise, a simple task is unable to be completed. Hence, the honeypot operator should consider these factors in practical deployment to avoid wearing out the attacker's patience.

Meanwhile, the honeypot operator should prevent the attacker from obtaining local command execution response time on the target device (i.e., front-end honeypot). If the

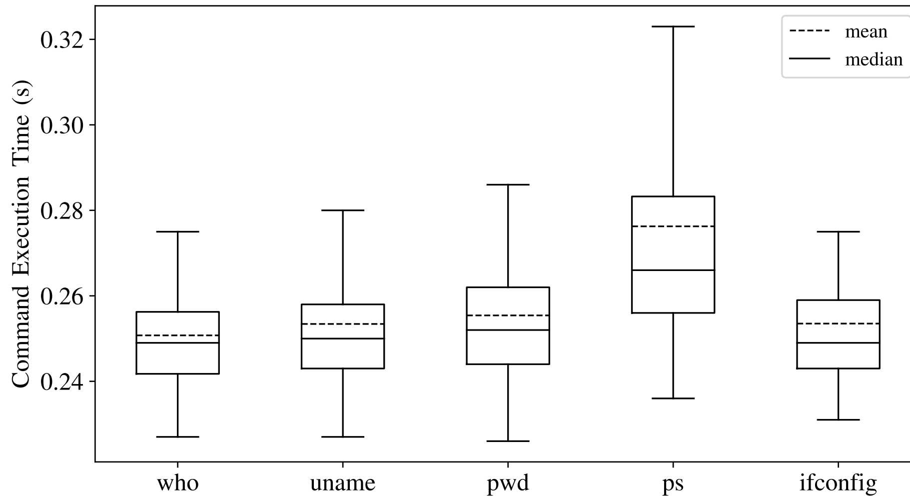


Figure 2.5: Latency for Redirected Commands (`system()`)

local execution response time is obtainable, the attacker can compare it with the HoneyBog to detect the hybrid honeypot architecture. One possible method for obtaining the local execution response time is to construct a system shell on the front-end honeypot via exploiting other vulnerabilities on the web server. In this case, the honeypot operator needs to intercept and redirect commands from the system shell to the back end. We discuss the existing solution in Section 2.7.

Synchronization Latency

In our testbed, the file transmission speed is around 1.5 MB/s. In our experiment webshell dataset, the largest webshell is more than 200 KB, while most webshells are around 10 KB. For a 10 KB webshell, the synchronization latency is around 0.006s. For a 200 KB webshell, the synchronization latency is around 0.13s. For these small-size files, the latency is less than that of command redirection. Therefore, the attacker can always access these uploaded webshells via injected commands.

System Overhead

To measure the system overhead, we focus on our interceptor and synchronizer in the front-end honeypot. We measure the overhead under the 32 concurrent command *who* injection requests for function `system()`. The interceptor costs around 23 MB of memory. Its CPU usage increases by less than 1%. To synchronize our experiment webshell dataset, the running synchronizer causes around 7% CPU usage and costs around 11 MB of memory. The experimental results show that HoneyBog can be readily deployed in the real world without excessive performance overhead.

2.7 Discussion

Liability

HoneyBog aims to provide a hybrid honeypot framework for the honeypot operator to monitor and analyze the attack based on webshell-based command injection in the back-end honeypot. The back-end honeypot can be any honeypot production. Besides, because the back-end network is completely controlled by honeypot operators, they can tune the isolation level of the network via configuring the firewall and IPS system. It is optional to allow the attacker to establish a new system shell from the honeypot. Similarly, if the attacker attacks other targets on the Internet, the honeypot operator can decide whether intercept the malicious payloads or terminate this connection according to the practical requirements.

Database

Besides the command injection, one common usage of webshell is to access the database. In a practical deployment, the honeypot operator may want to support the database for the web server. As a hybrid honeypot architecture, the database can be deployed in the back-end network to facilitate management. The web server in the front end connects the database remotely. Also, the operator can use the database honeypot [23] to monitor and

analyze specific attacks on it.

System Shell

Although we only consider webshell-based command injection in our threat model, the attacker may also exploit the vulnerability (e.g., CVE-2019-0211 [24]) of the web server itself to launch the remote code injection attack. The attacker can establish an independent system shell on the web server. Then the attacker can directly execute the OS commands on the front end of HoneyBog. To defend against this attack, we can embed the solution proposed in [20] into our system, which can intercept and forward commands of the system shell (e.g., Bash Shell) to be executed in the back end.

2.8 Related Work

2.8.1 Hybrid Honeypot

In 2004, Jiang et al. [25] propose Collapsar, a hybrid architecture to improve the coverage of the high-interaction honeypot using a number of redirectors in different production networks to tunnel attack traffic to the remote high-interaction honeypots. In the same year, Bailey et al. [26] also propose a hybrid architecture, which deploys the low-interaction honeypot as the front end to redirect interesting attack traffic to the high-interaction honeypot for detailed investigation. Potemkin [27] constructs a large-scale hybrid honeypot system by leveraging the virtual machine, late binding, and aggressive memory sharing techniques. Fan et al. [28] propose to use SDN to support a transparent TCP connection hand-off mechanism for the hybrid honeypot system. HoneyDOC [29] designs an SDN-based all-around hybrid honeypot system. The redirected traffic is not limited to TCP. Sun et al. [20] propose a hybrid decoy architecture that redirects injected commands of system shell (e.g., Bash Shell, PowerShell) to be executed in the remote back-end honeypot. Their work cannot intercept the webshell-based command injection, since those commands are executed by the interpreted language. Compare to Sun's work, our work focuses on the webshell-based

command injection.

2.8.2 PHP Webshell

Due to its ubiquitous presence in server environments, PHP is the typical language of choice to construct webshell [5]. The mitigation methods on webshell mainly go in two directions: detection and prevention. To detect PHP webshell, the traditional methods are using signature-based intrusion detection systems (IDS). With the development of machine learning, researchers have tried to apply various machine learning techniques to the webshell detection [9–15]. To prevent webshell, the core idea is to confine the privilege and the available resources for the webshell script, which requires the administrator to configure the web server correctly. Bulekov et al. [30] focus on the interpreted language itself. They propose to create a system-call allowlist for each PHP application to prevent remote code execution. Their solution does not consider the built-in functions (i.e., `system()`), which could execute shell commands.

2.9 Chapter Summary

This chapter demonstrates HoneyBog, a hybrid honeypot framework for constructing a high-fidelity environment against the webshell-based command injection. By redirecting the injected commands from the front end to the back end for execution, HoneyBog can deploy lightweight front-end honeypots containing a web server and basic supporting resources in any product network flexibly. The back-end honeypots can be centralized in a constrained environment to facilitate management. In addition, we synchronize the website files in bi-direction. It improves the fidelity of HoneyBog by allowing the attacker to exchange files with the back-end honeypot. We implement a prototype of HoneyBog over the Apache web server and PHP. The experiments show that HoneyBog can effectively intercept and redirect the injected commands from webshells with low overhead.

Chapter 3: Consistency is All I Ask: Attacks and Countermeasures on the Network Context of Distributed Honeypots

3.1 Introduction

Over the past 20 years, the honeypot technique has demonstrated its unique value in identifying unauthorized or illicit use of computer resources [1]. By monitoring and analyzing suspicious activities in honeypot systems, administrators have a better chance to detect and constrain attacks. To confront massive sophisticated attackers (e.g., APT attackers [31]), today's honeypot solutions pursue the scalability in the deployment and the capability to provide a detailed understanding of the attack. The local honeypot implementations are limited by their architecture and could achieve only one of the two.

The typical goal of honeypot operators target detecting early reconnaissance and reducing the attacker's dwell time. The detection is necessary while reducing the attacker's dwell time could cause the honeypot operators cannot identify all the tactics, techniques, and procedures (TTP) used in the sophisticated attack. It is not sufficient to simply eliminate APT threats. Therefore, in some cases, the honeypot operators need to increase the attacker's dwell time in order to feed fake information to APT attackers and observe the complete sophisticated attack. The key point of increasing the attacker's dwell time is to prevent the attacker from being aware of the honeypot environment.

Based on the distinct characteristics of each standalone honeypot, the attacker can leverage fingerprinting-based detection techniques [3,32] to identify the honeypots and avoid them. Besides, although the distributed honeypot systems can provide more flexible deployment and centralized management, they also introduce extra network context inconsistency between the front ends and the back ends, which could be exploited by attackers to identify

the distributed architecture of the deployed honeypot system. For instance, the back-end network may have different network configurations (e.g., host number, IP address, services) and network traffic than the front-end network. Moreover, most distributed honeypot systems [20, 25, 27, 33–35] allow the front end to reply to initial simple requests (e.g., ICMP) of the attacker and forward more sophisticated requests to the back-end honeypot; however, this division in the function may be exploited by an attacker, since those requests replied in the front end will not be logged in the access history of the back-end honeypot.

Since the fingerprinting of the standalone honeypot has been well studied in previous work [3, 32], we focus on the inconsistent network context caused by the architecture of the distributed honeypot systems in this work. We perform a systematic study on existing typical distributed honeypot systems to check if they are robust under a new anti-honeypot technique called *network consistency cross-checking attack* (NC3).

This attack consists of two stages, namely, *pre-exploitation reconnaissance* and *post-exploitation reconnaissance*. Before initiating an attack on a target machine, the attacker collects information (artifacts) related to the target machine during the pre-exploitation reconnaissance stage. The attacker may transition to the post-exploitation reconnaissance stage to retrieve a more comprehensive collection of artifacts on the target machine after successful exploitation efforts. We summarize nine types of network context artifacts that may be collected in the two aforementioned stages that attackers may leverage to identify distributed honeypot systems by means of inconsistencies in the data set. We study two types of attackers based on their locations, including an *insider* who resides in the same local subnetwork as the front end, and a *semi-insider* who is in the same local network but in a different subnetwork as the front end. Our theoretical analysis and experimental results show that they can utilize NC3 attacks to successfully identify all popular distributed honeypot systems.

To protect those distributed honeypots, whose goal is to increase the attacker's dwell time, from the NC3 attack, we develop a countermeasure called HoneyPortal to remove the network context inconsistency between the pre-exploitation and post-exploitation stages.

Precisely, the overall approach projects the remote back-end honeypot into the front-end network using a transparent traffic redirection strategy, where the front end redirects all incoming traffic (i.e., layer 2 and above) to the back end and vice versa. A back-end controller is responsible for forwarding the packets between the front end and the back end by using the redirection channel. For instance, when attackers break into the back-end honeypot and generate network traffic (flows) between the attacker's host machine and the back-end honeypot, HoneyPortal redirects all the outbound traffic of the back-end honeypot back to the front end. The front end then forwards the traffic to the attacker's host machine and forwards the attacker's response traffic to the back-end honeypot. By integrating the front end and the back end as one logic honeypot, we can eliminate the network context inconsistency introduced by the distributed honeypot systems.

Since the attacker may compromise the back-end honeypot with the root privilege, it is critical to ensure that our countermeasure, particularly, the back-end controller, cannot be detected by the attacker. We propose to use XDP (eXpress Data Path) [36] to isolate the back-end controller from back-end honeypots as XDP can transparently intercept and redirect the traffic flow of honeypots without interference. When we implement the back-end honeypot using a virtual machine, the XDP program runs in its corresponding virtual NIC in the back-end controller. When we implement the back-end honeypot using a physical machine, the XDP program runs on its corresponding NIC in the back-end controller.

We implement a prototype of HoneyPortal that supports the running of the front end on a physical machine and the deployment of the back-end honeypot on the virtual machine. We build a testbed for evaluating the effectiveness and performance of our HoneyPortal system. The experimental results show that HoneyPortal can successfully defeat NC3 attacks with low overhead. When packet sending rates reach 700 kpps, the front end only costs less than 1% CPU utilization in our testbed. Our back-end controller costs about 8% CPU utilization.

In summary, we make the following contributions:

- We perform a systematic study on one anti-honeypot technique called the network

context cross-checking (NC3) attack that exploits various network context information to detect distributed honeypots. We summarize nine types of artifacts that attackers could exploit to defeat all existing distributed honeypot systems.

- We propose a countermeasure to defeat NC3 attacks against the distributed honeypots. By projecting the remote back-end honeypot into the local network as a host machine, our defense can successfully remove the network context inconsistency introduced by the distributed honeypot systems.
- We implement a prototype of HoneyPortal. The experimental results in our testbed show that HoneyPortal can effectively defeat NC3 attack with low overhead.

3.2 Related Work

3.2.1 Distributed Honeypot

A distributed honeypot [26,37] typically consists of three parts, i.e., a front end, a back end, and a redirection channel. Its design aims to (1) combine the benefits of the lightweight low-interaction honeypots and the highly believable high-interaction honeypots and (2) achieve centralized management and control of the high-interaction honeypots. The front end is deployed in the protected network and may simulate several services and applications to directly respond to simple network reconnaissance requests. Meanwhile, more complex and suspicious network requests are redirected to a remote back-end network via the redirection channel. A back-end controller controls and logs network traffic between the front-end honeypot and the back-end honeypot. The back end is capable of generating more believable responses by running application services on a full-fledged operating system.

Bailey et al. [26] propose a globally distributed architecture, which deploys the low-interaction honeypot as the front end to redirect interesting attack traffic to the high-interaction honeypot for detailed attack information investigation. This architecture provides distributed deployment and centralized management. Jiang et al. [25] propose Collapsar, a distributed architecture to improve the coverage of the high-interaction honeypot

using a number of redirectors in different production networks to tunnel attack traffic to the remote high-interaction honeypots. Potemkin [27] follows a similar architecture, exploiting late binding and aggressive memory sharing to accommodate more VM-based honeypots. Hyhoneydv6 [34] provides a distributed architecture in the IPv6 address space.

Emerging techniques such as software-defined networking (SDN) and moving target defense (MTD) have been integrated to extend the capability of hybrid honeypots. Fan et al. [28] propose an SDN-based distributed honeypot to support a transparent TCP connection hand-off mechanism. Honeyprox [38] enhances the current honeynet design via SDN to defer the internal propagation of attackers within the honeynet. HoneyDOC [29] leverages SDN to support all-around honeypot, which provides transparent traffic redirection and high-quality attack capture. Artail et al. [33] introduce an adaptable distributed honeypot dynamically changing with the organizational network. Sun et al. [20] propose a distributed decoy architecture that may dynamically shift the network attack surface. Chovancova et al. [35] propose an autonomous distributed honeypot that is capable of adapting to environmental changes in real-time.

3.2.2 Anti-Honeypot Mechanism

As the weapon for attackers to identify honeypots, anti-honeypot mechanisms can be divided into three categories: *network-level fingerprinting*, *system-level fingerprinting*, and *operation-level fingerprinting* [3, 32].

Network-level honeypot fingerprinting focuses on detecting the discrepancy in network activities and network latency. Since honeypot systems may provide only a limited number of services and/or have constrained interactions with other real hosts and the Internet, attackers may observe fewer network activities of honeypots compared to those of legitimate hosts [4,39]. When the honeypot system is deployed in a separate network, it may introduce higher network latency than other hosts in the protected local network, and attackers can compare the latency differences to detect honeypots [40, 41]. System-level honeypot fingerprinting is based on the information collected from operating systems and applications.

A honeypot and a real host may have different system artifacts [41–43], such as operating system flags, running processes, volatile user information, files, and installed programs. By analyzing this information, attackers can infer if a host is in a honeypot. Operation-level honeypot fingerprinting is to measure if attackers can use the victim host to communicate with other hosts [44]. Since most honeypot systems are located in a constrained environment to prevent compromised honeypots from attacking other real hosts, these operation limitations may be notable honeypot indicators.

3.3 Threat Model

In this work, we assume that the attacker intends to compromise the legitimate hosts inside the intranet and maintains long-term access, while the honeypot operator intends to increase the attacker's dwell time as long as possible to observe all the TTP used in the attack. The attacker is able to conduct network reconnaissance to collect network and system artifacts after fully controlling a computer. Based on the location of the attacker, we classify them into three types, namely, *insider*, *semi-insider*, and *external attacker*.

The first type of attackers is the insiders located in the front-end network, the same subnetwork as the front-end honeypot. If there is an internal firewall in the subnetwork, we assume that the insider and the target host are in the same network segmentation. They are capable of scanning the hosts in this subnetwork or network segmentation. If the subnetwork is constructed with a hub, an insider can monitor all the network flows by setting the NIC into promiscuous mode. In practice, more and more networks are constructed with switches directly. In this case, the insider needs to conduct an ARP spoofing attack to redirect all the traffic to its own machine to monitor all the network flows. With the deployment of static ARP and PC firewall, the insiders may not always conduct the ARP spoofing attack successfully. Therefore, we assume the insider can only observe the network flows between itself and the target host.

The second type is the semi-insiders, who are located in the intranet but with different subnetworks or network segmentations as the front-end honeypot. Due to the security

policies of the internal firewall, the semi-insiders may only scan partial hosts in the front-end network. Both insiders and semi-insiders could be disgruntled employees or attackers who have successfully broken into the intranet via social engineering or unfixed security vulnerabilities.

The third type is the external attackers who are located outside the intranet. Due to the strict firewall policy or DMZ configuration, the external attackers can only interact with the hosts open to the public and usually have quite limited knowledge about the intranet. They cannot conduct the network context cross-checking, thus we don't discuss the external attacker in this work. However, in some specific cases [45], external attackers may collect partial information about the intranet across the gateway. Here we treat this kind of external attacker as semi-insiders.

After breaking into the back-end honeypot, we assume that attackers may gain the root privilege. They can access the logs of network services, monitor the network flow in the honeypot, and conduct one more round of reconnaissance from the back-end honeypot. Then they perform a cross-checking with the scanning results before attacking the honeypot.

3.4 Network Context Cross-Checking Attacks

Figure 3.1 shows the entire attack procedure of NC3, which consists of three steps, namely, *pre-exploitation reconnaissance*, *post-exploitation reconnaissance*, and *network context cross-checking*. Before attacking the target machine (i.e., the front-end honeypot), the attacker may collect the artifacts related to the target machine via traditional network reconnaissance [46]. We call this stage of attack as *Pre-exploitation Reconnaissance*. After breaking into the target machine (or the honeypot), the attacker can reissue network reconnaissance from the compromised target machine to collect network and system artifacts. We call this stage of attack as *Post-exploitation Reconnaissance*. Meanwhile, the attacker can reissue the network reconnaissance from their host machine once more. Finally, the attacker will compare the three sets of artifacts collected in two stages and detect potential inconsistency to check if the target machine is a honeypot.

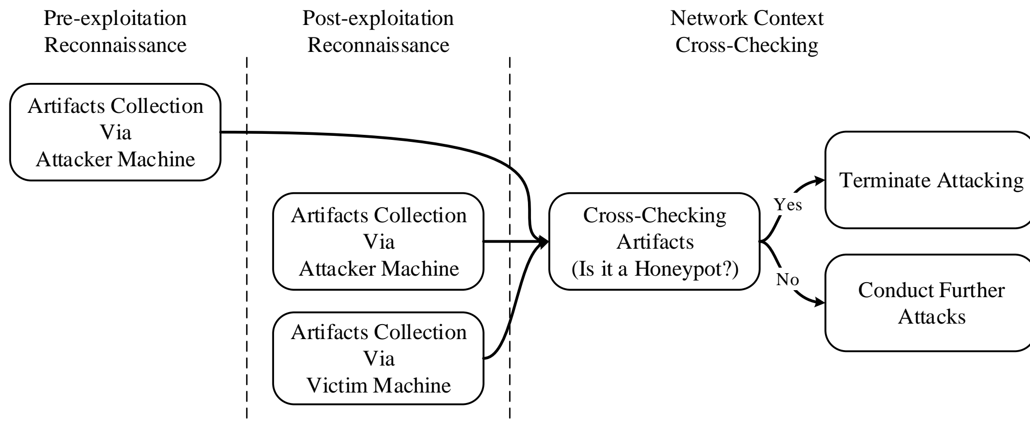


Figure 3.1: The Attack Procedure of NC3

3.4.1 Pre-exploitation Reconnaissance

Suppose the attacker resides on a host machine and is interested in attacking the victim machine. By running network scanning tools (e.g., Nmap[1], Nessus[2]), the insider may first inquire about the total number of live hosts in the local network. The insider can collect the victim machine's network address (IP and MAC addresses), OS information (i.e., OS type and version), open network ports (i.e., TCP or UDP ports), and service information (i.e., application types and versions). Similarly, the insider may collect the artifacts of another legitimate machine in the same subnetwork. The insider can also use network diagnostic tools (e.g., Traceroute) to collect the routing information (i.e., the outermost router in the path) from its host to an external machine on the Internet. All the interactions between the insider and the victim machine are recorded as the access history (e.g., SSH, HTTP, FTP, etc.). The insider can obtain the access history manually by using the packet introspect tools (e.g., Wireshark[3], Tcpdump[4]) to record the timestamp and metadata of each packet. Also, the insider can measure the link latency between the host and the victim machine using the *ping* command.

[1] https://nmap.org/
[2] https://www.tenable.com/products/nessus
[3] https://www.wireshark.org/
[4] http://www.tcpdump.org/

Due to the security policies on the internal firewall, the semi-insider may only scan a small portion of hosts in the target subnetwork thereby retrieving fewer artifacts than an insider. Also, since the semi-insider locates in a different subnetwork, the routing information is unable considered as a criterion to perform the cross checking.

3.4.2 Post-exploitation Reconnaissance

In this stage, the attacker may collect artifacts from both the victim machine and the attacker's host machine respectively. From the victim machine, the insider can collect the number of hosts in the local network and its own network address, OS information, open ports, and service information. It can also collect another legitimate host's network addresses, OS information, open ports, and service information. Moreover, the insider can record the routing path to an external server. By filtering the log of network services, the insider may extract the past access history that should have happened in the pre-exploitation stage. Normally, legitimate hosts log their network activities for troubleshooting. The insider can expect to obtain these logs from a real host. The insider can measure the link latency from the victim node to its host machine. The insider can also generate network traffic (e.g., ARP, UDP, TCP, and ICMP flow) between its host machine and the victim machine and then observe the network flow on the victim machine. From the attacker's host machine, the insider has the ability to recollect similar artifacts as those in the pre-exploitation stage. Additionally, the insider can observe the network flow on its host machine. The insider can observe these packets on both the victim machine and its own machine with the packet introspect tools (e.g., Wireshark, Tcpdump) in real-time.

Similar to insiders, the semi-insiders can collect the artifacts from the victim machine, while quantifiable differences limit the collection to only the victim machine and other visible hosts in the pre-exploitation stage. Also, due to the internal firewalls, the traffic flow between the semi-insider and the victim may differ when observing from both sides. From the view of the victim machine, the insider can discover more hosts in the target subnetwork. However, for NC3 attack, the semi-insider only needs to collect the artifacts

related to the victim machine and visible hosts in the pre-exploitation stage.

3.4.3 Artifacts Cross-Checking

In this stage, the attacker can cross-check the artifacts collected in the pre-exploitation stage and post-exploitation stage to check if the target machine is a honeypot. For the nine types of artifacts collected in three sets (i.e., pre-exploitation artifacts from the host machine, post-exploitation artifacts from the victim machine, and post-exploitation artifacts from the host machine), the attacker checks the consistency of artifacts in the same type of artifacts from different sets. As the insiders and semi-insiders have different capabilities and confidence in collecting reliable artifacts, they may adopt different decision policies on identifying potentially distributed honeypots.

Insider

The insider can directly interact with the victim machine, thus it can use all the nine artifacts to conduct NC3 attacks. Among these artifacts, the insider defines four types of consistency criteria. First, for the number of hosts, network addresses, open ports, services, routing information, and access history, consistency means the artifacts should be the same. Second, for the OS information, since the remotely obtained OS information may not be as accurate as the one collected on the local host, the consistency means that the artifacts obtained from the attacker's host machine should be the subset of the artifacts obtained from the local host. Third, for the link latency, considering the uncertainty of network transmission and request handling, the attacker can use T-test to compare them. If the *p-value* is greater than a given threshold (e.g., 0.05), the attacker can consider the link latency is consistent. Fourth, the network flows between the insider host machine and the victim machine are considered consistent when the network flows observed on the insider's host machine match the network flows observed on the victim machine.

Semi-insider

Due to the existence of internal firewalls, some network traffic may be filtered, and only partial live hosts are visible from the outside of the victim's subnetwork. Therefore, the semi-insider adopts a different decision policy. First, the number of hosts observed from the attacker machine stage should be less than or equal to the number of hosts observed from the victim machine. Second, for the network addresses and the access history of visible hosts, consistency means they should be the same. Third, the judgments for consistency on the link latency and OS information of visible hosts are the same as the insider's scenario. Fourth, some open ports and services are not visible from the attacker's machine. Therefore, those artifacts observed from the attacker's host machine are subsets of the artifacts observed from the victim machine. Similarly, considering some network traffic may not pass the internal firewall, the attacker cannot observe all outbound traffic sent from the victim machine. Moreover, subnetworks may have different access control privileges in the intranet. For example, the semi-insider may access some external servers, while the hosts in the target subnetwork cannot do it. Also, the victim machine and the attacker's host machine may have different routes to the Internet. It is similar to the isolation mechanism of the honeypot system. Therefore, when detecting distributed honeypots, the semi-insider relies more on cross-checking the consistency of network address, OS information, access history, open ports, services, and link latency.

3.4.4 Attack Effectiveness

Based on our threat model, we examine whether existing distributed honeypot systems can provide a stateful environment to the attacker under NC3 attack. Among these systems, we only obtain the source code of Sun et al. [20] to test, while we analyze others based on their technique details described in the papers. The examination results are summarized in Table 3.1, which shows the usefulness of each type of artifact by two types of attackers in identifying each specific distributed honeypot system.

Table 3.1: The Effectiveness of NC3 Attacks against Distributed Honeypot Systems

Distributed Honeypot System	Network Context Consistency																	
	Number of Hosts		Network Address		OS Information		Open Ports		Services		Route		Access History		Link Latency		Network Flow	
	I	II	I	II	I	II	I	II	I	II	I	II	I	II	I	II	I	II
Collapsar [25]	●	●	○	○	○	○	●	◐	●	◐	●	◐	●	●	●	●	○	◐
Potemkin [27]	○	◐	○	○	○	○	●	◐	●	◐	○	◐	●	●	●	●	○	◐
Artail et al. [33]	○	◐	○	○	●	●	●	◐	●	◐	●	◐	○	○	○	○	○	◐
Hyhoneydv6 [34]	○	◐	●	●	○	○	●	◐	●	◐	●	◐	○	○	○	○	○	◐
Chovancová et al. [35]	○	◐	○	○	●	●	○	◐	○	◐	●	◐	○	○	○	○	○	◐
Sun et al. [20]	○	◐	●	●	●	●	○	◐	○	◐	○	◐	○	○	●	●	○	◐
Jafarian et al. [2]	○	◐	○	○	○	○	○	◐	○	◐	○	◐	○	○	●	●	○	◐
HoneyPortal	●	●	●	●	●	●	●	◐	●	◐	●	◐	●	●	●	●	●	◐

1. For each artifact, column I indicates an insider; II indicates a semi-insider.
2. Level of consistency: ● Consistent; ○ Inconsistent; ◐ Uncertain (depending on the network and firewall configuration).

Insider

We first assume that all these distributed systems are deployed inside the subnetwork and suffered attacks from the insider.

Number of Hosts: Collapsar [25] allows network flow initiated by the back-end honeypot to return to the front-end network. As a result, the insider can continue to interact with the same network from the back-end honeypot during the post-exploitation stage. Its number of hosts doesn't change. Other distributed honeypot systems put the back-end honeypot in an isolated network environment from any legitimate hosts due to the containment requirements, and they don't duplicate the same number of decoy hosts in the back-end network. Thus, the numbers of hosts are different in two separated local networks when scanning from the attacker's host machine and the compromised victim machine respectively.

Network Address: Only Hyhoneydv6 [34] and Sun et al. [20] mentioned that they will use the same network address for both the front-end and back-end honeypot, while others assign different network addresses.

OS Information: Artail et al. [33], Chovancová et al. [35] and Sun et al. [20] mentioned that they use the same OS version in both the front-end and back-end honeypot. Others deploy different OSes in the back-end honeypot.

Open Ports & Services: For those combining the low-interaction honeypot and the high-interaction honeypot [2, 20, 35], the front-end honeypot may not provide the same

open ports and services as the back-end honeypot. Particularly, when the attacker is able to create new open ports or services on the back-end honeypot, the front-end honeypot may not be well synchronized to conduct the same operations.

Access Route: Potemkin [27], Sun et al. [20] and Jafarian et al. [2] redirect the outbound traffic initiated by the back-end honeypot to other places for further analysis. They could have different routes to the Internet, particularly, with different default gateways. Others don't have this functionality so their access routes remain the same to the insider.

Access History: Only Collapsar [25] and Potemkin [27] allow the insider to interact with the back-end honeypot directly during the pre-exploitation stage. These interactions can be logged in the back-end honeypot. For others, the insider can detect the absence of previous interaction history easily.

Link Latency: Similar to the access history, the insider interacts with the back-end honeypot in Collapsar [25] and Potemkin [27] during the whole exploitation stage. Besides, Sun et al. [20] and Jafarian et al. [2] only redirect network flows of specific protocols to the back-end honeypot, while other requests are replied by the front-end honeypot. There is no latency change for them. Artail et al. [33], Hyhoneydv6 [34] and Chovancová et al. [35] redirect the attacker's connections from the front end to the back end after exploitation. The change of response speed and location can both cause the difference in link delay to the insider.

Network Flow: The network flows may deviate between the back-end honeypot and the attacker's machine. As honeypot systems [25, 33–35] handle specific protocols (e.g., ARP, ICMP) in the front-end honeypot to speed up the response, the insider cannot receive them in the back-end honeypot. Since the containment policy prevents the insider from sending the packets from the back-end honeypot to other legitimate hosts, that traffic only exists in the back-end network [20, 27].

Semi-insider

When these distributed honeypot solutions are deployed in isolated subnetworks and protected by internal firewalls, some outbound network flows of the victim host may be blocked from the semi-insiders. Therefore, the semi-insider cannot rely too much on the inconsistency in the network flow and routing information. Meanwhile, the compromised victim machine still can interact with more hosts in the Intranet and scan more open ports and services other than the attacker's host machine. As shown in Table 3.1, the semi-insider can still detect all these distributed honeypot solutions by checking the inconsistency from network address, OS information, access history, and link latency.

3.5 HoneyPortal: The Countermeasure

Since it is difficult to simply rely on firewall configurations to defeat NC3 attacks, particularly, those conducted by the insiders, we develop a countermeasure called HoneyPortal.

3.5.1 System Design

Figure 3.2 shows the overall architecture of HoneyPortal, which consists of three main components: a *front end* residing in the protected local network, a *back end* in the honeypot service provider's local network, and a *redirection channel* connecting the front end and the back end. The front end is responsible for redirecting all network traffic to the back end. A controller on the back end forwards the packets between the front end and the corresponding back-end honeypot. A redirection channel is established to transparently connect the front end to the back end.

When the attacker sends a packet to the front end, the packet passes through the redirection channel and reaches the back-end honeypot. All the network interactions between the back-end honeypot and other hosts (e.g., an external server on the Internet or another legitimate host in the front end's local network) are redirected back to the front end, which conducts the network interactions on behalf of the back end and then sends the results to

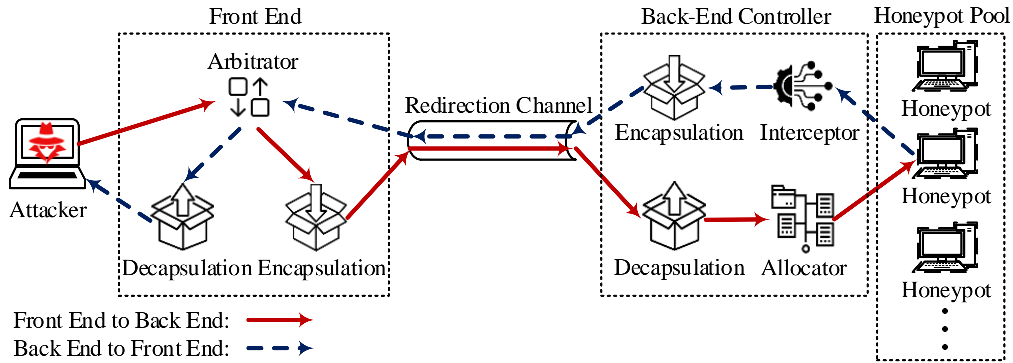


Figure 3.2: The Architecture of HoneyPortal System

the back end.

Front End

The network admins can easily deploy the small-sized front end into the protected local network as the entrance of the honeypot system. When the front end receives any network packets from the front-end network, the arbitrator module in the front end forwards all those packets, including broadcast traffic (e.g., ARP requests), to the encapsulation module, which adds a new header to the packets and send them to the back-end network via the redirection channel. In other words, the front end does not directly respond to any network requests received from the front-end network. When the front end receives a packet from the back end via the redirection channel, the arbitrator forwards the packet to the decapsulation module to remove the outer header added by the back-end controller. After that, the front end sends out the packet into the front-end network.

Back End

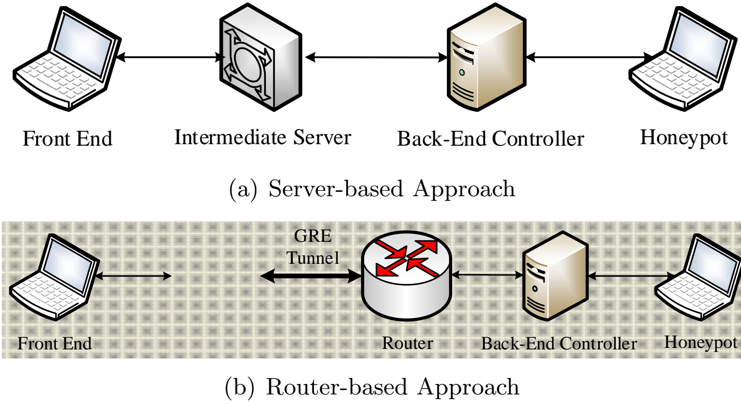
The back end includes a back-end controller and a pool of high-interaction honeypots. When the packets arrive from the front end, the back-end controller processes them in the decapsulation module to remove the outer packet headers and then forwards them to the corresponding honeypot. The interceptor module is responsible for capturing the outbound

traffic of the honeypots and forwarding them to the encapsulation module, which sends the encapsulated packets to the front end. In this way, the honeypot can only communicate with the outside via the front end.

To maintain the consistency of network context, the back-end honeypot needs to share the same IP and MAC addresses with its corresponding front end. The back-end controller maintains a configuration table inside the allocator module to establish a one-to-one mapping between the front end and the back-end honeypot. This table contains the IP and MAC addresses of the front-end host with the corresponding NIC or vNIC of the back-end controller. Honeypots can be deployed as virtual or physical machines, but they should be well isolated from each other.

Redirection Channel

A redirection channel is established to transparently bridge the front end and the back end. We have two options for HoneyPortal to redirect the traffic, namely, *server-based* approach and *router-based* approach. The server-based approach uses an intermediate server to forward the network packets between the front-end network and the back-end network, as shown in Figure 3.3(a). This approach is flexible in deployment. Honeypot operators can deploy the intermediate server in any place on the Internet, which is accessible to both the front end and the back end. The trade-off is that link latency will increase since the packets need to go through an extra device. The router-based approach needs to deploy or configure one edge router in both the front-end and back-end networks, and the GRE (Generic Routing Encapsulation) tunnel can be established between the two edge routers, as shown in Figure 3.3(b). If the operators can control and configure the edge routers, they can choose the router-based approach. End-to-end packets exchange can decrease the transmission latency.



(a) Server-based Approach

(b) Router-based Approach

Figure 3.3: Two Options of the Redirection Channel

3.5.2 Implementation of HoneyPortal

We implement a prototype of HoneyPortal using the eXpress Data Path (XDP) [47] technique. The XDP program can attach a lower-level hook inside the kernel. The hook is implemented by the NIC driver, inside the ingress traffic processing function (NAPI Poll method) before an skb is allocated for the current packet. By processing the packets inside the XDP program, we can reduce the processing delay. Meanwhile, since the XDP program is not running inside the honeypot, we keep the processing stealthy to the constrained attackers. We implement the front end on a physical machine and the back end on a virtual machine.

Front End

We implement the front end on top of an XDP-enabled NIC (see Figure 3.4), Intel X550-T2 network adapter with ixgbe 5.1.0-k driver. Specifically, we bind the XDP program to the XDP-enabled NIC within *Native* mode, since the native mode not only reduces the time of waiting for the memory allocation but also avoids extra packet duplication costs. By hooking the NAPI Poll method (i.e., `ixgbe_poll()`) of the NIC driver after the DMA of the buffer descriptor, the intercepted packets can be processed directly before the skb allocation. After encapsulation/decapsulation in the kernel space, we leverage the XDP_TX

action to send out the processed packets from the same NIC. Since the processed packets do not enter the network stack, we can accelerate the packet processing.

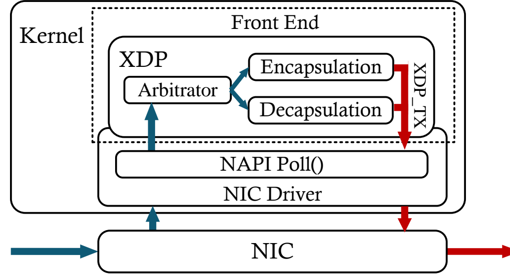


Figure 3.4: Implementation of the Front End

Back End

We implement two prototypes of the back end on the virtual and physical machines, respectively.

Virtual machine-based back end. We implement the back-end controller in the host OS and run multiple back-end honeypots in separate VMs, as shown in Figure 3.5. The setup of the physical machine for the back-end controller is the same as that for the front end. For the inbound traffic, the back-end controller leverages the AF_XDP socket to process the packets intercepted by XDP in the user space through zero_copy. The XDP R module intercepts packets from the NAPI Poll method of the NIC driver and filters all UDP packets from the front end in the kernel space. Then, the receiver app decapsulates original front-end packets and looks up the configuration table to seek the entrance of the target virtual machine in the user space. Next, we create AF_RAW socket S_i to inject front-end packets into vNIC $veth_i$ through the network stack. Finally, these packets are sent to the virtual machine via the veth pair.

For the current version XDP, all the devices must attach an XDP program using its

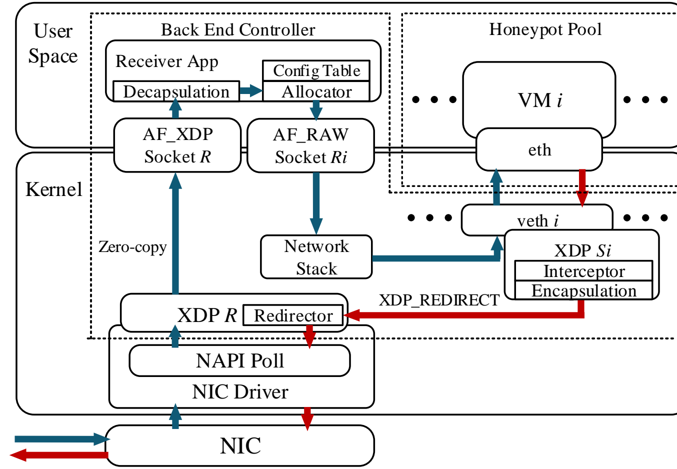


Figure 3.5: Implementation of the Virtual Machine-based Back End

default redirector module (`xdp_do_redirect()`) to receive packets redirected by other XDP programs. Though XDP supports XDP_REDIRECT action to deliver packets between NICs and vNICs, it will inevitably leave traces inside the honeypot. Particularly, the attacker can easily find out the existence of the XDP receiver module inside the honeypot. To solve this problem, we use the raw socket to inject packets into a veth pair in the user space. The veth pair virtual NIC (vNIC) is the corresponding entry of the honeypot. The XDP program running on the veth pair vNIC with the skb model could intercept all packets from the virtual machine with no performance improvement. We create AF_RAW socket S_i to inject front-end packets into vNIC $veth_i$ through the network stack. Finally, these packets are sent to the virtual machine via the veth pair.

For outbound traffic, the XDP sender program S_i listens on $veth_i$ to intercept all network packets out of the honeypot before entering the virtual bridge. The XDP program running on the veth pair vNIC with the skb model can intercept all packets from the virtual machine with no performance improvement. In this way, each honeypot is well isolated to prevent the attacker from committing an ARP-spoofing attack in the back end. In practice, the VM doesn't calculate the checksum (IP, TCP, and UDP) of the packet. The checksum computation is done in the physical NIC, which is called Checksum Offloading [48]. Since the

XDP program intercepts the packets in the vNIC, these packets haven't obtained the correct checksum. Hence, we re-implement the checksum computation functions in the XDP sender program to generate a correct checksum for them. Then, the module encapsulates these packets with new Ethernet, IP, and UDP headers. Finally, it leverages XDP_REDIRECT action to send these encapsulated UDP packets to the XDP program R, whose redirector module forwards them to the corresponding front-end hosts.

The VM-based honeypots can be easily extended to support container-based honeypots for better scalabilities [49]. The main configuration difference is that the veth pair is available for containers by default, but we need to set up a veth pair to act as the entrance for the VM-based honeypot.

Physical machine-based back end. We also implement the back end on two bare-metal physical machines, as shown in Figure 3.6. The back-end controller is deployed on one physical machine with two XDP-enabled physical NICs, where the NIC1 is responsible for connecting the front end, and the NIC2 acts as the entrance of the honeypot on another physical machine. Compared to the VM-based solution, the physical machine-based solution can provide better isolation, defeat existing anti-virtualization technology, and achieve a higher fidelity environment. Meanwhile, the VM-based solution can achieve better scalability and support more flexible management. To avoid the potential ARP-spoofing attack in the back end, the NIC2 only connects one physical machine-based honeypot. Both NIC1 and NIC2 are XDP-enabled. Intel X550T network adapter with ixgbe 5.1.0-k driver. The physical machine honeypot has an Intel I217-LM network adapter with e1000e 3.2.6-k driver. The OS is Ubuntu 18.04 LTS with kernel version 5.3.0.

For the inbound traffic, the XDP receiver program R hooked on NIC1 decapsulates the UDP packets received from the front end and the allocator module forwards them to the NIC of the corresponding honeypot through XDP_REDIRECT action. For the outbound traffic, the XDP sender program S hooked on NIC2 intercepts all packets received from the back-end honeypot, encapsulates all packets with new headers, and sends new packets to corresponding the front end through NIC1. Specifically, we leverage the XDP program

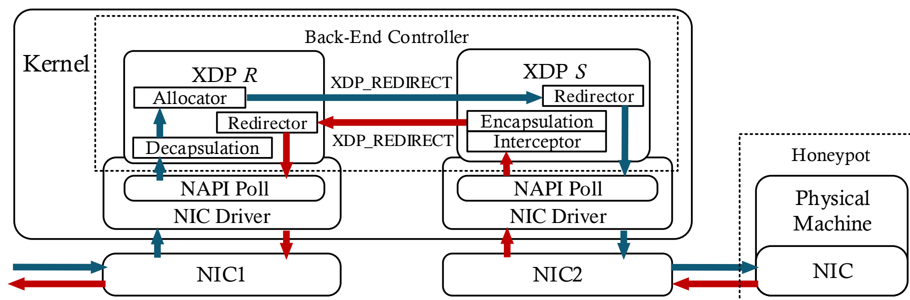


Figure 3.6: Implementation of the Physical Machine-based Back End

with XDP_REDIRECT action to build a high-speed connection between NIC1 and NIC2. Both the receiver R and sender S programs have a default redirector module to forward the packets from XDP_REDIRECT action. Both the receiver R and sender S programs are running in the *Native* mode, since they are binding to the physical NICs.

Redirection Channel

For the server-based approach, we implement the intermediate server on one physical machine. The intermediate server uses a redirection module based on the NAPT (Network Address Port Translation) [50] technique to allocate specific ports (4000 and 5000) for the front end and back end to connect. When the traffic arrives at the pre-configured ports of the server, the intermediate server redirects it to the corresponding device. Since the packets from the back-end honeypot are encapsulated with outer headers, we inspect the internal headers of the packets to build the map between front-end hosts and back-end honeypots. In the router-based approach, we set up two Cisco C921-4P as edge routers connecting the front-end and back-end networks in GRE mode. We configure the routing table of the routers to only allow the font-end hosts to connect the back-end controller, while other legitimate hosts cannot discover the back end.

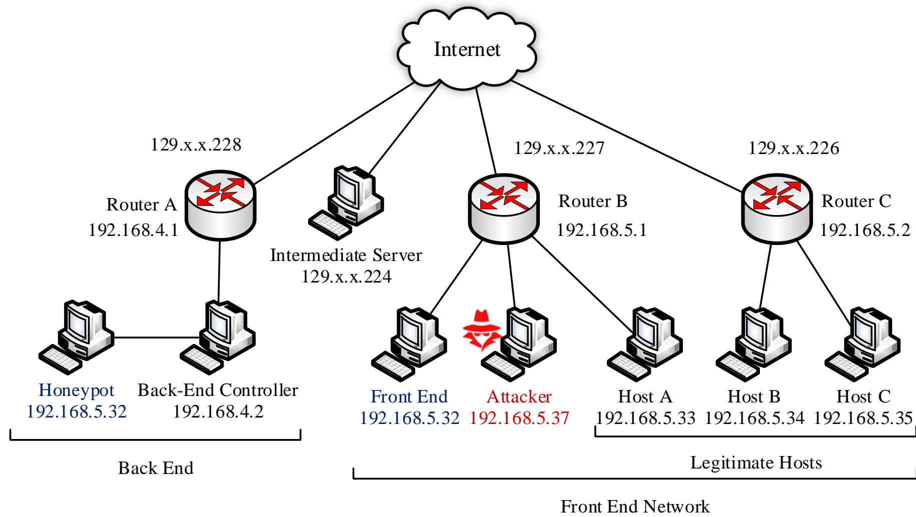


Figure 3.7: Testbed for HoneyPortal

3.6 Evaluation

3.6.1 Experiment Testbed

We built a testbed to evaluate the effectiveness of HoneyPortal using the VM-based back end, as shown in Figure 3.7. The protected local network consists of two subnetworks to simulate the scenarios that allow users to remotely connect to the local network. All host machines, including the intermediate server, use Ubuntu 18.04 LTS (kernel version 5.3.0) on Dell Precision 7810 desktops with Intel Xeon(R) E5-2620 CPU @ 2.40GHz and 16 GB memory. Both the front end and the back-end controller are equipped with XDP-enabled Intel X550-T2 network adapters. In the back end, we use VMWare 15.5 to create VM-based honeypots with 2 CPU cores and 2 GB memory. We use Cisco C921-4P as edge routers A, B, and C to connect the front-end network and the back-end network to the Internet. These routers support the maximal 1 Gigabit Ethernet (GbE) link.

3.6.2 Defense Effectiveness

In the pre-exploitation stage, the attacker first scans the entire local network (192.168.5.0/24) to locate the potential victim machine. Via *Nmap* 7.60, the attacker can scan the front end to collect the information including network address (IP: 192.168.5.32, MAC: A0:36:9F:28:64:6E), OS information (Linux), open ports (22, 80, etc.), and services (FTP, SSH, Apache HTTP, MySQL, etc.), as shown in Figure 3.8(a). Also, the attacker can obtain the access paths to outside servers (e.g., google.com, bing.com) via *traceroute* 2.1.0. Meanwhile, the attacker uses the *tcpdump* 4.9.3 to record the timestamp and protocol type of connections to the front end as the access history.

```
Starting Nmap 7.60 ( https://nmap.org ) at 2020-08-11 15:22 EDT
Nmap scan report for 192.168.5.32
Host is up (0.0021s latency).
Not shown: 995 closed ports
PORT     STATE SERVICE   VERSION
21/tcp   open  ftp       vsftpd 3.0.3
22/tcp   open  ssh       OpenSSH 8.2p1 Ubuntu 4ubuntu0.1 (Ubuntu Linux; protocol 2.0)
80/tcp   open  http      Apache httpd 2.4.41 ((Ubuntu))
443/tcp  open  ssl/https VMware Workstation SOAP API 15.5.6
3306/tcp open  mysql     MySQL (unauthorized)
MAC Address: A0:36:9F:28:61:44 (Intel Corporate)
Service Info: OSs: Unix, Linux; CPE: cpe:/o:linux:linux_kernel, cpe:/o:vmware:Workstation:15.5
.6
```

(a) Pre-exploitation Stage

```
:~$ ifconfig enp0s25
enp0s25: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.5.32  netmask 255.255.255.0  broadcast 192.168.5.255
        inet6 fe80::7719:293d:c4ef:de82  prefixlen 64  scopeid 0x20<link>
        ether a0:36:9f:28:61:44  txqueuelen 1000  (Ethernet)
        RX packets 48257  bytes 4950522 (4.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 40383  bytes 4219808 (4.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xfb200000-fb220000

:~$ uname
Linux
:~$ sudo netstat -tlpn |grep -E "ftp|apache|ssh|mysql|vmware"
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1331/sshd: /usr/sbi
tcp        0      0 0.0.0.0:902             0.0.0.0:*               LISTEN      189040/vmware-authd
tcp        0      0 0.0.0.0:3306            0.0.0.0:*               LISTEN      142589/mysqld
tcp6       0      0 :::80                   :::*                    LISTEN      1500/apache2
tcp6       0      0 :::21                   :::*                    LISTEN      1285/vsftpd
tcp6       0      0 :::22                   :::*                    LISTEN      1331/sshd: /usr/sbi
tcp6       0      0 :::33060                :::*                    LISTEN      142589/mysqld
tcp6       0      0 :::902                  :::*                    LISTEN      189040/vmware-authd
:~$ service --status-all | grep + | grep -E "ftp|apache|ssh|mysql|vmware"
 [ + ]  apache2
 [ + ]  mysql
 [ + ]  ssh
 [ + ]  vmware-workstation-server
 [ + ]  vsftpd
```

(b) Post-exploitation Stage

Figure 3.8: Network Recon Results

In the post-exploitation stage, after breaking into the back-end honeypot, the insider can redo the network reconnaissance from its host machine (192.168.5.37). On the back-end honeypot, the locally collected network information is shown in Figure 3.8(b). Our experimental results show that HoneyPortal can ensure that the collected information is consistent with those collected in the pre-exploitation stage. The attacker can also obtain the application and network related logs (e.g., */var/log/apache2/access.log*, */var/log/auth.log*), which record the interaction between the attacker and victim, to check if the pre-exploitation recon actions are conducted on the back-end honeypot. Our experiments show that HoneyPortal can ensure the log records matching to the historical network activities in the pre-exploitation stage.

The front end, back-end controller, and redirection channel are well protected from attackers. First, since the front end redirects all traffic from the front-end network to the back end, it is transparent to the attacker. Meanwhile, as a redirector, the front end only includes a simple XDP application hooked to the NIC driver to process the receiving packets in the kernel space. The XDP application only uses simple processing logic to parse packet header instead of payload. Thus, the attack surface of the front end is minimized. Second, similar to the front end, the back-end controller only uses the XDP technique to process the packet headers and forward the packets, so it also has a small attack surface. Since HoneyPortal does not make any modifications to the back-end honeypot, the security of the back-end honeypot relies on the adopted honeypot productions themselves. Third, for the server-based redirection channel, the intermediate server could be a public server that only provides UDP redirection service. Besides applying mature security mechanisms on the intermediate server, we could enable a white list on its firewall to filter unauthorized connections. Fourth, traditional anti-honeypot techniques only focus on the fingerprinting of honeypot production. Thus, these techniques can only be used to identify the honeypot production itself, not the transparent redirection procedure of HoneyPortal.

3.6.3 Processing Latency

We measure the network latency on both two types of back-end honeypot platforms. Figure 3.9 shows that our redirection framework introduces additional latency in all platforms when the payload sizes increase from 16 bytes to 1024 bytes. Compared to the router-based approach (GRE), the server-based approach requires the packets to go through an extra device (a.k.a. the intermediate server), so the link latency of the server-based approach is a little higher than the router-based approach. Also, the link latency increases with the payload size.

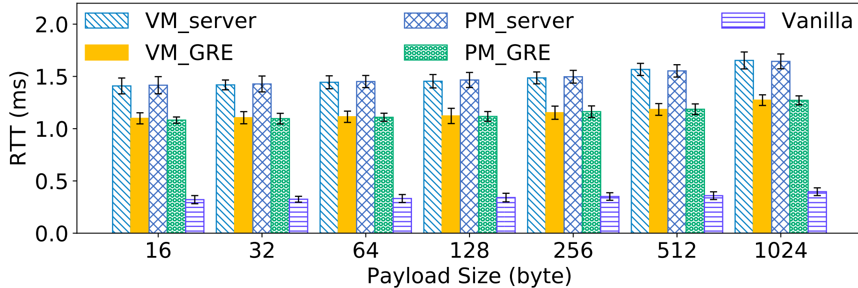


Figure 3.9: Link Latency (RTT) under Different Packet Sizes

To explore the main cause of increased delay, we break down the processing latency of each HoneyPortal component, as shown in Figure 3.10. The packet payload size is set to 64 bytes (default ICMP packet size in Linux system). Since AF_XDP requires cooperation in both kernel space and user space, the AF_XDP based back-end controller's processing latency is higher than the XDP based back-end controller. We can see the main contributor to the network delay is the link transmission delay, increasing along with the physical distance between the front end and the back end.

When deploying HoneyPortal in traditional local area networks that only consist of local computers and routers, the armored attacker may detect HoneyPortal by measuring the network latency differences if it can generate and observe the network flows between

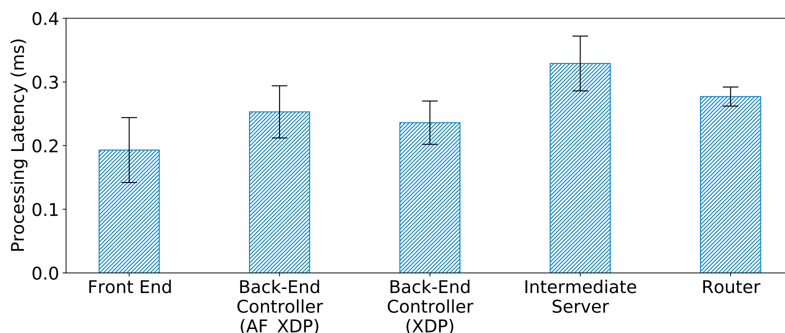


Figure 3.10: Processing Latency Breakdown of HoneyPortal

the target machine and the host machine. However, the deployment of virtual networks and the integration of cloud servers make it difficult for attackers to distinguish our honeypot systems from those legitimate remote host machines. First, large companies usually construct intercontinental virtual networks to connect remote locations with the help of MPLS [51] or SD-WAN [52], and the link latency varies significantly depending on the geographic distance. As a rule of thumb, 300 ms of network latency is technically acceptable to most business digital applications [53]. Second, an increasing number of companies have integrated public cloud services into their private networks [54]. Amazon has already provided a mature solution [55]. Therefore, when we deploy our intermediate server and the back end on the public cloud servers, the attacker cannot easily use the network latency to distinguish HoneyPortal, since other cloud-based servers may have the same amount of network latency.

3.6.4 System Overhead

We measure its CPU usage and memory consumption under the 700 kpps packet rates. For the front end, its loader in the user space costs lower than 0.1% CPU usage and around 2 MB memory. The kernel module of the front end costs around 8 KB memory. For the back-end controller, it costs around 8% CPU usage and 105 MB memory in the user space, while it costs less than 1% CPU usage and around 12 KB memory in the kernel space. The

experiment results show that HoneyPortal can be readily deployed in the real world without excessive performance overhead.

3.7 Discussion

Liability

HoneyPortal focuses on supporting distributed honeypot systems to defeat NC3 attacks, which enable insiders to detect the inconsistency of the network context before and after exploiting a victim machine. Although honeypot operators can deploy various isolation mechanisms (e.g., firewall, IPS system) in HoneyPortal based on their requirements, those constraints can be exploited by insiders to distinguish honeypots from real systems. Sophisticated insiders can always stay in the target network long enough to detect honeypots prior to launching an attack. In this case, a critical concern for the honeypot operator is to generate an alert as early as possible; even a compromised honeypot can be exploited to attack other legitimate hosts in the same local network. Hence, as a trade-off, the honeypot operator should not add any isolation mechanism between the front end and the back end. To minimize the impact of an attack on a network, the honeypot operator can configure the back-end honeypot to mirror network protections to that of other local (legitimate) hosts with an internal firewall. The internal firewall is responsible for detecting malicious activities and reduces the potential spread or propagation of a network attack to other networks in the environment thereby reducing the overall impact and risk of an attack. To defeat the semi-insider, the operator can deploy an isolation mechanism (e.g., firewall) between the front end and back end to detect and limit the spread of malicious payloads via the outbound traffic of the back-end honeypot. Section 3.4 indicated that semi-insiders may not be able to use the inconsistency in network flow to detect the distributed honeypot systems.

Scalability

In the stateless environment, the back-end honeypot only starts when the attack occurs. The sophisticated attacker (e.g., APT attacker) can easily detect the absence of previously regular interaction artifacts by continuously accessing the victim device over a long-term period. We intend to provide a stateful environment in HoneyPortal. Hence, we make a trade-off to keep the back-end honeypot running simultaneously with the front end.

Our prototype of HoneyPortal uses the XDP-enabled NIC and XDP techniques to reduce the packet processing overhead on both the front and back-end controller. However, the design of HoneyPortal framework supports the use of the NIC without XDP support. For example, the front end could be a software-based packet redirecting program instead of an XDP program to achieve the same goal. For the back end, the honeypot operators can deploy multiple honeypots on one physical machine. Therefore, HoneyPortal can smoothly integrate into existing honeypot solutions. The main performance bottleneck is the network bandwidth and available resources for the virtual machine in the back end.

3.8 Chapter Summary

In this chapter, we first present a new anti-honeypot technique called network context cross-checking (NC3) against the distributed honeypot systems. We show that all existing distributed honeypot systems can be detected by attackers using NC3. As a countermeasure, we propose HoneyPortal, a transparent attack redirection framework to eliminate the inconsistency of network context in distributed honeypot systems while leveraging XDP as a method to improve network performance. We implement a prototype of HoneyPortal to demonstrate that HoneyPortal can effectively defeat an NC3 attack with low overhead.

Chapter 4: Enhancing Malware Analysis Sandboxes with Emulated User Behavior

4.1 Introduction

Along with the increasing number of sophisticated malware that adopts packing and obfuscation techniques [56, 57], malware analysis sandbox systems have been widely used to help malware analysts provide the fine-grained dissection of malicious functionalities via monitoring malware's run-time behaviors [58–60]. Meanwhile, malware developers began to develop countermeasures to circumvent sandboxes [61]. For Example, malware can identify sandbox environments by checking system traits such as usernames, system settings, analysis instrumentation files, and installed drivers [62–65]. Moreover, advanced malware may evade sandboxes by performing timing attacks [66, 67], detecting CPU virtualization [68–70], checking process introspection indicators [71, 72], or even leveraging reverse Turing test [73, 74].

Researchers have proposed various mitigation approaches such as hiding environmental artifacts [75], binary modification [76], path exploration [77], state modification [78], heterogeneous analysis [79, 80], and bare-metal analysis [81] to minimize or eliminate discrepancies between malware analysis sandboxes and real systems. However, it remains a challenge to defeat a new anti-sandbox technique that leverages system artifacts (i.e., the registry entries, the event logs, the browsing histories, and the cached files) accumulated by normal user operations to distinguish sandbox environments from real systems [4].

To tackle the defect of lacking authentic system artifacts in the existing sandbox designs, one spontaneous approach is to construct the sandbox environments by directly cloning real user systems. However, this approach has some limitations [4]. First, the system artifacts from a real user system may contain the user's private information that needs to

be quarantined. It is challenging to thoroughly clean user private information and retain the most authentic system artifacts. Second, the system artifacts in the cloned system become outdated quickly without continuous user interactions. It is time-consuming to clone the latest real user system and then conduct data cleaning for each round of malware analysis.

We propose a new framework called *User Behavior Emulator (UBER)* that profiles authentic user activities to generate realistic system artifacts via user behavior emulation for the sandbox environments. Our design is based on one basic observation, namely, most malware is developed for mass attacks that do not aim at compromising a targeted computer. Therefore, it is plausible to profile a user behavior model using any normal authentic user and then simulate user activities according to an abstracted user behavior model. UBER consists of four components, namely, *computer usage collector*, *user profile generator*, *artifacts generator*, and *update scheduler*. The computer usage collector first leverages the system-event monitor technique to capture long-time computer usage information from real users. Then, the user profile generator performs statistical analysis on the collected usage information to construct a user behavior profile. Based on the user behavior profile, the artifacts generator continuously generates realistic system artifacts by emulating authentic user behavior via automation control techniques. Finally, the update scheduler periodically integrates the emulated system artifacts into the malware analysis sandbox to ensure "up-to-date" artifacts in the sandbox.

Instead of directly modeling the patterns of various low-level system artifacts, we choose to emulate the high-level user behavior, which is the source of artifact generation. It is difficult to generate complete and consistent system artifacts; however, we can derive the user behavior from the collected system artifacts, similar to the user profile in the intrusion detection field [82]. UBER collects comprehensive computer usage information including system events and application logs to construct user behavior profiles. To minimize privacy leakage, UBER only records the statistical characteristics (i.e., application usage times, file operations, and UI events) of the computer usage information. Moreover, UBER relies

on the statistical data from public websites (e.g., Alexa[1], Google Trends[2]) to profile web browser activities, such as accessing top sites, searching common terms.

We adopt a new deployment strategy in UBER to perform user behavior emulation within an isolated always-on system (continuously running to accumulate the user artifacts) and then copy this system to the malware analysis sandboxes on demand. This approach has three advantages over the solution that deploys the emulator directly in the malware analysis sandboxes. First, it can prevent the emulation processes from being exploited as an indicator for the evasion malware to identify sandboxes. In other words, it keeps our design much stealthier. Second, it can avoid enlarging the attack surface of the sandboxes, in contrast to emulating user behavior directly in sandbox environments. Third, it prevents the emulation processes from competing for system resources and interfering with the analysis results. UBER includes an update scheduler to ensure up-to-date artifacts in sandboxes by performing the copy process regularly or on-demand since the system artifacts copied into the sandbox environments will become obsolete without persistent user operations. Given that one malware analysis sandbox is usually rolled back to its initial state after each malware analysis [74], UBER replaces the initial state with the always-on emulated system containing the most up-to-date artifacts.

We implement a prototype of UBER based on python system-event monitor and automation control modules [83–87]. To evaluate the effectiveness of the generated artifacts of UBER, we deploy the artifacts generator on a virtual machine with a fresh (newly installed) Windows OS as a sandbox and manually operate the cloned fresh virtual machine as a "real" system simultaneously for comparison. After running these two systems for one month, we observe that both systems accumulate overall comparable amounts of system artifacts. We further explore the daily variation of artifacts generation and find that UBER is able to simulate the artifacts accumulation processes of real user systems. Finally, we leverage the state-of-the-art classifier provided by [4] to verify the authenticity of the system

[1] https://www.alexa.com/topsites
[2] https://trends.google.com/trends/

deploying with UBER. The experimental results indicate that UBER can effectively generate realistic artifacts via the emulation of real user operations to defeat the anti-sandbox technique that leverages system artifacts. Moreover, UBER can be extended to analyze targeted malware that aims to compromise a specific target machine by automatically profiling user behavior models representing specific user activities.

In summary, we make the following contributions:

- We design an emulation-based system called the User Behavior Emulators (UBER) to enhance malware analysis sandboxes by generating realistic system artifacts based on the automatically derived user profile model.
- We develop a new approach to create high-fidelity sandbox environments by emulating the high-level user behavior in an isolated always-on system and then stealthily merging this system to the malware analysis sandboxes.
- We implement a prototype of UBER and our experimental results demonstrate the effectiveness of UBER in defeating the sandbox evasion technique that exploits various system fingerprinting.

4.2 Threat Model

In this work, we focus on defeating the malware-used evasion techniques of checking authentic *system artifacts* generated in normal user activities [4,74]. In a real user system, normal users perform various actions such as browsing websites, editing office software, and coding computer programs. All those actions can generate accumulated system artifacts such as temporary files, DNS records, bookmarks, cookies, log entries, etc. In contrast, the sandbox environment only runs specific analysis software and lacks abundant system functions and user activities, thus missing those system artifacts. The lack of authentic system artifacts could become the fingerprinting of the sandbox. By checking the existence of those system artifacts, the malware can distinguish sandbox environments from real systems. After that,

malware may camouflage as a “benign” process and avoid being analyzed by the malware analysis sandbox mechanisms.

In the practical scenario, the attacker needs to increase the stealthiness of malware to increase their odds of survival. Minimizing the size and reducing resource consumption are common methods adopted in malware. In this case, we consider that the capability of malware in artifacts collection and analysis is limited. The malware can only analyze the collected artifacts with relatively simple logic and limited computation resource. Thus, we assume that the malware is unable to conduct an over-complicated analysis. For example, it can not analyze a large amount of real-time activities to check the meaningfulness of user behavior in semantics or recover the deleted data from the disk to detect the existence of the emulator.

Besides, we assume that the sandbox is a transparent malware analysis environment. The transparency indicates that there are no artifacts related to the sandbox production or introspection tools. Other sandbox evasion techniques such as checking instrumentation or introspection artifacts at runtime [61] are out of the scope of this work. Some sandboxes may allow the outbound connection to the Internet. In this case, the attacker can retrieve usage artifacts and then conduct an independent examination of the user’s behaviors outside the sandbox. Here we focus on the malware itself. Defeating the examination conducted by the real person is not within the scope of this work.

4.3 System Design

In this section, we first present the overview architecture of the UBER system, then we introduce the design of each component.

4.3.1 System Overview

The goal of UBER is to provide realistic system artifacts for the sandbox systems to counter malware that leverages system fingerprinting for detecting sandbox environments. The overall architecture of UBER is shown in Figure 4.1, which consists of four main components,

namely, *computer usage collector*, *user profile generator*, *artifacts generator*, and *update scheduler*, to emulate the interaction between users and the system according to an abstracted user behavior profile.

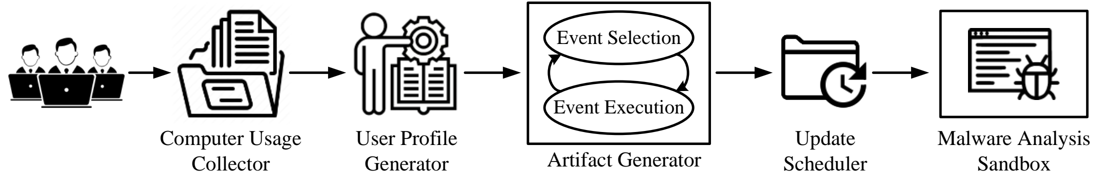


Figure 4.1: The Architecture of UBER System

The computer usage collector first gathers the raw usage data (e.g., the web access log) from the system used by real users. In other words, the raw usage data is generated in the user's daily activities. Then, the user profile generator performs statistical and correlation analysis on the raw data to profile user behaviors and then abstracts the user behavior profile as a configuration file to represent typical user activities. Next, the configuration file is fed into the artifacts generator, which consists of an event selector module and an event executor module to select suitable behavior events and execute these selected behavior events, respectively. The event selector and the event executor continuously generate realistic system artifacts in an execution environment. Finally, based on the practical requirements, the update scheduler periodically clones the newest emulated execution environment into the malware analysis framework as the sandbox.

Considering that the system artifacts are the result of continuous normal user operations, we choose to model user behavior from computer usage raw data and then emulate user activities based on the profile. It can achieve a better authenticity than directly modeling various complex patterns of system artifacts. Our emulation system is transparent to malware since it emulates user behavior in a secure environment and then copies the system artifacts into the malware analysis sandbox. Besides, the copy process is regularly

performed to keep the latest system artifacts. Before cloning the emulated system to the malware analysis sandbox, we remove the UBER emulator and clean its related components from the system, preventing the malware from using the UBER emulator as an indicator to detect the sandbox.

4.3.2 Computer Usage Collector

The computer usage collector runs on the machine of real users to gather the usage information and derive the user profile. To characterize user behavior, we collect computer usage information such as application interaction, file manipulation, network activities, and system activities. The application interaction indicates how the user interacts with the application in the system. UBER records the usage time of each application. The file manipulation indicates how the user manipulates the files in the system. UBER records the events of file access, creation, deletion, and modification. For the network activities, UBER collects the metadata of network connections, such as the number of connections, the interval of each connection. For the system activities, UBER counts their related trace in the system, such as running processes and system events.

To protect users' privacy, we only abstract the statistical usage metadata such as usage time. We do not record detailed sensitive information such as the Uniform Resource Locator (URL) access history [88] and content of edited documents [89].

4.3.3 User Profile Generator

After successfully collecting the usage information, we use the user profile generator to perform a statistical analysis and generate the user profile. The user behavior profile structure is shown in Figure 4.2. We define the behavior profile with the web and application activities as well as the file-related operations. The profile represents how a real user performs normal activities. After observing the computer usage of a real user, this component abstracts the user behavior profile as a configuration file. The configuration file consists of the duration of daily usage and the web/application activities with corresponding execution probabilities.

These probability values indicate the likelihood that a user would perform specific web and application activities.

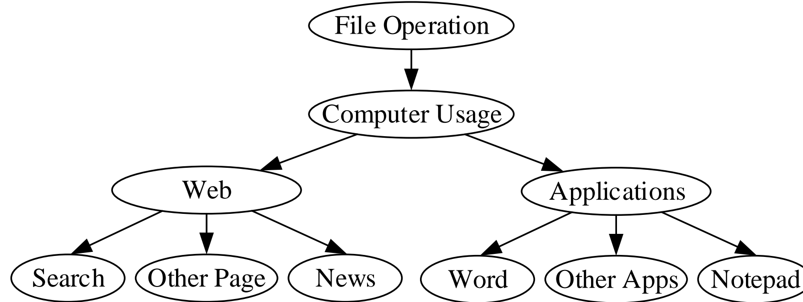


Figure 4.2: The Structure of User Behavior Profile

4.3.4 Artifacts Generator

Due to the complexity of the operating system (OS), it is difficult to ensure the consistency and completeness of the system artifacts in the timeline by simply generating them all at once. To solve this issue, we design the artifacts generator to continuously generate user activities and keep the system artifacts up-to-date. The generator consists of two modules: the event selector and the event executor. The event selector module selects specific activities based on the configuration files, and then the event executor module executes the predefined user actions. Hereinafter, we first introduce the identified artifacts that characterize the usage history of a real system, then present the workflow of the event selector and event executor module, and finally describe how these two modules generate the system artifacts.

System Artifacts

Sandboxes usually apply scoring mechanisms to determine if a file is likely malicious, while malware could also adopt similar mechanisms to determine if it is running in a sandbox

environment. UBER tries to complement these artifacts via user behavior emulation. We classify the artifacts generated by UBER into four main categories: *system*, *network*, *disk*, and *browser*.

System. It consists of both volatile artifacts and nonvolatile artifacts. The volatile artifacts require real-time system activities. For instance, the number of running processes is volatile and influenced by the active applications. Therefore, it could be a direct indicator of the usage of a system. In contrast, the nonvolatile artifacts represent the accumulated logs and configurations of the system. The system log contains plenty of information about the current state and historical usage of the system. It records various types of events, such as applications, security, setup, and system events of different risk ratings. Intuitively, the longer the system has been used, the more records that would be accumulated in the log. Since the system log varies among different systems, we generate usage patterns only by counting statistical properties such as the number of applications and system events, the number of event sources, and the time difference between the first and the last event. Besides the system logs, Windows has a hierarchical database (i.e., registry) that contains a wealth of information about the computer system and its historical usage. The kernel, device drivers, services, Security Accounts Manager, and user interfaces all use the registry to store data.

Disk. Various files including user-generated cached data and auto-generated temporary files may be created, modified, and deleted during the normal usage of a computer system. Files directly created by a real user can usually be found in the folder "Documents", "Desktop", "Pictures", "Downloads", etc. When there are limited or even no user-generated files in those disk directories, malware may use these temporary files as an indicator of the sandbox environment.

Network. Various network-related artifacts are associated with an authentic system. The amount, type, and variety of the network information in sandbox systems will be vastly different from those in real systems where users usually browse websites, execute various client applications, and install OS/application updates. Moreover, the Address Resolution

Protocol (ARP) and Domain Name System (DNS) cache entries as well as a few wireless SSIDs and active TCP connections can be indicators of the sandbox environment. The list of the previously downloaded Certificate Revocation Lists (CRLs) can also be a clue of accumulated network activities.

Browser. The web browser is one of the most commonly used applications in computer systems. Daily websites browsing can accumulate a large number of artifacts as historical usage. The rich diversity of its accumulated artifacts is worth treating it as a separate category. Since web browsers typically store webpages to the disk for speeding up future retrieval, it introduces a number of URL access histories and associated cookies. The presence of cookies with multiple timestamps usually indicates the regular browser usage of a real user. Specifically, cookies can be stored in separated files, a single SQLite file, or a format file. For example, cookies are separate files in Internet Explorer. The other indicators of browser usage also include artifacts such as the number of installed browsers, the total amount of visited URLs, the unique domains, and the saved bookmarks.

Event Selector & Event Executor

In UBER, the artifacts generator conducts groups of actions to generate related artifacts in the system. We define an event as a group of meaningful actions to be completed. The event selector picks up the event from the profile, and then the event executor executes it. The workflow of the event selector and event executor is illustrated in Figure 4.3.

Event Selector. This module makes decisions on which events will be performed according to the configuration files. Each event may trigger different subsequent events (sub-activities) to respond to the execution results of the primary event. The Probability & Randomization function within this module takes the probabilities and additional parameters provided by the configuration file to select the activities and the following sub-activities. This function mainly determines the event to be executed based on the usage profile that indicates which and how user actions will be performed.

The workflow of the Probability & Randomization function is presented in Algorithm 2.

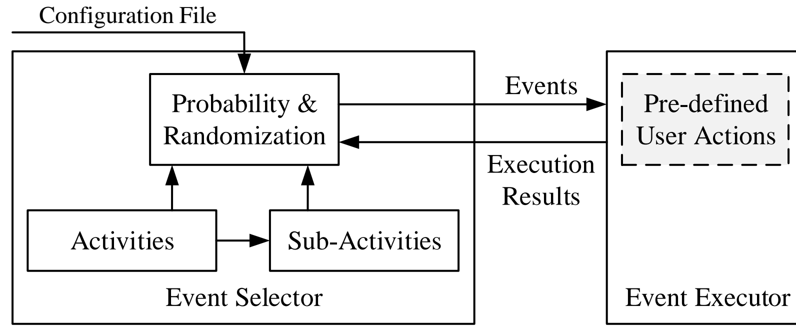


Figure 4.3: The Workflow of Event Selector & Event Executor

It takes the configuration files as inputs and outputs a list of activities that have been executed. First, it loads a configuration file to obtain the daily operation time, the probability, and the average duration of each activity type and sub-activity type. Second, it sets two schedulers that emulate user actions in the morning and the afternoon, respectively. The emulation times are set according to the statistic of daily computer usage. Third, it selects the activities and further sub-activities according to their corresponding probabilities. Finally, it selects specific new sub-activities based on the operation results of previous sub-activities. In the above process, we record the time for each sub-activity and activity to ensure the execution time does not exceed the limits defined in the configuration file.

Event Executor. This module is responsible for executing the events determined by the event selector. Each event performs the predefined user actions, where the event selector module provides all the needed variables to perform the corresponding actions. The outputs of these actions, such as results returned from a particular web search, are sent to the event selector module to make a new decision. Once the new decision is made, the event executor module executes the predefined actions.

Algorithm 2 Probability & Randomization Function

Input: $config$: The configuration file of user behavior profile
Output: E: A set of executed activities

```
1:  function MAIN( )
2:      Load configuration from config to U
3:      Initial E
4:      Scheduler EMULATOR(U.morning, E)
5:      Scheduler EMULATOR(U.afternoon, E)
6:      return E
7:  function EMULATOR(U, E)
8:      while sys.run < U.sys.time do
9:          Select type from U.types
10:         while type.run > U.type.time do
11:             Select other type' from U.types
12:         while type.run < U.type.time do
13:             Select subtype from U.type
14:             while subtype.run > U.subtype.time do
15:                 Select other subtype from U.type
16:             while subtype.run < U.subtype.time do
17:                 Load configuration from U_subtype
18:                 Perform operation of subtype
19:                 Get subtype operation results
20:                 E = E ∪ {subtype, subtype.run}
21:                 while results contains subtype' do
22:                     Load configuration from of U_subtype'
23:                     Perform operation of subtype'
24:                     Get subtype' operation results
25:                     E = E ∪ {subtype', subtype'.run}
26:                 Record operation time in subtype.run
27:             Record operation time in type.run
28:         Record operation time in sys.run
```

Artifacts Generation

In the following, we illustrate how the above two modules collaborate to generate realistic system artifacts. The event selector module is mainly responsible for selecting a user-like event based on the configuration file. Besides, this module is important in ensuring the validity of artifacts. The event executor module performs predefined user actions and generates system artifacts. The system artifacts generation procedure includes two stages, namely, *pre-emulation* and *emulation*.

In the pre-emulation stage, we manually install popular applications and browsers, which provide the necessary operation environments for both humans and UBER. For the file system, we create copies of non-sensitive files from the publicly accessed source (e.g., Internet) as decoy user's files. We need the pre-emulation stage for two reasons. The first reason is that the above artifacts remain unchanged after the long-term user operations and are thus not necessary for the daily update. The second reason is that completing these tasks in advance can lower the complexity of the emulation.

In the emulation stage, UBER starts to emulate user behavior. For the system artifacts, the event selector module first selects an application from the installed applications based on the user profile, and then the event executor module controls this application to perform pre-defined actions (e.g., mouse clicks, keyboard input). UBER also executes various background processes accordingly to emulate the realistic running computer. In this way, UBER could trigger various applications, enrich user actions, generate abundant system events, and increase log records. For the disk artifacts, the event executor module selects various types of files to create, modify and delete in the desktop folder according to the file operation distributions defined in the configuration file. UBER controls the browser to access various URLs and executes multiple client applications to generate network artifacts. The event executor module emulates the web activities for the browser artifacts, such as opening a browser, entering URLs, and browsing web content with user interactions (e.g., scrolling, clicking). When performing internet searching activities, the event selection module extracts effective terms from Google Trends and then glances at multiple search results

for emulating the "real" searching operations. When one website is accessed several times in one day, this website would be saved as a bookmark. In addition, the event executor module accesses news websites, mail websites, and websites from Alexa to cover a large amount of URL access for simulating the browser operations of a normal user. The event executor module randomly downloads specific files (e.g., .pdf, .docx, .xlsx) and script code (e.g., .java or .py) when performing the internet search to reflect the regular download habits.

4.3.5 Update Scheduler

The update scheduler is responsible for periodically duplicating the sandbox running the artifacts generator as the malware analysis sandbox. Due to practical requirements, the malware analyzers can schedule the duplication at a fixed cycle (e.g., every day) or just on demand. Because we need to emulate the real user usage on the OS, the emulation sandbox is only running during the work time (e.g., daytime) based on the profile during the artifacts generation procedure. The update scheduler can duplicate the emulation sandbox as the analysis sandbox when it is off. Once completed, the duplicated sandbox contains the up-to-date system artifacts generated by UBER.

The next task is to clean the duplicated sandbox by removing the residue of the artifacts generator from it. There are two reasons that the artifacts generator should not remain in the sandbox. First, it may compete for resources with the malware and interfere with the subsequent analysis. Second, the malware may evade the sandbox environments by identifying the existence of the artifacts generator.

Since UBER can work in virtual machines (VM), it conforms to the mainstream of malware dynamic analysis platforms [90]. The malware analyzers can import the cleaned sandbox copy (i.e., VM) into their desired malware analysis platform. Then they can execute the malware inside the sandbox and gather run-time information to derive its behaviors with the tools provided by the analysis platform. The malware analyzers can hinder malware from identifying the sandbox environment through system fingerprinting by using the up-to-date sandbox.

4.4 Prototype Implementation

We implement a prototype of UBER on Windows OS. The main reason is that the most effective sandbox detection solution [4] of checking the wear-and-tear artifacts was developed on Windows. Thus, we implement UBER on Windows to show that our sandbox system can defeat the detection mechanism proposed by [4]. The implementation architecture is shown in Figure 4.4. UBER first leverages python monitor modules including *watchdog* [83], *pynput* [84], *psutil* [91] and *pywin32* [85] to collect the computer usage of real users. Then, it performs statistical analysis to summarize the collected information into user behavior profiles that represent computer usage patterns. Next, based on the profiles, UBER applies python control modules Selenium [86], *pywin32*, and *pywinauto* [87] to implement the automatic control and execution of browser and other applications in a realistic way. Finally, the system with realistic artifacts is duplicated and imported into the malware analysis platform as a sandbox for malware analysis.

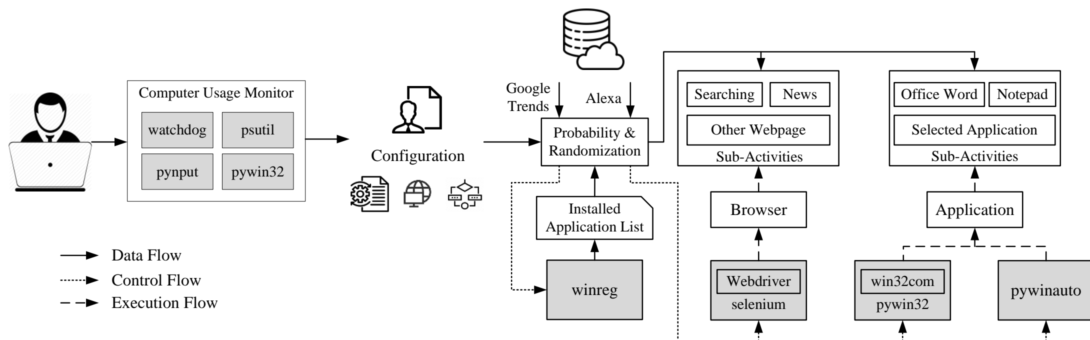


Figure 4.4: Implementation of UBER

We construct the real computer usage profiles based on our own daily operations. We collect computer application usage patterns and extract generalized user behavior profiles. We use these profiles as a proof of concept to present a real computer usage pattern for UBER. Taking the configuration file and the public data from Google Trends and Alexa sites

as inputs, UBER selects the browser or application activity and further selects sub-activities to emulate user operations realistically.

4.4.1 Computer Usage Collector

To characterize user behaviors, we implement python scripts based on monitor modules to collect comprehensive computer usage information. The implementation of this component is shown in Figure 4.5. We leverage the hook functions provided by python library *watchdog* and *pynput* to collect the file operations and UI events of user behaviors in real-time. We further log the UI events for every specific application and summarize the average UI operation speeds. We also adopt a polling-based method to regularly execute commands to record the background process and TCP connection information. The foreground window is obtained via Win32 API *win32gui.GetForegroundWindow()*. The foreground application usage is monitored via continuously polling the foreground window. Considering the delay in human operation, we set the polling interval for the foreground window to 0.1 seconds. To balance the efficiency and precision, we set the polling interval for background processes and TCP connections as 10 minutes and 1 minute, respectively. In our experimental platform (Intel Xeon(R) E5-2620 CPU @ 2.40GHz and 16 GB memory), the average memory consumption of the computer usage collector is 131.4 MB (0.8%), and the CPU utilization of it is below 0.1%. Therefore, this component can be deployed on a practical user computer without excessive performance overhead.

To collect the foreground application usage time, we design a polling algorithm, shown in Algorithm 3. This algorithm implements the application usage recording during the variation of the focused application. The function *get_fore_window()* represents getting currently focused application and its tab. The function *get_last_event()* represents retrieving the last input information and its intervals. First, the function obtains the current application and tab. Then, it leverages polling-based methods to detect the variation of the current application and tab. Next, it records usage time when the current application or tab changes, which means the application or tab is switched. Finally, the function stops the recording

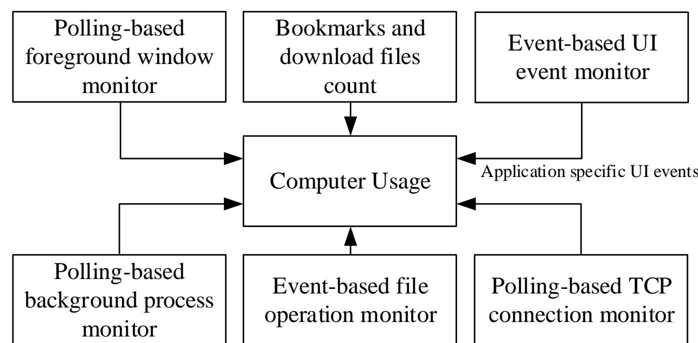


Figure 4.5: Implementation of Computer Usage Collector

of application usage when the system goes idle. The *idle* represents that the system has enough available resources after no user interaction for this interval. Typically, we set *idle* as 10 minutes.

4.4.2 User Profile Generator

We develop a Python script to generate the user profile from the collected usage metadata. The configuration file summarizes the overall user usage metadata and linked two JSON files (*backprocess_usage.json* and *application_usage_probability.json*) containing usage metadata for each process and application.

Figure 4.6 shows an example of the configuration file. The example represents an employee who regularly starts to work at about 8 am and 1 pm, and spends around 8 hours handling computer tasks every day. This configuration file indicates that this particular user spends approximately 34.3% of usage time performing web browsing among the whole computer usage time. This configuration file also contains the statistical results of file operations, as well as the quantitative distributions of TCP connections and background processes. There will be a large variance in the number of TCP connections. Specifically, the minimum value means that the computer is just booting and the maximum value means that the user opens a lot of web pages. The file *backprocess_usage.json* contains all the observed background processes and their corresponding usage probabilities. The

Algorithm 3 Foreground App Usage Collection

Input: *idle*: The idle time interval
Output: U: The list of application usage

```
1: /* Initializing */
2: Initial U
3: app_tab, app_id = get_fore_window()
4: cur_tab, cur_id = app_tab, app_id
5: lastinput, intervals = get_last_event()
6: flag_idle = False
7: while True do
8:     sleep(0.1)
9:     lastinput_tmp, intervals_tmp = get_last_event()
10:    if lastinput_tmp == lastinput then
11:        /* System Idle */
12:        if intervals_tmp > idle then
13:            if !flag_idle then
14:                Record app_tab, app_id usage time app_time
15:                Store {app_id, app_time} in U
16:            flag_idle = True
17:    else
18:        /* System Reactive */
19:        if flag_idle then
20:            app_id, app_tab = get_fore_window()
21:            cur_tab = app_tab; cur_id = app_id
22:        flag_idle = False
23:        lastinput = lastinput_t
24:        app_id, app_tab = get_fore_window()
25:        if app_id == cur_id then
26:            if app_tab != cur_tab then
27:                /* Tab Switch */
28:                Record app_tab usage time app_time
29:                cur_tab = app_tab
30:        else
31:            /* Application Switch */
32:            Record app_tab, app_id usage time tab_time
33:            Store {app_id, app_time} in U
34:            cur_tab = app_tab; cur_id = app_id
```

usage probability is the process usage time rate over the whole record period. The file *application_usage_probability.json* contains the information of the operated applications, the accessed websites, the search terms, the duration of browsing each site, etc. This file also defines the likelihood that a user performs sub-activities (e.g., reading news, using webmail, searching for information via a search engine) during a browsing session. Each sub-activity has additional configurations including a pre-defined popular website list and the selection probabilities of specific websites.

```
computer_usage:07:58:38
morning:{"min":"07:52:40","average":"08:51:26","max":"10:39:20"}
afternoon:{"min":"12:30:46","average":"13:23:57","max":"15:07:17"}

BackprocessType:backprocess_usage.json
BackprocessNum:{"min": 196, "average": 227, "max": 271}
ForeprocessTypes:Web,34.4|App,65.6
WebTypes:application_usage_probability.json,Web
AppTypes:application_usage_probability.json,App

TcpconnectionNum:{"min": 2, "average": 112, "max": 906}
DesktopOperation:{"Moved file": 45, "Modified file": 127, "Deleted file":
68, "Created file": 116, "Modified directory": 69, "Created directory": 5,
"Moved directory": 1}
```

Figure 4.6: An Example of Configuration File

As shown in Figure 4.6, the user typically turns on their computer at approximately 8:51, as early as 7:52, and as late as 10:39, in the morning. The startup time distribution is inputted as the computer's possible start times on that day. Each possible time value is then given a probability respective to its distance from the average start time of 8:51. Assume T_{ave}, T_{min}, and T_{max} represents the average, minimum, and maximum start time,

respectively. The probability $P(t)$ of each time value t is calculated by the formula:

$$P(t) = \begin{cases} \frac{T_{ave}-t}{T_{ave}-T_{min}}, & t \leq T_{ave} \\ \frac{t-T_{ave}}{T_{max}-T_{ave}}, & t > T_{ave} \end{cases} . \tag{4.1}$$

Then, UBER would select the time values based on calculated probability $P(t)$ to start the emulation.

To protect user privacy, we do not collect context-sensitive data of the real user. Hence, the configuration file does not include the specific URLs or textual documents from the real user. For web access, we define generic web activities according to the public stats of popular web activities.[3] The pre-defined popular websites are selected from Alexa, while the searching topics are collected from Google Trends. Alexa contains the most frequently visited websites, the time spent by a user in browsing these websites, and daily counts of browsing each site per visitor. We use the top 500 sites in the USA as the target websites to be accessed. The time spent by a user in browsing these websites is used as a time limit during the emulation. Google Trends provides daily trending items of what users are commonly searching on the web. We maintain a file to store hot topics in the USA over the past seven days. By using *pytrends* [92], we update the search topic every week. For textual document manipulation, we use the non-sensitive and publicly accessible documents collected from the Internet.

4.4.3 Artifacts Generator

The artifacts generator locates inside the virtual machine to emulate the user's behaviors. In Table 4.1, we list all the artifacts that are checked in paper [4] and note how they can be generated by UBER. The Windows registry is a part of the system category and we list the registry as a separate category. UBER completes these tasks at the pre-emulation stage for

[3]https://www.infoplease.com/science-health/internet-statistics-and-resources/most-popular-internet-activities

artifacts generated by setting values, installing software, and copying files.

For files in the Desktop and the Recycle Bin, UBER automatically copies various types of non-sensitive files from the Internet. UBER also copies the bookmarks and download files and removes personal information among these items. For artifacts including ARP entries, wireless SSIDs, auto-run programs, firewall rules, and connected USB devices, we develop python scripts to set these values directly.

Table 4.1: System Artifacts and Corresponding Emulation Strategies

Category	Artifacts	Description	Generation Strategy	
			Pre-emulation	Emulation
System	totalProcesses	# running processes	/	run various processes
	winupdt	# installed Windows updates	pre-install	regularly install updates
	sysevt	# system events	/	run various applications
	appevt	# application events	/	run various applications
	syssrc	# sources of system events	/	run various applications
	appsrc	# sources of application events	/	run various applications
	sysdiffdays	# days since the first system event	/	run various applications
	appdiffdays	# days since the first application event	/	run various applications
Disk	recycleBinSize	# bytes of the recycle bin	pre-copy	regularly delete directories & files
	recycleBinCount	# recycle bin files	pre-copy	regularly delete directories & files
	tempFilesSize	# bytes of temporary system files	/	access various urls
	tempFilesCount	# temporary system files	/	access various urls
	miniDumpSize	# bytes of process crash minidump files	/	run various applications
	miniDumpCount	# process crash minidump files	/	run various applications
	ThumbsFloderSize	# bytes of thumbnails	/	run various applications
	desktopFileCount	# desktop files	pre-copy	regularly construct directories & files
Network	ARPCacheEntries	# ARP cache entries	pre-set	/
	dnscacheEntries	# DNS cache entries	/	access various urls
	certUtilEntries	# URLs of downloaded CRLs	/	access various urls
	wirelessnetCount	# cached wireless SSIDs	pre-set	/
	tcpConnections	# active TCP connections	/	access various urls
Registry (System)	regSize	# bytes of the registry	pre-install	run various applications
	uninstallCount	# registered software uninstallers	pre-install	/
	autoRunCount	# autorun programs when startup	pre-set	/
	totalSharedDlls	# reference count	pre-copy	/
	totalAppPaths	# registered application paths	pre-install	/
	totalActiveSetup	# active setup application entries	pre-install	/
	orphanedCount	# leftover registry entries	pre-set	/
	totalMissingDlls	# registered missing DLLs	pre-copy	/
	usrassistCount	# UserAssist cache entries	/	run various applications
	shimCacheCount	# shim cache entries	/	run various applications
	MUICacheEntries	# MUI cache entries	/	run various applications
	FireruleCount	# Windows Firewall rules	pre-set	/
	deviceClsCount	# connected USB devices	pre-set	/
	USBStorCount	# connected USB storage devices	pre-set	/
Browser	browserNum	# installed browsers	pre-install	/
	uniqueURLs	# unique vistied URLs	/	access multiple urls
	totalTypeURLs	# typed URLs	/	access multiple urls
	totalCookies	# HTTP cookies	/	access multiple urls
	uniqueCookieDomains	# unique HTTP cookie domains	/	access multiple urls
	totalBookmarks	# bookmarks	pre-copy	regularly save bookmarks
	totalDownloadedFiles	# download files	pre-copy	regularly download files
	urlDiffDays	# days since the oldest to newest visited URL	/	access various urls
	cookieDiffDays	# days since the oldest to newest HTTP cookie	/	access various urls

Other artifacts can be accumulated from the user behavior emulation. UBER controls the execution of the browser and applications to generate artifacts. To simulate authentic user activities, UBER performs all user interactions including scrolling, mouse-clicking, and keyboard inputting, at the user's average operation speed. In addition, UBER emulates user behaviors based on the user profiles to ensure the execution of web and application activities in human-like habits and ways. We manually parse the commonly accessed websites, such as Google, Bing, CNN, BBC, etc., to help UBER extract meaningful URLs from the searching results and news pages. As an example, UBER selects the Internet Explorer browser to perform a Google search. It employs the method *find_elements_by_xpath("//*[@id='rso']/div/div//*/div[@class='rc']/div[1]/a")* provided by *Selenium* to extract effective Google searching results. Then, UBER selects one or multiple pages from results to glance. When it comes to conducting a Bing search, UBER leverages the method *find_elements_by_xpath("//*[@id='b_results']/li[@class='b_algo']/h2/a — //*[@id='b_results']/li[@class='b_algo']/div/div/h2/a")* to extract available search results. When one website is accessed multiple times, UBER is able to save bookmarks in the browser automatically. We also manually parse the GUI elements from popular applications such as Notepad and PDF reader to help UBER perform realistic operations. UBER extracts meaningful paragraphs from the pre-chosen websites and then performs text editing using applications such as Office Word and Notepad. For the file operations, UBER regularly creates, moves, modifies, or deletes files and directories according to the user profiles, and downloads various files from the pre-set websites. UBER looks up the installed application list and then selects one of them according to the usage patterns to determine which applications to be executed.

4.4.4 Update Scheduler

The update scheduler runs on the hypervisor to manipulate the VM via the VirtualBox python API. The VM duplication can be conducted during the system downtime. When there is a request to update the sandbox, it first ensures the VM is paused or turned off.

Then it duplicates the snapshot of the VM with the up-to-date artifacts. Considering that the new VM still contains the artifacts generator, we develop a system cleaning script based on PowerShell, which uninstalls those python modules for emulations and then destroys itself. Besides, to remove the related traces from Windows event logs, we also delete all the event records[4] (e.g., access, delete) related to UBER scripts before sandbox deployment. According to our threat model, the malware cannot recover the deleted files from the disk by itself. Hence, the malware cannot discover the existence of UBER scripts. The original VM is preserved to accumulate artifacts continuously.

In our implementation, we use a full-featured Windows VM. Its size is about 90 GB (including 80 GB virtual disk and 10 GB snapshot). The duplication procedure costs about 35 minutes. The time of the clean procedure is less than 5 minutes. For the daily update, the total 40 minutes effort is acceptable. The time cost is mainly influenced by the size of the VM image and disk read/write speed. If the analyzer uses some simplified VM with a smaller size, the time cost can be further reduced. We test our VM on the Cuckoo[5] analysis platform. After configuring the network and installing a Cuckoo agent, we successfully configured our VM as a Cuckoo sandbox. The whole procedure is less than 10 minutes.

4.5 Effectiveness Evaluation

We evaluate the defense effectiveness of UBER against user behavior artifacts detection. To the best of our knowledge, there is no publicly released malware that detects the sandbox with the technique introduced by paper [4]. Therefore, it is difficult to evaluate our solution using real malware. To overcome this challenge, we present the defense effectiveness of UBER in three aspects. First, we compare the long-term accumulated artifacts generated by the real user and UBER. Second, we compare their daily change of artifacts measured over a month. Third, we examine the authenticity of the generated artifacts with the verification tool provided by the authors of [4].

[4]https://github.com/3gstudent/Eventlogedit-evtx–Evolution
[5]https://cuckoosandbox.org/

4.5.1 Experiment Setup

We set up our testbed on a Dell Precision 7810 desktop with Intel Xeon(R) E5-2620 CPU @ 2.40GHz and 16 GB memory. The OS of the hypervisor is Ubuntu 18.04 LTS (kernel version 5.3.0). We use VirtualBox 6.0 to create the VMs (i.e., sandbox). The Windows 10 VMs are configured with 4 vCPUs and 4 GB memory.

To collect the real computer usage artifacts, we use an independent desktop and leverage it to deal with daily work every day (from Monday to Friday). The work time is about 8 hours each day. Since we have about one hour of lunchtime, we record two start timestamps for each day: one for the morning and one for the afternoon. Usually, the morning start time is around 8:00 am. The collection procedure for our experiment lasts one and a half months. For the volatile artifacts (e.g., running processes, active GUI interfaces, and network connections), the computer usage collector conducts the polling-based query in the background as described in Section 4.4.1. Its polling interval is 0.1 seconds. For other artifacts (e.g., downloaded files, browser bookmarks), which are relatively stable, our computer usage collector only queries once at the end of each day (work time). A summary of user artifacts will be produced every day. With these computer usage artifacts, we generate the user profile for the emulator. We also use them to compare with the artifacts generated by UBER. Since all the real computer usage data is from authors, IRB approval is not required in our experiments.

4.5.2 Accumulated Artifacts

We first want to demonstrate that the artifacts generated by UBER are indistinguishable from those in real systems. We deploy UBER on the VM with fresh (installed) OS, serving as the sandbox system. As for comparison, we manually operate another cloned VM that represents the real system with normal user operation. According to Table 4.1, we manually set the necessary operation environment before emulation. We install popular applications, copy common files, and set real firewall rules on the fresh OS. Then, we continuously run these machines for one month and measure all the accumulated artifacts mentioned in

Section 4.3.4 using an automatic artifacts collector based on PowerShell 5.0.

Table 4.2 shows the average values of each artifact for the three categories of systems. The "Baseline" column represents the artifact values of the fresh OS. The "Baseline+Real_User" column represents the artifact values of the system operated by the real user manually. The "Baseline+UBER" column represents the artifact values of the system deployed with UBER. The "Ratio" column represents the ratio of the "Baseline+UBER" column to the "Baseline+Real_User" column.

Table 4.2 shows a clear distinction in these artifacts between the fresh OS and the used system. The difference demonstrates that the identified artifacts can serve as strong indicators of OS without user activities. As mentioned in paper [4], the malware could exploit these artifacts to differentiate a sandbox system from the real system. Besides, we can see that the system with UBER has a comparable amount of system artifacts accumulated as used systems that we manually operate. On average, the system with UBER is able to generate 94.25% of the artifacts of real systems. Although there is a variation in the artifact values between these two systems, they both own realistic system artifacts. In particular, the values of "sysdiffdays", "appdiffdays", "urlDiffDays", "cookieDiffDays", and "ARPCacheEntries" in both systems are the same since they are created on the same day and in the same LAN network. The values of "wirelessnetCount", "unistallCount" and "FireruleCount", etc. are same for pre-emulation strategies. In addition, these artifacts are infrequently changed during daily activities. The values of "sysevt", "appevt", "usrassistCount", "MUICacheEntries", "uniqueCookieDomains" and "dnscacheEntries" may be quite different among these two systems. The difference is mainly caused by the fact that UBER performs different application activities and accesses different URLs from real users. However, the malware is unable to identify the sandbox environment only based on the different URL-access patterns because the access pattern varies significantly among different users. Therefore, UBER is effective in generating realistic artifacts in sandbox environments in terms of statistical characteristics.

Table 4.2: System Artifacts Comparison between Baseline, Baseline+Real_User, and Baseline+UBER

Artifacts	Baseline	Baseline+Real_User	Baseline+UBER	Ratio
totalProcesses	47	97	105	108.25%
winupdt	0	68	68	100%
sysevt	106	3.7k	3.6k	97.30%
appevt	123	9.8K	10K	102.41%
syssrc	34	56	52	82.96%
appsrc	12	34	36	105.88%
sysdiffdays	0	30	30	100%
appdiffdays	0	30	30	100%
recycleBinSize	0	102 MB	114.6MB	112.35%
recycleBinCount	0	163	153	93.86%
tempFilesSize	1.9M	41.2MB	39.7 MB	96.36%
tempFilesCount	21	79	62	78.48%
miniDumpSize	0	1.6MB	0.7MB	43.75%
miniDumpCount	0	10	6	60%
ThumbsFloderSize	3.5MB	54MB	46MB	95.19%
desktopFileCount	3	1690	1488	88.05%
ARPCacheEntries	7	14	14	100%
dnscacheEntries	0	149	141	94.63%
certUtilEntries	2	375	328	87.47%
wirelessnetCount	0	8	8	100%
tcpConnections	15	49	42	85.71%
regSize	30.2MB	58.2MB	57.8MB	99.31%
uninstallCount	13	81	81	100%
autoRunCount	3	11	11	100%
totalSharedDlls	24	76	76	100%
totalAppPaths	9	35	35	100%
totalActiveSetup	9	14	14	100%
orphanedCount	0	3	3	100%
totalMissingDlls	12	18	18	100%
usrassistCount	26	187	134	71.66%
shimCacheCount	3.2K	73k	71k	97.26%
MUICacheEntries	14	53	57	107.54%
FireruleCount	381	446	446	100%
deviceClsCount	3	24	24	100%
USBStorCount	0	5	5	100%
browserNum	1	3	3	100%
uniqueURLs	1	221	193	87.33%
totalTypeURLs	1	1.1K	975	88.64%
totalCookies	0	775	687	88.64%
uniqueCookieDomains	0	164	146	89.02%
totalBookmarks	0	363	379	104.41%
totalDownloadedFiles	0	21	17	80.95%
urlDiffDays	0	30	30	100%
cookieDiffDays	0	30	30	100%
Average	—	—	—	**94.25%**

4.5.3 Variation of Artifacts Generation

We conduct a second experiment to compare the average daily variation of sandbox with UBER and real systems over one month. We select the following artifacts including “sysevt”, “appevt”, “tempFileSize”, “regSize”, “MUICacheCount”, “usrassistCount”, “shimCacheCount”, “uniqueURLs” and “uniqueCookieDomains” from Table 4.1. These artifacts can represent the accumulation of user activities in the system. They are steadily accumulated each day and do not randomly change as artifacts “miniDumpSize”, “tcpConnections”, and “ThumbsFloderSize”, etc. Therefore, we measure the variation of the selected artifacts to show the progress of UBER in daily artifacts generation. The variation curve is shown in Figure 4.7, where the x-axis represents the observed days, and the y-axis represents the value of a specific artifact.

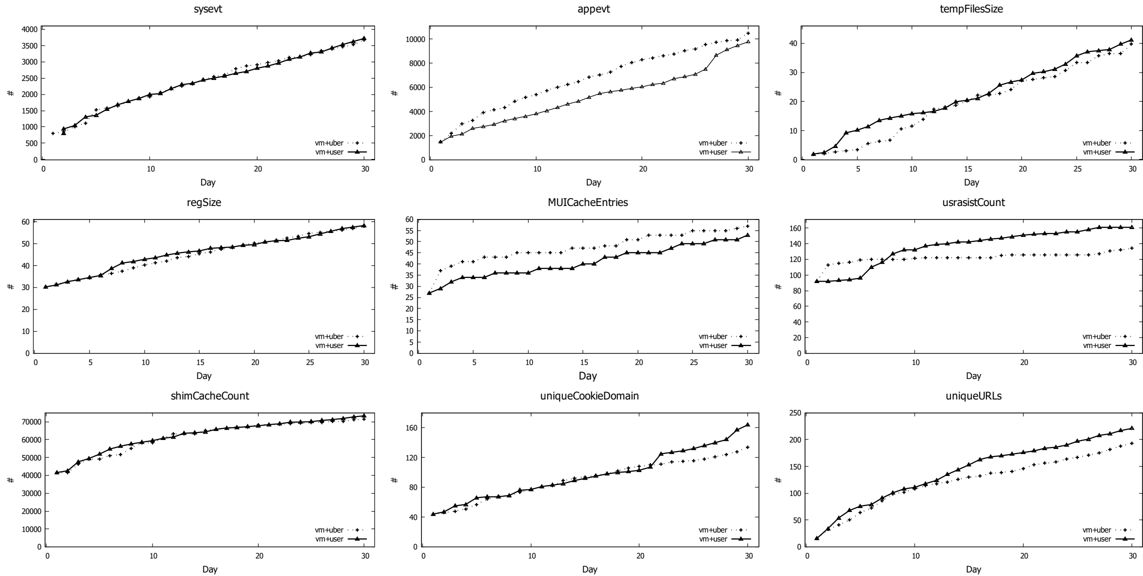


Figure 4.7: The Daily Variation of Artifacts Generation

We can observe that UBER could accumulate artifacts every day, which is similar to real user operations. The initial values of selected artifacts are from manually setting

the necessary operation environment. Artifacts “sysevt”, “tempFileSize”, “regSize” and “shimCacheCount” record the quantity information of general user usage. Their variation curves of UBER are almost the same as the real user since these two systems have the same operating environment and perform similar application activities. Artifacts “appevt”, “usrassistCount”, “MUICacheCount”, “uniqueURLs” and “uniqueCookieDomains” are also influenced by the application revoking sequence and the uniqueness of URL access. The variation curves of UBER are different from the real user since our UBER emulator revokes applications in a different sequence and accesses different URLs from the real user.

From the similarity in the variation curves, we can conclude that the artifact accumulation progress of UBER enabled sandbox is close to the real system. When the malware selects seven days as the time length of usage, UBER can emulate the same seven days of usage and generate a close quantity of artifacts as what the real user can do. Hence, the malware cannot choose a specific time length of usage to distinguish our emulation-based sandbox from the real system.

4.5.4 Artifacts Validation in Third-Party Tool

To further prove the validation of generated artifacts, we examine if UBER can pass the checking proposed in [4]. Its authors provide us with a well-trained decision tree classifier to evaluate generated artifacts of UBER. This classifier is trained with the baselines and real users used in [4]. It receives the artifacts listed in Table 4.1 and outputs the prediction: “Base” or “User”. The “Base” indicates the fresh OS (i.e., sandbox), while the “User” indicates the real system.

To explore the effectiveness of UBER under the detection of the third-party tool, we test the artifacts in the span of 30 days. The UBER emulator starts to work from a fresh OS. The test results are shown in Figure 4.8. We can observe that the classifier identifies our sandbox as “User” with 50% prediction confidence after the first day. Two days later, the confidence reached 90%. After running UBER for one week, the classifier identifies our sandbox as a real system with 100% prediction confidence. In other words, the classifier

considers real user activities generate these artifacts. In practice, the malware may only check a subset of these artifacts generated by UBER due to its small size. Even if any malware adopts a full-featured classifier from paper [4] to check all the mentioned artifacts, one-week emulation of UBER can effectively deceive it.

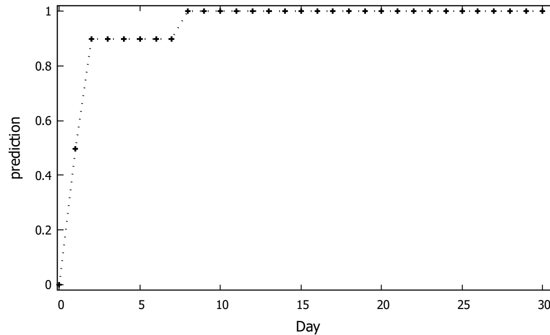


Figure 4.8: The Predictions of Sandbox Deployed with UBER

From the above three experiments, we can conclude that UBER can successfully emulate authentic user operations and generate realistic artifacts in sandbox systems to make them indistinguishable from real systems.

4.6 Discussions

More Fine-grained Profiling

UBER generates the most commonly identified user artifacts by emulating the relevant activities. This approach does not apply to generating artifacts associated with proprietary, customized, or otherwise less popular software. This type of software usually generates its own unique artifacts. If one attacker aims at a target utilizing this specific software, this approach will become less effective. To defeat the sandbox evasion technique that utilizes these software-specific artifacts, we could extend UBER to emulate the specific software

used to generate those unique artifacts.

For those sandboxes allowing outbound connection to the Internet, the attacker may repeatedly upload small benign samples to collect artifacts and then conduct the independent examination with more computation resources. In this way, the attacker will try to dig out the unique features of UBER sandboxes (e.g., a specific file related to the emulator or a similar behavior pattern among these sandboxes. To defend against this kind of attack, we would like to introduce more random factors to our UBER emulator in the future. Those random factors intend to minimize the similarity among UBER sandboxes at both file and behavior levels and increase the difficulty of detecting unique features.

It is usually sufficient to create realistic user behavior profiles with generic operations from normal real users and public available web access data. If malware targets one specific individual or organization, generic profiles may become invalid. However, this type of targeted malware can be handled by collecting computer usage information from specific targets and then deriving specialized user profiles.

Since the attack surface of system artifacts is huge, with the evolution of evasion techniques, malware may identify the sandbox environment by leveraging a variety of artifacts not included in Table 4.1. Therefore, we plan to derive a more precise user profile including CPU usage, I/O usage, network usage, system calls, and GUI events to perform a more accurate emulation of user behavior and integrate methods like FORGE [93] into UBER to generate validated artifacts for confusing advanced malware in the future.

Our prototype of UBER runs in a Windows virtual machine and focuses on creating believable artifacts in Windows OS. However, the architecture of UBER could be extended to work in the analysis sandbox based on other operating systems (e.g., Linux, Android, Mac). In other words, the framework proposed by UBER could be applied to complement artifacts related to application usage, the file system, and the various network caches, regardless of the exact operating system. Therefore, we leave the generation of system artifacts on the sandbox system from other platforms, such as Android, as our future work.

Alternative Solutions

There are other solutions to generate user artifacts against sandbox evasion techniques; however, after studying their limitations, we choose to derive a user behavior model from real user systems and then use the model to guide user behavior and generate system artifacts.

User Behavior Replay. It involves capturing usage activities from an actual user and replaying these activities on a sandbox system. However, it remains a challenge to fully capture user activities and replay to the sandbox system, especially due to the inconsistent semantic gaps between real and replay activities [94]. Another problem with this solution is the privacy issue. When it comes to real user activities, personal information, such as emails, passwords, and other sensitive data, requires a sanity mechanism to protect the user's privacy. However, the screening of privacy information may easily crash the replaying process.

Clone Real User System. By directly cloning real user systems to sandbox environments, we could ensure sandbox systems own realistic artifacts. However, this solution also faces privacy issues, such as sanitizing sensitive information, including certificates, documents, and pictures stored in systems. Besides privacy issues, the cloned realistic artifacts become expired after a period of time.

Directly User Behavior Simulation. This solution directly simulates user behaviors, including mouse clicking and keyboard inputting in sandbox environments. However, it is still difficult to authentically mimic the user activities in various application scenarios without disturbing the run-time resource and analysis results of malware.

Practical Usage

The sandbox deployment strategy decides the image size and response time. In our implementation, we allocate 80 GB to our VM virtual disk to satisfy the essential daily usage for our artifacts generation. In the practical scenario, the size of the VM varies. For example,

the default virtual disk size of FireEye's FLARE VM[6] is 60 GB, while the size of Any.Run[7] is 255 GB. Their virtual disk size can be increased easily. Similarly, the virtual disk size of UBER can be increased for practical usage.

Compared to current sandbox solutions, our VM needs to keep renewing the artifacts, thus the update has a higher frequency. The current solutions only update the VM when it is necessary. According to the practical requirements of analyzers, the periodic time of UBER could be one day or longer (e.g., one week) to avoid over-high time cost. In the USA, the length of the paid holiday varies on the contract between the employer and the employee. Thus, the maximal update interval should vary with the practical scenario. For example, a company provides around 10 days of paid holiday each year and the continuous holiday is no longer than 5 workdays. In this case, we suggest that the maximal update interval could be one week (a 2-day weekend and 5 workdays) to emulate it. The maximal update interval would also vary based on the local paid holiday policy for other countries.

To keep pace with the real user, we suggest that the emulation VM (sandbox) run 8 hours per day from Monday to Friday, which is common for general employees. After each day's emulation, the update scheduler can duplicate the emulation VM as an analysis sandbox. The emulation VM could accumulate artifacts and produce the newest duplication as the analysis sandbox day by day. If the analysis sandbox is running 24/7, the operator could switch the user access entrance to the newest analysis sandbox when an update has been completed. After that, the updated analysis sandbox accepts the newly arrived tasks. For each malware analyzed in one update cycle, the analyzer can still return to the previous snapshot in a short time. This procedure is as same as the current sandbox solutions. The analyzer will not be influenced during work time.

[6]https://github.com/mandiant/flare-vm
[7]https://any.run/

4.7 Related Work

4.7.1 Artifacts Identification-based Sandbox Evasion Techniques

Many researchers have studied sandbox evasion techniques and corresponding mitigation strategies. Afianian et al. [61] present a detailed survey on existing sandbox evasion techniques and summarize possible countermeasures towards that evasion malware. Alexei et al. [95] systematically review evasion techniques against automated dynamic malware analysis on PC, mobile, and web. Dilshan [65] provides an overview of commonly used evasion techniques and corresponding mitigation countering these evasions.

The identification of typical artifacts on analysis sandboxes has been recognized as effective evasion techniques [4, 72, 74, 96]. By checking the distinguishable characteristics of one given sandbox, malware could alter its behavior to evade detection. Vashisht et al. [96] point out that malware usually checks the interaction activities generated by a real user on the run-time environment. Malware will not exhibit any malicious behavior until human interactions, such as moving the mouse at a realistic speed or mouse buttons clicking, are performed. Blackthorne et al. present AVLeak [72] to extract characteristics from emulators running inside commercial antivirus (AV) software. AVLeak treats these emulators as black boxes and leverages the side channel to extract typical artifacts from AV. Yokoyama et al. developed SandPrint [74] to exfiltrate characteristics of malware analysis sandboxes and further prove that these characteristics extracted from sandbox systems could be used by malware to perform evasion analysis. Spotless Sandboxes [4] further points out that the type of activities conducted in a sandbox environment varies greatly from a typical real system. Therefore, this method identifies an indicative set of common, easily accessible, and historical system artifacts that are generated by normal user activities. Malware could easily derive sandbox environments based on those typical artifacts. UBER follows this idea and tries to complement these defects by generating realistic historical artifacts via user activities emulation based on derived usage patterns.

4.7.2 Countering Sandbox Evasion

There are two major technical routes to defeat the artifacts identification-based sandbox evasion techniques: detecting evasion techniques and producing realistic artifacts. Detecting evasion techniques adopted by malware is a typical way to prevent sandbox evasion techniques. Lindorfer et al. [62] propose to detect environment-sensitive malware that usually alters its behavior when running on different analysis environments. This method could be used to combat previously unknown sandbox evasion techniques. Barecloud [97] is similar to Disarm, while it uses four different platforms. Kang et al. [78] adopt instruction-level traces from multiple analysis platforms to detect evasion behaviors. MalGene [98] is proposed as an automatic method to extract evasion behavior signatures based on system call sequences. MORRIGU [99] is developed to leverage dynamic sandbox reconfiguration techniques to detect anti-evasion malware triggers. These evasion detection methods could be the first line of defense for defeating artifact identification-based sandbox evasion.

Deploying realistic system artifacts intends to deceive the malware that the running environment is not a malware analysis sandbox but a real user's device. Emulation is an effective method to continuously generate system artifacts. It typically regards hardware emulation for testing the stability of applications on different platforms, or software execution emulation to identify implementation bugs and verify the effectiveness of functions. The user emulation intends to perform user-similar operations on the software and generate corresponding results. In the security area, emulation tools are used to provide an even more realistic application interaction and network environment (e.g., cyber range). LARIAT [100] and K0ALA [101] were designed by MIT Lincoln Laboratory model and generate network traffic directly. Megyesi et al. [102, 103] present a traffic generation framework based on automatic user behavior emulation. These frameworks generate realistic traffic based on user behavior scenario extraction and the recording of user interactions on specific applications. The effectiveness of generated traffic is verified via experiments. Dutta et al. [104] design the user bots which leverage the recorded user behaviors to generate similar user network traffic. DASH [105] is an agent platform for building agents that simulate

human behavior (such as responding to phishing emails, downloading and using security software). By using different structural modeling decisions and sequence models, D2U [106] explores the emulation solutions for the application usage sequence. All these emulated application interactions could generate system artifacts and network activities more or less.

Meanwhile, as a countermeasure, the malware could adopt a variety of emulation identification techniques. Kruegel [107] presents some sandbox evasion techniques via hardware emulation identification. In UBER, we use continuous user behavior emulation to generate realistic system artifacts for the sandbox to deceive the malware.

4.8 Chapter Summary

With the wide application of malware analysis sandboxes, malware authors have also developed sophisticated evasion techniques to evade sandbox environments. Among them, one particular technique is to fingerprint various artifacts generated during the normal usage of the real system, which cannot be countered via state-of-the-art mitigation strategies. Given existing malware analysis sandboxes deployed on pristine operating system images, we investigate the typical system artifacts and propose a generic anti-evasion system UBER, which can effectively generate realistic system artifacts based on the automatically derived user profile. We implement a prototype of UBER and our experimental results indicate that UBER is able to faithfully emulate user behaviors and generate realistic artifacts in sandbox systems. UBER is flexible in constructing user behavior profiles of specific individuals, organizations, or activities, which could further improve its capability to mitigate increasingly sophisticated and targeted sandbox evasion techniques.

Chapter 5: HoneyMustard: Towards Enhancing the Fidelity of Honeypot with Real-Time User Behavior Emulation

5.1 Introduction

Since Lance Spitzer [1] introduced the term "honeypot" in 2002, the honeypot technique has been widely adopted in security systems. Over the past 20 years, honeypot has evolved from merely a toolkit into a real system equipped with "real" data [108]. It can capture, analyze, and derive the motivation behind an attack or serve as an early warning to intrusion.

Attackers are well-motivated to develop various anti-honeypot techniques that leverage the differences between honeypots and real systems as indicators to detect the existence of honeypots. Particularly, due to the lack of real user operations on the honeypot, attackers can check "wear and tear" artifacts [4] or user activities [96] to detect if the system is in use and thus identify the honeypot. In general, to defeat the anti-honeypot methods, the countermeasures are to remove the major differences between the honeypots and the real systems [3]. Recently, UBER [109] emulates the user activities to generate static artifacts, defeating the "wear and tear" artifacts checking from malware. However, it cannot be directly deployed on the honeypot, since its artifact emulator is visible in the system and can be easily detected by attackers. To emulate real user activities, D2U [106] designs a mechanism to generate an application usage sequence; however, it cannot generate believable and logical user operations at the application level.

In this work, we propose an application-level real-time user behavior emulation framework called HoneyMustard to enhance the fidelity of the honeypot system. HoneyMustard can emulate logical application-level user operations in real-time and maintain the stealthiness of the emulator. The workflow of HoneyMustard consists of two major stages: user operation collection and computer vision-based emulation. In the user operation collection

stage, it collects the user operations on applications from both real user activities and user manuals to construct an action dataset. The collected user operations share the same logic as the real users, which ensures that attackers cannot easily distinguish them from real user activities. In the emulation stage, it leverages computer vision techniques to manipulate the GUI interface of the honeypot via a remote desktop connection (e.g., VNC). This design choice can achieve a high stealthiness of honeypot. When HoneyMustard is emulating user activities, attackers can only observe a remote desktop connection on the target machine, so we can protect our emulation system on a remote server that is inaccessible to the attacker. Considering that more companies and organizations allow employees to work remotely in the past several years due to COVID, it is normal for attackers to discover a remote desktop connection in the victim's device.

We implement a prototype of HoneyMustard to demonstrate the effectiveness of our solution in attacker deception. We record the emulated user activities and real user activities into videos for five common tasks separately, then we conduct a user study with an online survey with 100 participants to identify the emulated videos from them. The average success rate of deception is 71.8%. The experimental results show that HoneyMustard can effectively deceive attackers with real-time emulated user activities.

In summary, we make the following contributions:

- We propose a real-time application-level user behavior emulation framework, HoneyMustard, to enhance the fidelity of the honeypot system. It emulates the GUI-based user activities based on the logic of real users and user manuals to deceive the sophisticated attackers.

- Our design leverages computer vision techniques to emulate GUI-based user behaviors, which can support various applications and platforms via the remote desktop connection in real-time. The remote desktop connection also ensures that we can hide the emulation procedure as a normal activity.

- We implement a prototype of HoneyMustard and evaluate it from both user study and

performance measurement. The experimental results demonstrate that our solution can effectively improve the fidelity of honeypots via real-time GUI-based user behavior emulation.

5.2 Related Work

5.2.1 Honeypot Detection

Sophisticated attackers are well-motivated to detect honeypots to maintain the stealthiness of attack. The honeypot detection techniques can be integrated into the malware or conducted by the attacker directly. In general, honeypot detection techniques focus on the artifacts in two major categories [3, 32]: *network-related fingerprinting* and *system-related fingerprinting*.

Network-related fingerprinting is to detect the discrepancy in network activities and network latency. Usually, honeypot systems may provide only a limited number of services and/or have constrained interactions with other real hosts and the Internet. If attackers observe fewer network activities in the compromised host compared to legitimate hosts, they may consider it a honeypot [39]. Some honeypots are deployed in separate locations, which may introduce extra network latency. Attackers can detect those honeypots with higher network latency [40].

System-related fingerprinting is related to the setup of operating systems and applications. Compared with real hosts, honeypot systems may use a different configuration to monitor and constrain the malware and attackers. Attackers can detect these unique system artifacts [42, 61, 95], such as operating system flags, running processes, volatile user information, files, installed programs, etc. Also, honeypots may lack usage from real users. Spotless Sandboxes [4] proposed to check the "wear and tear" artifacts inside the compromised system to detect the honeypot. Those artifacts are inevitably generated by real users during daily usage. Similar to network activities, some attackers also leverage real-time user activities to detect honeypots. For example, malware may wait for more than two

mouse clicks before executing [96]. As a person, the sophisticated attacker can enhance the user behavior-related detection by staying in the compromised system long enough to check various user operations before conducting further attacks.

5.2.2 User Emulation

Among the countermeasures for honeypot detection, user emulation is an effective method to generate user activities and "wear and tear" artifacts. LARIAT [100] and K0ALA [101] were designed for cyber range to emulate network traffic. Megyesi et al. [102, 103] proposed a framework to generate real traffic from the recorded user interactions on specific applications. Similarly, Dutta et al. [104] leverage the recorded user behaviors to generate network traffic. Besides generating network traffic, UBER [109] generates the usage artifacts for the honeypot by continuously emulating user behaviors. D2U [106] explores emulating the application usage sequence using generative models based on actual user data. Both UBER and D2U didn't propose a real-time application-level emulation solution. For attackers, illogical user operations could be an obvious indicator of the honeypot. Compared with previous work, our framework focuses on supporting the finer-grained emulation, i.e., generating the logical user operations for the specific applications, which can enhance the fidelity of the honeypot system.

5.3 Threat Model and Assumption

In this work, we focus on defeating the honeypot detection technique of checking GUI-based user behaviors. In the practical scenario, real users usually interact with their computers via the GUI interface. Hence, the attackers could expect to observe the GUI-based user behaviors in real user devices. If not, the compromised system could be a honeypot. The lack of GUI-based user behaviors could become one of fingerprinting for honeypot detection.

In our threat model, we consider the sophisticated attackers (e.g., APT), who can obtain the root privilege after breaking in and are able to dwell in the compromised system as

long as possible to collect sensitive information or modify (system and user) files. Since they intend to maintain their long-term foothold in the compromised system, the attackers have enough time to detect the honeypot based on the user behaviors. We assume that the attackers can monitor the GUI desktop and user's operations on the GUI interface via periodically taking screenshots or direct remote desktop connection (e.g., RDP, VNC). By observing and analyzing the operations on the GUI interface, the attackers can decide whether there are real user activities in the compromised system. If there are no user activities, the attackers may consider that the compromised system is a low-value object or has a high probability of being a honeypot. If the observed user activities are illogical or completely random, the attackers can consider them committed by an emulator. The compromised system also could be a honeypot. This decision procedure is similar to the Turing test [110] to some degree. Unlike the full-featured Turing test, the attackers would avoid interacting with the user or interfering with the user's operations to maintain their stealthiness because any abnormal interaction could raise an alert. Besides observing the user activities, the attackers would detect the existence of an emulator in the compromised system to check whether these user activities are emulated.

5.4 System Design

5.4.1 Overview of HoneyMustard

We propose the HoneyMustard framework to enhance the fidelity of honeypot. Its design principle includes:

- **Stealthiness** - The trace of the emulator is invisible in the honeypot.
- **Logicalness** - The emulated user activities are as logical as the real user did.
- **Real Time** - The attackers can observe emulated user activities in real-time.

As shown in Figure 5.1, HoneyMustard consists of two core components: collected user operations and emulator. Both of them are deployed in an independent device from the

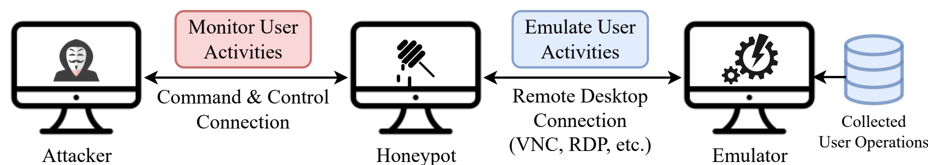


Figure 5.1: The Overview of HoneyMustard

honeypot. The emulator connects to the honeypot via the remote desktop connection. It leverages computer vision techniques to redo the collected user operations on the GUI interface (i.e., desktop) of the honeypot. Because the remote desktop connection allows a computer's desktop to be run remotely on one system while being displayed on a separate device, the attackers can only notice the existence of the remote desktop connection in the compromised system other than the emulator. With the popularization of cloud services in the industry, more and more companies and organizations deploy their working platforms in the cloud and let employees access them via a remote desktop connection. Attackers cannot simply take the remote desktop connection as the indicator of the emulator. Therefore, the remote desktop connection ensures the **stealthiness** of HoneyMustard. We collected the GUI-related user operations from real users to ensure the **logicalness** of HoneyMustard. To achieve **real time**, HoneyMustard would emulate user activities periodically, which means attackers can monitor the emulated user activities.

5.4.2 User Behavior Emulation

Figure 5.2 shows the workflow of HoneyMustard to emulate user activities on the GUI interface. According to the two core components of HoneyMustard, we divide it into two stages: user operation collection and computer vision-based emulation. In the user operation collection stage, we construct the dataset of user operations. In the computer vision-based emulation stage, HoneyMustard starts to emulate user activities in the honeypot.

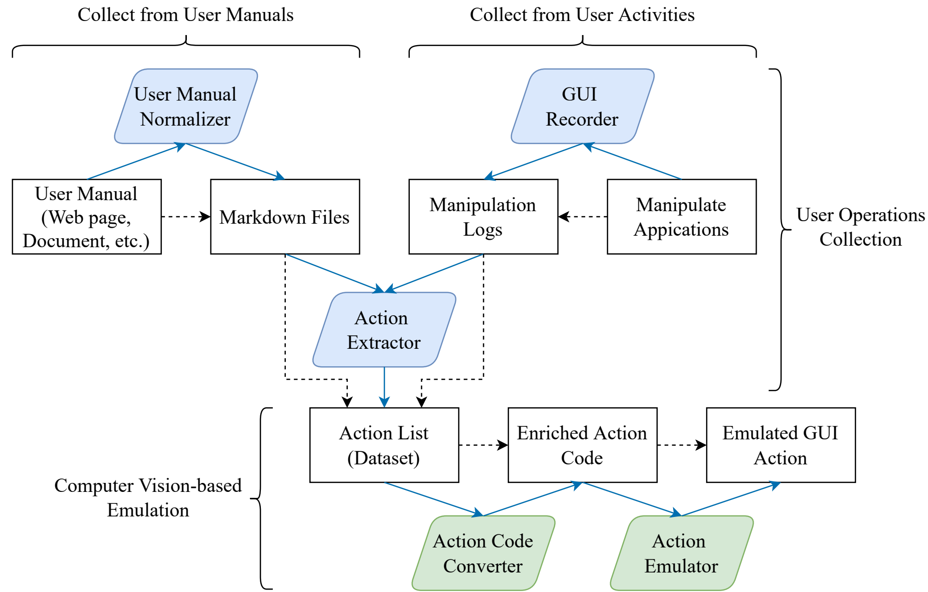


Figure 5.2: The Workflow of HoneyMustard

User Operations Collection

Intuitively, because of the large number of users, it is most convenient to collect user operations from the daily activities of real users. Under the practical circumstance, daily activities inevitably contain private information, which requires us to identify and eliminate it to protect the user's privacy. To overcome this problem, we use two types of data sources to construct the dataset of user operations. One type is the real user activities. Another one is the user manuals. For the first solution, we invite real users to complete assigned specific tasks without using their personal information. Instead, they may use decoy information. In this way, the collected user operations don't contain private information. For the second solution, we extract the user operations from the user manuals. The user manuals consist of logical user operations to complete tasks for specific applications. Since the manual creators (e.g., software vendors) have sanitized the content, there is usually no private information in the user manuals.

To collect user operations from user activities, we invite users to complete some basic

tasks on the common applications. Each task contains several essential operations. Meanwhile, we leverage a GUI recorder to record each step conducted by users over the GUI interface. For each task, we convert the user operations to a manipulation log, which contains the actions (e.g., mouse click, input, etc.) and GUI elements (e.g., button, check box, label button, etc.).

To collect user operations from user manuals, we leverage a user manual normalizer to convert various user manual formats to a similar but common format. We choose the Markdown format as the normalized format. The Markdown format [111] is well-structured, which can facilitate our parser to analyze the manual content. Similar to manipulation logs, the Markdown-formatted user manuals preserve the actions and GUI elements.

After obtaining manipulation logs and Markdown-formatted user manuals, the action extractor extracts the actions from descriptive text to generate action lists. For each task, we generate an action list to preserve the operation logic. For the text-only manual, we can obtain the action keywords (e.g., Click, Select, Type, etc.) and the target objects (e.g., GUI elements, text, etc.). For the hybrid manual, we also obtain the images of the GUI element as one part of the target objects. We define a uniform action record format to represent the information of each action:

```
<Action Keyword>, [Target Type], [Target Object], [Target Path]
```

Here the `Action Keyword` indicates the operation. The common keywords include click, right-click, select, type, input, and so on. The `Target Type` indicates the type of the object. We use it to distinguish the GUI elements (e.g., button, check box, label button, etc.) and text. The `Target Object` shows the name of the GUI elements or content of the text. For the hybrid manual, we could obtain the images of some GUI elements. The `Target Path` stores the location of the image. For each action record, it must contain an action keyword at the head, while other elements are optional.

When processing manipulation logs, we don't extract the timestamp of each operation. The timestamps may contain the pattern of real user behaviors and belong to the user's privacy. The aggregation of multiple action lists is the dataset for user activity emulation.

Computer Vision-based Emulation

When conducting the user activity emulation, HoneyMustard picks up the desired task (i.e., action list) from the dataset and emulates actions over the honeypot. To emulate a logical application usage sequence, we follow the solution proposed in D2U [106]. When emulating the usage sequence, D2U has considered the work hours and break times.

For each task, HoneyMustard converts it to the enriched action code via an action code converter. The enriched action code makes these actions executable for the emulator. It also introduces more helper action details for emulation, which are not provided by the user manuals. To enrich the action code, the converter completes the following tasks:

1) Converting the action keywords to the corresponding operations of input peripherals (e.g., mouse, keyboard). In the descriptive instructions, multiple action keywords represent one operation of input peripherals. For example, the action keywords Click, Select and Check could be conducted by the mouse click operation to the desired GUI elements. By conducting this task, we can reduce the complexity of the action types and simplify the code.

2) Adding waiting time slots. In the practical scenario, there is a processing time between two continuous actions. After the processing, the software can switch the status or interface to receive the following action. Besides, real users always pause for a while between two consecutive actions. They may think for a while before conducting the next move, or they may be disturbed by other things during the operation procedure. For these reasons, we add the waiting time slot after each action. The waiting time length is pre-defined. It is decided by the emulation environment performance and the software response speed.

3) Adding pre-collected GUI element image path. As HoneyMustard leverages the computer vision technique to emulate the user action, it needs to identify the GUI elements from the screen. Some software interfaces leverage icons as shortcuts. For the text-only manual, the extractor can only extract the type and name of GUI elements. In this case, the emulator may not locate it successfully via its name. To solve this problem, we collected these GUI elements as the pre-knowledge data and link them to the object names. When

actions related to icon-based GUI elements or specific action records appear, the converter can read the corresponding pre-collected GUI element images and add their paths into the code. For example, the "Close" button does not contain any text indicator in some software interfaces. The converter translates the action "*Close*" to click the "Close" button in the code. The target path of the "Close" button is from the pre-knowledge data.

4) Converting partial actions into pre-defined action sequences. Sometimes, an extracted action could be not detailed enough. This usually appears in the system configuration manuals. The author of the user manual considers these actions to be relatively fixed and well-known, thus the audience can conduct them easily. For example, a manual could start with "*Open settings*". In its context, this action includes searching keyword settings, locating the settings item, and clicking the setting icon. To convert this kind of action to specific operations, we also need to pre-define some short operation routines for them. When the converter encounters them, it revokes the pre-defined operation routines into code.

5) Adding pre-defined action parameters. Some action requires users to fill in custom information, such as installation path, IP address, etc. In most cases, we may use the default value provided by the software/system. For those who do not provide a default value or cannot use the default value, we introduce the pre-defined values fitting for the setup of the emulation environment. In this way, we can keep the continuity of the emulation.

With the enriched action code, HoneyMustard can emulate the operation of input peripherals on the software/system interfaces. As mentioned above, the code has two major types of target objects: image and text. The action emulator leverages the computer vision technique to locate the target objects shown on the screen. For the image, the emulator conducts the template matching to search for the corresponding GUI elements. For the text, the emulator searches the matching text from the screen by using optical character recognition (OCR). After knowing the actions and the target objects, the emulator can execute the actions sequentially on the honeypot via a remote desktop connection. From the attacker's perspective, a remote user is using the system.

5.5 Implementation

Our prototype of HoneyMustard is built on a Linux host installed Ubuntu 20.04 (kernel version 5.4.0). We use a Windows 10 host as the honeypot. We configure a *UltraVNC server* 1.3.2a on the honeypot, so the emulator connects to the honeypot via a VNC connection.

5.5.1 User Operations Collection

GUI recorder

Since our prototype focuses on emulating user behavior on the Windows-based host, we use the *WinAppDriver* [112] for the GUI recorder. The recorder tracks both keyboard and mouse interactions against an application interface – representing a GUI action. When recording is active, it generates the log including GUI actions and information of selected GUI elements.

User Manual Normalizer

We use the *pandoc* 2.5 [113] as the normalizer to convert the user manual in various formats to the Markdown format. Its supported formats include HTML, DOCX, and EPUB. For the user manuals in other formats, HoneyMustard can use corresponding Markdown format converters as the normalizer.

Action Extractor

We develop the action extractor in Python 3.8.10. It processes the descriptive instructions from manipulation logs and Markdown-formatted manuals to obtain the action list. For the text-only content, the extractor processes the descriptive instructions to obtain the action list. For the hybrid content, besides processing the instructions, the extractor extracts the GUI elements from the manual images and links them to the corresponding actions.

Instruction Processing:

1) Text Tagging: The extractor leverages a pre-defined action keyword list to identify

the actions from instructions. Its basic idea is to search these pre-defined keywords in the instructions, while not every matched word represents an action. A matched word indicates an action keyword only when it is a verb. For example, the keyword "*Input*" could appear as a noun in a descriptive phrase, while it should be a verb to indicate an action for users. Hence, we use the lexical category analysis to distinguish the action words from the descriptive text. The extractor uses the *NLTK* [114] package to do this work. The extractor conducts the word tokenization to split the instructions into words. Then, it attaches a part of the speech tag to each word.

2) Tag Revise: In the practical scenario, the tagging can not reach 100% accuracy. The main reason is that user manuals have various writing styles and prefer to use terms from the computer area, while the *NLTK* is trained with the common corpses. Sometimes the action keywords in the imperative sentences will not be tagged as verbs. Another reason is that we keep the symbols among text in the Markdown format. These symbols are the typographical emphasis in the text and can be the indicator target objects, while they could mislead the tagging results. Sometimes, the action keyword before these symbols would be wrongly tagged as nouns. To eliminate error tags, the extractor would automatically recheck the tags of the found keywords and then revise them.

3) Target Object Identification: After identifying the action keywords, the extractor searches the corresponding target object. The target object follows the action keyword. Usually, it should be a noun or a short phrase. The extractor can conduct a forward search for the nouns from the identified action keyword to the next action keyword or the end of the current sentence. In some Markdown-formatted user manuals, the identified action keywords may not have target objects. Besides, the target object could locate among the symbols, such as quotation marks, asterisks, etc. Sometimes, this kind of target object could be a descriptive sentence, which is common in system configuration manuals. These situations are also processed by our action extractor.

Image Processing:

In the hybrid content, there are images that follow the corresponding instructions to

provide extra information about GUI elements. Hence, the extractor can request image processing after each identified action if an image follows the instruction. There are two types of images. One is the precisely cropped GUI elements, such as button, check box, Label button, etc. The audience can learn the exact look of the GUI elements mentioned in the instruction. The emulator can directly use them to search the matched GUI elements on the screen. Another is the interface of software or system, which contains the target GUI element. It intends to give the audience an overview of the current status after completing an action. In this case, the extractor needs to extract the GUI elements from the images.

We leverage the *OpenCV* [115] and *Tesseract-OCR* [116] to extract the specific GUI elements from the images. From the instruction processing, we obtain the type and name of the target object. According to the target type, the extractor conducts the Canny boundary detection on the image. For example, the button is rectangular. To detect a specific button, the extractor needs to detect all the possible rectangular bounding boxes in the image. In some cases, boundary detection could provide lots of results. The main reason is that the boundary of GUI elements could be obscure, which is caused by low resolution. A GUI element could be detected multiple times, which produces multiple overlapped detected bounding boxes. Hence, we use Non-Maximum Suppression (NMS) [117] to eliminate the redundant bounding boxes. Then the extractor detects the name or content from the remaining bounding boxes via OCR. Compared with the target name, the extractor can find out the correct bounding box for the GUI element. Similarly, the detected text from low-resolution images would be inaccurate. We calculate the Character Error Rate (CER) of the identified text with the target name. As we know, the target GUI element is in the processed image. Therefore, the bounding box, which has the highest CER score, contains the target GUI elements. With the correct bounding box, the extractor can corp the image of the GUI element and fill its path in the action record.

There are some icon-based GUI elements without description text showing aside on the screen. The OCR technique cannot detect them from the provided images, thus the extractor doesn't extract them. For them, we currently rely on the pre-knowledge data to

fill this gap in the action converter.

5.5.2 Computer Vision-based Emulation

Action Code Converter

Similar to the extractor, our action converter is also developed in Python. It converts the action list into the Robot Framework code. The Robot Framework code has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach, which ensures our converter is simple and effective. The converter translates the action records into the test cases in sequence with the provided action keywords as the code keywords and the target object information as the parameters. As mentioned in Section 5.4, we introduce the pre-knowledge data to enrich the generated code. The pre-knowledge data includes the scale of waiting time slots, images of icon-based GUI elements, and action parameters.

Action Emulator

As shown in Figure 5.3, the action emulator consists of the *Robot Framework* 4.1 [118] and *Sikuli* 2.0.5 [119]. Robot Framework is a Python-based, extensible keyword-driven automation framework. It can be used for test automation and robotic process automation. The framework has a rich ecosystem around it consisting of various generic libraries and supports custom libraries. We use the Sikuli library of Robot Framework to manipulate the GUI elements on the screen. It uses image recognition powered by OpenCV to identify GUI elements and OCR to identify text.

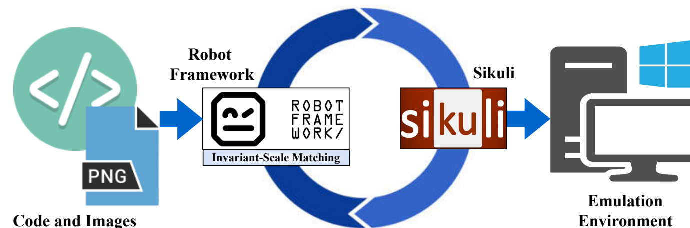


Figure 5.3: Implement of Action Emulator

One known restriction of Sikuli is that the image to be searched for must be the same in width and height (pixels) as the image to search in. It is difficult to ensure the images from manuals share the same resolutions with the screen, which could cause failure in image matching easily. To solve this problem, we develop an invariant-scale matching module for Robot Framework working with the Sikuli module. The core idea of invariant-scale matching is to loop over the input image at multiple scales to find a matching one.

As shown in Algorithm 4, the Robot Framework receives the code and related images. If the input image has the same resolution as the screen, the Robot Framework forwards it to the Sikuli module. The Sikuli module would emulate the corresponding action on the identified area of the screen (Line 5). If the resolution of the input image is different from the screen, the Robot Framework drives the Sikuli module to capture a screenshot from the emulation environment. Then it delivers the screenshot and input image to the invariant-scale matching module (Line 6). The invariant-scale matching module searches the matching area from the screenshot under various scales (Line 16) and returns the corresponding coordinate to the Sikuli module via Robot Framework. With the coordinate, the Sikuli module can continually emulate the action on the correct GUI element (Line 10).

5.6 Evaluation

In our experiment, HoneyMustard is running on a Ubuntu 20.04 (kernel version 5.4.0) device with Intel Xeon E5-2620 CPU @ 2.40GHz and 16 GB memory. The honeypot is a Windows 10 device with Intel Core i7-6500U CPU @ 25.0GHz and 8 GB memory.

5.6.1 Deception Effectiveness

Study design

To evaluate the deception effectiveness of HoneyMustard, we conduct a user study to check whether the emulated user activities can be considered as the real user activities by the real person. Since we cannot recruit real attackers, we invite normal users to this study.

Algorithm 4 Action Emulation with Input Image

```
Input:  A: Action to be executed on the GUI element
        I: Input image for searching the corresponding GUI element
 1: function ROBOT_EMULATOR(A, I)
 2:     S = Sikuli.screenshot()
 3:     if I.resolution() == S.resolution() then
 4:         /* Same Resolution */
 5:         Sikuli.execute(A, I)
 6:     else
 7:         /* Different Resolutions */
 8:         C = INVARIANT_SCALE_MATCH(I, S)
 9:         if C is not None then
10:             Sikuli.execute(A, C);

11: function INVARIANT_SCALE_MATCH(I, S)
12:     found = None
13:     similarity = None
14:     coordinate = None
15:     /* Loop Over the Scales */
16:     for scale in range(Min, Max, Step) do
17:         /* Resize the Input Image */
18:         I_r = resize(I, scale)
19:         /* Conduct the Template Matching */
20:         (similarity, coordinate) = matchTemplate(I_r, S)
21:         /* Update the Matched Area */
22:         if found is None or similarity > found[0] then
23:             found = (similarity, coordinate)
24:     return coordinate
```

To simulate the real-time environment, we present the user activities in the video. By watching these videos, the users need to decide whether these activities are conducted by real users. Considering the real attackers may be more sensitive than the normal users, we also provide videos of real user activities as a baseline, which can assist the normal users to notice the difference between these two types of activities. We collect users' subjective and direct feelings about these videos through an online survey via the crowdsourcing platform - Amazon Mechanical Turk.[1] After posting our survey, we received 100 unique responses from anonymous users. We check the response patterns manually to confirm there is no malicious filling on the questionnaire.

Materials. We select five common tasks for the user study. These tasks include OpenVPN installation (**T1**), LibreOffice Marco editing (**T2**), Adobe Reader trust configuration (**T3**), setting windows system environment (**T4**), and windows network discovery configuration (**T5**). These tasks are from their corresponding official user manuals. There are an average of 10 steps for each task. Each task has two related videos. One records the emulated user activities, while another one records the real user activities. To create the videos of emulated user activities, HoneyMustard converts these user manuals to emulated user operations on the honeypot. We record these operations on the honeypot as videos. To create the videos of real user activities, we manually conduct the same tasks by ourselves and make the videos. Since all the collected data is from authors and public user manuals, IRB approval is not required in our experiments.

Procedure. In this experiment, each participant needs to watch ten videos (i.e., five tasks) and answer five questions. For each task, video A contains the real user activities, while video B contains the emulated user activities. After watching two videos for one task, each participant will answer a question to indicate which one may contain the emulated user activities. The question provides three choices: video A, video B, and uncertain. We set the maximum completion time of the questionnaire as 30 minutes.

[1]https://www.mturk.com/

Results

As shown in Figure 5.4, participants from normal users cannot identify the emulated user activities easily. Only the video B for each task presents the emulated user activities, so the average identification accuracy of these five tasks is 29.2%. Average 32.2% participants consider the video A as the emulated user activities, and average 39.6% participants cannot make the decision. Therefore, the success rate of deception reaches 71.8%, which indicates that emulated user activities of HoneyMustard can effectively deceive the attackers.

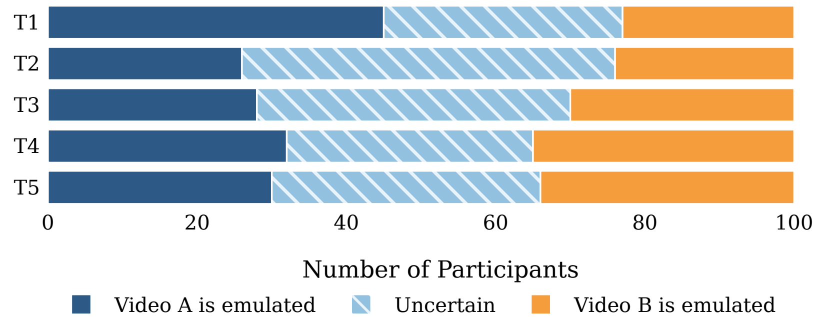


Figure 5.4: Results of Emulated Activities Identification Survey

The success rate of deception varies among different tasks. It is higher in the software manipulation-related activities (77% for **T1**, 76% for **T2**, 70% for **T3**) than the system configuration-related activities (65% for **T4**, 66% for **T5**). Among these five tasks, software manipulation-related activities are relatively more complicated and contain more mouse and keyboard operations, which increases the success rate of deception. According to the comments from participants, they don't find out any logical issues in these videos and mainly observe the cursor movement. They tend to choose the video with more mechanical and faster cursor movement as the emulated one. These criteria also lead some of them to choose the video A as the emulated one.

In the practical scenario, the VNC connection can introduce the network delay. Because

the action emulator manipulates the GUI interface projected to the local, the network delay only influences the response time. Real users confront the same issue with using a remote VNC connection thus the attackers cannot leverage the network delay to identify the emulated user activities.

5.6.2 System Overhead

Here we focus on the performance of the components without using third-party tools, i.e., the action extractor and action code converter. We measure the average memory consumption and CPU usage under processing the user manual of these five tasks. The action extractor costs about 190 MB memory and 12% CPU usage. The action code converter costs about 35 MB memory and less than 1% CPU usage. Since we don't modify third-party tools used in other components, their system overhead can be found in corresponding official documents. The experiment results show that HoneyMustard can be readily deployed in the real world without excessive performance overhead.

5.7 Discussion

Enhancing User Emulation

Our current solution inserts a random waiting time slot between operations to emulate the real user. A real user usually has a specific operation pattern in the waiting time slot. This pattern is decided by the content of the activity, personal character, job type, and so on. From a long-term observation, sophisticated attackers usually can extract an access pattern of waiting slots from real users, while the randomly generated waiting time slot cannot provide a fixed pattern. In this case, attackers may detect the emulated user activities. In future work, we plan to create various user profiles to increase the success rate of deception. Based on the various profiles, HoneyMustard can insert specific patterned waiting time slots to personate different types of real users. Besides, the cursor movement speed should also adapt to different user profiles.

Automatic Parser Selection

There are some pre-trained parsing models [120–123] that can parse the natural language instructions into executable actions. Their design goal is to parse several specific types of user manuals. To support various types of software/system-related user manuals, the action extractor needs to leverage different text parsers to process instructions and extract actions. Currently, we can manually select the appropriate parsing models or create parsing rules for user manuals from different sources. We plan to make HoneyMustard recognize the styles of input user manuals and select the appropriate parsers automatically in the future.

5.8 Chapter Summary

This chapter presents HoneyMustard, an application-level real-time user behavior emulation framework to enhance the fidelity of the honeypot system. HoneyMustard collects logical user operation sequences of various applications from the real user activities and user manuals to construct an action dataset. After obtaining this collected action dataset, it can leverage computer vision techniques to emulate user activities in the honeypot. With the commonly used VNC connection, HoneyMustard manipulates the GUI interface of the honeypot remotely without deploying any extra suspicious agent in the honeypot. Besides, HoneyMustard periodically emulates user activities, which allows the attackers to observe these activities in real-time. We implement a prototype of HoneyMustard and evaluate it from both an online survey with 100 participants and its performance overhead. According to the experimental results, our framework can effectively deceive attackers with real-time emulated user activities.

Chapter 6: Conclusions

6.1 Summary

In this research, we investigate novel solutions to several challenges in constructing distributed hybrid honeypot systems. Our research aims at constructing scalable and believable distributed hybrid honeypot frameworks to provide the honeypots as a service. It could benefit the security research community and industry to use high-fidelity, hard-to-detect, and customer-tailored honeypots in deterring sophisticated cyber threats, without needing to purchase expensive hardware and software and hiring professionals for honeypot configuration, analysis, and maintenance.

In the following, we summarize the results from this research:

- We develop an application-level distributed hybrid honeypot framework called HoneyBog to monitor and analyze webshell-based command injection. It intercepts and redirects malicious injected commands from the front-end honeypot to the high-fidelity back-end honeypot for execution. HoneyBog can prevent the attacker from launching further attacks in the protected network because the webshell-based injected commands are transferred into a remote constrained execution environment. The hybrid structure facilitates the centralized management of high-fidelity honeypots for remote honeypot service providers. Our experiments on 260 PHP webshells show that HoneyBog can effectively intercept and redirect injected commands with low overhead.

- We uncover that the distributed honeypot systems may suffer from one type of anti-honeypot technique called network context cross-checking (NC3) that enables attackers to detect network context inconsistencies before and after breaking into a targeted

system. We perform a systematic study of NC3 and summarize nine types of network context artifacts that may be collected during the exploitation that attackers may leverage to identify distributed honeypot systems by means of inconsistencies in the data set. Our theoretical analysis and experimental results show that they can utilize NC3 attacks to successfully identify all popular distributed honeypot systems.

- To protect distributed honeypots from the NC3 attack, we develop a countermeasure called HoneyPortal to remove the network context inconsistency between the pre-exploitation and post-exploitation stages. Precisely, the overall approach projects the remote back-end honeypot into the front-end network using a transparent traffic redirection strategy, where the front end redirects all incoming traffic (i.e., layer 2 and above) to the back end and vice versa. To improve its performance, we use the XDP technique in network traffic processing. The experimental results show that HoneyPortal can successfully defeat NC3 attacks with low overhead.

- We propose an emulation-based system called *User Behavior Emulator (UBER)* to deceive malware with the auto-generated usage artifacts in the sandbox. UBER automatically profiles user models from real user activities. Then it emulates the high-level user behavior in an isolated always-on system and stealthily merges this system into the malware analysis sandboxes. We explore the daily variation of artifacts generation and find that UBER is able to simulate the artifacts accumulation processes of real user systems. We also leverage the state-of-the-art classifier provided by [4] to verify the authenticity of the system deploying with UBER. The experimental results indicate that UBER can effectively generate realistic artifacts.

- We propose a real-time application-level user behavior emulation framework called HoneyMustard to enhance the fidelity of the honeypot. HoneyMustard leverages computer vision techniques to emulate GUI-based user activities in the honeypot via a remote desktop connection, achieving both believable and stealthy design goals. Using the remote desktop connection and GUI-based emulation ensures the real-time

emulation without leaving any traces of the emulator inside the honeypot. To evaluate its deception effectiveness, we conduct a user study with 100 participants. The experimental results show that our solution can effectively deceive the participants with emulated user operations and improve the believability of honeypots.

6.2 Future Work

In this dissertation, we demonstrate how our work can enhance the scalability and believability of the honeypot system. We believe much more can be done in this area. We summarize and propose an additional set of enhancements that can extend the capabilities of our work as follows:

- HoneyBog focuses on capturing the webshell-based command injections. The web server can also suffer other exploitation, and the attackers can construct an independent system shell on it. As a complete solution, we intend to integrate HoneyBog with our previous hybrid honeypot framework on system-based command injection [20].

- For the NC3 attack, we currently identify nine types of artifacts, while there could be more artifacts that can be used by attackers. In the future, we will keep exploring new artifacts to enrich the NC3.

- HoneyPortal provide an effective defense mechanism to NC3. In future work, we would like to explore how to integrate it into the practical network environment with an appropriate isolation configuration, which can effectively maintain consistency and delay the spreading of attacks in the protected network.

- UBER can automatically generate emulated usage artifacts, while HoneyMustard can emulate application-level user activities in real-time. We aim at combining these two systems together to construct a full-featured system to deceive malware and attackers with both static usage artifacts and real-time emulated activities.

Bibliography

[1] L. Spitzner, "Honeypots: Catching the insider threat," in *19th Annual Computer Security Applications Conference, 2003. Proceedings.* IEEE, 2003, pp. 170–179.

[2] J. H. Jafarian and A. Niakanlahiji, "Delivering honeypots as a service," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020.

[3] J. Uitto, S. Rauti, S. Laurén, and V. Leppänen, "A survey on anti-honeypot and anti-introspection methods," in *World Conference on Information Systems and Technologies.* Springer, 2017, pp. 125–134.

[4] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *2017 IEEE Symposium on Security and Privacy (SP).* IEEE, 2017, pp. 1009–1024.

[5] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis, "No honor among thieves: A large-scale analysis of malicious web shells," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 1021–1032.

[6] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.

[7] IBM X-Force Research, "Understanding the webshell game: How command injection webshell attacks are a rising threat to networks," 2017. [Online]. Available: http://index-of.es/Attacks/RFI%20injection%20attacks/webshell_game.pdf

[8] NSA, "Detect and prevent web shell malware," April 2020. [Online]. Available: https://media.defense.gov/2020/Jun/09/2002313081/-1/-1/0/CSI-DETECT-AND-PREVENT-WEB-SHELL-MALWARE-20200422.PDF

[9] B. Hawkins and B. Demsky, "Zenids: introspective intrusion detection for php applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 2017, pp. 232–243.

[10] Y. Fang, Y. Qiu, L. Liu, and C. Huang, "Detecting webshell based on random forest with fasttext," in *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, 2018, pp. 52–56.

[11] H. Cui, D. Huang, Y. Fang, L. Liu, and C. Huang, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC).* IEEE, 2018, pp. 153–160.

[12] N.-H. Nguyen, V.-H. Le, V.-O. Phung, and P.-H. Du, “Toward a deep learning approach for detecting php webshell,” in *Proceedings of the Tenth International Symposium on Information and Communication Technology*, 2019, pp. 514–521.

[13] A. Croix, T. Debatty, and W. Mees, “Training a multi-criteria decision system and application to the detection of php webshells,” in *2019 International Conference on Military Communications and Information Systems (ICMCIS)*. IEEE, 2019, pp. 1–8.

[14] W. Kang, S. Zhong, K. Chen, J. Lai, and G. Xu, “Rf-adacost: Webshell detection method that combines statistical features and opcode,” in *International Conference on Frontiers in Cyber Security*. Springer, 2020, pp. 667–682.

[15] Z. Pan, Y. Chen, Y. Chen, Y. Shen, and X. Guo, “Webshell detection based on executable data characteristics of php code,” *Wireless Communications and Mobile Computing*, vol. 2021, 2021.

[16] Kitploit, “slopshell - the only php webshell you need,” May 2021. [Online]. Available: https://www.kitploit.com/2021/05/slopshell-only-php-webshell-you-need.html

[17] W3Techs, “Usage statistics of server-side programming languages for websites,” May 2021. [Online]. Available: https://w3techs.com/technologies/overview/programming_language

[18] PHP Group *et al.*, “PHP: Hypertext preprocessor,” 2021. [Online]. Available: http://www.php.net/

[19] ——, “Function and method listing,” 2021. [Online]. Available: https://www.php.net/manual/en/indexes.functions.php

[20] J. Sun, S. Liu, and K. Sun, “A scalable high fidelity decoy framework against sophisticated cyber attacks,” in *Proceedings of the 6th ACM Workshop on Moving Target Defense*, 2019, pp. 37–46.

[21] A. Bench, “ab-Apache HTTP server benchmarking tool,” *Accessed on*, May 2021.

[22] J. Nielsen, “Powers of 10: Time scales in user experience,” *Retrieved January*, vol. 5, p. 2015, 2009.

[23] M. Wegerer and S. Tjoa, “Defeating the database adversary using deception-a mysql database honeypot,” in *2016 International Conference on Software Security and Assurance (ICSSA)*. IEEE, 2016, pp. 6–10.

[24] “CVE-2019-0211.” National Vulnerability Database, 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-0211

[25] X. Jiang and D. Xu, “Collapsar: A vm-based architecture for network attack detention center.” in *USENIX Security Symposium*, 2004, pp. 15–28.

[26] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and N. Provos, “A hybrid honeypot architecture for scalable network monitoring,” *University of Michigan, Ann Arbor, Michigan, USA, Tech. Rep. CSE-TR-499–04*, 2004.

[27] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 148–162.

[28] W. Fan and D. Fernández, "A novel sdn based stealthy tcp connection handover mechanism for hybrid honeypot systems," in *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–9.

[29] W. Fan, Z. Du, M. Smith-Creasey, and D. Fernández, "Honeydoc: an efficient honeypot architecture enabling all-round design," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 683–697, 2019.

[30] A. Bulekov, R. Jahanshahi, and M. Egele, "Saphire: Sandboxing PHP applications with tailored system call allowlists," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[31] E. Cole, *Advanced persistent threat: understanding the danger and how to protect your organization*. Newnes, 2012.

[32] S. Morishita, T. Hoizumi, W. Ueno, R. Tanabe, C. Gañán, M. J. van Eeten, K. Yoshioka, and T. Matsumoto, "Detect me if you… oh wait. an internet-wide view of self-revealing honeypots," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2019, pp. 134–143.

[33] H. Artail, H. Safa, M. Sraj, I. Kuwatly, and Z. Al-Masri, "A hybrid honeypot framework for improving intrusion detection systems in protecting organizational networks," *computers & security*, vol. 25, no. 4, pp. 274–288, 2006.

[34] S. Schindler, B. Schnor, and T. Scheffler, "Hyhoneydv6: A hybrid honeypot architecture for ipv6 networks," *International Journal of Intelligent Computing Research*, vol. 6, 2015.

[35] E. Chovancová, N. Adám, A. Baláž, E. Pietriková, P. Fecil'ak, S. Šimoňák, and M. Chovanec, "Securing distributed computer systems using an advanced sophisticated hybrid honeypot technology," *Computing and Informatics*, vol. 36, no. 1, pp. 113–139, 2017.

[36] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018, pp. 54–66.

[37] W. Fan, "Contribution to the design of a flexible and adaptive solution for the management of heterogeneous honeypot systems," Ph.D. dissertation, ETSI Telecomunicación (UPM), 2017.

[38] S. Kyung, W. Han, N. Tiwari, V. H. Dixit, L. Srinivas, Z. Zhao, A. Doupé, and G.-J. Ahn, "Honeyproxy: Design and implementation of next-generation honeynet via sdn," in *2017 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2017, pp. 1–9.

[39] J. Rrushi, "Honeypot evader: Activity-guided propagation versus counter-evasion via decoy os activity," in *Proceedings of the 14th IEEE International Conference on Malicious and Unwanted Software*, 2019.

[40] X. Fu, W. Yu, D. Cheng, X. Tan, K. Streff, and S. Graham, "On recognizing virtual honeypots and countermeasures," in *2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*. IEEE, 2006, pp. 211–218.

[41] S. Mukkamala, K. Yendrapalli, R. Basnet, M. Shankarapani, and A. Sung, "Detection of virtual environments and low interaction honeypots," in *2007 IEEE SMC Information Assurance and Security Workshop*. IEEE, 2007, pp. 92–98.

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

[43] M. Dornseif, T. Holz, and C. N. Klein, "Nosebreak-attacking honeynets," in *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004*. IEEE, 2004, pp. 123–129.

[44] C. C. Zou and R. Cunningham, "Honeypot-aware advanced botnet construction and maintenance," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 199–208.

[45] T. Rytilahti and T. Holz, "On using application-layer middlebox protocols for peeking behind nat gateways." in *NDSS*, 2020.

[46] S. A. Shaikh, H. Chivers, P. Nobles, J. A. Clark, and H. Chen, "Network reconnaissance," *Network Security*, vol. 2008, no. 11, pp. 12–16, 2008.

[47] XDP-project, "The express data path (xdp) inside the linux kernel," https://github.com/xdp-project, 2020, accessed May, 2020.

[48] F. Mantog, "System and method for checksum offloading," Feb. 20 2007, US Patent 7,181,675.

[49] N. Memari, S. J. B. Hashim, and K. B. Samsudin, "Towards virtual honeynet based on lxc virtualization," in *2014 IEEE REGION 10 SYMPOSIUM*. IEEE, 2014, pp. 496–501.

[50] P. Srisuresh and K. Egevang, "Traditional ip network address translator (traditional nat)," RFC 3022, January, Tech. Rep., 2001.

[51] X. Xiao, A. Hannan, B. Bailey, and L. M. Ni, "Traffic engineering with mpls in the internet," *IEEE network*, vol. 14, no. 2, pp. 28–33, 2000.

[52] Z. Yang, Y. Cui, B. Li, Y. Liu, and Y. Xu, "Software-defined wide area network (sd-wan): Architecture, advances and opportunities," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2019, pp. 1–9.

[53] D. W. Wang, *Software Defined-WAN for the Digital Age: A Bold Transition to Next Generation Networking*. CRC Press, 2018.

[54] M. Attaran and J. Woods, "Cloud computing technology: improving small business performance using the internet," *Journal of Small Business & Entrepreneurship*, vol. 31, no. 6, pp. 495–519, 2019.

[55] S. Larbi, "Options for extending layer 2 on-premises networks to vmware cloud on aws," 2020. [Online]. Available: https://aws.amazon.com/blogs/apn/options-for-extending-layer-2-on-premises-networks-to-vmware-cloud-on-aws/

[56] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 481–500.

[57] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 395–411.

[58] McAfee, "Mcafee advanced threat defense - advanced detection for stealthy, zero-day malware," https://www.mcafee.com/enterprise/en-us/products/advanced-threat-defense.html, accessed in May. 2021.

[59] B. Inc, "Symantec content analysis - dynamic sandboxing," https://docs.broadcom.com/doc/malware-analysis-s400-s500-en, accessed in May. 2021.

[60] F. M. Analysis, "Safely execute and analyze malware in a secure environment," https://www.fireeye.com/solutions/malware-analysis.html, accessed in May. 2021.

[61] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–28, 2019.

[62] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *International Workshop on Recent Advances in Intrusion Detection.* Springer, 2011, pp. 338–357.

[63] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN).* IEEE, 2008, pp. 177–186.

[64] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna, "Escape from monkey island: Evading high-interaction honeyclients," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2011, pp. 124–143.

[65] D. Keragala, "Detecting malware and sandbox evasion techniques," *SANS Institute InfoSec Reading Room*, vol. 16, 2016.

[66] G. Pék, B. Bencsáth, and L. Buttyán, "nether: In-guest detection of out-of-the-guest malware analyzers," in *Proceedings of the Fourth European Workshop on System Security*, 2011, pp. 1–6.

[67] M. Brengel, M. Backes, and C. Rossow, "Detecting hardware-assisted virtualization," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2016, pp. 207–227.

[68] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing system virtual machines," in *Proceedings of the 19th international symposium on software testing and analysis*, 2010, pp. 171–182.

[69] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect cpu emulators," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, vol. 41, 2009, p. 86.

[70] A. Alwabel, H. Shi, G. Bartlett, and J. Mirkovic, "Safe and automated live malware experimentation on public testbeds," in *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, 2014.

[71] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the seventh european workshop on system security*, 2014, pp. 1–6.

[72] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, and B. Yener, "Avleak: fingerprinting antivirus emulators through black-box testing," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[73] H. S. Baird, A. L. Coates, and R. J. Fateman, "Pessimalprint: a reverse turing test," *International Journal on Document Analysis and Recognition*, vol. 5, no. 2, pp. 158–163, 2003.

[74] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes *et al.*, "Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion," in *International Symposium on Research in Attacks, Intrusions, and Defenses.* Springer, 2016, pp. 165–187.

[75] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[76] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained malware analysis using stealth localized-executions," in *2006 IEEE Symposium on Security and Privacy (S&P'06).* IEEE, 2006, pp. 15–pp.

[77] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies," *Black Hat*, vol. 1, pp. 1–27, 2012.

[78] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proceedings of the 1st ACM workshop on Virtual machine security*, 2009, pp. 11–22.

[79] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 227–238.

[80] M. Xu and T. Kim, "Platpal: Detecting malicious documents with platform diversity," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 271–287.

[81] C. Spensky, H. Hu, and K. Leach, "Lo-phi: Low-observable physical host instrumentation for malware analysis." in *NDSS*, 2016.

[82] J. Peng, K.-K. R. Choo, and H. Ashman, "User profiling in intrusion detection: A review," *Journal of Network and Computer Applications*, vol. 72, pp. 14–27, 2016.

[83] Y. Mangalapilly, "watchdog," https://github.com/gorakhargosh/watchdog, accessed in May. 2021.

[84] Moses-palmer, "pynput," https://github.com/moses-palmer/pynput, accessed in May. 2021.

[85] M. Hammond, "pywin32," https://github.com/mhammond/pywin32, accessed in May. 2021.

[86] Selenium, "selenium," https://github.com/SeleniumHQ/selenium/, accessed in May. 2021.

[87] Pywinauto, "pywinauto," https://github.com/pywinauto/pywinauto, accessed in May. 2021.

[88] R. Ferreira and R. L. Aguiar, "Repositioning privacy concerns: Web servers controlling url metadata," *Journal of Information Security and Applications*, vol. 46, pp. 121–137, 2019.

[89] F. Hassan, D. Sanchez, and J. Domingo-Ferrer, "Utility-preserving privacy protection of textual documents via word embeddings," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[90] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–48, 2019.

[91] G. Rodola, "psutil," https://github.com/giampaolo/psutil, accessed in May. 2021.

[92] G. Mills, "pytrends," https://github.com/GeneralMills/pytrends, accessed in May. 2021.

[93] T. Chakraborty, S. Jajodia, J. Katz, A. Picariello, G. Sperli, and V. Subrahmanian, "Forge: A fake online repository generation engine for cyber deception," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[94] P. M. Cao, Y. Wu, S. S. Banerjee, J. Azoff, A. Withers, Z. T. Kalbarczyk, and R. K. Iyer, "CAUDIT: Continuous auditing of SSH servers to mitigate brute-force attacks," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 667–682.

[95] A. Bulazel and B. Yener, "A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, 2017, pp. 1–21.

[96] S. O. Vashisht and A. Singh, “Turing test in reverse: new sandbox-evasion techniques seek human interaction,” 2014.

[97] D. Kirat, G. Vigna, and C. Kruegel, “Barecloud: Bare-metal analysis-based evasive malware detection,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 287–301.

[98] D. Kirat and G. Vigna, “Malgene: Automatic extraction of malware analysis evasion signature,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 769–780.

[99] A. Mills and P. Legg, “Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques,” *Journal of Cybersecurity and Privacy*, vol. 1, no. 1, pp. 19–39, 2021.

[100] L. M. Rossey, R. K. Cunningham, D. J. Fried, J. C. Rabek, R. P. Lippmann, J. W. Haines, and M. A. Zissman, “Lariat: Lincoln adaptable real-time information assurance testbed,” in *Proceedings, ieee aerospace conference*, vol. 6. IEEE, 2002, pp. 6–6.

[101] T. M. Braje, “Advanced tools for cyber ranges,” MIT Lincoln Laboratory Lexington United States, Tech. Rep., 2016.

[102] P. Megyesi, G. Szabó, and S. Molnár, “User behavior based traffic emulator: A framework for generating test data for dpi tools,” *Computer Networks*, vol. 92, pp. 41–54, 2015.

[103] S. Molnár, P. Megyesi, and G. Szabó, “Multi-functional traffic generation framework based on accurate user behavior emulation,” in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2013, pp. 13–14.

[104] P. Dutta, G. Ryan, A. Zieba, and S. Stolfo, “Simulated user bots: Real time testing of insider threat detection systems,” in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 228–236.

[105] J. Wroclawski, T. Benzel, J. Blythe, T. Faber, A. Hussain, J. Mirkovic, and S. Schwab, “Deterlab and the deter project,” in *The GENI Book*. Springer, 2016, pp. 35–62.

[106] S. Oesch, R. A. Bridges, M. Verma, B. Weber, and O. Diallo, “D2u: Data driven user emulation for the enhancement of cyber testing, training, and data set generation,” in *Cyber Security Experimentation and Test Workshop*, 2021, pp. 17–26.

[107] C. Kruegel, “Full system emulation: Achieving successful automated dynamic analysis of evasive malware,” in *Proc. BlackHat USA Security Conference*, 2014, pp. 1–7.

[108] C. K. Ng, L. Pan, and Y. Xiang, *Honeypot frameworks and their applications: a new framework*. Springer, 2018.

[109] S. Liu, P. Feng, S. Wang, K. Sun, and J. Cao, “Enhancing malware analysis sandboxes with emulated user behavior,” *Computers & Security*, p. 102613, 2022.

[110] S. M. Shieber, *The Turing test: verbal behavior as the hallmark of intelligence.* Mit Press, 2004.

[111] J. Gruber, "Daring fireball: Markdown," *Récupéré le*, vol. 3, no. 04, 2004. [Online]. Available: https://daringfireball.net/

[112] Microsoft, "Windows application driver," https://github.com/microsoft/WinAppDriver, accessed in March. 2022.

[113] J. MacFarlane, "Pandoc - a universal document converter," https://pandoc.org/releases.html, accessed in March. 2022.

[114] N. Team, "Nltk," https://www.nltk.org/, accessed in March. 2022.

[115] OpenCV, "Opencv on wheels," https://github.com/opencv/opencv-python, accessed in March. 2022.

[116] M. A. Lee, "Python tesseract," https://github.com/madmaze/pytesseract, accessed in March. 2022.

[117] A. Neubeck and L. Van Gool, "Efficient non-maximum suppression," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 3. IEEE, 2006, pp. 850–855.

[118] R. F. Foundation, "Robot framework," https://robotframework.org/, accessed in March. 2022.

[119] R. Hocke, "Sikulix," https://sikulix.github.io/, accessed in March. 2022.

[120] Y. Li, J. He, X. Zhou, Y. Zhang, and J. Baldridge, "Mapping natural language instructions to mobile ui action sequences," in *Annual Conference of the Association for Computational Linguistics (ACL 2020)*, 2020.

[121] L. She, Y. Cheng, J. Y. Chai, Y. Jia, S. Yang, and N. Xi, "Teaching robots new actions through natural language instructions," in *The 23rd IEEE International Symposium on Robot and Human Interactive Communication.* IEEE, 2014, pp. 868–873.

[122] S. R. Branavan, H. Chen, L. Zettlemoyer, and R. Barzilay, "Reinforcement learning for mapping instructions to actions," in *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, 2009, pp. 82–90.

[123] R. Manuvinakurike, J. Brixey, T. Bui, W. Chang, D. S. Kim, R. Artstein, and K. Georgila, "Edit me: A corpus and a framework for understanding natural language image editing," in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

Curriculum Vitae

Songsong Liu received his Bachelor of Engineering degree in Information Security from Wuhan University in 2012 and his Master of Engineering degree in Information Security from Huazhong University of Science and Technology in 2015. His research interests include system security, moving target defense, and digital forensics.