Cyprus University of Technology
Faculty of Engineering and Technology

Bachelor's Thesis

Exploring the fingerprintability of honeypot systems, based on observed discrepancies, and designing and proposing techniques for preventing detection

Stylianos Kyriakou

Limassol, May 2025

Τεχνολογικό Πανεπιστήμιο Κύπρου

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΗΜΠ 412 -Διπλωματική Εργασία Ακαδημαϊκό έτος 2024-2025

Όνομα Φοιτητή / ΑΦΤ : Στυλιανός Κυριάκου / 22603

Βαθμός: 9.0

Τίτλος: Study the fingerprintability of honeypot systems, based on observed discrepancies and designing and proposing techniques for prevention

Επιβλέπων Καθηγητής:

Όνομα	Υπογραφή	Ημερ.
Παναγιώτης Ηλία		3/6/2025

Εξεταστής 1:

Όνομα	Υπογραφή	Ημερ.
Αντρέας Δημοσθένους		3/6/2025

Εξεταστής 2:

Όνομα	Υπογραφή	Ημερ.
Χρίστος Λοΐζου		3/6/2025

CYPRUS UNIVERSITY OF TECHNOLOGY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER ENGINEERING AND INFORMATICS

Bachelor's Thesis

Exploring the fingerprintability of honeypot systems, based on observed discrepancies, and designing and proposing techniques for preventing detection

Stylianos Kyriakou

Supervisor

Faculty of Engineering and Technology

Dr. Panagiotis Ilia

Lecturer in the Department of Electrical Engineering,

Computer Engineering and Informatics

Limassol, May 2025

Copyrights

I would like to thank my professors for helping me achieve my goals and complete this thesis. This wouldn’t have been possible without my supervisor Dr. Panagiotis Ilia. I am very grateful for the support that my friends and family have given me throughout my academic journey.

ABSTRACT

Cybersecurity is a crucial aspect of today's highly interconnected world. Nowadays, everything is connected to the internet, exposing a potentially broad surface to attacks, since everything is susceptible. From personal computers, to servers, cloud infrastructure, mobile devices, sensors, and even electrical appliances - almost every device we use in our daily life - is potentially vulnerable and susceptible to attacks, and is being attacked daily. One technology that has been designed and deployed to help us better understand how attackers and malicious actors act, is honeypots. Honeypots are decoy systems that aim to entrap attackers into thinking that they managed to gain access to a system but in reality, they are in a separate environment aimed at monitoring their behavior and letting the administrators know how an attacker is carrying out their attack, what vulnerabilities they might go after, and what tools or techniques they are using. This in the end can help the developers improve their system’s security. The problem arises when an attacker figures out that they are interacting with a honeypot - a process known as fingerprinting - and, as a result, they either avoid carrying out their attack or, in some cases, even turn the honeypot against its owner. This thesis aims to help make honeypots undetectable so they can spy on the attackers and let the developers know their system’s weakest links. By analyzing headers, banners, and service behaviors, and comparing them to those of real-world machines, I aim to suggest practical techniques that enhance the stealth and effectiveness of honeypots.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

In our days there are so many electronic devices connected to the internet. They are a huge part of our lives and they hold a significant amount of important data. Being connected to the internet imposes many threats. A large number of devices are being attacked daily by bad actors. It is a widespread issue of modern-day technology where data theft and unauthorized access occur constantly, especially in big organizations. Cybersecurity specialists are constantly working on safeguarding personal devices and servers to avoid attacks as much as possible but it is never an easy thing to do nor can they guarantee absolute protection. Especially when it comes to servers, organizations face attacks by bad actors constantly and have their data stolen which heavily impacts their financial stability and their customers. One of the tools that organizations use to detect and study attacks is honeypots. Honeypots provide the ability to emulate a vulnerable system which in turn observes the attacker's behavior and helps the administrator pinpoint areas where they can improve the security of their system. They also distract the attackers from the real system.

1.1 Research Contribution & Objectives

Honeypots are a tool of cybersecurity; they are intruder traps designed to deceive attackers by making them believe that they gained access to an organization’s system. It's a common practice for organizations to deploy decoy systems (i.e., honeypots) in order to lure attackers, capture their behavior, and detect and understand potential threats and attacks. However, in some cases, attackers may realize that they are actually interacting with a decoy system and refrain from performing any reconnaissance actions or deploying any attacks. This thesis aims to study the most prominent honeypot systems, assess whether they are fingerprintable, based on the services running on them and whether discrepancies exist between the expected and observed behavior. Furthermore, to propose techniques to minimize their fingerprintability vector, in order to prevent detection. Honeypots run separately from the actual system and their purpose is to spy on the attacker while they are performing an attack. That helps the system administrator understand what attacks might be carried against their system in order to

improve it and make it more secure. In the meantime, the actual system is safe from that attack because it is separate from the honeypot. The problem is that the attackers might sometimes figure out, by using fingerprinting tools, that they are interacting with a honeypot and not carry out their attack or even use the honeypot to their advantage.

We identified discrepancies in popular honeypot systems and proposed solutions to make them less vulnerable to fingerprinting attacks. Our analysis involved examining open ports, running services, and the headers and banners they exposed. We also searched for machines listed in a search engine that gathers information about systems, that matched the fingerprint of known honeypots, revealing inconsistencies that could compromise stealth. By addressing these issues, we contributed to the design of a more secure honeypot capable of monitoring attackers while remaining undetected. We recommended ways to improve honeypots and enhance their effectiveness, ensuring they can achieve their goal and stay under the radar, avoiding detection. As a result, our work strengthened existing research by eliminating critical vulnerabilities.

Our target is to improve the security mechanisms of organizations using advanced technology (honeypots), which we improved accordingly, based on our research. Honeypots rely on their ability to stay under the radar, to spy on unsuspecting attackers. While many have tried to make honeypots less fingerprintable, they overlooked discrepancies in headers and banners which makes their research less accurate in this regard. We aimed to complement their work. We first analyzed how attackers fingerprint honeypots. We examined how widespread the problem with discrepancies in headers and banners for services is, in the real world. We then suggested methods, rules and techniques, in order to eliminate discrepancies and reduce the honeypots' fingerprintability vector. This will in turn result in honeypots that are harder to detect.

To summarize, the main contributions of our research are as follows:

- We analyze how attackers fingerprint honeypots using discrepancies in service headers and banners.
- We complement existing research by addressing overlooked discrepancies in honeypot deployments.
- We conduct a large-scale measurement study to validate our fingerprinting detection methodology.

- We propose improvements to minimize honeypot fingerprintability and enhance stealthiness.

1.2 Related Work

We analyzed existing techniques to identify their limitations and found that our approach - detecting discrepancies in service headers and banners - effectively complements and extends prior research. To validate this methodology and determine the real-world extent of the issue, we conducted a large-scale measurement study focused on: (a) confirming the reliability of our discrepancy-based detection method, and (b) evaluating how widespread the problem is across deployed systems.

We focused on identifying subtle discrepancies in headers and banners that attackers use to fingerprint honeypots. By addressing these overlooked details, my research will lead to more realistic and dynamic honeypots, making them harder to detect and more effective against advanced attackers. This will help in creating a more effective security tool that can deceive attackers for longer periods, ultimately enhancing its ability to gather valuable intelligence and prevent attacks.

Many different techniques are used by attackers to fingerprint honeypots, some of which are using fingerprinting tools Nmap and Xprobe2 to collect the device type, OS details, OS version number, device model, fingerprintable architecture and finally, the version of the service running on the port. This is achieved by OS detection, version detection, script scanning, traceroute, aggressive fingerprinting and OS predictions, providing very close OS match results when they can't find the perfect OS match [6]. Others use information from OSI layers 4 and 7, from each network service, such as network application banner, application version and network protocols [11]. Some manage to fingerprint the host’s online state, IP/MAC address, OS and incoming/outgoing traffic. Active fingerprinting, which is based on TCP and ICMP is achieved using signature detection in which unique messages are sent as probes to the target system and the header fields of the responses are analyzed to determine the OS. The difference in the responses shows the difference in deployment. Passive fingerprinting is achieved by sniffing the network for packets transmitted by the target system. This data is compared with the OS signatures database to fingerprint the remote

host and its services because each OS uses a different implementation of the TCP/IP stack and ICMP [12]. Another method is by using ZMap and doing a one-packet scan sending TCP SYN packets to ports 22, 23 and 80 using the DNS OARC exclusion list. Then, a custom scanner that connects on the appropriate port visits responsive IPv4 addresses and sends probes to fingerprint honeypots. Disconnection messages at various stages of the protocol exchange result in a large source of differences that can act as distinguishing factors, and explain its origin [13].

Developers are trying to make honeypots less fingerprintable by catching fingerprinting attacks in real time. This is achieved by analyzing irregularities in TCP/IP packets, such as TCP options, TCP flags, ICMP requests, ICMP packet sizes, and UDP requests. They are using a fuzzy technique for recognizing the fingerprinting attack on the honeypot based on the input packets and their outputs [7]. Some, perform Principal Component Analysis (PCA) to determine the key factors by which a fingerprinting attack can be uncovered [8]. Others, use Dynamic Fuzzy Rule Interpolation (D-FRI) where the honeypot uses a dynamic rule-based system that adjusts over time to identify and address new fingerprinting attempts by learning from network traffic and constantly updating its rule base, avoiding consistent behaviors that could be fingerprinted [9]. Another technique is imitating the network stack behavior of operating systems to trick fingerprinting tools, creating virtual routing topologies [10].

Despite these attempts, honeypots still face many challenges when it comes to their fingerprintability. The developers most of the time don’t implement a protocol the same way as the server they impersonate, but instead rely on off-the-shelf libraries, they use deployment scripts, docker containers or just copy and paste the source files, including the same host key, to all their honeypots and use old versions of the services they impersonate [13].

1.3 Methodology

By looking for inconsistencies/ discrepancies in honeypot systems, we understood how they are being fingerprinted by attackers. We examined the headers and banners of the services running on their open ports. First, we set up some of the honeypots from this list [1] in Virtual Machines and observed which of their ports are open, what services

are running on them, and performed banner grabbing to analyze their headers and banners. This allowed us to document the observed characteristics in a controlled environment and understand how they differ from real systems. We then looked for discrepancies in machines given by shodan.io that fit the ones that we found with our technique and analysis. By comparing these findings with the data provided by Shodan, we were able to confirm that the discrepancies we identified are not only theoretical but actually present in honeypots. Finally, we suggested improvements for their weak points, in order for them to remain undercover and observe the attacker's behavior.

We managed to identify honeypots due to an outdated service version they had, which matched a honeypot banner. This underlines the importance of keeping service versions in honeypots updated. Honeypots that are not regularly updated to the latest versions of the services they mimic run the risk of being fingerprinted by attackers. We recommended that banners should be kept updated with the latest software versions of a service.

2 Background

2.1 Basic Concepts

In computer networks, ports are logical communication endpoints that allow a device to differentiate between services running on a system. They are part of the transport layer (Layer 4) in the OSI model and in conjunction with IP addresses enable communication between devices in a network. Ports are identified by a 16-bit number from 0 to 65535 and each service has its own port. When a service starts it listens for incoming requests on its port. Here are lists of ports and the services running on them [2], [3].

The transport layer is responsible for end-to-end communication with reliability and efficiency in data transfer between devices in a network. It breaks and reassembles data and ensures that a sender does not send too much data to a receiver at one. It also checks for corrupted data to retransmit it, allows many applications to use the network at the same time and uses port numbers to tell communication streams apart. Finally, it establishes and terminates connections.

IP addresses (Internet Protocol addresses) are unique identifiers assigned to each device connected to a network. It enables them to communicate with each other over the internet. There are two types, IPv4 (Internet Protocol version 4) and IPv6 (Internet Protocol version 6) and they are further classified into public, private, static, dynamic and loopback IP addresses. Devices use their IP addresses to identify and communicate with each other.

2.2 Headers, Banners and Banner Grabbing

Services have headers and banners which provide information about the service running on a port. They can reveal the software, version or something else which can be useful to identify the service. Headers are metadata or information sent by a service to manage communication by providing instructions for data exchange and banners are a response sent by a service when a connection is established to provide identifying information. Headers and banners are important for administrators because they help identify a service that runs to assist in troubleshooting. They are also useful to attackers because they are used in fingerprinting to identify the service and its version to help find vulnerabilities for it. If they are too revealing they can expose a service to threats.

Banner grabbing is a technique used to gather information about a service running on a device that is connected to a network. This is achieved by establishing a connection to a specific port on a device and retrieving the banner, which is information sent by the service running on that port. This technique is used by administrators to check what services and versions are running, by security analysts to perform penetration testing and identify vulnerabilities and by attackers to identify software versions to exploit. There are many tools that can be used for banner grabbing like Telnet, Netcat, Python scripts or by connecting to the service using the appropriate command. Banner grabbing can be detected by an IDS (Intrusion Detection System) as a threat. Administrators should hide the banners, use firewalls and IDS, update the software and encrypt communication.

2.3 Honeypots

A honeypot is a cybersecurity tool that attracts attackers by simulating a real, vulnerable and valuable system, and analyses their behavior. It deceives attackers because in reality, it is not a real system. This allows security analysts to observe the attacker's tactics and tools without exposing the actual system. Honeypots are used for detecting unauthorized intrusion attempts, to warn for threats. In addition, they help improve the system's security based on the attack data that is gathered and turn the attacker's attention away from the real system by consuming their time in a fake environment. The attackers believe that they have gained access to an actual system but all their actions are being logged for analyzing and further strengthening of the security of the actual system.

Honeypots are classified based on interaction level and purpose. In regards to interaction level there are low-interaction and high-interaction honeypots and when it comes to purpose there are production and research honeypots. Low-interaction honeypots simulate very basic functionality, they are easy to set up but are more easily fingerprintable by attackers and they provide limited data on their behavior. High-interaction honeypots on the other hand, provide fully functional systems, they are more complex to set up but they are harder to identify and they provide rich insight into the attacker's behavior. Production honeypots are deployed in live environments to protect real systems and they are focused on detecting and avoiding threats. However, research honeypots are used for studying how attackers behave and have nothing to do with protecting actual systems. They are deployed either at the network perimeter, like the DMZ (demilitarized zone) to attract external attackers or inside the network to detect insider threats and lateral movement. They can be set up as VMs (virtual machines), containers, or specialized software that mimics vulnerable systems or services, depending on the deployment goal. Depending on their purpose, they may be placed outside the firewall to monitor unwelcome internet traffic or inside the firewall to observe internal or targeted attacks.

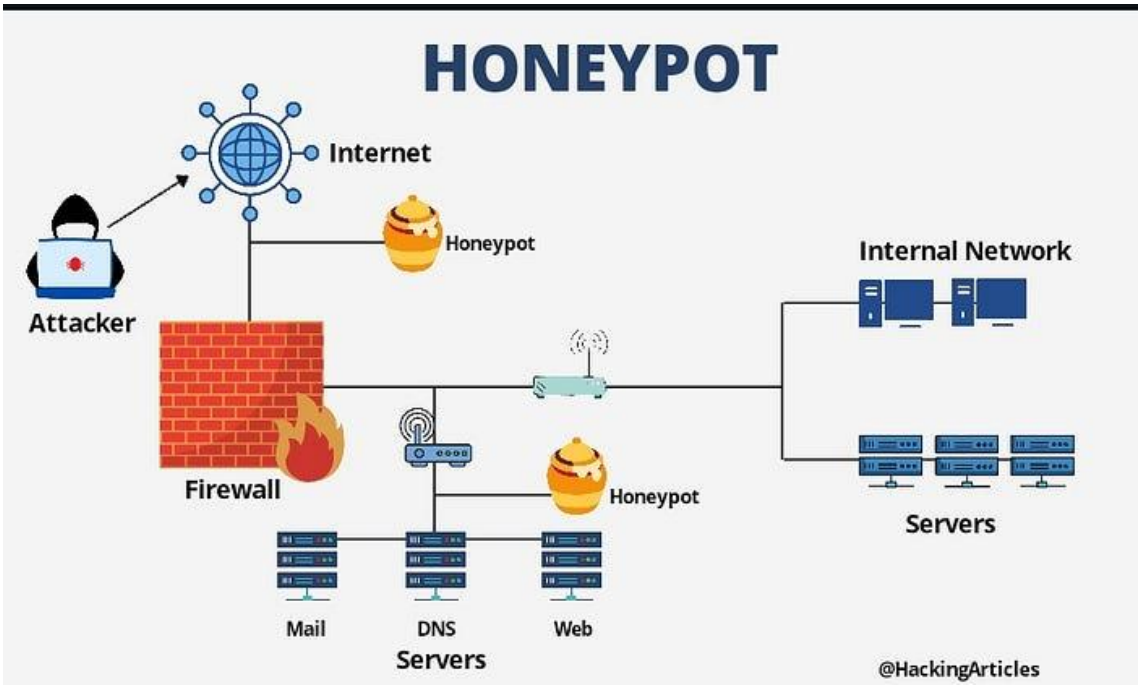


Figure 1: Where are honeypots deployed

Source: [4]

Honeypots pose some risks because they might get fingerprinted by attackers. Also, if they are not properly isolated, attackers can use a compromised honeypot to turn it against the administrator and launch an attack. This is why it is extremely important for administrators to make their honeypots hard to identify.

2.4 Fingerprinting

Fingerprinting is the process of identifying a system or application based on its characteristics and behavior. It is widely used in cybersecurity to gain information about a system or service. Attackers use it to identify Operating Systems, software or services running on a system, in order to exploit vulnerabilities in the identified system or service. Administrators use it to identify unauthorized devices in a network and they can understand the attacker's behavior by analyzing their fingerprinting techniques.

There are two types of fingerprinting, active fingerprinting and passive fingerprinting. Active fingerprinting involves sending specially crafted packets to a system and

analyzing its responses. Some examples are, sending modified TCP or UDP packets and observing the target's response or finding discrepancies in protocol implementation or error messages. Active fingerprinting provides detailed and accurate information but can be detected by IDS (Intrusion Detection Systems) and alert the target for reconnaissance activity. Passive fingerprinting, on the other hand, involves analyzing traffic or communication from the target without sending any packets. For example, packet headers, TTL values or sequence numbers in existing traffic can be observed. Passive fingerprinting is stealthier and harder to detect but provides less information in comparison to active fingerprinting.

Tools that can be used for fingerprinting, for offensive purposes are Nmap, Netcat, Zmap, Amap and Metasploit. When it comes to defensive purposes, Wireshark, Snort or Zeek can be used.

Nmap (Network Mapper) is a popular, open-source tool for network discovery and security checking. It is used to scan networks, identify hosts and services and perform vulnerability assessments. It scans ports to detect open, closed or filtered ports on a target. Also, it identifies running services and their versions and determines the Operating System and its version. Finally, it has a scripting engine, NSE (Nmap Scripting Engine) for vulnerability scanning.

To understand how fingerprinting works, an OS fingerprinting example is with TCP/IP. This method involves sending a SYN packet with specific flags or parameters to a target and analyzing the SYN-ACK response for TTL value, window size or TCP options. A connection to a service can be made and then the headers or banners can be retrieved and analyzed for software and version details.

In order to defend against fingerprinting, developers should obfuscate or hide information by disabling or modifying banners. Also, they should use security tools like firewalls, IDS, IPS to detect and block fingerprinting. Finally, they should update their software for known vulnerabilities to get patched and implement honeypots to trap the attackers. Honeypots should have good protocol emulation, random behavior and should mimic real configurations in order for them to match the environment they are protecting. Finally, they should have hidden network artifacts, they should be deployed alongside real systems and they should be kept updated.

2.5 Tools

Shodan is a specialized search engine that finds devices, services and systems connected to the internet. It is unlike traditional search engines like Google Search which provide websites and web content. It finds internet connected devices and provides details about them. Shodan scans the internet for open ports and services and collects data from these devices like device type, service headers and banners, metadata, geographic location, Operating Systems, software versions, vulnerabilities and provides information on whether it believes the device to be a honeypot. It works by sending probes and analyzing the data from the systems that respond. Finally, it identifies IoT devices, industrial systems, databases, servers, webcams, printers and smart appliances.

T-Pot is a multi-honeypot platform developed by the German telecommunications company, Deutsche Telekom. It can deploy and help the administrator manage multiple honeypots on a system. This allows them to gather information about the attackers' behavior and techniques. It integrates many open-source honeypots and monitoring tools into one platform, which makes it a good network cybersecurity tool. Firstly, by combining many different honeypots it can emulate many systems and services, in order to attract many attackers and record different attacks. In addition, it provides a dashboard for monitoring, visualizing and analyzing the collected attack data. It can also show attack trends, the geographic location of attackers and expose their tools. It is easy to deploy and set up using Docker and the honeypots are in a container to make it difficult to attack the host or other systems. Finally, it includes a NIDS (Network Intrusion Detection System) to help analyze traffic. This makes T-Pot suitable to be used to gather threat intelligence, for security research, for network defense and for educational purposes. It is comprehensive in the tools that it provides, scalable because by using Docker the administrator can add or remove components and is open-source. On the other hand, because it provides many services it can be resource-intensive and will require maintenance to update.

3 Methodology

3.1 Overview

We installed four popular honeypots in Virtual Machines and scanned them using Nmap to identify their open ports and running services. We also performed banner grabbing on the honeypots. In addition, we developed Python scripts to search Shodan, to collect 10000 IP addresses, scan their open ports, and perform banner grabbing. We also developed Python scripts to analyze the gathered data from the 10000 real-world systems and compare them with the honeypots.

Specifically, we wrote a Python script to search Shodan for 10000 server IP addresses, using filters apache, openssh, mysql, https and email. Then, we wrote a Python script to extract these 10000 IPs from Shodan's results. Next, we wrote a Python script to scan these IPs using Nmap to find their open ports. Also, we wrote a Python script to banner grab these IPs using the commands we used to banner grab the honeypots, to find matching banners. Additionally, we wrote a Python script to analyze the number of open ports associated with each IP address. Moreover, we wrote a Python script to analyze the number of matching ports for each IP with the honeypots and the number of IPs with all the ports of a honeypot open. Finally, we wrote a Python script to analyze the number of unique IPs with matching banners for each honeypot port and the number of matching banners for each IP.

Initially, we concluded that we had not yet identified any honeypots. Given our limited resources, in order to find honeypots, we had to search Shodan using more specific filters. To improve our chances of fingerprinting honeypots on servers, we installed four additional honeypots, allowing us to search for more matching ports and banners. Finally, we searched Shodan for 1000 more IPs using filters for specific ports.

Overall, our methodology involved a total of eight honeypots and 11000 real-world IP addresses. This broader setup enabled a more detailed comparison between honeypots and real systems. By expanding both the variety of honeypots and the scale of real-world data, we improved our chances of identifying patterns, revealing discrepancies, and ultimately assessing the fingerprintability of honeypot deployments.

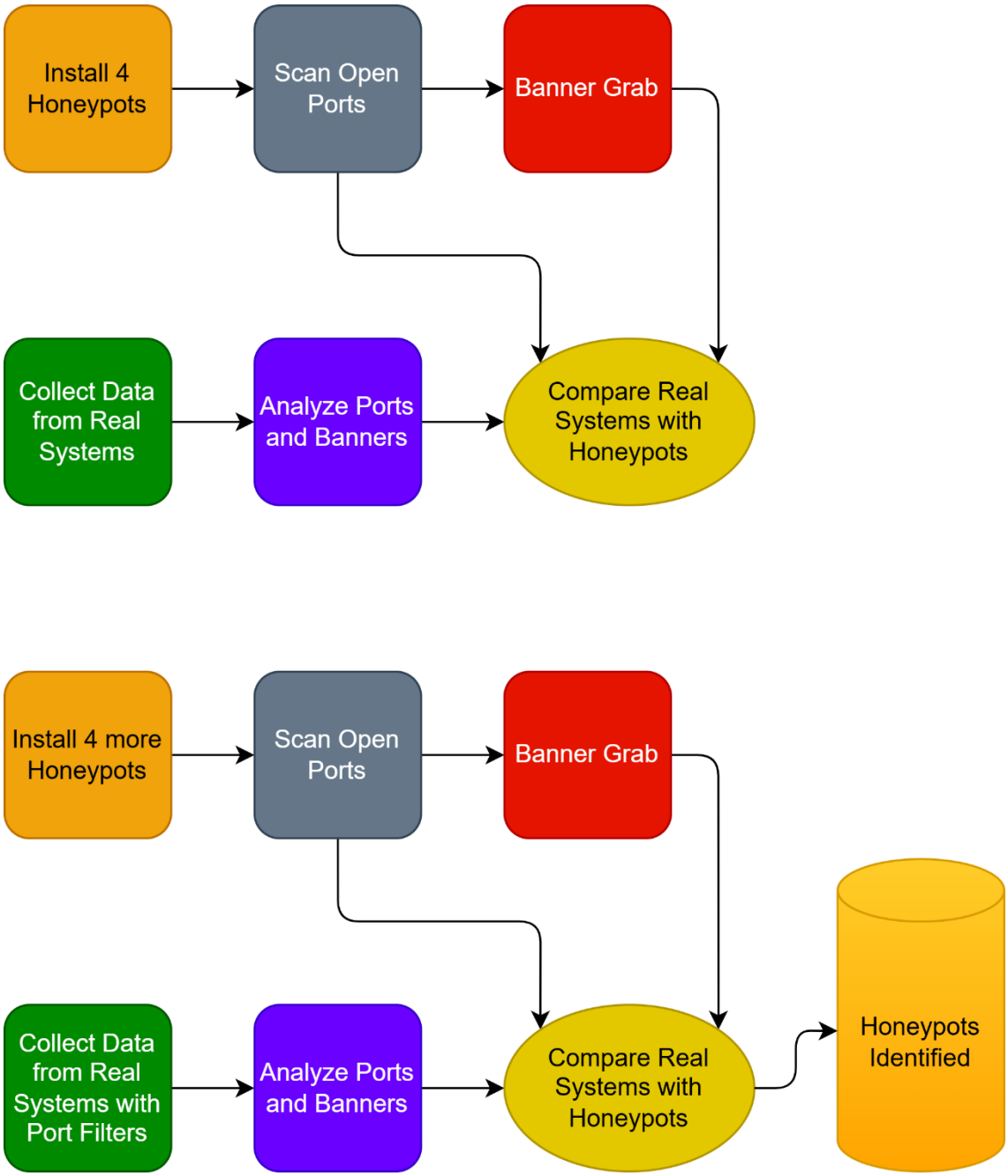


Figure 2: Methodology

3.2 Experiment 1

We started off by downloading, installing and configuring four popular honeypots, Conpot, Cowrie, Dionaea and T-Pot from this list [1] in Virtual Machines. In order to be able to install the honeypots we also had to install their dependencies. This website is

for T-Pot honeypot, developed by Deutsche Telekom. We used Oracle VirtualBox and our VMs used Linux Ubuntu. We downloaded Conpot using Python pip, we couldn't get Docker to work. We downloaded Cowrie using Docker. We downloaded Dionaea using Docker, we couldn't get Ubuntu installation to work. Finally, we downloaded T-Pot with Sensor installation type which consists of these honeypots: adbhoney, ciscoasa, citrixhoneypot, conpot, cowrie, dicompot, dionaea, elasticpot, heralding, honeypy, honeysap, honeytrap, mailoney, medpot, rdpy, snare & tanner and these tools: cockpit, ewsposter, fatt, p0f & suricata. We then scanned these honeypots using Nmap to find which ports they have open and which services are running on them. We then proceeded to banner grab these honeypots. Below are the open ports, services, headers and banners we got for each honeypot.

3.2.1 Conpot

Ports

```
PORT      STATE SERVICE
631/tcp   open  ipp
2121/tcp  open  ccproxy-ftp
5020/tcp  open  zenginkyo-1
8800/tcp  open  sunwebadmin
10201/tcp open  rsms
44818/tcp open  EtherNetIP-2
```

Services

```
PORT     STATE SERVICE      VERSION
631/tcp  open  ipp          CUPS 2.4
2121/tcp open  ccproxy-ftp?
8800/tcp open  sunwebadmin?
```

Conpot has port 631 open with ipp CUPS 2.4 running, port 2121 open with ccproxy-ftp running, port 5020 open with zenginkyo-1 running, port 8800 open with sunwebadmin

running, port 10201 open with rsms running and port 44818 open with EtherNetIP-2 running.

Headers and Banners

Port 631

```
root1@Conpot:~/Conpot$ curl -s -I http://localhost:631
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Language: en_US
Content-Length: 2262
Content-Type: text/html; charset=utf-8
Date: Sat, 04 Jan 2025 08:51:44 GMT
Keep-Alive: timeout=10
Last-Modified: Thu, 26 Sep 2024 11:15:36 GMT
Accept-Encoding: gzip, deflate, identity
Server: CUPS/2.4 IPP/2.1
X-Frame-Options: DENY
Content-Security-Policy: frame-ancestors 'none'
```

Port 2121

```
root1@Conpot:~/Conpot$ curl ftp://localhost:2121 -v
* Host localhost:2121 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:2121...
* connect to ::1 port 2121 from ::1 port 37176 failed: Connection refused
*   Trying 127.0.0.1:2121...
* Connected to localhost (127.0.0.1) port 2121
< 200 FTP server ready.
* Got a 200 ftp-server response when 220 was expected
* Closing connection
curl: (8) Got a 200 ftp-server response when 220 was expected
```

Port 5020

```
root1@Conpot:~/Conpot$ curl localhost:5020 -v
* Host localhost:5020 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:5020...
* connect to ::1 port 5020 from ::1 port 46658 failed: Connection refused
*   Trying 127.0.0.1:5020...
* Connected to localhost (127.0.0.1) port 5020
> GET / HTTP/1.1
> Host: localhost:5020
> User-Agent: curl/8.5.0
> Accept: */*
>
* Empty reply from server
* Closing connection
curl: (52) Empty reply from server
```

Port 8800

```
root1@Conpot:~/Conpot$ curl -I http://localhost:8800
HTTP/1.1 302 Found
Date: Sat, 04 Jan 2025 08:29:20 GMT
Content-Type: text/html
Location: /index.html
Content-Length: 0
```

Port 10201

```
root1@Conpot:~/Conpot$ curl -v http://localhost:10201
* Host localhost:10201 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
```

```
*   Trying [::1]:10201...
* connect to ::1 port 10201 from ::1 port 38696 failed: Connection refused
*   Trying 127.0.0.1:10201...
* Connected to localhost (127.0.0.1) port 10201
> GET / HTTP/1.1
> Host: localhost:10201
> User-Agent: curl/8.5.0
> Accept: */*
>
* Empty reply from server
* Closing connection
curl: (52) Empty reply from server
```

```
root1@Conpot:~/Conpot$ openssl s_client -connect localhost:10201
CONNECTED(00000003)
40479AA3D5740000:error:0A000126:SSL routines:ssl3_read_n:unexpected eof while reading:../ssl/record/rec_layer_s3.c:316:
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 0 bytes and written 300 bytes
Verification: OK
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
```

Port 44818

```
root1@Conpot:~/Conpot$ curl -vk https://localhost:44818
* Host localhost:44818 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:44818...
* connect to ::1 port 44818 from ::1 port 42898 failed: Connection refused
*   Trying 127.0.0.1:44818...
* Connected to localhost (127.0.0.1) port 44818
* ALPN: curl offers h2,http/1.1
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
```

```
root1@Conpot:~/Conpot$ openssl s_client -connect localhost:44818
CONNECTED(00000003)
4097584AC9760000:error:0A000126:SSL routines:ssl3_read_n:unexpected eof while reading:../ssl/record/rec_layer_s3.c:316:
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 0 bytes and written 300 bytes
Verification: OK
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
```

Below is a table showing for each port whether it has a header, a banner or none.

Table 1: Conpot ports headers and banners

Port	Header	Banner
631	✓	✗
2121	✗	✓
5020	✓	✗
8800	✓	✗
10201	✓	✗
44818	✓	✗

3.2.2 Cowrie

Ports

```
PORT     STATE    SERVICE
2222/tcp filtered EtherNetIP-1
```

Services

```
PORT     STATE SERVICE VERSION
2222/tcp open  ssh     OpenSSH 6.0p1 Debian 4+deb7u2 (protocol 2.0)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Cowrie has port 2222 open with EtherNetIP-1/ ssh version OpenSSH 6.0p1 Debian 4+deb7u2 (protocol 2.0) running.

Headers and Banners

Port 2222

```
root1@Cowrie:~$ ssh -vvv 172.17.0.1
```

```
OpenSSH_9.6p1 Ubuntu-3ubuntu13.5, OpenSSL 3.0.13 30 Jan 2024
```

```
root1@Cowrie:~$ openssl s_client -connect 172.17.0.1:2222
CONNECTED(00000003)
40C7D45C6F7F0000:error:0A00010B:SSL routines:ssl3_get_record:wrong version number:../ssl/record/ssl3_record.c:354:
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 5 bytes and written 293 bytes
Verification: OK
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
```

Below is a table showing for each port whether it has a header, a banner or none.

Table 2: Cowrie ports headers and banners

Port	Header	Banner
2222	✓	✓

3.2.3 Dionaea

Ports

```
PORT      STATE    SERVICE
21/tcp    filtered ftp
42/tcp    filtered nameserver
80/tcp    filtered http
135/tcp   filtered msrpc
443/tcp   filtered https
445/tcp   filtered microsoft-ds
1433/tcp  filtered ms-sql-s
1723/tcp  filtered pptp
1883/tcp  filtered mqtt
3306/tcp  filtered mysql
5060/tcp  filtered sip
5061/tcp  filtered sip-tls
11211/tcp filtered memcache
```

Services

```
PORT     STATE    SERVICE      VERSION
21/tcp   filtered ftp
42/tcp   filtered nameserver
80/tcp   filtered http
135/tcp  filtered msrpc
443/tcp  filtered https
445/tcp  filtered microsoft-ds
1433/tcp filtered ms-sql-s
1723/tcp filtered pptp
3306/tcp filtered mysql
5060/tcp filtered sip
5061/tcp filtered sip-tls
```

Dionaea has port 21 open with ftp running, port 42 open with nameserver running, port 80 open with http running, port 135 open with msrpc running, port 443 open with https running, port 445 open with microsoft-ds running, port 1433 open with ms-sql-s running, port 1723 open with pptp running, port 1833 open with mqtt running, port

3306 open with mysql running, port 5060 open with sip running, port 5061 open with sip-tls running and port 11211 open with memcache running.

Headers and Banners

Port 21

```
root1@Dionaea:~$ ftp 172.17.0.1
Connected to 172.17.0.1.
220 DiskStation FTP server ready.
Name (172.17.0.1:root1): root1
331 Password required for root1.
Password:
230 User logged in, proceed
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

Port 80

```
root1@Ubuntu:~$ curl -I http://172.17.0.1
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/html; charset=ascii
Content-Length: 204
Connection: close
```

```
root1@Dionaea:~$ wget --server-response --spider http://172.17.0.1:80
Spider mode enabled. Check if remote file exists.
--2025-01-04 10:39:39--  http://172.17.0.1/
Connecting to 172.17.0.1:80... connected.
HTTP request sent, awaiting response...
  HTTP/1.1 200 OK
  Server: nginx
```

```
  Content-Type: text/html; charset=ascii
  Content-Length: 204
  Connection: close
Length: 204 [text/html]
Remote file exists and could contain further links,
but recursion is disabled -- not retrieving.
```

Port 443

```
root1@Dionaea:~$ curl -I -k http://172.17.0.1
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/html; charset=ascii
Content-Length: 204
Connection: close
```

```
root1@Ubuntu:~$ wget --server-response --spider --no-check-certificate
https://172.17.0.1
Spider mode enabled. Check if remote file exists.
--2025-01-03 15:27:29--  https://172.17.0.1/
Connecting to 172.17.0.1:443... connected.
WARNING: cannot verify 172.17.0.1's certificate, issued by
‘OU=anv,O=dionaea.carnivore.it,CN=Nepenthes Development Team,C=DE’:
  Self-signed certificate encountered.
    WARNING: certificate common name ‘Nepenthes Development Team’ doesn't
match requested host name ‘172.17.0.1’.
HTTP request sent, awaiting response...
  HTTP/1.1 200 OK
  Server: nginx
  Content-Type: text/html; charset=ascii
  Content-Length: 204
  Connection: close
Length: 204 [text/html]
Remote file exists and could contain further links,
but recursion is disabled -- not retrieving.
```

Port 3306

```
root1@Dionaea:~$ mysql -h 172.17.0.1 -u root1 -p -e "SELECT @@version;"
Enter password:
+------------+
| @@version  |
+------------+
| 5.7.16     |
+------------+
```

```
root1@Dionaea:~$ mysql -h 172.17.0.1 -P 3306 -- protocol=tcp
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1729232896
Server version: 5.7.16 MySQL Community Server (GPL)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
[3]+  Stopped                 mysql -h 172.17.0.1 -P 3306 -- protocol=tcp
```

Port 11211

```
root1@Dionaea:~$ curl 172.17.0.1:11211 -v -X "STATS"
*   Trying 172.17.0.1:11211...
* Connected to 172.17.0.1 (172.17.0.1) port 11211
> STATS / HTTP/1.1
> Host: 172.17.0.1:11211
```

```
> User-Agent: curl/8.5.0
> Accept: */*
>
* Received HTTP/0.9 when not allowed
* Closing connection
curl: (1) Received HTTP/0.9 when not allowed
```

Below is a table showing for each port whether it has a header, a banner or none.

Table 3: Dionaea ports headers and banners

Port	Header	Banner
21	✗	✓
42	✗	✗
80	✓	✗
135	✗	✗
443	✓	✗
445	✗	✗
1433	✗	✗
1723	✗	✗
1883	✗	✗
3306	✗	✓
5060	✗	✗
5061	✗	✗
11211	✓	✗

3.2.4 T-Pot

Ports

```
root1@tpot:~$ sudo nmap -p- localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-03-28 18:39 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000040s latency).
Not shown: 65534 closed tcp ports (reset)
PORT      STATE SERVICE
64295/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 0.77 seconds
```

Services

```
root1@tpot:~$ sudo nmap -sV localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-03-28 18:42 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000040s latency).
All 1000 scanned ports on localhost (127.0.0.1) are in ignored states.
Not shown: 1000 closed tcp ports (reset)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.35 seconds
```

Headers and Banners

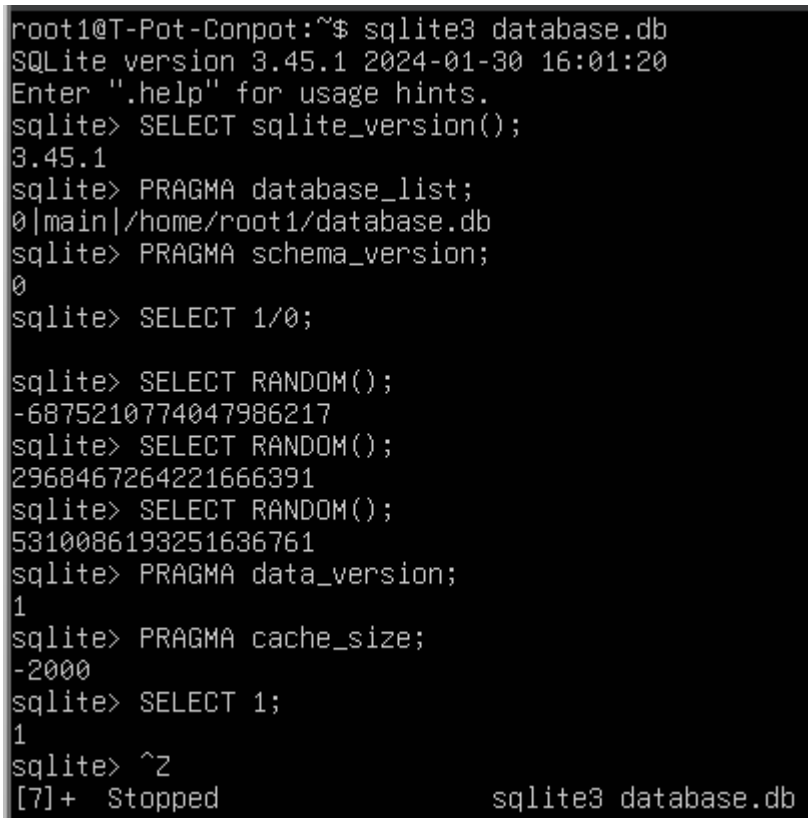
Port 64295

```
root1@tpot:~$ telnet
telnet> open localhost 64295
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SSH-2.0-OpenSSH_9.6p1 Ubuntu-3ubuntu13.9

Invalid SSH identification string.
Connection closed by foreign host.
```

```
root1@tpot:~$ nc -v localhost 64295
Connection to localhost (127.0.0.1) 64295 port [tcp/*] succeeded!
SSH-2.0-OpenSSH_9.6p1 Ubuntu-3ubuntu13.9
^C
root1@tpot:~$ nmap -sV -p 64295 localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-03-28 18:50 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000083s latency).

PORT      STATE SERVICE VERSION
64295/tcp open  ssh     OpenSSH 9.6p1 Ubuntu 3ubuntu13.9 (Ubuntu Linux; protocol 2.0)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.19 seconds
root1@tpot:~$ nmap --script=banner -p 64295 localhost
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-03-28 18:52 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00014s latency).

PORT      STATE SERVICE
64295/tcp open  unknown
|_banner: SSH-2.0-OpenSSH_9.6p1 Ubuntu-3ubuntu13.9

Nmap done: 1 IP address (1 host up) scanned in 0.06 seconds
```

```
root1@tpot:~$ curl -v http://localhost:64295
* Host localhost:64295 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:64295...
* Connected to localhost (::1) port 64295
> GET / HTTP/1.1
> Host: localhost:64295
> User-Agent: curl/8.5.0
> Accept: */*
>
* Received HTTP/0.9 when not allowed
* Closing connection
curl: (1) Received HTTP/0.9 when not allowed
root1@tpot:~$ openssl s_client -connect localhost:64295
CONNECTED(00000003)
406704B8D8720000:error:0A00010B:SSL routines:ssl3_get_record:wrong version number:../ssl/record/ssl3_record.c:354:
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 5 bytes and written 293 bytes
Verification: OK
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
```

```
root1@tpot:~$ ftp localhost 64295
Connected to localhost.
SSH-2.0-OpenSSH_9.6p1 Ubuntu-3ubuntu13.9
ftp>
[4]+  Stopped                 ftp localhost 64295
```



Below is a table showing for each port whether it has a header, a banner or none.

Table 4: T-Pot ports headers and banners

Port	Header	Banner
64295	✓	✓

T-Pot has one port open but it has multiple headers and banners. We downloaded T-Pot four times, one with all the honeypots removed except Conpot, one with all the honeypots removed except Cowrie, one with all the honeypots removed except Dionaea and one with all the honeypots intact. All four instances had just port 64295 open and they all had the same headers and banners.

3.3 Python Scripts

We wrote a Python script to search Shodan for 10 thousand server IP addresses, using filters apache, openssh, mysql, https and email. Shodan provides instructions on how to search in its documentation [5].

```
import os
import shodan
from shodan import Shodan
from dotenv import load_dotenv

load_dotenv()

api = Shodan(os.getenv("api_key"))

# try/except block to catch errors
try:
    # Search Shodan
    for i in range(0,5):
        for query in ["apache", "openssh", "mysql", "https", "email"]:
            results = api.search(query)

            # Write the results
            print('Results found: {}'.format(results['total']))
            with open("results1.txt", "a", encoding="utf-8") as f:
                for result in results['matches']:
                    f.write(f"IP: {result['ip_str']}\n")
                    f.write(result['data'] + "\n")
                    f.write("\n" + "-"*50 + "\n")

except shodan.APIError as e:
    print('Error: {}'.format(e))
```

We wrote a Python script to extract these 10 thousand IPs from Shodan's results.

```
# Count unique IP addresses
unique_ips = set()

try:
    with open("results1.txt", "r", encoding="utf-8") as f:
        for line in f:
            if line.startswith("IP:"):  # Identify lines with IP addresses
                ip_address = line.split("IP: ")[1].strip()
                unique_ips.add(ip_address)

    # Write the unique IPs to a file
    with open("unique_ips1.txt", "w", encoding="utf-8") as output_file:
        for ip in unique_ips:
            output_file.write(ip + "\n")

    print(f"Total unique IP addresses found: {len(unique_ips)}")
    print("Unique IP addresses have been written to 'unique_ips1.txt'.")

except FileNotFoundError:
    print("Error: The file 'results1.txt' was not found.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

We wrote a Python script to scan these IPs using Nmap to find their open ports.

```
import subprocess

# Read IPs from the 'unique_ips1.txt' file
with open("unique_ips1.txt", "r") as file:
    ips = [ip.strip() for ip in file.readlines()]  # Strip IPs
```

```
# Define the range of IPs to scan
start_line = 1
end_line = 10000

# Slice the IPs list based on the specified range
ips_to_scan = ips[start_line - 1:end_line]

# Open the output file to write the results
with open("nmap_scan_results1.txt", "a") as file:
    # Perform the scan for each IP in the specified range
    for index, target_ip in enumerate(ips_to_scan, start=start_line):
        print(f"\n[{index}/{len(ips_to_scan) + start_line - 1}] Scanning {target_ip} for all ports and services...")

        # Nmap command for all ports and service version detection
        command = ["nmap", "-p-", "-sV", "-T5", target_ip]

        try:
            # Run the nmap command using subprocess
            result = subprocess.run(command, capture_output=True, text=True, check=True)

            # Write the results to the file
            file.write(f"\n\nScan Results for {target_ip}:\n")
            file.write(result.stdout)

            print(f"Results for {target_ip} saved.")

        except subprocess.CalledProcessError as e:
            print(f"Error scanning {target_ip}: {e.stderr}")
            file.write(f"\nError scanning {target_ip}: {e.stderr}\n")

        except Exception as e:
            print(f"Unexpected error for {target_ip}: {e}")
```

```
            file.write(f"\nUnexpected error for {target_ip}: {e}\n")

    print("\nAll results saved to 'nmap_scan_results1.txt'.")
```

We wrote a Python script to banner grab these IPs using the commands we used to banner grab the honeypots, to find matching banners.

```
import subprocess
import shlex

# Define commands
commands = {
    "command1": "curl -s -I http://target_ip:631",
    "command2": "curl ftp://target_ip:2121 -v",
    "command3": "curl target_ip:5020 -v",
    "command4": "curl -I http://target_ip:8800",
    "command5": "curl -v http://target_ip:10201",
    "command6": "openssl s_client -connect target_ip:10201",
    "command7": "curl -vk https://target_ip:44818",
    "command8": "openssl s_client -connect target_ip:44818",
    "command9": "ssh -vvv target_ip",
    "command10": "openssl s_client -connect target_ip:2222",
    "command11": "ftp target_ip",
    "command12": "curl -I http://target_ip",
    "command13": "wget --server-response --spider http://target_ip:80",
    "command14": "curl -I -k https://target_ip",
    "command15": "wget --server-response --spider --no-check-certificate
https://target_ip",
    "command16": "mysql -h target_ip -u root1 -p -e \"SELECT @@version;\"",
    "command17": "curl target_ip:11211 -v -X \"STATS\"",
    "command18": "echo -e \"stats\\r\\n\" | nc target_ip 11211",
    "command19": "telnet target_ip 64295",
    "command20": "nc -v target_ip 64295",
```

```
    "command21": "curl -v http://target_ip:64295",
    "command22": "openssl s_client -connect target_ip:64295",
    "command23": "ftp target_ip 64295",
    "command24": "sqlite3 database.db 'SELECT sqlite_version();'",
    "command25": "sudo nping --tcp -p 64295 target_ip --flags syn,ack,fin",
    "command26": "curl -v http://target_ip",
    "command27": "nc -v target_ip 2525",
    "command28": "nc -v target_ip 6379",
    "command29": "nmap -p 6379 --script redis-info target_ip",
    "command30": "nmap -p 6379 -sV target_ip",
    "command31": "redis-cli -h target_ip -p 6379 INFO",
    "command32": "redis-cli -h target_ip -p 6379 PING",
    "command33": "curl -v --connect-to target_ip:36885",
    "command34": "curl -v http://target_ip:8080",
    "command35": "ssh -v target_ip",
    "command36": "curl -v http://target_ip:2112",
    "command37": "nc -v target_ip 3306",
    "command38": "telnet target_ip 3306",
    "command39": "curl -v http://target_ip:8081/",
    "command40": "ssh -p 44475 target_ip -v",
    "command41": "mysql -h target_ip -P 3306 --protocol=tcp"
}

def run_command(command_template, ip):

    # Run a specific command on the given IP

    try:
        command = command_template.replace("target_ip", ip)
        process = subprocess.Popen(
            shlex.split(command),
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
```

```
        )

        stdout, stderr = process.communicate(timeout=15)  # Timeout in 15 seconds

        return stdout.strip() if stdout else stderr.strip() or "No output."

    except subprocess.TimeoutExpired:
        process.kill()  # Kill the process
        return "Timeout."

    except Exception as e:
        return f"Error: {e}"

def grab_banners(ip):

    # Run all banner grabbing commands on the given IP

    results = []
    for cmd_name, command_template in commands.items():
        print(f"  Running {cmd_name} on {ip}...")
        output = run_command(command_template, ip)
        results.append((cmd_name, command_template, output))
    return results

def write_results(ip, banners, output_file):

    # Write the banner grabbing results to the output file

    with open(output_file, "a") as file:
        file.write(f"\n\nResults for {ip}:\n")
        for cmd_name, command, result in banners:
            file.write(f"{cmd_name}: {command}\nResult:\n{result}\n\n")
    print(f"Results for {ip} saved to {output_file}.")
```

```
def main():
    unique_ips_file = "unique_ips1.txt"
    output_file = "banner_grabbing_results.txt"

    # Define the range of IPs
    start_line = 1
    end_line = 10000

    # Read the list of unique IPs
    with open(unique_ips_file, "r") as file:
        ips = [ip.strip() for ip in file.readlines()]

    # Slice the IP list based on the specified range
    ips_to_process = ips[start_line - 1:end_line]

    # Banner grab each IP in the specified range
    for index, ip in enumerate(ips_to_process, start=start_line):
        print(f"\n[{index}/{len(ips_to_process)}] Processing {ip}...")
        banners = grab_banners(ip)
        write_results(ip, banners, output_file)

    print("\nAll results saved to 'banner_grabbing_results.txt'.")

if __name__ == "__main__":
    main()
```

We wrote a Python script to analyze the number of open ports associated with each IP address.

```
import re
from collections import defaultdict

def parse_nmap_results(file_path):
```

```
    # Parse the nmap results file to extract open ports for each IP

    open_ports = {}
    current_ip = None

    with open(file_path, "r") as file:
        for line in file:
            # Detect an IP address line
            ip_match = re.match(r"Scan Results for (\d+\.\d+\.\d+\.\d+):", line)
            if ip_match:
                current_ip = ip_match.group(1)
                open_ports[current_ip] = []

            # Detect open port lines
            if current_ip:
                port_match = re.match(r"(\d+)/tcp\s+open", line)
                if port_match:
                    open_ports[current_ip].append(port_match.group(1))

    return open_ports

def summarize_port_counts(open_ports):

    # Count how many IPs have the same number of open ports and list which ports

    port_summary = defaultdict(list)
    for ip, ports in open_ports.items():
        port_summary[len(ports)].append((ip, ports))
    return port_summary

def write_results_to_file(port_summary, output_file):

    # Write the summary of open ports across IPs
```

```
    with open(output_file, "w") as file:
        file.write("Summary of open ports across IPs:\n\n")
        for port_count, ip_list in sorted(port_summary.items()):
            file.write(f"{len(ip_list)} IPs have {port_count} open ports:\n")
            for ip, ports in ip_list:
                file.write(f"  {ip}: {', '.join(ports)}\n")
            file.write("\n")

if __name__ == "__main__":
    nmap_results_file = "nmap_scan_results1.txt"
    output_file = "ips_ports.txt"

    open_ports = parse_nmap_results(nmap_results_file)
    port_summary = summarize_port_counts(open_ports)

    write_results_to_file(port_summary, output_file)
    print(f"Results written to {output_file}")
```

We wrote a Python script to analyze the number of matching ports for each IP with the honeypots and the number of IPs with all the ports of a honeypot open.

```
import re

def find_honeypot_like_ips(open_ports, honeypot_profiles):

    # Identify IPs that have at least some ports matching each honeypot

    matching_ips = {honeypot: {} for honeypot in honeypot_profiles}
    all_ports_matching_count = {honeypot: 0 for honeypot in honeypot_profiles}

    for ip, ports in open_ports.items():
        for honeypot, honeypot_ports in honeypot_profiles.items():
```

```
            matched_ports = set(ports) & honeypot_ports  # Find common ports
            if matched_ports:
                matching_ips[honeypot][ip] = list(matched_ports)
                if honeypot_ports.issubset(set(ports)):  # Ensure all honeypot ports are open even with extra ports
                    all_ports_matching_count[honeypot] += 1

    return matching_ips, all_ports_matching_count

def parse_nmap_results(file_path):

    # Parse the nmap results file to extract open ports for each IP

    open_ports = {}
    current_ip = None

    with open(file_path, "r") as file:
        for line in file:
            ip_match = re.match(r"Scan Results for (\d+\.\d+\.\d+\.\d+):", line)
            if ip_match:
                current_ip = ip_match.group(1)
                open_ports[current_ip] = []

            port_match = re.match(r"(\d+)/tcp\s+open", line)
            if current_ip and port_match:
                open_ports[current_ip].append(port_match.group(1))

    return open_ports

def write_matching_ips_to_file(matching_ips, all_ports_matching_count, output_file, honeypot_profiles):

    # Write the IPs and their matching ports to a file for each honeypot and the total count of IPs with all ports matching
```

```
    with open(output_file, "w") as file:
        file.write("IPs matching honeypot ports:\n\n")
        for honeypot, matches in matching_ips.items():
            file.write(f"Honeypot: {honeypot}\n")
            for ip, ports in matches.items():
                if honeypot_profiles[honeypot].issubset(set(ports)):
                    file.write(f"  {ip} (ALL {len(ports)} ports matching): {', '.join(ports)}\n")
                else:
                    file.write(f"  {ip} ({len(ports)} matching ports): {', '.join(ports)}\n")
            file.write(f"Total IPs with ALL ports matching: {all_ports_matching_count[honeypot]}\n\n")

def main():
    input_file = "nmap_scan_results1.txt"
    output_file = "port_matching.txt"

    # Define honeypot profiles with their ports
    honeypot_profiles = {
        "Conpot": {"631", "2121", "5020", "8800", "10201", "44818"},
        "Cowrie": {"2222"},
        "Dionaea": {"21", "42", "80", "135", "443", "445", "1433", "1723", "1883", "3306", "5060", "5061", "11211"},
        "T-Pot": {"64295"}
    }

    open_ports = parse_nmap_results(input_file)
    matching_ips, all_ports_matching_count = find_honeypot_like_ips(open_ports, honeypot_profiles)
    write_matching_ips_to_file(matching_ips, all_ports_matching_count, output_file, honeypot_profiles)

if __name__ == "__main__":
    main()
```

Finally, we wrote a Python script to analyze the number of unique IPs with matching banners for each honeypot port. The script also finds the number of matching banners for each IP.

```
import json

def count_matching_results(banner_json_path, results_json_path, output_file_path):
    # Open file with banners
    with open(banner_json_path, 'r', encoding='utf-8') as banner_file:
        banner_data = json.load(banner_file)

    # Open file with results
    with open(results_json_path, 'r', encoding='utf-8') as results_file:
        results_data = json.load(results_file)

    # Dictionary to count unique IP matches per port for every honeypot
    honeypot_port_match_count = {}
    # Dictionary to count matched ports per IP per honeypot
    ip_matched_ports = {}

    # Write match results
    with open(output_file_path, 'w', encoding='utf-8') as output_file:
        for honeypot, ports in banner_data.items():
            for port, banner_info in ports.items():
                # Set for unique IPs that have matched a banner for a port
                matched_ips = set()

                for banner in banner_info['banners']:
                    banner = banner.strip()  # Strip banner
                    # Check each IP in results
                    for ip, commands in results_data.items():
                        matched = False  # Each IP is counted only once per banner
```

```
                for command_key, command_data in commands.items():
                    result = command_data['result'].strip()  # Strip result

                    # Find matching banner
                    if banner in result:
                        if not matched:  # Only count the first match per banner per IP
                            match_string = (f"Matching banner found. IP: {ip}, Honeypot:
{honeypot}, "
                                            f"Port: {port}, Matched Banner: {banner}\n")
                            output_file.write(match_string)

                            # Add IP to set for port matching count
                            matched_ips.add(ip)

                            # Only one match per port per IP
                            if (honeypot, ip, port) not in ip_matched_ports:
                                ip_matched_ports[(honeypot, ip, port)] = True

                            matched = True  # Each IP is counted only once per banner

            # Count unique IPs with a matching banner for a port
            if matched_ips:
                honeypot_port_match_count[(honeypot, port)] = len(matched_ips)

    # Convert port matching count to IP matching count
    ip_banner_count = {}
    for (honeypot, ip, port) in ip_matched_ports:
        if (honeypot, ip) not in ip_banner_count:
            ip_banner_count[(honeypot, ip)] = 0
        ip_banner_count[(honeypot, ip)] += 1  # Count only once per port per IP

    # Write summary for port matching banners
    output_file.write("\nMatching banners count for each honeypot and port:\n")
    for (honeypot, port), count in honeypot_port_match_count.items():
```

```
            result_string = f"Honeypot: {honeypot}, Port: {port} - Unique IPs with matches: {count}\n"
            output_file.write(result_string)

        # Write summary for IP matching banners
        output_file.write("\nNumber of banners each IP matched per honeypot:\n")
        for (honeypot, ip), count in ip_banner_count.items():
            result_string = f"Honeypot: {honeypot}, IP: {ip} - Matched banners count: {count}\n"
            output_file.write(result_string)

# File paths
banner_json_path = 'all.json'     # Path for honeypot banners
results_json_path = 'output.json'  # Path for result banners
output_file_path = 'match_results.txt'  # Path for output

count_matching_results(banner_json_path, results_json_path, output_file_path)
```

Up to this point, we believe that we have not yet found any honeypots. Only a very small number of IPs had all the ports of a honeypot open, and the matching banners we found were generic. It is apparent that finding honeypots using filters only for the server type is quite rare. With more resources, scanning a larger number of IPs would likely allow us to discover honeypots. However, given our limited resources, we needed to search Shodan using more specific filters in order to increase our chances of finding honeypots.

3.4 Experiment 2

We decided to download, install and configure four more honeypots, Mailoney, HellPot, Beelzebub and ElasticPot, in order to increase our chances of finding honeypots in servers because that would enable us to search for more matching ports and banners. We downloaded Mailoney using Python pip, we couldn't get Docker to work. We downloaded HellPot using Go Compiler. We downloaded Beelzebub using Docker, Go

Compiler is also available. Finally, we downloaded ElasticPot, using Docker because we couldn't get Ubuntu installation to work. Below are the open ports, services, headers and banners we got for each honeypot.

3.4.1 Mailoney

Ports

```
PORT      STATE SERVICE
80/tcp    open  http
631/tcp   open  ipp
2525/tcp  open  ms-v-worlds
6379/tcp  open  redis
36885/tcp open  unknown
```

Services

```
PORT     STATE SERVICE    VERSION
80/tcp   open  tcpwrapped
631/tcp  open  ipp        CUPS 2.4
2525/tcp open  smtp-proxy Python SMTP Proxy 0.3
Service Info: Host: mailoney
```

Mailoney has port 80 open with http/ tcpwrapped running, port 631 open with ipp CUPS 2.4 running, port 2525 open with ms-v-worlds/ smtp-proxy version Python SMTP Proxy 0.3 running, port 6379 open with redis running and port 36885 open with an unknown service running.

Headers and Banners

Port 80

```
root1@mailoney:~/mailoney$ curl -v http://localhost
* Host localhost:80 was resolved.
```

```
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:80...
* Connected to localhost (::1) port 80
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/8.5.0
> Accept: */*
>
* Recv failure: Connection reset by peer
* Closing connection
curl: (56) Recv failure: Connection reset by peer
```

Port 631

```
root1@mailoney:~/mailoney$ curl -s -I http://localhost:631
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Language: en_US
Content-Length: 2262
Content-Type: text/html; charset=utf-8
Date: Mon, 17 Feb 2025 14:29:27 GMT
Keep-Alive: timeout=10
Last-Modified: Thu, 26 Sep 2024 11:15:36 GMT
Accept-Encoding: gzip, deflate, identity
Server: CUPS/2.4 IPP/2.1
X-Frame-Options: DENY
Content-Security-Policy: frame-ancestors 'none'
```

Port 2525

```
root1@mailoney:~/mailoney$ nc -v localhost 2525
Connection to localhost (127.0.0.1) 2525 port [tcp/*] succeeded!
220 mailoney Python SMTP proxy version 0.3
```

Port 6379

```
root1@mailoney:~/mailoney$ nc -v localhost 6379
Connection to localhost (127.0.0.1) 6379 port [tcp/redis] succeeded!
INFO
$4937
# Server
redis_version:7.0.15
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:d81b8ff71cfb150e
redis_mode:standalone
os:Linux 6.8.0-52-generic x86_64
arch_bits:64
monotonic_clock:POSIX clock_gettime
multiplexing_api:epoll
atomicvar_api:c11-builtin
gcc_version:13.2.0
process_id:1157
process_supervised:systemd
run_id:48a96336a5639b585aae4d666dc273f05488f743
tcp_port:6379
server_time_usec:1739804404091031
uptime_in_seconds:4906
uptime_in_days:0
hz:10
configured_hz:10
lru_clock:11751156
executable:/usr/bin/redis-server
config_file:/etc/redis/redis.conf
io_threads_active:0

# Clients
```

```
connected_clients:1
cluster_connections:0
maxclients:10000
client_recent_max_input_buffer:8
client_recent_max_output_buffer:0
blocked_clients:0
tracking_clients:0
clients_in_timeout_table:0
```

```
PORT     STATE SERVICE
6379/tcp open  redis
| redis-info:
|   Version: 7.0.15
|   Operating System: Linux 6.8.0-52-generic x86_64
|   Architecture: 64 bits
|   Process ID: 1157
|   Used CPU (sys): 5.946720
|   Used CPU (user): 3.343973
|   Connected clients: 1
|   Connected slaves: 0
|   Used memory: 1.02M
|   Role: master
|   Bind addresses:
|     127.0.0.1
|     -::1
|   Client connections:
|_    127.0.0.1
```

```
PORT     STATE SERVICE VERSION
6379/tcp open  redis   Redis key-value store 7.0.15
```

Port 36885

```
root1@mailoney:~/mailoney$ curl -v --connect-to localhost:36885
http://localhost:36885
* Host localhost:36885 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:36885...
* connect to ::1 port 36885 from ::1 port 43128 failed: Connection refused
*   Trying 127.0.0.1:36885...
* connect to 127.0.0.1 port 36885 from 127.0.0.1 port 46776 failed: Connection refused
* Failed to connect to localhost port 36885 after 0 ms: Couldn't connect to server
* Closing connection
curl: (7) Failed to connect to localhost port 36885 after 0 ms: Couldn't connect to server
```

Below is a table showing for each port whether it has a header, a banner or none.

Table 5: Mailoney ports headers and banners

Port	Header	Banner
80	✓	✗
631	✓	✗
2525	✗	✓
6379	✗	✓
36885	✓	✗

3.4.2 HellPot

Ports

```
PORT     STATE SERVICE
631/tcp  open  ipp
8080/tcp open  http-proxy
```

Services

```
PORT     STATE SERVICE VERSION
631/tcp  open  ipp     CUPS 2.4
8080/tcp open  http    nginx
```

HellPot has port 631 open with ipp CUPS 2.4 running and port 8080 open with http-proxy/ http nginx running.

Headers and Banners

Port 631

```
root1@HellPot:~/HellPot$ curl -s -I http://localhost:631
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Language: en_US
Content-Length: 2262
Content-Type: text/html; charset=utf-8
Date: Mon, 17 Feb 2025 15:25:56 GMT
Keep-Alive: timeout=10
Last-Modified: Thu, 26 Sep 2024 11:15:36 GMT
Accept-Encoding: gzip, deflate, identity
Server: CUPS/2.4 IPP/2.1
X-Frame-Options: DENY
Content-Security-Policy: frame-ancestors 'none'
```

Port 8080

```
root1@HellPot:~/HellPot$ curl -v http://localhost:8080
* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
* connect to ::1 port 8080 from ::1 port 48572 failed: Connection refused
*   Trying 127.0.0.1:8080...
```

```
* Connected to localhost (127.0.0.1) port 8080
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Server: nginx
< Date: Mon, 17 Feb 2025 15:53:16 GMT
< Content-Type: text/plain; charset=utf-8
< Content-Length: 9
< Connection: close
<
* Closing connection
```

Below is a table showing for each port whether it has a header, a banner or none.

Table 6: HellPot ports headers and banners

Port	Header	Banner
631	✓	✕
8080	✓	✕

3.4.3 Beelzebub

Ports

```
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
631/tcp   open  ipp
2112/tcp  open  kip
2222/tcp  open  EtherNetIP-1
```

```
3306/tcp  open  mysql
8080/tcp  open  http-proxy
8081/tcp  open  blackice-icecap
44475/tcp open  unknown
```

Services

```
PORT     STATE SERVICE    VERSION
22/tcp   open  ssh        Linksys WRT45G modified dropbear sshd (protocol 2.0)
80/tcp   open  http       Apache httpd 2.4.53 ((Debian))
631/tcp  open  ipp        CUPS 2.4
2222/tcp open  ssh        Linksys WRT45G modified dropbear sshd (protocol 2.0)
3306/tcp open  mysql?
8080/tcp open  http       Apache httpd
8081/tcp open  tcpwrapped
```

Beelzebub has port 22 open with ssh Linksys WRT45G modified dropbear sshd (protocol 2.0) running, port 80 open with http Apache httpd 2.4.53 ((Debian)) running, port 631 open with ipp CUPS 2.4 running, port 2112 open with kip running, port 2222 open with EtherNetIP-1/ ssh Linksys WRT45G modified dropbear sshd (protocol 2.0) running, port 3306 open with mysql running, port 8080 open with http-proxy/ http Apache httpd running, port 8081 open with blackice-icecap/ tcpwrapped running and port 44475 open with an unknown service running.

Headers and Banners

Port 22

```
root1@Beelzebub:~/beelzebub$ ssh -v localhost
OpenSSH_9.6p1 Ubuntu-3ubuntu13.5, OpenSSL 3.0.13 30 Jan 2024
```

Port 80

```
root1@Beelzebub:~/beelzebub$ wget --server-response --spider http://localhost:80
```

```
Spider mode enabled. Check if remote file exists.
--2025-02-17 16:27:28--  http://localhost/
Resolving localhost (localhost)... 127.0.0.1
Connecting to localhost (localhost)|127.0.0.1|:80... connected.
HTTP request sent, awaiting response...
  HTTP/1.1 200 OK
  Content-Type: text/html
  Server: Apache/2.4.53 (Debian)
  X-Powered-By: PHP/7.4.29
  Date: Mon, 17 Feb 2025 16:27:28 GMT
  Content-Length: 115
Length: 115 [text/html]
Remote file exists and could contain further links,
but recursion is disabled -- not retrieving.
```

Port 631

```
root1@Beelzebub:~/beelzebub$ curl -s -I http://localhost:631
HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Language: en_US
Content-Length: 2262
Content-Type: text/html; charset=utf-8
Date: Mon, 17 Feb 2025 16:30:21 GMT
Keep-Alive: timeout=10
Last-Modified: Thu, 26 Sep 2024 11:15:36 GMT
Accept-Encoding: gzip, deflate, identity
Server: CUPS/2.4 IPP/2.1
X-Frame-Options: DENY
Content-Security-Policy: frame-ancestors 'none'
```

Port 2112

```
root1@Beelzebub:~/beelzebub$ curl -v http://localhost:2112
* Host localhost:2112 was resolved.
```

```
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:2112...
* Connected to localhost (::1) port 2112
> GET / HTTP/1.1
> Host: localhost:2112
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Content-Type: text/plain; charset=utf-8
< X-Content-Type-Options: nosniff
< Date: Mon, 17 Feb 2025 16:38:58 GMT
< Content-Length: 19
<
404 page not found
* Connection #0 to host localhost left intact
```

Port 2222

```
root1@Beelzebub:~/beelzebub$ ssh -vvv localhost
OpenSSH_9.6p1 Ubuntu-3ubuntu13.5, OpenSSL 3.0.13 30 Jan 2024
```

```
root1@Beelzebub:~/beelzebub$ openssl s_client -connect localhost:2222
CONNECTED(00000003)
40571E671D7C0000:error:0A00010B:SSL routines:ssl3_get_record:wrong version number:../ssl/record/ssl3_record.c:354:
---
no peer certificate available
---
No client certificate CA names sent
---
SSL handshake has read 5 bytes and written 293 bytes
Verification: OK
```

```
---
New, (NONE), Cipher is (NONE)
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
---
```

Port 3306

```
root1@Beelzebub:~/beelzebub$ nc -v localhost 3306
Connection to localhost (127.0.0.1) 3306 port [tcp/mysql] succeeded!
8.0.29
```

```
root1@Beelzebub:~/beelzebub$ telnet localhost 3306
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
8.0.29
```

Port 8080

```
root1@Beelzebub:~/beelzebub$ curl -v http://localhost:8080
* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.5.0
> Accept: */*
```

```
>
< HTTP/1.1 401 Unauthorized
< Server: Apache
< Www-Authenticate: Basic
< Date: Mon, 17 Feb 2025 17:09:23 GMT
< Content-Length: 12
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
```

Port 8081

```
root1@Beelzebub:~/beelzebub$ curl -v http://localhost:8081/
* Host localhost:8081 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8081...
* Connected to localhost (::1) port 8081
> GET / HTTP/1.1
> Host: localhost:8081
> User-Agent: curl/8.5.0
> Accept: */*
>
* Recv failure: Connection reset by peer
* Closing connection
curl: (56) Recv failure: Connection reset by peer
```

Port 44475

```
root1@Beelzebub:~/beelzebub$ ssh -p 44475 localhost -v
OpenSSH_9.6p1 Ubuntu-3ubuntu13.5, OpenSSL 3.0.13 30 Jan 2024
```

Below is a table showing for each port whether it has a header, a banner or none.

Table 7: Beelzebub ports headers and banners

Port	Header	Banner
22	✗	✓
80	✓	✗
631	✓	✗
2112	✓	✗
2222	✓	✓
3306	✗	✓
8080	✓	✗
8081	✓	✗
44475	✗	✓

3.4.4 ElasticPot

Ports

```
PORT      STATE SERVICE
631/tcp   open  ipp
9200/tcp  open  wap-wsp
39911/tcp open  unknown
```

Services

```
PORT     STATE SERVICE VERSION
631/tcp  open  ipp     CUPS 2.4
9200/tcp open  http    Apache httpd
```

ElasticPot has port 631 open with ipp CUPS 2.4 running, port 9200 open with wap-wsp/ http Apache httpd running and port 39911 open with an unknown service running.

Headers and Banners

Port 631

```
root1@elasticpot:~$ curl -v http://localhost:631
* Host localhost:631 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:631...
* Connected to localhost (::1) port 631
* using HTTP/1.x
> GET / HTTP/1.1
> Host: localhost:631
> User-Agent: curl/8.12.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Connection: Keep-Alive
< Content-Language: en_US
< Content-Length: 2262
< Content-Type: text/html; charset=utf-8
< Date: Sun, 23 Mar 2025 18:55:37 GMT
< Keep-Alive: timeout=10
< Last-Modified: Thu, 26 Sep 2024 11:15:36 GMT
< Accept-Encoding: gzip, deflate, identity
< Server: CUPS/2.4 IPP/2.1
< X-Frame-Options: DENY
< Content-Security-Policy: frame-ancestors 'none'
<
<!DOCTYPE HTML>
<html>
  <head>
    <link rel="stylesheet" href="/cups.css" type="text/css">
    <link rel="shortcut icon" href="/apple-touch-icon.png" type="image/png">
    <meta charset="utf-8">
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=9">
```

```
    <meta name="viewport" content="width=device-width">
    <title>Home - CUPS 2.4.7</title>
  </head>
  <body>
    <div class="cups-header">
      <ul>
        <li><a href="https://openprinting.github.io/cups/"
target="_blank">OpenPrinting CUPS</a></li>
        <li><a class="active" href="/">Home</a></li>
        <li><a href="/admin">Administration</a></li>
        <li><a href="/classes/">Classes</a></li>
        <li><a href="/help/">Help</a></li>
        <li><a href="/jobs/">Jobs</a></li>
        <li><a href="/printers/">Printers</a></li>
      </ul>
    </div>
    <div class="cups-body">
      <div class="row">
        <h1>OpenPrinting CUPS 2.4.7</h1>
        <p>The standards-based, open source printing system developed by <a
class="jumbolink" href="https://openprinting.github.io/"
target="_blank">OpenPrinting</a> for Linux® and other Unix®-like operating
systems. CUPS uses <a href="https://www.pwg.org/ipp/everywhere.html"
target="_blank">IPP Everywhere™</a> to support printing to local and network
printers.</p>
      </div>
      <div class="row">
        <div class="thirds">
          <h2>CUPS for Users</h2>
          <p><a href="help/overview.html">Overview of CUPS</a></p>
          <p><a href="help/options.html">Command-Line Printing and
Options</a></p>
        </div>
        <div class="thirds">
          <h2>CUPS for Administrators</h2>
          <p><a href="help/admin.html">Adding Printers and Classes</a></p>
```

```
          <p><a href="help/policies.html">Managing Operation Policies</a></p>
          <p><a href="help/network.html">Using Network Printers</a></p>
          <p><a href="help/firewalls.html">Firewalls</a></p>
          <p><a href="help/man-cupsd.conf.html">cupsd.conf Reference</a></p>
         </div>
         <div class="thirds">
          <h2>CUPS for Developers</h2>
          <p><a href="help/cupspm.html">CUPS Programming Manual</a></p>
          <p><a href="help/api-filter.html">Filter and Backend Programming</a></p>
         </div>
     </div>
   </div>
   <div class="cups-footer">Copyright &copy; 2021-2023 OpenPrinting. All rights
reserved.</div>
  </body>
</html>
* Connection #0 to host localhost left intact
```

Port 9200

```
root1@elasticpot:~$ curl -v http://localhost:9200
* Host localhost:9200 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:9200...
* Connected to localhost (::1) port 9200
* using HTTP/1.x
> GET / HTTP/1.1
> Host: localhost:9200
> User-Agent: curl/8.12.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Server: Apache
```

```
< Date: Sun, 23 Mar 2025 18:48:45 GMT
< Content-Length: 339
< Content-Type: application/json; charset=UTF-8
< Connection: Close
<
{
  "status" : 200,
  "name" : "Green Goblin",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "1.4.1",
    "build_hash" : "b88f43fc40b0bcd7f173a1f9ee2e97816de80b19",
    "build_timestamp" : "2015-07-29T09:54:16Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
* shutting down connection #0
}
```

Port 39911

```
root1@elasticpot:~$ curl -v http://localhost:39911
* Host localhost:39911 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:39911...
* connect to ::1 port 39911 from ::1 port 37854 failed: Connection refused
*   Trying 127.0.0.1:39911...
* Connected to localhost (127.0.0.1) port 39911
* using HTTP/1.x
> GET / HTTP/1.1
> Host: localhost:39911
> User-Agent: curl/8.12.1
```

```
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 404 Not Found
< Date: Sun, 23 Mar 2025 19:07:30 GMT
< Content-Length: 19
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
404: Page Not Found
```

Below is a table showing for each port whether it has a header, a banner or none.

Table 8: ElasticPot ports headers and banners

Port	Header	Banner
631	✓	✗
9200	✓	✓
39911	✓	✗

Finally, we searched Shodan for one thousand more IPs using filters for specific ports. We scanned 410 IPs using the filter: port 2525, this port has a unique banner for Mailoney and 590 IPs using the filter: port 9200, this port has a unique banner for ElasticPot.

4 Result Analysis

4.1 Python Scripts

We wrote a Python script to visualize the number of open ports associated with each IP address.

```
import re
import matplotlib.pyplot as plt
from collections import defaultdict

def parse_summary_file(file_path):

    # Read the file and extract the number of IPs that have a given number of open ports

    port_summary = defaultdict(int)

    with open(file_path, "r") as file:
        for line in file:
            match = re.match(r"(\d+) IPs have (\d+) open ports:", line)
            if match:
                num_ips = int(match.group(1))
                num_ports = int(match.group(2))
                port_summary[num_ports] += num_ips

    return port_summary

def plot_port_distribution(port_summary):

    # Scatter plot of open ports across IPs

    x_values = list(port_summary.keys())  # Number of open ports per IP
    y_values = list(port_summary.values())  # Number of IPs with that many open ports

    plt.figure(figsize=(8, 5))
    plt.scatter(x_values, y_values, color='b', alpha=0.6, edgecolors='black')
    plt.xscale("log")  # Apply logarithmic scale to x-axis
    plt.yscale("log")  # Apply logarithmic scale to y-axis
    plt.xlabel("Number of Open Ports per IP (Log Scale)", fontsize=18)
    plt.ylabel("Number of IPs (Log Scale)", fontsize=18)
    plt.title("Distribution of Open Ports Across IPs", fontsize=20)
```

```
    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)
    plt.grid(True, linestyle="--", alpha=0.5)
    plt.show()

if __name__ == "__main__":
    summary_file = "ips_ports.txt"
    port_summary = parse_summary_file(summary_file)
    plot_port_distribution(port_summary)
```

We wrote a Python script to visualize the number of matching ports of the IPs with the honeypots.

```
import re
import matplotlib.pyplot as plt
import numpy as np
from collections import defaultdict


def parse_port_matching_results(file_path):
    matching_port_distribution = defaultdict(lambda: defaultdict(int))
    current_honeypot = None


    with open(file_path, "r") as file:
        for line in file:
            honeypot_match = re.match(r"Honeypot: (.+)", line)
            if honeypot_match:
                current_honeypot = honeypot_match.group(1)
```

```
            if current_honeypot:
                match = re.search(r"\((\d+) matching ports\)|ALL (\d+) ports matching", line)
                if match:
                    num_ports = int(match.group(1) or match.group(2))
                    matching_port_distribution[current_honeypot][num_ports] += 1

    return matching_port_distribution


def visualize_results(matching_port_distribution):
    plt.figure(figsize=(12, 6))

    all_ports = sorted(set(port for counts in matching_port_distribution.values() for port
in counts))
    honeypots = list(matching_port_distribution.keys())

    port_to_honeypot_data = defaultdict(list)
    for honeypot in honeypots:
        for port, count in matching_port_distribution[honeypot].items():
            port_to_honeypot_data[port].append((honeypot, count))

    color_map = dict(zip(honeypots, plt.get_cmap("tab10").colors))

    fixed_width = 0.125
    spacing = 0.02

    for port_index, port in enumerate(all_ports):
```

```
        group = port_to_honeypot_data[port]
        group_size = len(group)
        total_width = group_size * (fixed_width + spacing)
        left_edge = port - total_width / 2

        for i, (honeypot, count) in enumerate(group):
            xpos = left_edge + i * (fixed_width + spacing) + fixed_width / 2
            plt.bar(xpos, count, width=fixed_width, label=honeypot if port_index == 0 else
"",
                    color=color_map[honeypot], alpha=0.8)

    plt.xlabel('Number of Matching Ports', fontsize=18)
    plt.ylabel('Number of IPs', fontsize=18)
    plt.yscale('log')
    plt.ylim(top=5000)
    plt.xticks(all_ports, fontsize=15)
    plt.yticks(fontsize=15)
    plt.tick_params(axis='x', which='major', length=8)
    plt.tick_params(axis='x', which='minor', length=4)
    plt.title('Number of IPs with a Number of Matching Ports', fontsize=20)
    plt.legend(fontsize=18)
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.show()

def main():
    input_file = "port_matching.txt"
```

```
    matching_port_distribution = parse_port_matching_results(input_file)
    visualize_results(matching_port_distribution)


if __name__ == "__main__":
    main()
```

We wrote a Python script to visualize the number of unique IPs with matching banners for each honeypot port.

```
import matplotlib.pyplot as plt

def parse_match_results(file_path):
    honeypots = []
    ports = []
    matches = []

    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            if line.startswith("Honeypot:"):
                parts = line.strip().split(" - ")
                honeypot_port = parts[0].split(", ")
                honeypot = honeypot_port[0].split(": ")[1]
                port = int(honeypot_port[1].split(": ")[1])
                match_count = int(parts[1].split(": ")[1])

                honeypots.append(honeypot)
                ports.append(port)
                matches.append(match_count)

    return honeypots, ports, matches
```

```
def visualize_match_results(file_path):
    honeypots, ports, matches = parse_match_results(file_path)

    labels = [f"{h}\n({p})" for h, p in zip(honeypots, ports)]

    plt.figure(figsize=(12, 8))
    bars = plt.bar(labels, matches, color='#C58E00', width=0.8)

    plt.ylabel("Unique IPs with Matching Banners", fontsize=18)
    plt.xlabel("Honeypot Ports", fontsize=18)
    plt.title("Matching Banners Count for Each Honeypot Port", fontsize=20)
    plt.yscale('log')

    plt.xticks(rotation=60, ha='center', fontsize=18)
    plt.yticks(fontsize=18)

    # Write amount of unique IPs with matching banners on top of each bar
    for bar, count in zip(bars, matches):
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width() / 2, height, str(count),
                 ha='center', va='bottom', fontsize=18)

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    file_path = "final_match_results.txt"
    visualize_match_results(file_path)
```

Finally, we wrote a Python script to visualize the number of matching banners for each IP per honeypot.

```
import matplotlib.pyplot as plt
```

```
import collections
import re

# Choose file
file_path = "conpot_ips.txt"
with open(file_path, "r") as file:
    lines = file.readlines()

# Extract honeypot name, IPs, and matching banner counts
honeypot_data = collections.defaultdict(list)
pattern = re.compile(r"Honeypot: (.*?), IP: (.*?) - Matched banners count: (\d+)")

ip_honeypot_mapping = {}
ip_banner_counts = {}
honeypot_name = None

for line in lines:
    match = pattern.search(line)
    if match:
        honeypot, ip, count = match.groups()
        count = int(count)
        ip_honeypot_mapping[ip] = honeypot
        ip_banner_counts[ip] = count
        if honeypot_name is None:
            honeypot_name = honeypot

# Sort IPs by matched banner count (descending order)
sorted_ips = sorted(ip_banner_counts, key=lambda ip: ip_banner_counts[ip],
reverse=True)
sorted_banner_counts = [ip_banner_counts[ip] for ip in sorted_ips]

plt.figure(figsize=(16, 9))
plt.scatter(range(len(sorted_ips)), sorted_banner_counts, s=15, edgecolors="black",
linewidth=0.1)
```

```
# IP count for every matched banner count
from collections import Counter
banner_counts = Counter(sorted_banner_counts)

# Positions and labels for x-axis
xticks = []
xtick_labels = []
start = 0
for count in sorted(banner_counts.keys(), reverse=True):
    num_ips = banner_counts[count]
    midpoint = start + num_ips // 2
    xticks.append(midpoint)
    xtick_labels.append(str(num_ips))
    start += num_ips

plt.xticks(xticks, xtick_labels, fontsize=15)
plt.xlabel("Number of IPs per Matched Banners Count", fontsize=18)
plt.ylabel("Matched Banners Count", fontsize=18)
plt.yticks(range(min(sorted_banner_counts), max(sorted_banner_counts) + 1),
fontsize=15)
plt.title(f"{honeypot_name} Matched Banners Count per IP", fontsize=20)
plt.grid(axis="y", linestyle="--", alpha=0.1)

plt.show()
```

4.2 Graphs and Table

Below are the graphs and a table to visualize our results.

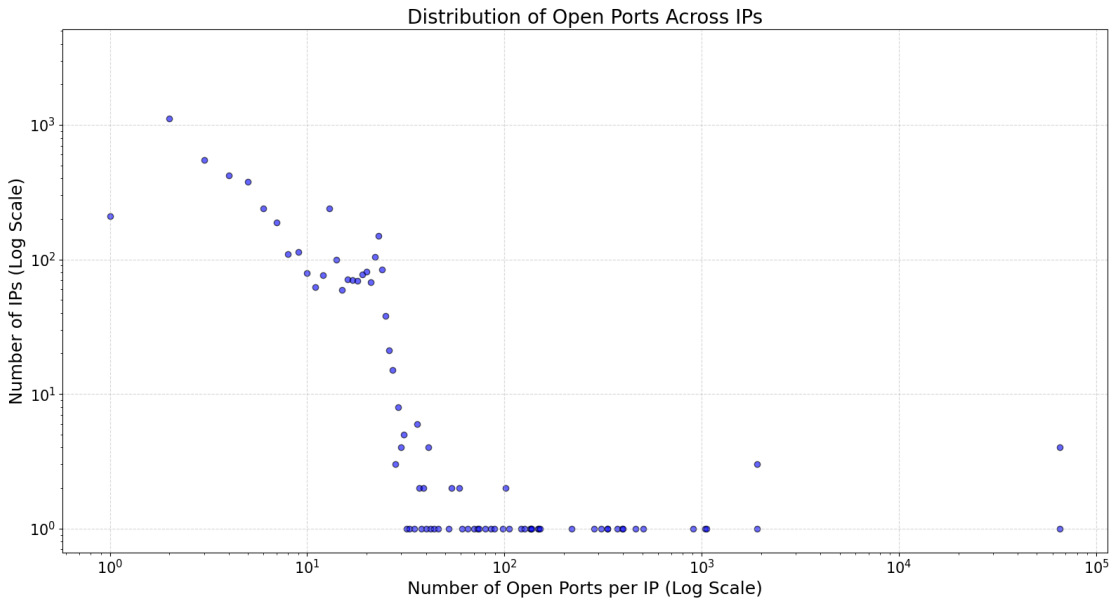


Figure 3: Number of open ports associated with each IP address

Most IPs have a small number of open ports (around 1–10 ports), while only a few IPs have unusually high numbers of open ports. This pattern suggests that typical hosts expose only a limited set of services, whereas a minority have hundreds or even thousands of open ports.

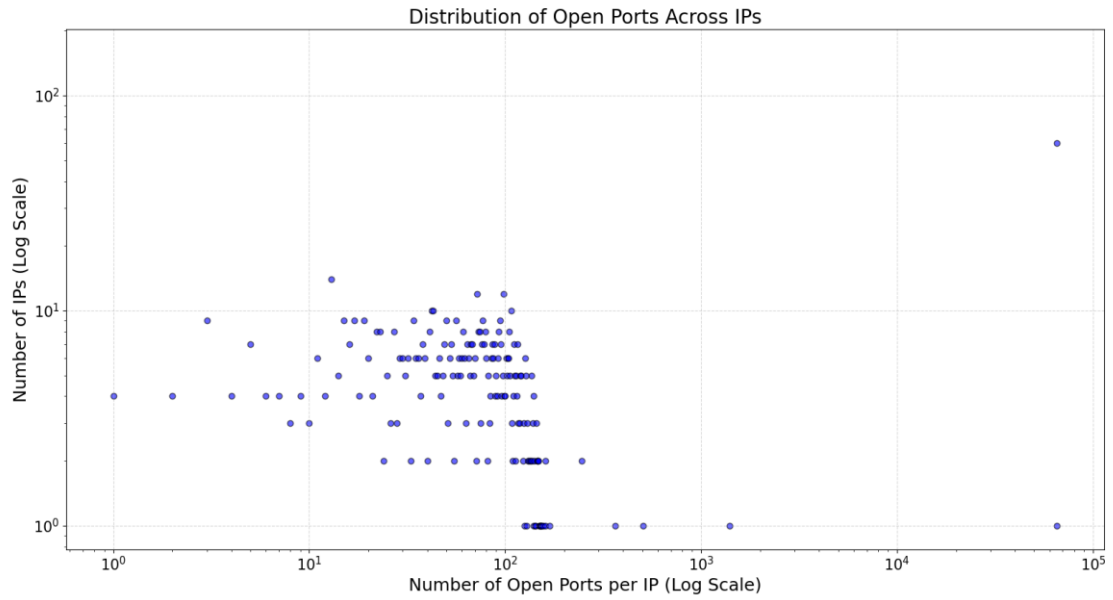


Figure 4: Number of open ports associated with each IP address with a specified port

Compared to the previous figure, the distribution here is more concentrated around IPs with 10–100 open ports. This suggests that when filtering for a specified port, many IP addresses have a moderate number of open ports. Very few IPs have extremely low or extremely high numbers.

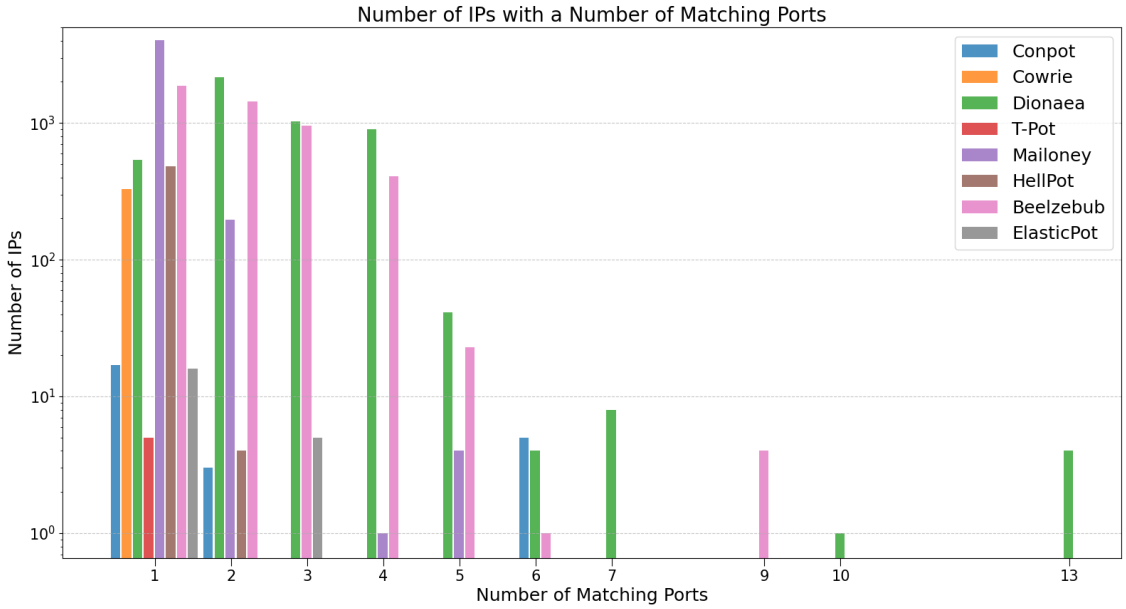


Figure 5: Number of matching ports of the IPs with the honeypots

Most IPs match only 1–4 ports, with a sharp drop afterward. Dionaea and Beelzebub have more IPs matching many ports, while Conpot and T-Pot have very few matches. A small number of IPs match many ports.

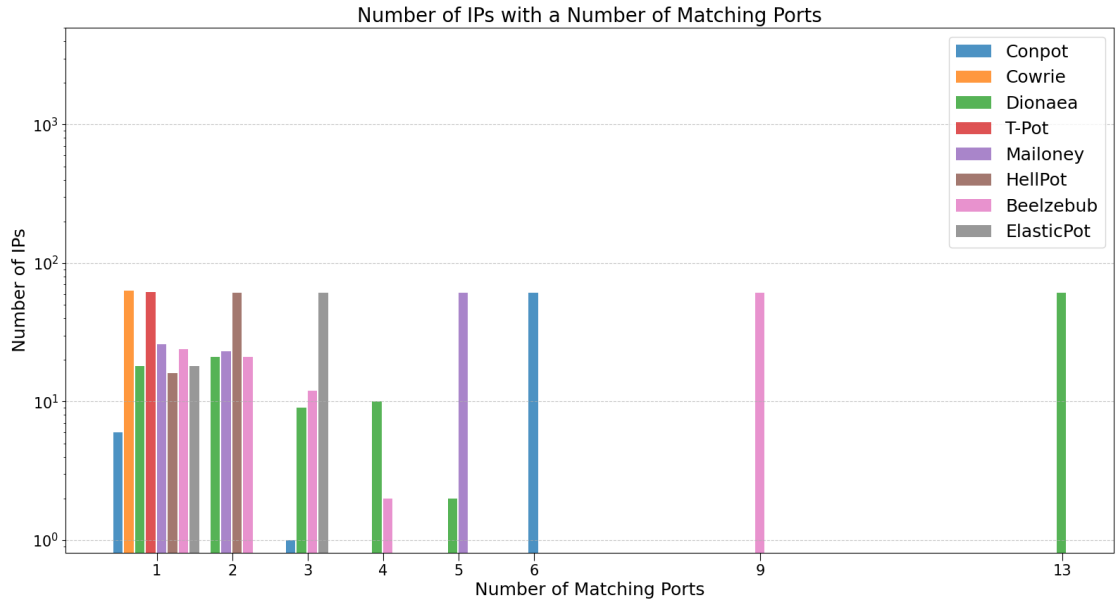


Figure 6: Number of matching ports of the IPs with a specified port with the honeypots

Compared to the previous figure, here the number of IPs increases with the number of matching ports for some honeypots. Dionaea and Beelzebub again show more IPs matching many ports, while Conpot has relatively few matches overall.

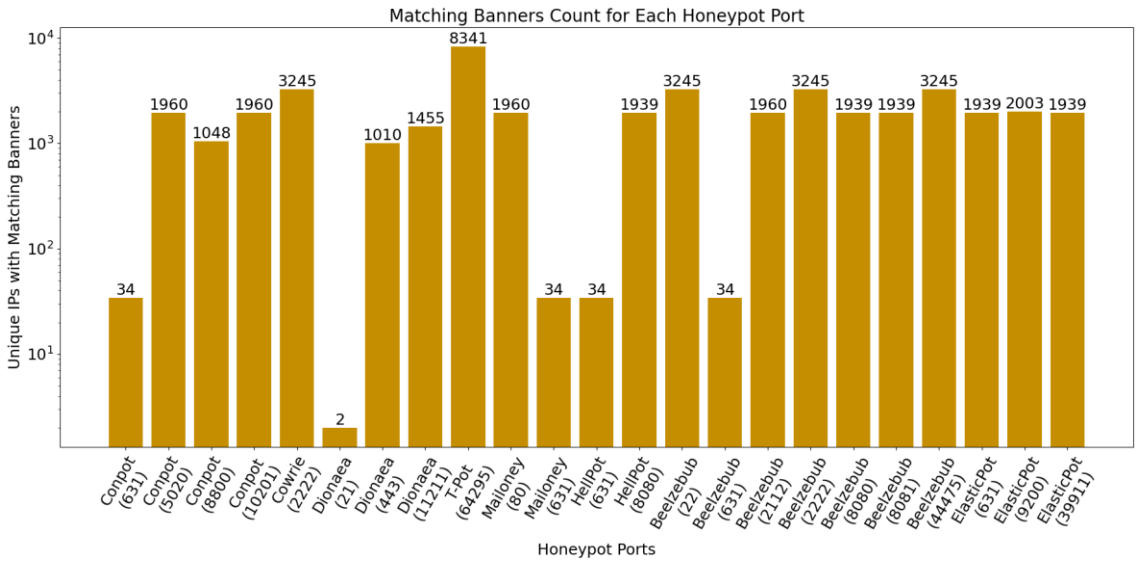


Figure 7: Number of unique IPs with matching banners for each honeypot port

Table 9: Number of unique IPs with matching banners for each honeypot port

Honeypot	Port	Unique IPs
Conpot	631	34
Conpot	5020	1960
Conpot	8800	1048
Conpot	10201	1960
Cowrie	2222	3245
Dionaea	21	2
Dionaea	443	1010
Dionaea	11211	1455
T-Pot	64295	8341
Mailoney	80	1960
Mailoney	631	34
HellPot	631	34
HellPot	8080	1939
Beelzebub	22	3245
Beelzebub	631	34
Beelzebub	2112	1960
Beelzebub	2222	3245
Beelzebub	8080	1939
Beelzebub	8081	1939
Beelzebub	44475	3245
ElasticPot	631	1939
ElasticPot	9200	2003
ElasticPot	39911	1939

As shown in Figure 7 and detailed in Table 9, T-Pot has the highest number of matching banners by a large margin. Beelzebub and ElasticPot ports show consistently high match counts, suggesting they imitate common services well. In contrast, Dionaea on port 21 has only two matches. The logarithmic y-axis shows an uneven distribution.

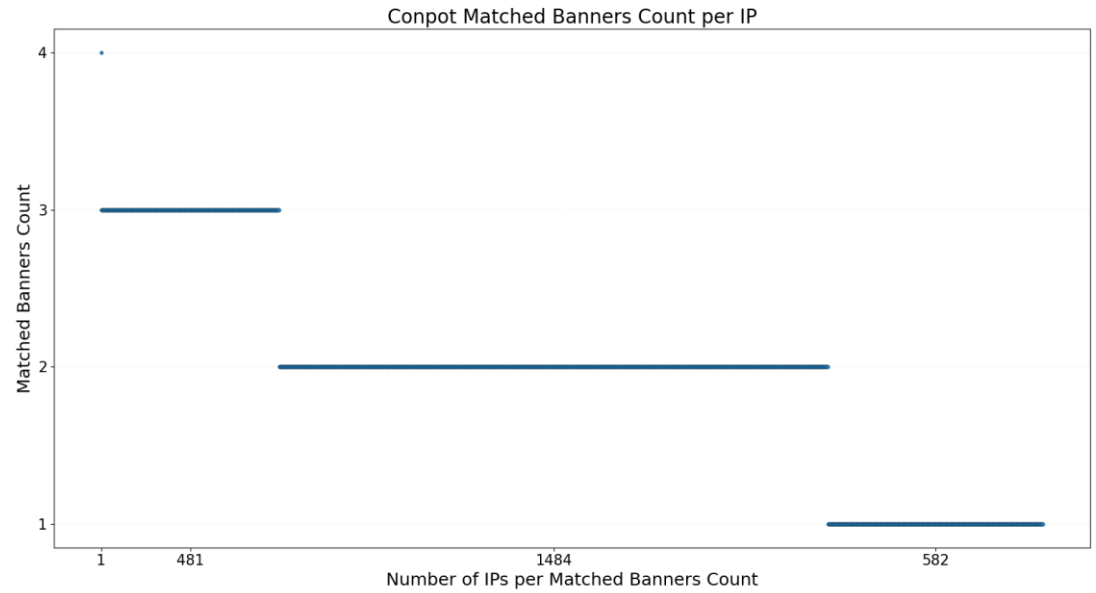


Figure 8: Number of Conpot matching banners for each IP

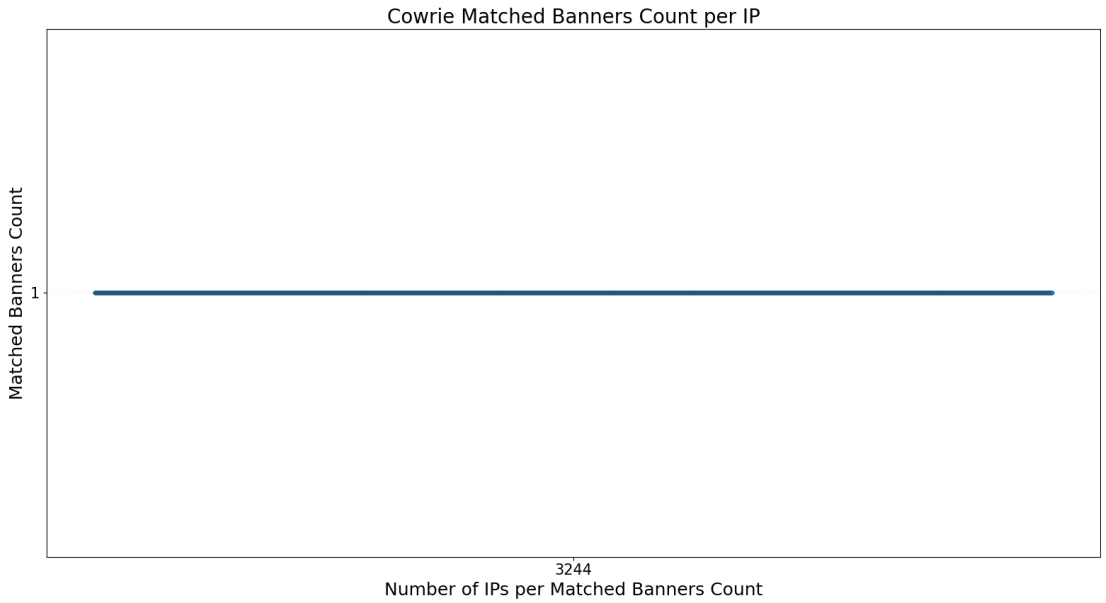


Figure 9: Number of Cowrie matching banners for each IP

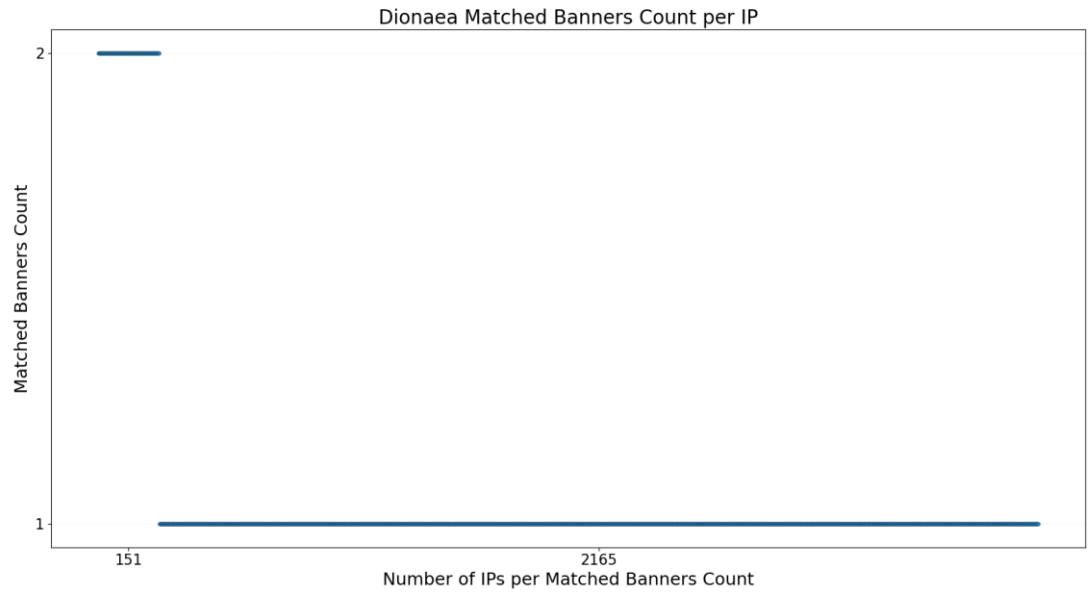


Figure 10: Number of Dionaea matching banners for each IP

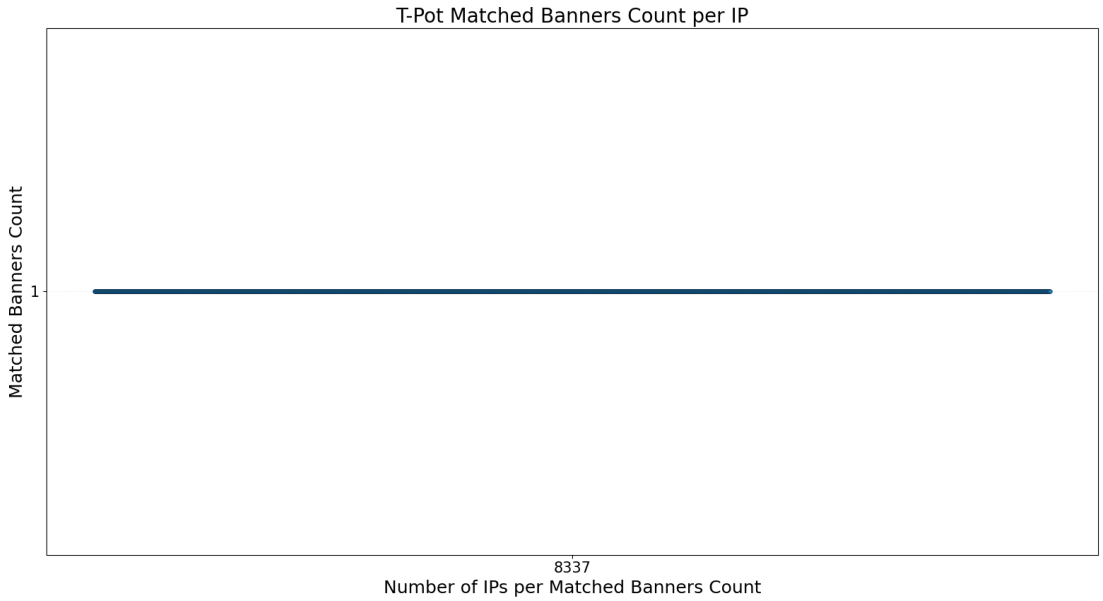


Figure 11: Number of T-Pot matching banners for each IP

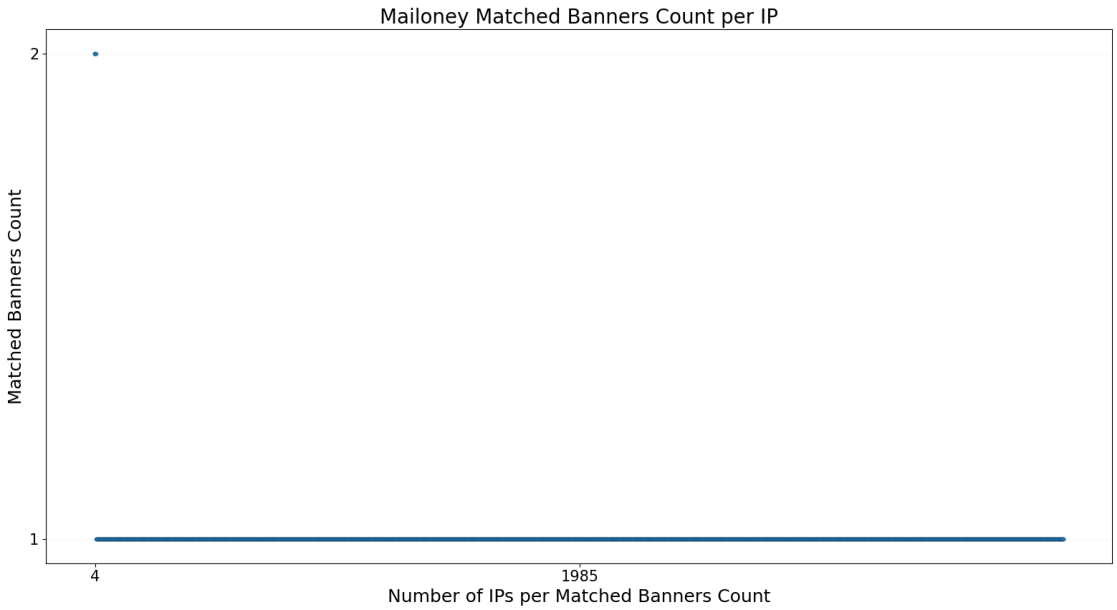


Figure 12: Number of Mailoney matching banners for each IP

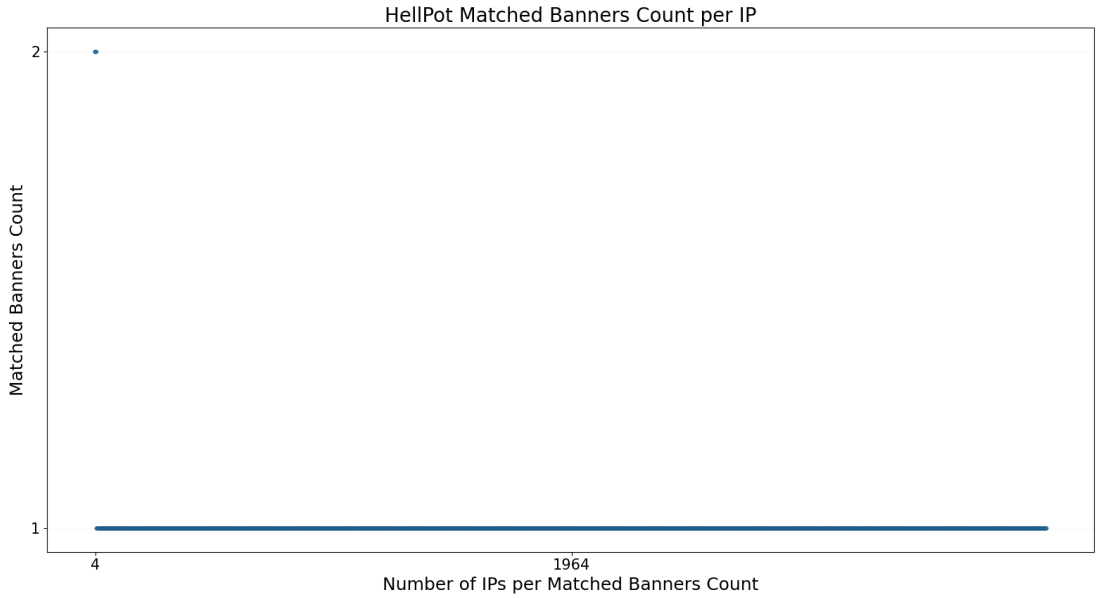


Figure 13: Number of HellPot matching banners for each IP

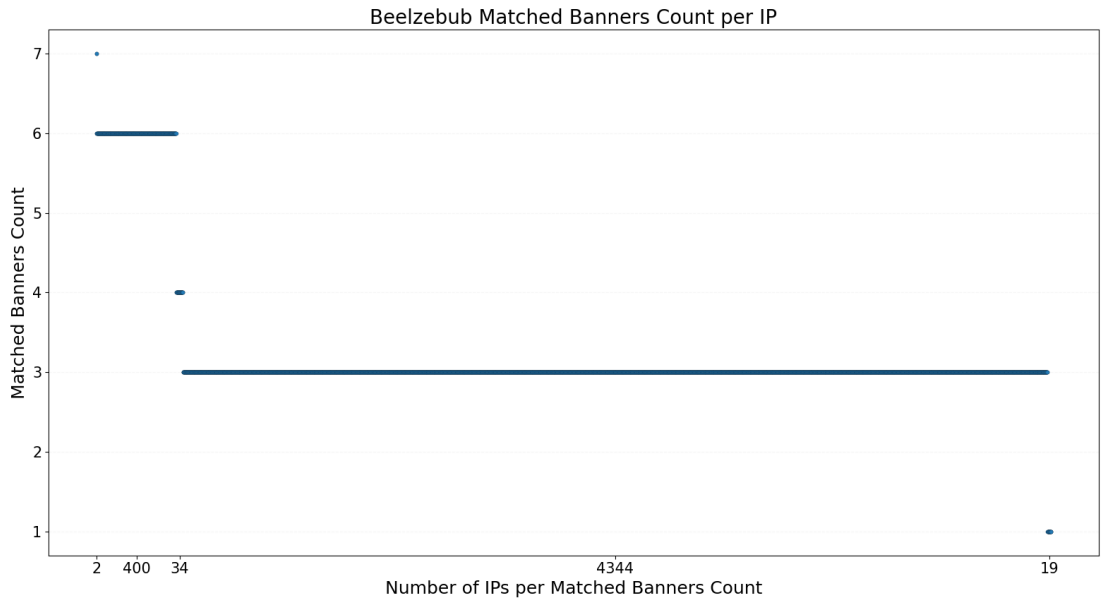


Figure 14: Number of Beelzebub matching banners for each IP

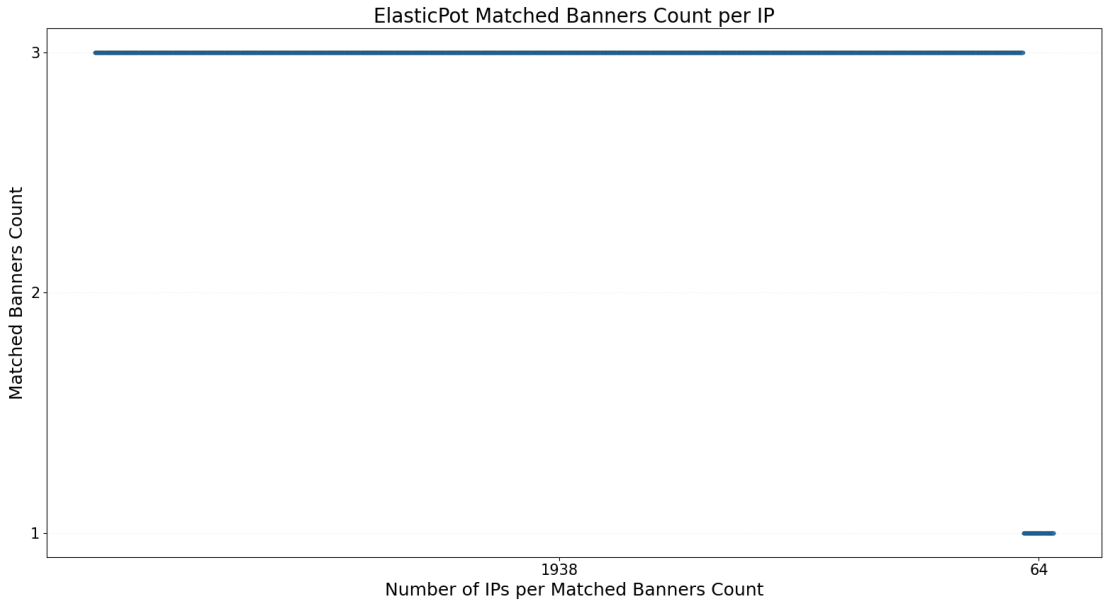


Figure 15: Number of ElasticPot matching banners for each IP

Conpot has a mix of matched banners per IP. Most IPs have 2 matched banners, but there are also cases with 1, 3, and even 4 matches, indicating some variation. Cowrie has one port, so all IPs have exactly 1 matched banner. Dionaea also mainly has 1

matched banner per IP, with very few IPs having 2 matched banners. T-Pot has one port like Cowrie, so all IPs have exactly 1 matched banner, which makes it harder to fingerprint based on banner count alone. Mailoney mostly has 1 matched banner per IP, but a very small number of IPs have 2 matched banners. HellPot is very similar to Mailoney in terms of matched banners per IP. It mainly shows 1 matched banner, with very few exceptions having 2 matches. Beelzebub shows more variation. Most IPs have 3 matched banners, but there are also IPs with 1, 4, 6, and even 7 matched banners. ElasticPot shows a pattern where most IPs have 3 matched banners, but there are some IPs with only 1 matched banner.

In summary, Cowrie and T-Pot have matches for their single ports. Dionaea, Mailoney, and HellPot mostly have 1 matched banner and rarely 2 matched banners per IP. Conpot and Beelzebub have more variation, while ElasticPot mostly has 3 matched banners per IP and a few cases with only 1 matched banner per IP.

4.3 Honeypots Fingerprinted

We found a unique banner ("lucene_version" : "4.10.4") in some of the systems we scanned from Shodan, matching with ElasticPot's port 9200 banner, which we considered a honeypot characteristic because the current version of Lucene is 10.1.0. Honeypots are frequently not updated to the latest software versions of the services they imitate which makes them fingerprintable. There were 23 IPs that we suspected to be honeypots because they had this banner on port 9200. We wrote a Python script that checks whether IPs have the honeypot tag in Shodan.

```
import os
from shodan import Shodan
from shodan.exception import APIError
from dotenv import load_dotenv

load_dotenv()

api_key = os.getenv("api_key")
```

```
if not api_key:
    raise ValueError("API key not found. Please set it in your .env file.")

api = Shodan(api_key)

INPUT_FILE = 'honeypot_suspects.txt'
OUTPUT_FILE = 'honeypots.txt'

honeypot_count = 0

with open(INPUT_FILE, 'r') as f:
    ips = [line.strip() for line in f if line.strip()]

with open(OUTPUT_FILE, 'w') as out:
    for ip in ips:
        try:
            result = api.host(ip)
            tags = result.get('tags', [])
            if 'honeypot' in tags:
                out.write(f"{ip}: HONEYPOT\n")
                honeypot_count += 1
            else:
                out.write(f"{ip}: Not honeypot\n")
        except APIError as e:
            out.write(f"{ip}: Error - {e}\n")

# Write total count to output file
with open(OUTPUT_FILE, 'a') as out:
    out.write(f"\nTotal honeypots detected: {honeypot_count}\n")
```

According to Shodan’s tags, 21 out of our 23 suspects are honeypots.

```
45.76.86.231: HONEYPOT
```

98.184.61.43: HONEYPOT
20.28.196.139: HONEYPOT
41.90.231.68: HONEYPOT
57.155.113.5: HONEYPOT
144.38.0.238: HONEYPOT
45.56.126.22: HONEYPOT
91.216.172.11: HONEYPOT
37.27.37.148: HONEYPOT
2.47.51.219: HONEYPOT
46.38.231.87: HONEYPOT
72.138.24.194: Not honeypot
92.151.15.59: HONEYPOT
114.160.33.189: HONEYPOT
133.4.180.135: HONEYPOT
23.186.168.47: HONEYPOT
80.240.29.123: HONEYPOT
37.97.81.190: HONEYPOT
87.63.238.94: HONEYPOT
37.27.90.5: HONEYPOT
136.243.222.1: HONEYPOT
45.32.116.91: HONEYPOT
172.233.89.39: Error - Unable to fetch information

Total honeypots detected: 21

5 Related Work

N. Naik et al. [6] have tried to make honeypots less fingerprintable. They developed a technique to identify and predict fingerprinting attacks in real-time by analyzing fingerprinting patterns from popular fingerprinting tools to build a predictive model. They started by fingerprinting a Windows-based honeypot, KFSensor using Nmap and Xprobe2. They run different fingerprinting attacks many times and captured and analyzed the responses using Wireshark and KFSensor logs. They tested five Nmap OS fingerprinting techniques, Basic OS detection, Aggressive mode, Fuzzy matching,

Service-based OS detection and Repeated fingerprinting attempts. They also tested Xprobe2 OS fingerprinting techniques that use ICMP and UDP based probes instead of TCP. They logged every probe and response and identified abnormalities like TCP Options & Flags, UDP Probing, ICMP Requests and ICMP Packet Size. By doing that, they made some key observations. Abnormal ICMP responses were a strong indicator of fingerprinting, TCP flag combinations were commonly used and UDP/ICMP based scanning using Xbrobe2 was stealthier but less effective on newer OS versions. Finally, they developed a 5-level risk prediction model for fingerprinting attacks ranging from lower to imminent risk. Using fuzzy logic-based correlation to refine detection, if many abnormalities appear from the same attacker, then the honeypot flags it as a potential fingerprinting attack. Their method is promising and covers several types of OS fingerprinting attacks but may be less accurate if new fingerprinting probes are used which are not covered in this method.

Others [7], used a fuzzy logic-based technique to identify and predict fingerprinting attacks in real time. They did an experiment with fingerprinting attacks on the honeypot, KFSensor and developed a fingerprinting detection mechanism that uses attack signatures from the fingerprinting tools Nmap and Xprobe2. They used a fuzzy inference system to determine how likely is there an attack occurring. They used Nmap to perform Basic OS fingerprinting, Aggressive fingerprinting, Fuzzy matching OS detection, Service based OS fingerprinting and Repeated OS fingerprinting. They collected the attack data using Wireshark, KFSensor logs and compared normal and abnormal TCP/UDP/ICMP packets. The TCP based fingerprinting indicators are TCP Options Analysis, FIN Probing, SYN/FIN Probing, XMAS Scans, Null Packets and ECN-Echo Probing. The UDP and ICMP based fingerprinting indicators are UDP Probing, ICMP Timestamp Requests, ICMP Router Solicitation Requests and ICMP Packet Anomalies. They then proposed a fingerprinting detection mechanism that uses anomaly-based detection of TCP, UDP and ICMP packets. Finally, they used their fuzzy logic-based approach to analyze and predict the Fingerprinting Attack Probability. Their fuzzy rule base combines Mamdani's fuzzy inference system with rules that define the attack probability based on the severity of anomalies. Their technique can detect fingerprinting attacks in real time but may not work when unknown fingerprinting probes are used which are not covered in this method.

Other researchers [8], used PCA (Principal Component Analysis) to discover fingerprinting attacks in real time, predict the severity of the attack and classify network anomalies based on the attack patterns. They used the honeypot KFSensor and the fingerprinting tools Nmap and Xprobe2. They executed the attack using the Nmap OS fingerprinting techniques, Basic OS fingerprinting, Aggressive OS and service detection, Fuzzy matching OS detection, Service based OS detection and Repeated OS detection attempts. They also, used Xprobe2 which relies on ICMP and UDP packets instead of TCP and sent crafted ICMP Echo, Timestamp and Router Solicitation packets. They collected the data using Wireshark and KFSensor logs. In addition, they examined network anomalies in TCP, UDP and ICMP packets to detect fingerprinting attacks. The fingerprinting attack indicators for TCP packets are FIN Probing, SYN/FIN Probing, XMAS Scans, NULL Packets and ECN-Echo Probing. When it comes to UDP packets, the fingerprinting attack indicators are ICMP Port Unreachable responses and variations in UDP payload responses. The ICMP based indicators for a fingerprinting attack are Echo Request packet size anomalies and Router Solicitation and Timestamp requests. They then used PCA to reduce dimensionality and identify the important features to detect an attack. The features they selected were TCP Flags, TCP Options, ICMP Requests, ICMP Packet Size, UDP Requests, TCP Window Size, IP Time-To-Live (TTL), IP Identification (IPID), IP Type of Service (TOS) and TCP Urgent Pointer. PCA reduced these 10 features to 5 principal components that characterize a fingerprinting attack. They confirmed using Eigenvalue analysis, that the most valuable fingerprinting indicators are TCP Flags, TCP Options, ICMP Requests, ICMP Packet Size and UDP Requests. Based on these results, they developed a 5-level risk assessment model ranging from lower to imminent risk. By using PCA, the detection rate's accuracy significantly improved with a reduction in false positives by relying on the important attack indicators. Their method can predict fingerprinting attacks in real time but may not detect some attacks which use other attack techniques.

Others [9], developed a smart honeypot using D-FRI (Dynamic Fuzzy Rule Interpolation) that detects fingerprinting attacks in real time, even when exact attack signatures are not present and it learns from attack patterns to dynamically update its rule base. D-FRI uses fuzzy inference which works best with a large rule base, which is not practical. It also uses FRI (Fuzzy Rule Interpolation) which enables reasoning with a

limited rule base. D-FRI improves FRI by dynamically updating rules, based on analyzed real time traffic. Other honeypots have static detection rules but attacks evolve and require new detection methods. They used the honeypot KFSensor and the fingerprinting tools Nmap, Xprobe2, NetScanTools Pro, SinFP3 and Nessus. They tested five fingerprinting attack types using Nmap. Namely, Basic OS detection, Aggressive fingerprinting, Fuzzy matching OS detection, Service-based OS detection and Repeated OS fingerprinting. They used Wireshark to capture attack packets to analyze them. They then listed the fingerprinting indicators in TCP/UDP/ICMP Packets. For TCP based fingerprinting they found TCP Flags and TCP Options, for UDP based fingerprinting they found ICMP "Port Unreachable" responses to UDP probes and for ICMP based fingerprinting they found ICMP Echo Requests, ICMP Timestamp Requests and ICMP Router Solicitation packets. Also, they found that FIN Scans can fingerprint Windows honeypots because they reply with RST (a TCP header flag), SYN/FIN Packets can fingerprint Linux honeypots because they reply with SYN/FIN/ACK and ICMP Timestamp Requests give away the system uptime and the clock skew. In order to develop their honeypot, they used KFSensor to log attacks, FIS (Fuzzy Inference System) to categorize fingerprinting threats and D-FRI to dynamically learn new attack patterns. Their FIS used four fuzzy input variables, MTCPF (Malicious TCP Flags), MTCPO (Malicious TCP Options), MICMPR (Malicious ICMP Requests) and MUDPR (Malicious UDP Requests). In, addition, it used one fuzzy output variable, PFT (Fingerprinting Threat Probability) and a Fuzzy Rule Base, limited to begin with and dynamically enlarged as attacks are detected. Moreover, when a new attack signature is detected, D-FRI adds missing rules to the fuzzy rule base using transformation-based rule interpolation in order to continue to improve accuracy over time. Their experiment showed significantly better attack classification after using D-FRI. Finally, their technique focuses on specific fingerprinting attacks, so it needs to be extended to detect unknown fingerprinting techniques.

N. Provos [10] developed Honeyd, which instead of simulating an entire OS, it simulates the network stack. It responds to network traffic aimed at unallocated IP addresses by using TCP, UDP and ICMP to interact with attackers. Honeyd has a packet dispatcher that processes incoming packets and sends them to the corresponding protocol handler. It changes response packets to mimic different operating systems

based on Nmap fingerprinting data so that fingerprinting tools identify it as a real system. Namely, it modifies TCP Initial Sequence Numbers, IP identification numbers, TCP options and ICMP responses. In order to mimic real applications, services are run as external scripts and traffic is redirected to real machines or back to the attacker. Honeyd has a configuration file to give the ability to select OS personalities, open and closed ports, service scripts and routing topologies. Its customizable network topologies with simulated routers, latency and packet loss give the impression of a structured network. Also, it has packet forwarding which results in realistic delays to simulate a real network. It has a Personality Engine which changes network responses to mimic real operating systems. Fingerprinting tools Nmap and Xprobe2 were used to test that it is not fingerprintable. In addition, traceroute was used to confirm that a realistic network topology is simulated and response latencies were measured and compared to configured values. As a result, Honeyd creates virtual honeypots that can fool attackers and capture their actions without the need for dedicated hardware.

R. N. Dahbul et al. [11], used threat modeling to find weaknesses in honeypots that make them fingerprintable. Honeypots are usually fingerprinted using port scanning tools, service banners and protocol inconsistencies in OSI layers 3, 4 and 7. Attackers, then poison the honeypot by sending a lot of junk traffic to it. They might compromise it and use it to launch an attack or to gather intelligence. These researchers used two systems, one with some honeypots and a real system. They fingerprinted those systems using Nmap and they compared the results for the honeypots and the real system to find inconsistencies. They found that honeypots have some uncommon ports that expose them so they should only have common ports. For Honeyd, the IIS emulation had wrong timestamps which should be changed to mimic real IIS servers. Its HTTP responses were wrong, so the Apache service script should be changed to support more methods. For Dionaea, the FTP banner was very generic, so it should be changed to resemble Microsoft FTP Service. Server Message Block (SMB) fingerprinting detected Dionaea because of fixed workgroup names, so SMB configuration should be changed to randomize workgroup names. The MS-SQL service could be fingerprinted using certain TDS response tokens, so the TokenType value in mssql.py should be changed to 0xAA. For Kippo, the ping command allowed unrealistic IPs, so ping.py should be changed to return “Unknown host” for non-existent IPs. For Glastopf, the LFI (Local File

Inclusion) error messages were too specific, so responses should be changed to "Permission Denied". They concluded that Glastopf and Honeyd had low probability of detection but Kippo and Dionaea had high probability of detection.

H. Mohammadzadeh et al. [12], developed a dynamic honeypot system in order to overcome the limitations of a static honeypot. They are dynamically adjusting the system to changes in the network by deploying and redeploying honeypots automatically based on the network state. The honeypots automatically map the network, they configure settings and they adjust to network changes. It is a Windows based platform, unlike most dynamic honeypots that are designed for Unix or Linux. It uses WinHoneyd which is a low interaction honeypot and it's the Windows version of Honeyd. The honeypots are being deployed automatically, requiring no prior knowledge of the network's topology. For analyzing the network, p0f was used for passive OS fingerprinting, Nmap was used for active OS fingerprinting, the WinPcap Library was used to capture network packets and Log Parser was used to process p0f logs to analyze the network topology. In addition, Honeypot Configuration Generator automated the honeypot setup and Analysis Module collected logs for review. The system starts by mapping the network, to decide how to deploy the honeypots. Nmap is used for active fingerprinting, by sending probe packets to determine OS and services by using TCP and ICMP analysis. For passive fingerprinting, they used p0f, to listen to network traffic without sending probes but it couldn't identify hosts that aren't sending packets. They experimented with a network with 64 Windows machines with their firewalls disabled and with a network with 10 different OS setups. They started by measuring the network mapping speed and the traffic generated. Active scanning was faster but it generated a lot of network traffic and passive scanning was slower but it didn't generate network traffic. They then tested fingerprinting accuracy for different OS versions. Nmap worked well when firewalls were disabled but it didn't work when they were enabled. P0f was less detailed but it worked even when firewalls were enabled. They chose Nmap SMB OS detection script which identified Windows OS correctly. In addition, they tested if fingerprinting methods can remain undetected by IDS (Intrusion Detection Systems). Active scanning was detected by Snort and Sax2 IDS, even with IDS evasion techniques, but p0f was not detected by IDS. The honeypots are then deployed based on network topology, using real world MAC addresses to avoid detection, and preventing

IP conflicts. Also, the honeypots are redeployed every 10 minutes based on network changes. To conclude, their system can automatically deploy, manage and redeploy honeypots on a Windows based network.

A. Vetterl et al. [13], say that honeypots use off the shelf transport layer libraries that result in protocol implementation discrepancies which makes them fingerprintable. They proposed a generic fingerprinting technique that identifies honeypots at internet scale with one probe packet and finds these protocol discrepancies that expose them. Also, they evaluated how honeypots were being deployed and how software was being updated. They experimented with Kippo, Cowrie, TPwd, MTPot, TIoT, Dionaea, Glastopf and Conpot. They then used custom probes on different protocol implementations to find protocol discrepancies, they used ZMap to scan the internet and they used cosine similarity analysis to measure how differently they responded. In addition, they selected the probe that showed the biggest response discrepancies. For SSH, fingerprinting was focused on Client version strings, SSH2 MSG KEXINIT (key exchange negotiation) packets and padding behavior in packets. For Telnet, fingerprinting was focused on responses to WILL/WON'T/DO/DON'T negotiations and responses to specific option requests. For HTTP, fingerprinting was focused on variations in response headers to uncommon HTTP methods. They used ZMap to scan the internet for SSH (port 22), Telnet (port 23), and HTTP (port 80) services. Finally, they sent fingerprinting probes to detected servers to identify honeypots. They found 7605 honeypots across 6125 IPv4 addresses and they had extremely accurate identification of honeypots. They noted that Kippo and Cowrie were easily identifiable because of SSH padding behavior discrepancies. HTTP honeypots were identifiable because of wrong HTTP header behavior. They also noted that 27% of honeypots had not been updated for more than 31 months and that many honeypots were mass-deployed with identical SSH host keys, which made them easy to fingerprint. Most honeypots were hosted on Amazon, DigitalOcean, and OVH cloud services. They concluded that honeypots can be fingerprinted at the transport layer no matter the application-level deception. Patching current honeypots is not enough so a new architecture is needed to be able to mimic real protocol behavior. At last, administrators should not use default libraries because they present distinguishable artifacts.

6 Conclusions

Honeypots that can be fingerprinted pose a security risk for the organizations that deploy them because attackers may recognize them and adjust their tactics accordingly. We downloaded popular honeypots; found the ports they have open and banner grabbed them. We then searched for real servers to try and fingerprint honeypots they might have by comparing their open ports, headers and banners with the honeypots. We managed to identify 21 honeypots due to an outdated service version they had, which matched an ElasticPot banner. If an attacker were to implement our technique and fingerprint a honeypot, they would refrain from carrying out their attack or even attempt to turn the honeypot against its organization.

It is clear that developers and administrators need to implement techniques to make honeypots less fingerprintable. They should try to match real system behavior by using realistic banners. They should make timestamps, version numbers and headers match with actual service versions. They should emulate real OS and network stack behavior by matching TTL values, TCP options, sequence numbers, and response delays. In addition, they should avoid using default honeypot signatures by modifying response content, error messages, and timing characteristics to be different from the defaults. They should change default ports to match the ports used by real services. Also, they should simulate real user activity by including real looking files, logs, user directories and uptime. Moreover, they should implement the appropriate timing to have realistic latency, by using jitter or delays. Furthermore, they should mimic vulnerabilities or misconfigurations carefully in order to emulate believable but not too easy to fingerprint vulnerabilities. They should not use outdated or obviously fake services just to lure attackers, but instead, they should mimic real misconfigurations like weak SSH keys or outdated certificates. Additionally, they should rotate banners, service fingerprints and response behavior to cycle between real headers and banners from real services. In addition, they should store logs remotely and not on the local filesystem. Also, they should periodically change open ports, banners and system characteristics to maintain a dynamic behavior. As established, developers and administrators should keep service versions up to date in banners. This is an extremely important part of banner obfuscation and realism, as we analyzed, because attackers look at version numbers to fingerprint honeypots that have old or fake service versions.

BIBLIOGRAPHY

[1] Deutsche Telekom Security GmbH and M. Ochse, "T-Pot Community Edition," GitHub repository, 2024. [Online]. Available: https://github.com/telekom-security/tpotce

[2] Wikipedia, "List of TCP and UDP port numbers," Wikipedia, the free encyclopedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

[3] Internet Assigned Numbers Authority (IANA), "Service Name and Transport Protocol Port Number Registry," 2024. [Online]. Available: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

[4] V. Gupta, "Honeypots 101 — A Beginner's Guide to Honeypots," Medium, 2020. [Online]. Available: https://bevijaygupta.medium.com/honeypots-101-a-beginners-guide-to-honeypots-f05eab2b3d17

[5] Shodan, "Tutorial: Searching Shodan," Shodan Documentation, 2024. [Online]. Available: https://shodan.readthedocs.io/en/latest/tutorial.html#searching-shodan

[6] N. Naik and P. Jenkins, "Discovering hackers by stealth: Predicting fingerprinting attacks on honeypot systems," in 2018 IEEE International Conference on Cyber Security and Protection of Digital Services (Cyber Security), 2018, pp. 1–8. doi: 10.1109/CyberSecPODS.2018.8560697.

[7] N. Naik, P. Jenkins, R. Cooke, and L. Yang, "Honeypots that bite back: A fuzzy technique for identifying and inhibiting fingerprinting attacks on low interaction honeypots," in 2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), Rio de Janeiro, Brazil, 2018, pp. 1–8. doi: 10.1109/FUZZ-IEEE.2018.8491425.

[8] N. Naik, P. Jenkins, and N. Savage, "Threat-aware honeypot for discovering and predicting fingerprinting attacks using principal components analysis," in 2018 IEEE Symposium Series on Computational Intelligence (SSCI), Bangalore, India, 2018, pp. 623–630. doi: 10.1109/SSCI.2018.8628662.

[9] N. Naik, C. Shang, Q. Shen, and P. Jenkins, "Intelligent dynamic honeypot enabled by dynamic fuzzy rule interpolation," in 2018 IEEE 20th Int. Conf. on High

Performance Computing and Communications; IEEE 16th Int. Conf. on Smart City; IEEE 4th Int. Conf. on Data Science and Systems (HPCC/SmartCity/DSS), 2018, pp. 1520–1527. doi: 10.1109/HPCC/SmartCity/DSS.2018.00250.

[10] N. Provos, “Honeyd: A virtual honeypot daemon (extended abstract),” Center for Information Technology Integration, University of Michigan, 2004. [Online]. Available: http://www.citi.umich.edu/u/provos/honeyd/

[11] R. N. Dahbul, C. Lim, and J. Purnama, “Enhancing honeypot deception capability through network service fingerprinting,” J. Phys.: Conf. Ser., vol. 801, no. 1, p. 012057, 2017. doi: 10.1088/1742-6596/801/1/012057.

[12] H. Mohammadzadeh, M. Mansoori, and I. Welch, “Evaluation of fingerprinting techniques and a Windows-based dynamic honeypot,” in Proc. 11th Australasian Information Security Conference (AISC 2013), Adelaide, Australia, 2013, CRPIT, vol. 138, pp. 59–66.

[13] A. Vetterl and R. Clayton, “Bitter harvest: Systematically fingerprinting low- and medium-interaction honeypots at Internet scale,” in 12th USENIX Workshop on Offensive Technologies (WOOT 18), Baltimore, MD, USA, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/vetterl