Understanding and Detecting Remote Infection on Linux-based IoT Devices

Hongda Li*
hongdal@g.clemson.edu
Clemson University
Clemson, USA

Qiqing Huang*
qiqinghu@buffalo.edu
University at Buffalo
Buffalo, USA

Fei Ding
feid@clemson.edu
Clemson University
Clemson, USA

Hongxin Hu
hongxinh@buffalo.edu
University at Buffalo
Buffalo, USA

Long Cheng
lcheng2@clemson.edu
Clemson University
Clemson, USA

Guofei Gu
guofei@cse.tamu.edu
Texas A&M University
College Station, USA

Ziming Zhao
zimingzh@buffalo.edu
University at Buffalo
Buffalo, USA

ABSTRACT

The rocketed population, poor security, and 24/7 online properties make Linux-based Internet of Things (IoT) devices ideal targets for attackers. However, due to the budget constraints and an enormous number of vulnerabilities on such devices, protecting them against attacks is very challenging. Therefore, understanding and detecting IoT malware *remote infection*, which is before the compromised IoT devices are monetized by adversaries, is crucial to mitigate damages and financial loss caused by IoT malware. In this paper, we conduct an empirical study on a large-scale dataset covering 403,464 samples collected from VirusShare and a large group of IoT honeypots to gain a deep insight into the characteristics of IoT malware remote infection. We share detailed statistics of shell commands found in our dataset, highlight malicious behaviors performed through those commands, investigate current states of fingerprinting methods of those commands, and offer a taxonomy of shell commands by introducing the notion of *infection capability*. To demonstrate the usefulness of the knowledge gained from our study, we develop an approach to detect on-going remote infection activities based on infection capabilities. Our evaluation shows that our detection approach can achieve a 99.22% detection rate for remote infections in the wild and introduce small performance overhead.

CCS CONCEPTS

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

*Both authors contributed equally to the paper.

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9140-5/22/05…$15.00
https://doi.org/10.1145/3488932.3517423

KEYWORDS

Linux-based IoT; Remote Infection; Malware Detection; Shell Command

ACM Reference Format:
Hongda Li, Qiqing Huang, Fei Ding, Hongxin Hu, Long Cheng, Guofei Gu, and Ziming Zhao. 2022. Understanding and Detecting Remote Infection on Linux-based IoT Devices. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3488932.3517423

1 INTRODUCTION

Over the past few years, the Internet of Things (IoT) has become ubiquitous in our daily lives being applied in many fields, such as smart home, healthcare, transportation, and industry control [31]. More than 71% of the tremendous number of IoT devices are powered by Linux, such as OpenWrt [44] and Raspbian [47], and such a trend has been growing constantly [9]. At the same time, due to poor security – such as weak passwords, unpatched software, and lack of antivirus – and the 24/7 online properties, Linux-based IoT devices become attractive targets for attackers. These compromised Linux-based IoT devices are monetized by attackers and could cause a range of security problems, such as launching large-scale DDoS attacks and stealing sensitive information. For example, in 2016, catastrophic damages to the Internet services, including GitHub, Twitter, and Spotify, were caused by several high-profile DDoS attacks [3, 18] originated from Mirai, which infected over 1.2 million Linux-based IoT devices [40]. To make matters worse, the design preferences of many IoT devices are put on low price and short time-to-market rather than costly and comprehensive security solutions. As a result, it presents a pressing challenge to protect IoT devices from being compromised by remote attackers.

Recent research [43, 45, 55] has recognized three main stages of IoT device compromise – intrusion, infection, and monetization. In the intrusion stage, the malware attempts to login to the target system by exploiting weak/default passwords or unpatched software vulnerabilities. Studies [37] show that more than 93% IoT malware

gains the shell access of the target system through cracked SSH or Telnet passwords. In the infection stage, which begins after malware successfully logins to the system, malware invokes a series of shell commands remotely to prepare the environments for the monetization that may happen later on. Finally, in the monetization stage, malware monetizes in various ways, such as launching DDoS attacks, stealing data, and cryptocurrency mining [37], which may cause real-world damages or financial loss. To mitigate damages caused by infected Linux-based IoT devices, it is crucial to fully understand the remote infections on those IoT devices to enable early detection before adversaries can exploit those devices for monetization.

There have been several efforts to investigate IoT malware. Efforts made in [6, 26] offer measurements only based on specific IoT malware families. Other efforts either devote to general Linux malware analysis [14] or understanding the status of IoT malware from a general perspective [5, 13, 45]. Beyond solely measuring and understanding IoT malware, there are also several efforts that develop detection approaches for IoT malware [2, 27, 52, 53]. However, they are all based on measurements of malicious payloads, which are used for various monetization purposes. Activities during the infection process that prepare the environment, deliver malicious payloads, and execute those payloads are overlooked.

In this work, we seek an in-depth understanding of the characteristics of IoT malware remote infection, by conducting an empirical study on shell commands found in a large-scale dataset covering samples retrieved from VirusShare [57] and logs of IoT honeypots deployed around the world. We first identify five common remote infection phases, namely *Settlement*, *Environment Preparation*, *Payload Delivery*, *Payload Execution*, and *Persistence & Covert*. We then analyze, 3,439 infection shell scripts, 48,099 malicious ELF files, and honeypot logs of 352,016 different infection incidents to extract the shell commands. After that, we share the statistics of the commands we found, highlight the malicious behaviors performed through those commands, and investigate current status of fingerprinting methods of those commands. Based on our observations, we finally propose a taxonomy of shell commands to abstract shell commands that achieve the same goal in a remote infection into groups.

To demonstrate the usefulness of the knowledge we obtain, we develop an infection detector based on the infection capabilities. We evaluate our infection detector on software IoT devices that we deployed as honeypots in the wide, where the detector achieves a 99.22% true positive rate. To evaluate the performance and functional impact introduced by our infection detector, we install and test our infection detector on three types of real IoT devices, representing low-, middle-, and high-end IoT platforms, respectively. Our testing results indicate that our infection detector introduces no more than 4% CPU load and consumes 2.7MB memory space without causing any false alerts or functional disruptions during a one-week operation in those real IoT devices.

The key contributions of this paper are as follows:

- We conduct the first empirical study on shell commands extracted from a large-scale dataset to understand the characteristics of remote infection on Linux-based IoT devices.
- We share our findings and propose a taxonomy of shell commands serving as a reference for future research on the early detection of IoT malware.
- We demonstrate the usefulness of the knowledge gained from our study by developing an infection detector that can detect ongoing infection activities with a high detection rate.
- We make the compiled dataset publicly available and open-source our detector implementation to the research community at https://github.com/soter-project/soter to benefit future research on defending IoT malware.

2 THREAT MODEL

We assume adversaries may exploit different vulnerabilities, such as weak passwords, device firmware flaws, and vulnerable applications, to compromise IoT devices. During the intrusion process, the intruder can get shell access to the target IoT device via any potential vulnerabilities. In the infection stage, a loader running on a remote (usually compromised) device or server interacts with the target device through the shell. Once the loader logins to the shell of the target device, it executes a series of shell commands to infect the device. Loaders running remotely can either be binary executables or shell scripts, which contain shell commands to be sent to (over the SSH or Telnet session) and executed in the target device. Our Study concentrates on the infection stage of IoT device compromise. This is because the infection stage happens before the monetization stage so that we can take any actions if necessary when infection activities are observed to mitigate any potential damages or financial loss in an early stage. We will not cover the intrusion stage in this work since 93% IoT device compromises are through exploiting vulnerable passwords [37], which is mainly caused by operational security issues [6, 26, 33].

3 UNDERSTANDING REMOTE INFECTION

3.1 Identifying Infection Process

Recent research on IoT malware provides a deep understanding on Mirai [6]. It is revealed that Mirai comes with a separate loader program, which asynchronously infects vulnerable IoT devices by logging in through brute-force attempts, determining system environment, downloading and executing architecture-specific binaries, and finally deleting downloaded binaries. We further investigate the source code of Mirai and recognize that actually a number of shell commands are invoked by the loader running on a remote server to perform the aforementioned operations. Another research devoted to a different IoT malware, Hajime [26], also reveals similar infection behaviors, which include logging in, downloading, and executing malicious binaries in the target system via shell commands. Besides, a prior study based on an online honeypot system, IoTPot [45], discovers that during the infection stage, IoT malware logins to the target system via Telnet and executes a series of shell commands to detect system architecture, download and execute malicious binaries, and remove the downloaded binaries.

Based on those observations, the IoT remote infection process can be generally divided into 5 phases – *Settlement*, *Environment Preparation*, *Payload Delivery*, *Payload Execution*, and *Persistence & Covert* as shown in Figure 1. In the *Settlement* phase, the loader (running remotely) tries to find a writable place as the working directory.

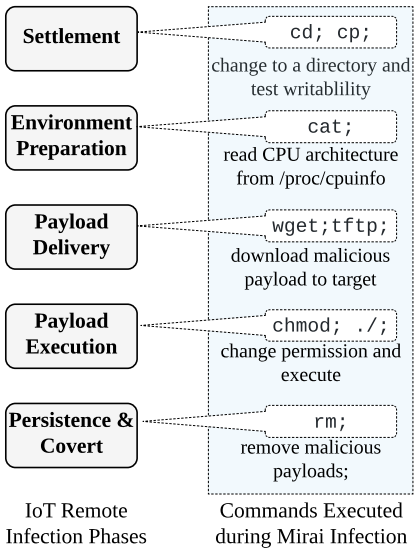

Figure 1: Five phases of IoT remote infection process that we identified (left), and shell commands found in Mirai source code for infection (right).

During the *Environment Preparation* phase, the loader collects necessary information from the system. During the *Payload Delivery* phase, the loader delivers the malicious payload to the target system. In the *Payload Execution* phase, the loader executes the delivered payload on the target system. Finally, the loader often seeks ways to persist and covert footprint in the *Persistence & Covert* phase. It is worth noting that not all loader requires all phases to accomplish a successful remote infection, nor the 5 phases must appear in exact order. For example, it is possible the *Settlement* phase is after the *Environment Preparation* phase and sometimes the *Persistence & Covert* phase is missing. Furthermore, leaked source code [40, 58], existing literature [16, 45], and reverse engineering reports [23, 29] imply that an IoT remote infection can be accomplished by executing a series of shell commands. For instance, one of the widely seen IoT malware, Mirai, achieves its infection via some common Linux shell commands as shown in Figure 1.

To fully understand how IoT remote infection proceeds, it is crucial to know what commands and in what sequence they are executed during each infection phase. This motivates us to seek a comprehensive study over large-scale data that could contain a rich number of shell commands used for remote infection.

3.2 Data Collection

3.2.1 VirusShare Dataset. We collect a dataset from VirusShare [57], which is a repository of malware samples to provide security researchers, incident responders, and forensic analysts with access to samples of live malicious code. The repository contains a large number of diverse malware samples, whose active time ranges from 2012 to 2020. In VirusShare, we find two types of files that include shell commands. One type is the bash scripts. The other is the malicious ELF (binary) files that include shell commands.

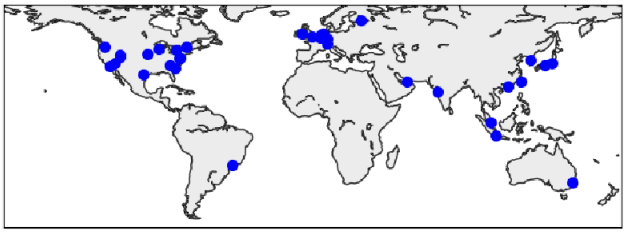

Figure 2: Geo-distribution of our deployed software IoT Devices.

We collect *all* the files on VirusShare with timestamp spanning from 2012-06-15 to 2020-04-05. To extract shell scripts, we parse all downloaded files using `base64`, `file`, and `bashlex` [7], an open-sourced bash shell parser. We obtain 3,620 Linux shell scripts through this step. We then query VirusTotal [56], a hub of more than 70 antivirus scanners and URL/domain blacklisting services, for each shell script to make sure that they are reported as malicious. We consider the report as malicious if at least one scanner or service indicating it as malicious. As a result, we retrieve 3,439 infection shell scripts. Besides, we download all the malicious ELF files available on VirusShare, which yields 48,099 malicious ELF files.

3.2.2 IoT Honeypots. We deploy software IoT devices with vulnerable passwords across the globe following the guidance introduced in HoneyCloud [15]. Worth noting, all the software IoT devices are provisioned with ARM CPU architecture – one of the most popular architectures for real IoT devices – emulated by QEMU emulator. In total, we deployed 182 software IoT devices on 4 public clouds – Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, and Vultr – distributing at 32 different sites (shown in Fig. 2) from 2020-06-25 to 2020-10-13. As a result, all the software IoT devices attract 352,016 remote infection incidents. For each remote infection incident, the honeypot records the shell command that has been executed in sequence. We distinguish different infection incidents by login sessions originating from different hosts.

To prevent our software IoT devices from being used by adversaries to launch attacks or infect more IoT devices, we employ `iptables` in the host machine (outside of software IoT devices) to block any SSH and Telnet sessions originated from our software IoT devices. Moreover, we reboot and replace the firmware of all the devices within 3 minutes after an SSH or Telnet login is detected to avoid our devices being exploited for malicious purposes.

3.3 Data Analysis

To analyze the shell scripts, we develop a tool based on `bashlex` to extract the commands from all shell scripts. We then get the statistics of the shell commands invoked by the script and understand the functionality of each shell command based on online manual pages [11, 34]. As such, we confirm that all the shell scripts in our dataset can align with a part or all of the phases of IoT remote infection we have identified. Fig. 3 shows an example of the infection script in our dataset. According to the functionalities of

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
whoami=$( whoami )  # Environment Preparation
if [ ${whoami}x != "root"x ];then
    curl http://e3sas6tzvehwgpak.tk/lowerv2.sh > /tmp/lower.sh # payload delivery
  chmod 777 /tmp/lower.sh  # payload execution
  nohup bash /tmp/lower.sh >/dev/null 2>&1 & # payload execution
    if [ ! -f "/tmp/lower.sh" ] ;then
        wget -P /tmp/ http://e3sas6tzvehwgpak.tk/lowerv2.sh  # payload delivery
        rm /tmp/lower.sh.*  # Persistence & covert
    rm /tmp/lowerv2.sh.*  # Persistence & covert
    fi
  chmod 777 /tmp/lowerv2.sh  # payload execution
  nohup bash /tmp/lowerv2.sh >/dev/null 2>&1 &  # payload execution
else
  echo "*/5 * * * * curl -fsSL http://e3sas6tzvehwgpak.tk/r88.sh|sh" > /var/spool/
  cron/root # Settlement
  mkdir -p /var/spool/cron/crontabs  # payload execution
  echo "*/5 * * * * curl -fsSL http://e3sas6tzvehwgpak.tk/r88.sh|sh" > /var/spool/
  cron/crontabs/root # Settlement
    curl http://e3sas6tzvehwgpak.tk/rootv2.sh > /tmp/root.sh  # payload delivery
  chmod 777 /tmp/root.sh  # payload execution
  nohup bash /tmp/root.sh>/dev/null 2>&1 &  # payload execution
    if [ ! -f "/tmp/root.sh" ] ;then
        wget -P /tmp/ http://e3sas6tzvehwgpak.tk/rootv2.sh  # payload delivery
        rm /tmp/root.sh.*  # Persistence & covert
    rm /tmp/rootv2.sh.*  # Persistence & covert
    fi
    chmod 777 /tmp/rootv2.sh  # payload execution
    nohup bash /tmp/rootv2.sh >/dev/null 2>&1 &  # payload execution
fi
```

Figure 3: A sample infection script in our dataset. SHA-256: 2a151e1148fb95c7696b05db4c58d1fd8e138f0f9c8c638228c203ad273523f8

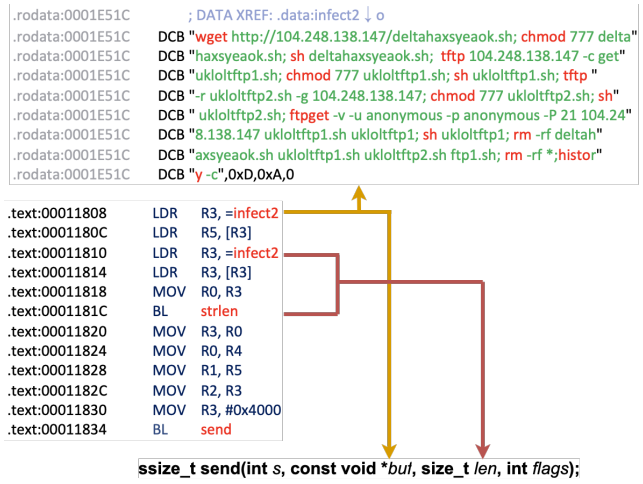

Figure 4: A sample ELF file in our dataset. SHA-256: cc0e1ff4ef6ae076c55c7435457dbd647789989fbfecdc04262f26bd02deac73

those commands, we can align them with five phases of the remote infection. A full list of the mapping relations of each command is shown in Table 6.

In addition to the infection scripts, we also extract shell commands from the ELF files in our dataset. Fig. 4 showcases an example of such ELF files that send shell commands to a remote target. By disassembling the ELF file, we can see the send system call is invoked with shell command strings as arguments, based on which we derive that this ELF is executed on a remote server and invokes those shell commands through the network.

For the IoT honeypot logs, we simply extract the command sequences executed by each remote infection and store them in dedicated text files. We consider the command sequence of a single infection incident as a data sample in our dataset.

3.4 Analysis Results

In this section, we provide detailed statistics of shell commands executed in a remote infection, discuss the malicious behaviors performed through those shell commands, and investigate the current state of fingerprints for those shell commands. We finally provide a taxonomy of shell commands based on each command's capabilities in a remote infection.

3.4.1 Statistics. The commands found in infection scripts, malicious ELF files, and the honeypot logs are quite concentrated. Table 1 lists the top-20 commands from those sources in our dataset. The percentages drop to 0.17%, 0.81%, and less than 0.01% for the 20th command in infection shell scripts, malicious ELF files, and honeypot logs, respectively. We totally find **169** different shell commands from the infection shell scripts and malicious ELF files while from the honeypot logs, we only find **52** different shell commands. Furthermore, we verified that the 169 shell commands cover all the

Scripts	Percentage	ELF File	Percentage	Honeypot Logs	Percentage
cd	52.28%	**cd**	13.39%	**cd**	37.55%
./executable	10.75%	**rm**	10.08%	**sh**	13.18%
wget	8.67%	**sh**	8.97%	*./executable*	9.72%
rm	7.98%	**chmod**	7.05%	**cat**	9.12%
chmod	7.72%	**wget**	5.07%	**echo**	8.82%
tftp	3.26%	**kill**	5.05%	**wget**	4.86%
echo	1.61%	**free**	4.82%	**id**	4.49%
curl	1.46%	**read**	4.74%	**su**	4.15%
sshd	0.63%	**tftp**	4.65%	**head**	4.01%
ftp	0.59%	**open**	4.64%	**rm**	2.63%
sh	0.57%	**sleep**	4.26%	**chmod**	0.64%
bash	0.57%	**busybox**	2.81%	**cp**	0.54%
cp	0.43%	**pkill**	2.49%	**tftp**	0.18%
chattr	0.38%	**history**	2.38%	**ps**	0.08%
busybox	0.31%	**printf**	1.69%	**uname**	0.01%
mv	0.24%	**service**	1.68%	**ls**	0.01%
ssh	0.24%	**ftpget**	1.63%	**grep**	<0.01%
ulimit	0.22%	**iptables**	1.16%	**killall**	<0.01%
touch	0.18%	**readlink**	1.06%	**ifconfig**	<0.01%
ftpget	0.17%	**sshd**	0.81%	**mkdir**	<0.01%

Table 1: Top 20 commands found in infection scripts, malicious ELF files, and honeypot logs.

52 shell commands found in honeypot logs. This statistic implies that shell commands executed by remote infections are limited to a small command set in a real-world setup. A full list of all the commands discovered in our dataset is provided in Table 6 in the appendix.

Takeaway: *shell commands executed during remote infections are concentrated and in practice, this number is much smaller than what we found in a static dataset.*

One reason remote infections via shell commands are widely employed to target the Linux-based IoT devices is that infecting via shell command is compatible with diverse IoT devices. The shell commands can be categorized into three categories, *external*, *built-in*, and *hybrid*. The *external* commands refer to utility programs that are installed in the system, such as under `/usr/bin`, `/usr/sbin`, `/bin`, `/sbin`, ect. Those commands are independent of the CPU

architecture and the shells. As long as those utility programs are installed in the system, they are ready to be executed and can be invoked by different shells (e.g., sh, zsh, csh, tcsh, etc.). *built-in* commands refer to the internal commands that are implemented by specific shells. For example, `kill`, `history`, and `cd` are implemented by shells, thus may differ from shell to shell. Nevertheless, all the Unix-like shell families share a large comment set of *built-in* commands. *hybrid* commands refer to those commands that are implemented by shells while there are also utility programs out there. For example, `echo`, `printf`, and `pwd` are implemented as shell built-in commands while there may be utility programs with the same name installed in the system. Our study reveals that the majority of the shell commands are external commands. The detailed statistics are listed in Table 5 in Appendix B.

Takeaway: *external commands are the majority of shell commands, which make remote infections via shell commands highly compatible with various IoT device architectures and shells.*

3.4.2 Trail and Error. *Trail and error* is a cost-effective way to figure out the right parameters required by certain commands given that limited information is exposed to the adversaries. Generally, we observe two scenarios where *trail and error* is used. In the first scenario, a remote loader logins to the target shell and sends commands like `cd || cd || cd` to test which path is accessible. We find **87.44%** of the samples in our dataset includes this behavior pattern. In the second scenario, a remote loader tries to download malicious payload with different tools, where the command patterns are isomorphic to `wget || curl || tftp`. We find **94.6%** of the samples in our dataset includes this behavior pattern. The reason that *trail and error* pattern is widely employed by remote infections is that it can significantly increase the infection success chance while remaining logically simple. It is worth noting that the *trail and error* behaviors are highly suspicious since a legitimate administrator should have the idea of where the best working place is and which tools are available in the system.

Takeaway: *trail and error is widely used by remote infection scripts and is highly suspicious.*

3.4.3 Embedded Malicious Payload. Malicious payload delivery is critical to the remote infection, even for the whole compromise lifecycle. We find that most of the samples in our dataset (97.44%) utilize one or more download commands to deliver the malicious payload. A small portion of samples (0.47%) in our dataset embeds malicious payload as part of the command arguments. The technique used to embed malicious payloads is the so-called *here document* [25]. One advantage of embedding malicious payloads over downloading is that it can bypass firewalls in some cases. For example, some networks disallow download originated from inside to an unknown external server. This will result in regular download tools failing to download. Another advantage of embedding malicious payloads is that it does not rely on any download tools on the target system. This makes the infection more robust to resource-constrained systems where download tools usually are not available. However, embedded malicious payloads usually rely on `base64`, a shell command that converts binary data into ASCII string format and vice versa. Thus, `base64` becomes the dependency of infections with malicious payload embedded.

Takeaway: *remote infection scripts embed malicious payloads in rare cases where shell commands like base64 are required to convert between binary data and ASCII string format.*

3.4.4 Fingerprints. There are two common fingerprinting methods of the 3,439 infection scripts. One is based on the IP addresses involved in the shell scripts. The other is based on the MD5 of the shell script. We investigate those two fingerprinting methods of the 3,439 infection scripts in our dataset. First, we obtain the statistics of unique IP addresses in all the infection scripts. We totally find **1963** unique IP addresses appearing in **2296** infection scripts. We query those IP addresses against IPsum [50], a threat intelligence feed based on 30+ different publicly available lists [51] of suspicious and/or malicious IP addresses. The databases are daily updated, and our results are cut off as of 2021-07-20. As a result, we find that only **28** out of 1963 unique IP addresses are included by those publicly available lists. We further test the availability of all the IP addresses found in our dataset, and it turns out none of them is still accessible.

Takeaway: *the IP addresses involved in shell scripts tend to be temporary and are unlikely to be covered by publicly available malicious IP address databases.*

Second, we study the reports of VirusTotal regarding the infection scripts in our dataset. Figure 5(a) shows the cumulative distribution function (CDF) graph of the number of VirusTotal engines that report each script in our dataset as malicious. The results imply that for 83.42% of all the infection scripts, there are 20 to 30 VirusTotal detection engines reporting them as malicious. Figure 5(b) shows the percentages of VirusTotal detection engines that reported them as malicious. The number of engine reports varies from 26 to 61. For most of the infection scripts (91.08%), the ratio that VirusTotal engines reporting it as malicious is between 17% and 31%. As a summary, VirusTotal detection engines have limited signature coverage of the infection scripts in our dataset. With the above limitation in mind, we recommend choosing the threshold of the report ratio at around 17% or the report engine number at approximately 20 to achieve low false positives as well as low false negatives. This recommendation is consistent with the results in [60].

Takeaway: *choosing a report ratio around 17% or report number around 20 of VirusTotal achieves a good trade-off between false positives and false negatives of infection scripts.*

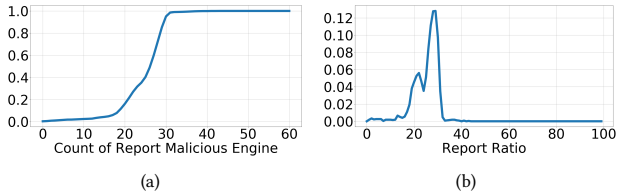

Figure 5: A CDF graph showing the number of VirusTotal engines that classify each script in our dataset as malicious (a), and the distribution of the ratio that VirusTotal engines reporting an infection script as malicious (b).

Infection Capabilities	Abbr	Explanation	Commands
Change Permission	**CH**	Change the permission of files	chmod usermod chattr umask chown chgrp
Remove History	**RM**	Remove activity history to evade forensic	rm history
Disable Security	**DS**	Disable security mechanisms	accton ufw
Download	**DW**	Download files from the Internet	wget tftp curl ftp apt-get lynx git ftpget mail
Find Place	**FP**	Try to find a working place	cd
Copy File	**CP**	Copy files	cp tail head mv cat read scp strings tee
Create and Write	**CW**	Create and write to new files	echo printf mkfifo open
Decompress	**DCP**	Decompress files	tar gunzip gzip unzip
Decode	**DCD**	Decode from encoded files	base64
Compile Code	**CC**	Compile source code	make gcc cc ldconfig
Process Text	**PT**	Search, cut, sort texts	grep awk sed cut tr sort egrep uniq wc
Kill Process	**KILL**	Kill processes	pkill killall kill
Exclude Others	**EXO**	Avoid others to infect or login	chpasswd iptables-restore passwd
Network Probe	**NP**	Probe internal or external networks	nmap zmap di
Implant Backdoor	**IBD**	Launch a deamon to enable access later	httpd squid sshd
Execute	**EXE**	Execute files	nohup xargs crontab perl sh service nc sshpass bash exec nice php python screen ssh busybox ./executable env
Collect Information	**CI**	Collect information from the system	uname which ls ifconfig lsof arp stat id whoami netstat lsb release ping hostname chkconfig df file free fuser getconf iptables-save nproc pidof socklist uptime who lspci ps du lastlog
Manage System	**MSYS**	Change users and environment variables	sysctl iptables userdel su mkisofs useradd defaults sudo reboot
Manage Software	**MSOFT**	Update or install software	mktemp mkdir export ln apt-key dpkg yum
Manage Resource	**MRES**	Set/reset resource hard and soft limit	ulimit
Get Time	**GT**	Get timestamp of files	date
Change Time	**CT**	Change timestamp of files	touch
Programming	**PR**	programming commands, e.g., break; continue	expr test set unset declare local continue break unalias exit enable return let true trap readonly getopts
Agnostic	**AGN**	Any commands do not align to the above	sleep yes lp find pwd md5sum clear fold kdialog logout wall
Unrecognized	**UN**	Any commands not including in our base	

Table 2: A list of infection capabilities we abstracted and corresponding explanations.

3.4.5 Shell Command Taxonomy. We provide a taxonomy of shell commands based on the notion of *infection capability*: an abstraction of a sort of shell commands that can achieve certain goals during the remote infection. Abstracting infection capabilities from commands rather than studying each specific command directly makes our understanding more general. This generality allows future work to add new commands or remove outdated commands to keep the knowledge base up-to-date and offers a general way to understand and organize how shell commands are executed in a remote infection. We label all the 169 shell commands in our dataset with 25 infection capabilities according to the goals that a command can achieve in remote infections.

For example, lots of remote infections download files from a server on the Internet in the *Payload Delivery* phase. There are a set of shell commands, such as `wget`, `tftp`, `curl`, and `git`, that can achieve this goal. Then we label these commands, which have the capability to download files from a remote server with *Download* infection capability. Table 2 shows our taxonomy of shell commands based on infection capabilities. Among the infection capabilities, there are three special ones, which are not abstracted according to the goals the shell commands:

- *Unrecognized.* Any new commands that are not included in our current dataset will be labeled with *unrecognized* infection capability in our taxonomy.
- *Programming.* Commands that are used for general programming purpose is labeled with *Programming* infection capability. Among the examples are `break`, `continue`, `local`, `set`, etc.
- *Agnostic.* Commands that cannot be labeled like any other infection capabilities will be labeled as *agnostic*. Among the examples are `sleep` (wait for a certain time), `pwd` (print working directory), `clear` (clear screen contents), etc.

Based on our taxonomy, we further study what infection capabilities are exploited in different infection phases. The *agnostic* and *programming* infection capabilities are exploited in all infection phases since they do not attach to any specific infection goals. Some infection capabilities, including *process text*, *copy file*, *manage software*, *manage resource*, *disable security mechanisms*, and *manage system* are exploited by more than one infection phases, while the remaining infection capabilities are specific to a single infection phase. In addition, we investigate how many infection phases are involved in each sample of our dataset. We find that 0% samples involve only 1 infection phase; 0.11% samples involve only 2 infection phases; 0.17% samples involve only 3 infection phases; 37.19% samples involve only 4 infection phases; and 62.05% samples involve all 5 infection phases. The statistic implies that not all 5 infection phases are necessarily presented for a remote infection. But most remote infections involve 4 or 5 infection phases. Refer to Figure 13 in Appendix C for a visualized summary of the exploit relations between each phase and infection capabilities.

4 DETECTING REMOTE INFECTION

To demonstrate the usefulness of our knowledge, we develop a preliminary infection detector utilizing the taxonomy proposed in Section 3 and evaluate our infection detector.

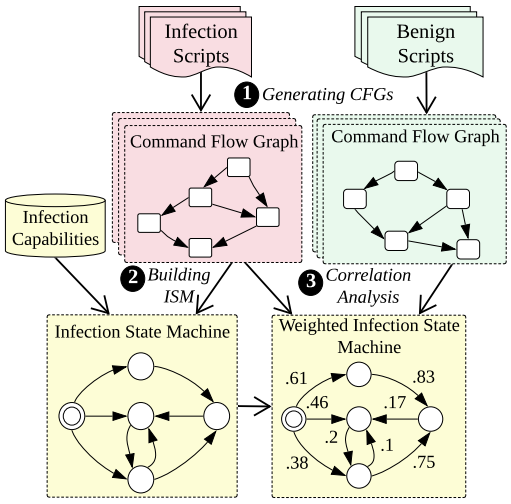

Figure 6: Modeling approach overview. Firstly, we generate the CFGs of infection and benign scripts in our dataset. Then we build an ISM that represents the behavior patterns of malware infection. After that, we assign weights to the ISM through a correlation analysis. The generated WISM works as a general model of malware infection.

4.1 Model Development

In this section, we model the remote infection process as a finite state machine, which is constructed using the infection scripts in our dataset. Since this model will be used for remote infection detection, a baseline is required to distinguish infection activities from benign activities. To determine the baseline between infection and benign activities, we collect benign shell scripts via FIRMADYNE [12], a tool that can download firmware images and associate metadata from supported IoT device vendor websites. After collecting the firmware images, we use the firmware walker [20] and Firmware Mod Kit [21] to search the firmware image for shell scripts. We finally acquired 9,337 unique benign shell scripts after de-duplicate the extracted shell scripts with MD5 values.

An overview of our modeling approach is illustrated in Fig. 6. The output of our modeling is a weighted infection state machine (WISM) that models the remote infection. Our modeling approach consists of 3 major steps. First, we generate the *command flow graph* (CFG) for all the infection scripts and benign scripts in our dataset. Second, we build an infection state machine (ISM) based upon the CFGs of all the infection scripts. Finally, we conducted a correlation analysis based on the CFGs of infection scripts and benign scripts. The correlation analysis tracks the capabilities in all CFGs and assigns a weight to each state transition in the ISM. The weights are maximized for infection scripts while minimized for benign scripts.

4.1.1 Generating Command Flow Graphs. The *command flow graph* (CFG) is a representation, using graph notation, of all paths containing shell commands in the sequence that might be traversed through a shell script during its execution. In our generated CFGs, each node represents a shell command, and each directed edge represents a transfer from one command to another one. We develop our own tool to generate the CFGs based on `Bashlex` [7], which is an open-sourced parser for bash scripts. We generate a CFG for each infection or benign script. Fig. 7 illustrates two infection scripts (script-a and script-b) and their corresponding CFGs. Each CFG is stored as a file using the `networkx` [54] Python library. Those CFGs will be used when we build the ISM and WISM.

4.1.2 Building Infection State Machine. We use an ISM to represent the relations between infection capabilities that are exploited by the remote infection. We formally define our ISM as a 5-tuple $(\Sigma, S, s_0, \Delta, F)$ where:

- Σ is the set of all the infection capabilities that we have abstracted;
- S is the set of states, each of which is mapped to an infection capability in Table 2;
- s_0 is the initial state, which also belongs to S but is not mapped to any infection capability;
- Δ is the state-transition function: $\Delta : S \times \Sigma \rightarrow S$; and
- F is the set of final states.

We only build a single ISM from all the infection scripts. As an example, Fig. 7 depicts the CFGs of two infection scripts and the ISM built from those two scripts.

In the ISM, we consider each node as a state where a specific infection capability has been exploited during the infection. Each state in our ISM is mapped to an infection capability except for the initial state, s_0. For example, a state that maps to `RM` means that if the infection goes into this state, the `RM` infection capability has been exploited. Therefore, some shell commands, such as `rm` and `history`, must be invoked. We associate a directed edge with an infection capability, covering different shell commands. For example, a directed edge coming into the `RM` state is associated with the `RM` infection capability, which abstracts the `rm` command in Fig. 7. It is worth noting that this abstraction captures the transitions at the infection capabilities level rather than the command level. To automatically build the ISM from all the infection scripts, we develop a tool that takes the CFGs as inputs and yields an ISM. Each state in the ISM is mapped to an infection capability. A state transition in the ISM means one more infection capability is exploited.

4.1.3 Correlation Analysis. Our correlation analysis tracks a sequence of state transitions in the ISM to determine a remote infection. The key idea is to assign each state transition in the ISM with weight and maintain a counter for the weights over a temporal window. Once the counter exceeds a threshold, a remote infection is detected. To determine the weight for each state transition, we employ the late acceptance hill-climbing (LAHC) algorithm [10]. The LAHC algorithm can be used to find a local optimum for an optimization problem in a bounded time frame.

Let's denote a WISM as θ, a series of state transitions that are triggered by a sequence of N infection capabilities as $T = (t_1, t_2, ..., t_{N-1})$, and $0 \leq \theta_{t_i} \leq 1$ as the weight that will be assigned to the state transition t_i ($i \in [1, ..., N-1]$) in WISM. Then, the risk score of a sequence of N infection capabilities is defined as

$$R(T, \theta) = \frac{1}{N} \sum_{i=1}^{N} \theta_{t_i} \quad (1)$$

With the above definition, we can calculate a risk score given a WISM and a sequence of infection capabilities.

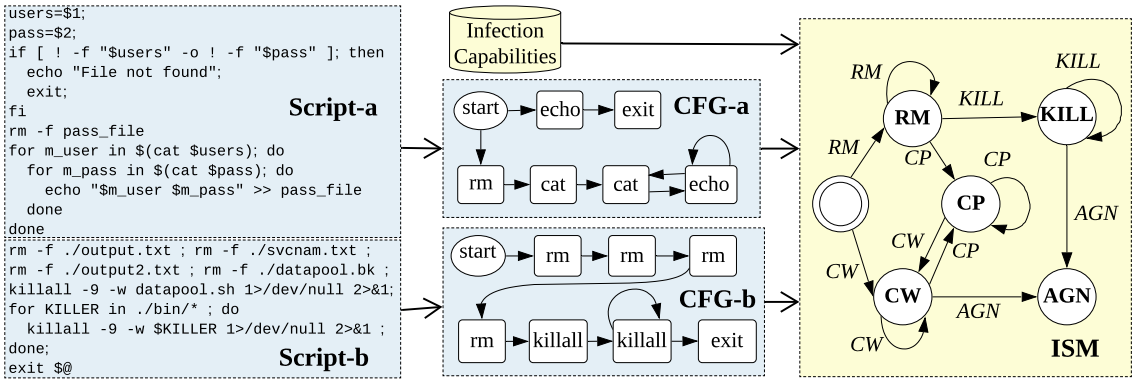

Figure 7: Building the ISM from two infection scripts. Refer to Table 2 for abbreviations of the capabilities and Table 6 in Appendix for command to infection capability mapping.
Script-a SHA-256: 9dd5a5ca05290aabb87e4472c78e316bcee6a37eb30bf7bbf8c3b4c3a3646941
Script-b SHA-256: 0bc440c8382a9fcf161d7c8496b270d59f4893ad0aa15ed1474ef5a99a2455f3

To obtain a WISM that best models the remote infection process, we *minimize* the risk score for all the sequences of infection capabilities found in benign scripts and *maximize* the risk score for all the sequences of infection capabilities found in infection scripts. Therefore, we can formulate the weight assignment as an optimization problem that maximizes the following objective function:

$$f(\mathbf{M}, \mathbf{B}, \theta) = \sum_{i=1}^{N_\mathbf{M}} R(T_i^\mathbf{M}, \theta) - \sum_{j=1}^{N_\mathbf{B}} R(T_j^\mathbf{B}, \theta) \quad (2)$$

Where $\mathbf{M}$ is a set of sequences of infection capabilities found in infection scripts, $N_\mathbf{M}$ is the size of $\mathbf{M}$, $T_i^\mathbf{M}$ is a series of state transitions triggered by the ith sequence of infection capabilities found in infection scripts, $\mathbf{B}$ is a set of sequences of infection capabilities found in benign scripts, $N_\mathbf{B}$ is the size of $\mathbf{B}$, $T_j^\mathbf{B}$ is a series of state transitions triggered by the jth sequence of infection capabilities found in benign scripts.

We then employ the LAHC algorithm to solve the optimization problem. The LAHC algorithm relies on a feedback loop to gradually improve the quality of weights in the WISM. For each iteration, the algorithm adjusts all the weights of the WISM with a small amount (0.01 in our case). The initial weights of each state transition are set to the frequency of the infection capabilities found in the infection scripts. Over a sufficiently large number of iterations (100,000 in our case), we observe the convergence and obtain the weights for a WISM.

4.2 Detector Implementation

Our infection detector is implemented to run in the kernel space and detects malware infection in real-time. Fig. 8 provides an overview of our infection detector . The *hook* is implemented as C function pointers that point to the entry point of the *classifier*. The *classifier* consists of the WISM, PID-keyed temporal windows, a threshold tester, and an alert function. Every time a command execution triggers the *hook* through `execve` system call, the infection detector retrieves the temporal window according to the PID value. Once the login session terminates, the `exit` system call is called, which is also hooked to trigger the removal of the relevant temporal window. Our infection detector recalculates the risk score based on the infection capabilities that have been exploited within each temporal window separately. If the risk score does not exceed a threshold, the *classifier* returns to `execve` to continue the normal routine. Otherwise, an alert function is called, which writes a detection log to the `rsyslog` system [48] and returns to `execve` with an error code to stop the following actions of this command. Our infection detector skips the shell build-in commands, such as `echo`, `set`, `continue`, etc., since those build-in commands do not trigger `execve`. However, the infection detector traces `cd` command by hooking the `chdir` system call because this build-in command has been widely used by remote infections (refer to Table 1 for statistics).

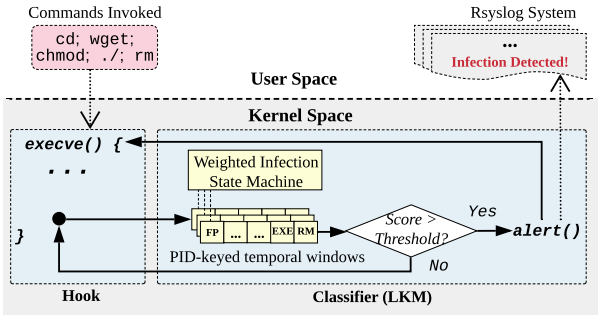

Figure 8: Overview of the infection detector implementation.

Two parameters need to be defined for the infection detector: *i)* the risk score threshold to distinguish remote infection from legitimate use; and *ii)* the temporal window size. When we choose the optimal threshold, we make use of infection scripts and benign scripts in our dataset. For each script, we compute the risk scores of all the infection capability sequences in that script. Then, we choose the highest risk score as the risk score of that script. After that, we compute the Cumulative Distribution Function (CDF) of risk scores for both infection scripts and benign scripts. The CDF is shown in

Fig. 12 in Appendix. By observing the CDFs, we estimate a threshold that satisfies a certain level of true alert rate while maintaining a relatively low false alert rate. For our implementation, we choose 0.50 as the threshold to achieve a reasonably high true alert rate while remaining relatively low false alert rate. To determine the temporal window size, we count the length of all the command sequences we extracted from our infection samples. We observe that all the command sequences are shorter than 34. As a result, we decide to choose 34 as the maximal length of the command temporal windows. When our infection detector is running, a number of 34-length temporal windows will slide over the infection capability sequences exploited by different login sessions respectively. This parameter is adjustable by reloading the infection detector.

4.3 Evaluation

4.3.1 Effectiveness Evaluation of Our Detector. We deployed our infection detector on the software IoT honeypots from 2021-03-29 to 2021-05-01 to evaluate its effectiveness in detecting ongoing remote infections. Since the software IoT devices are deployed as honeypots, all logins to those devices are considered remote infections. Table 3 lists the detecting results of the infection detector overall 147,860 remote infections, among which our infection detector raises 146,702 alerts. As a result, the FNR is 0.78%, and TPR is 99.22%.

	Total	Alert	FN	FNR	TPR
Remote Infections	147,860	146,702	1,158	0.78%	99.22%

Table 3: Real-time detection results of the infection detector. FN: False Negative; FNR: False Negative Rate; TPR: True Positive Rate.

We further investigated the false negative samples and found new infection patterns that are not in our dataset. For example, we observed some remote loaders download malicious payloads and execute the same payloads for multiple times. Between two consecutive executions, they invoke `wget` to download more files. Finally, they removed all downloaded files. It is likely that the consecutively downloaded files may convey fragmented information that will be used by the malicious payloads. Among those false negative samples, we also found incomplete infections. For example, we found some remote loaders login to invoke `ls`, `cat`, or `uname`, and then logout without invoking other shell commands. This likely happens because those remote loaders check the system information and may find that the system is not their target.

In addition, we explored how many login sessions can happen simultaneously. Every time a new SSH or Telnet login session happens, a PID-keyed temporal window is created to track the commands invoked through that SSH or Telnet session. On average, we observed 3 login sessions throughout our 33-day online deployment, and the maximum number of live login sessions is 57. These observations convince us that isolating commands invoked from different login sessions does not require much memory space.

4.3.2 Generalization Evaluation of Our Modeling Approach. In this section, we evaluated how general our modeling approach is by testing the trained model against *unseen* samples. In particular, we split the shell scripts in our dataset randomly into a *training* set (80%) and a *testing* set (20%) and made sure samples in the *testing* set do not appear in the *training* set. During the correlation analysis and threshold determination (a.k.a. training process), only the *training* set is used, and the *testing* set always remains unseen to the detection model. During the testing process, the *testing* set is used to test the prediction performance of the trained model. Table 4 lists the details of all the sets we used in this evaluation. The *total_b* and *total_m* are sets of benign and infection scripts, respectively. The *training_b* and *training_m* are sets of benign and infection scripts used for training. The *testing_b* and *testing_m* are sets of benign and infection scripts used for testing.

Sample Sets	Volume	Description
training_b	7,483	benign scripts for weight and threshold assignment.
training_m	2,802	infection scripts for weight and threshold assignment.
testing_b	1,854	unseen benign scripts to test our model
testing_m	637	unseen infection scripts to test our model
Total_b	9,337	all benign scripts use for the evaluation
Total_m	3,439	all infection scripts used for the evaluation

Table 4: Details of each sample set used in our evaluation.

The testing results are presented in Fig. 9(a). Most of the infection scripts yield a high risk score close to 1, while the majority of benign scripts gain a low risk score close to 0. Using the threshold 0.50, which is determined during the training process, our model achieves 0.17% false positive rate (FPR) and 96.33% true positive rate (TPR), with an overall accuracy of 98.83% and F-score of 0.98. We manually investigated the scripts that were falsely classified by our model. We found that the false positive samples are caused by extensively downloading and executing binaries, which is similar to the behavior of remote infection scripts. The false negative samples are mainly caused by extensive usage of commands – such as `declare`, `continue`, and `break` – that are abstracted with the *programming* infection capability. This capability allows the script to implement complex programming logic, which is not fundamental to a remote infection.

Furthermore, we performed our model's receiver operating characteristic (ROC) analysis based on the testing sets. The ROC curve shows the relationship between FPR and TPR. It provides a means of reviewing the performance of a model in terms of the *trade-off* between FPR and TPR. In our case, the ROC curve shown in Fig. 9(b) implies that the TPR grows rapidly when the FPR still remains low. The area under the curve (AUC) for our model is 0.973, which indicates a good balance between false positives and false negatives.

4.3.3 Performance and Functional Impacts on Real Devices. In this experiment, we evaluated the performance impact and functional impact introduced by our infection detector on three popular IoT platforms, representing *low-end*, *mid-end*, and *high-end* IoT platforms:

- D-Link DCS-932L IP Camera (DCS-932L), equipped with MIPS 24KEc processor, 32MB SDRAM and 4MB flash memory, representing a *low-end*, resource-constrained platform.
- Raspberry Pi Compute Model 3 (CM3), equipped with ARM Cortex-A53 quad-core processor, 1GB DRAM and 8GB micro-SD card, representing a *mid-end*, generic multi-purpose platform.

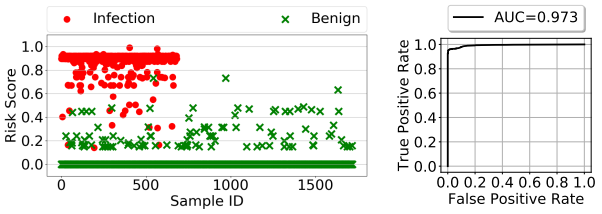

Figure 9: The testing results (a) and the ROC curve (b) of our infection detection model over the infection scripts that are unseen during the training process.

- SolidRun HummingBoard Edge (HBE), equipped with ARM Cortex-A9 quad-core processor, 2GB DRAM and 16GB micro-SD card, representing a *high-end*, powerful computing platform.

We installed OpenWrt on DCS-932L, Raspbian on CM3, and Debian on HBE. All those Linux versions are officially supported by the device vendors, respectively. In addition, we installed and ran different applications on the three devices to emulate a regular workload. We installed light sensor [32] and MJPGE streaming [41] packages on DCS-932L that allow it to capture and send frames continuously to a remote receiver. We installed the Motion package [42] on CM3 that allows it to work as a Digital Video Recorder (DVR). We configured HBE to make it an edge node that forwards network traffic. As a baseline, we first set up all the devices without the infection detector and observed the CPU and memory usage. As a comparison, we then installed the infection detector in the devices and observed the CPU and memory usage again.

Performance Impact. We quantified the CPU overhead using the CPU load average [22] within one minute. The CPU load average metric represents the total queue length of the active processes within a recent past time period (e.g., one minute in our experiment). If this number is greater than 1, it means at least one active process on average is in the queue waiting to use CPU. In contrast, if this number is less than 1, it means less than one active process on average is in the queue waiting to use the CPU. We employ the "CPU load average" value reported by `top` as the CPU load average for a device. Fig. 10(a) shows the results when the devices are running without human interaction and Fig. 10(b) shows the results when users login to the system and perform regular operations such as copying files, removing files, opening files, etc. In the first scenario, the infection detector introduces 3.73%, 3.61%, and 3.85% CPU load average for DCS-932L, CM3, and HBE, respectively. In the second scenario, the infection detector introduces 1.73%, 2.20%, and 2.54% CPU load average for DCS-932L, CM3, and HBE, respectively. In both scenarios, the infection detector introduces no more than 3.85% CPU load average, meaning that the length of the active process waiting queue only increases 3.85% on average due to the infection detector.

To measure the memory overhead, we added extra code in the kernel to trace how much memory is allocated to the process of the infection detector. Then, we compared that with free memory information obtained from `/proc/meminfo`. The results from different IoT platforms are quite similar, showing that our infection detector occupies 2.7MB of memory. This amount is only a small portion of the available memory, even for low-end IoT platforms like DCS-932L, which has around 12MB free memory in our experiments.

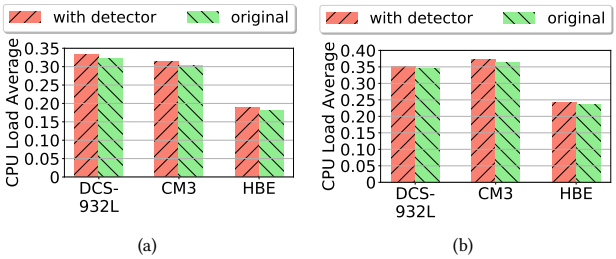

Figure 10: CPU load average of three types of IoT platforms w/o human interaction (a), and w/ human interaction (b).

Functional Impact. To evaluate the functional impact on real IoT devices, we exercised the three IoT devices as comprehensively as possible. For DCS-932L, we configured it as a surveillance camera with the `light-sensor`, `httpd`, and `jmpg-stream` packages installed. We operated the device through Telnet login and configured the services on the devices via shell commands. We then set up CM3 as a receiver of the stream from DCS-932L. We operated CM3 through SSH to check the logs and modify the configuration files. Especially, we updated the software source configuration, which relates to software upgrades. We also tested some other CM3 use cases that involve various shell commands under `/usr/bin`, `/usr/sbin`, `/bin`, and `/sbin`. Those use cases include programming, utilizing the general-purpose input/output (GPIO) pin to control the light on and off, and basic auditing. For HBE, we set it up as a network gateway. We operated the device through SSH to check the logs, modify the configuration files, and test HBE with the same shell commands as in CM3.

After operating these real IoT devices with real-world use cases for a week, we did not observe any false alerts raised by the infection detector. This observation implies that our infection detector introduces minimal functional impact on real IoT devices under real-world use cases.

4.3.4 Evasion Analysis. Evasions can happen in all aspects, from exploiting new zero-day vulnerabilities to switching to other attack surfaces. Specific to the infection detector, we limit the scope of our discussion with command-level evasions (a.k.a. mimicry attacks) where adversaries try to mimic benign operations while they actually perform infection activities such that our infection detector cannot capture the infections. Generally, three types of strategies may be used to alter the commands being executed to evade the infection detector.

Injecting Extra Commands. First, adversaries may execute *extra* commands other than those that are for the infection purpose. To address this issue, we relax our infection detector by skipping commands that trigger transitions with low weights in our detection model. We exhaustively try different values between 0 and 1 with

0.01 stepping distance and choose 0.42 for the infection detector to achieve a maximum detection accuracy on our training set. Note that, continuously triggering transitions with weights greater than the alert threshold is highly likely to trigger an alert.

We then conduct experiments to test the effectiveness of this countermeasure with the following evasion attack strategies. **Strategy 1**: randomly injecting different shell commands that trigger transitions with low weights (less than 0.42). **Strategy 2**: randomly injecting duplicated shell commands that trigger transitions with low weights. **Strategy 3**: randomly injecting different shell commands that trigger transitions with high weights (greater than 0.42).

The percentage of the injected commands grows from 10% to 100%, where 100% means injected, and original commands have the same number. Our testing results are shown in Fig 11. The detection accuracy of the infection detector decreases slightly as the number of extra commands increases. Specifically, our infection detector achieves 97.42%, 97.74%, and 97.58% with 10% extra commands injected for three strategies, respectively, and achieves 89.66%, 89.82%, and 87.72% with 100% extra commands injected for three strategies, respectively.

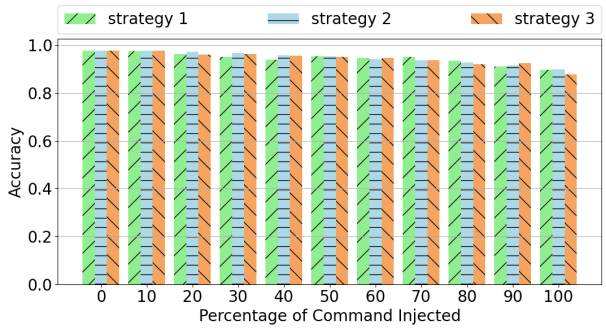

Figure 11: Evasion by injecting random commands.

Replacing with New Commands Adversaries may also replace existing shell commands with new shell commands that are not in our dataset to evade the infection detector. For example, replacing `wget` with `uget`, which is not used in any infection scripts in our dataset. To tackle this kind of evasion attacks, we can extend our infection detector by adding the new shell commands to our infection capabilities. Our infection detector is designed based on the *infection capabilities* rather than specific shell commands. Therefore, the infection detector can handle such replacement (without retraining), given that the infection capability of the new command is provided to our infection detector. To evaluate our countermeasures to this evasion attack, we choose at least one command for each infection capability. We then replace the chosen commands in all infection shell scripts with new commands that are not in our dataset. We then label those new commands with proper infection capabilities and run the detection test. The results show that our infection detector can still achieve a high detection accuracy of 97.58%.

Exploiting Variant Commands Other than injecting and replacing commands, adversaries may also employ elaborate variants of shell commands in a remote infection. For example, to delete a file, adversaries may use "`cat /dev/null > file`" instead of "`rm file`". The former only clears the content of a file without deleting it from the system. For some other examples, adversaries may use "`wget -o`" directly for downloading to obviate the "`cd`" command, and use "echo *" rather than "ls" to list a directory. However, sophisticated evasion attacks require extra effort to construct and may not always achieve the same remote infection objectives. Moreover, we have conducted experiments for the aforementioned three sophisticated evasion attacks. We simulate all the above attacks simultaneously overall testing scripts. As a result, our infection detector can still achieve a detection accuracy of 97.90%. This result implies that our infection detector is robust to certain variants of remote infections because the infection detector detects remote infections based on risk scores rather than command-level pattern matching.

5 DISCUSSION

Our remote infection model is trained only based on the shell command flows extracted from shell scripts in our dataset. We leave as our future work to utilize the log command sequences for our model training and extract the shell command flows from ELF files via static program analysis. Apart from static analysis, we also plan to dynamically execute ELF files as remote loaders to retrieve shell command flows, which requires the support of an infrastructure that involves multiple networked hosts (devices). Take the loader of Mirai as an example. It is running on a server, retrieving vulnerable passwords from another report server, and taking a remote IoT device as a target.

The detector developed in this work may not capture infections that were not seen in the past due to a static detection threshold. We plan to enhance our detector to adopt machine learning based algorithms and use adaptive thresholds that can be updated dynamically, which will make our detector more robust. In addition, our detector currently can only respond to remote infections by simply prohibiting commands from being executed by offending processes. We will also consider integrating our detector with dynamic mitigation approaches for remote infections, leveraging the output of the detector.

In our current testbed, we only seek the software IoT device as a solution to deploy IoT honeypots on a large scale due to its cost-efficiency. Although we have enhanced the fidelity of software IoT devices and enabled basic functions on them, they are still not able to fully emulate all features of hardware IoT devices. In addition, adversaries may leverage in-depth information (e.g., model-specific registers [19]) and advanced techniques (e.g., execution analysis [46]) to infer the identity of a virtual IoT device. The authors in [15] have already demonstrated the possibility of deploying hardware IoT devices as honeypots. We will also seek large-scale deployment of IoT honeypots with hardware IoT devices in the future.

The taxonomy of shell commands proposed in this work is one of the possible ways to abstract the shell commands with respect to remote infections. There may be other ways to label the shell commands with different infection capabilities other than what we proposed in this work. Nevertheless, as we have demonstrated with a preliminary infection detector, with our taxonomy, the infection

capabilities for each shell command can be used to detect infection in the wild with reasonably high accuracy.

6 RELATED WORK

There is a body of work devoted to the study of malware utilizing data mining and machine learning techniques [17, 28, 38, 39], which are OS-agnostic. However, those efforts limit analyzing the behaviors of malicious payload rather than the remote infection process. Another body of work devotes to the study of OS-specific malware. Those work either focuses on Windows-specific malware [24, 30, 35, 49] or Android-specific malware [1, 36, 59].

In the Linux area, there have been several research efforts made recently. Manos Antonakakis et al. [6] provide great details of the widely spread IoT malware, Mirai, including its infection mechanism. Another work [26] conducts measurement and analysis of Hajime botnet and finds that Hajime is a variant from Mirai but with new features that Mirai does not use. Although detailed infection techniques and measurements are provided in those works, they are limited to a specific IoT malware family. The research in [14] offers the first comprehensive analysis and measurement of malware for Linux platforms. This work unveils the challenges of Linux malware analysis, develops a toolset specific for Linux malware analysis, and documents the results of its analysis. Another recent research [5] studies the IoT malware life cycle. Authors in this work divide the IoT malware life cycle into five components – infection vector, payload, persistence, capabilities, and C&C infrastructure. Andrei Costin et al. in [13] provides a survey of IoT malware focusing on the vulnerabilities and detection approaches. Unfortunately, those research efforts fail to provide an in-depth measurement of the infection process of IoT malware. There is also a number of work employing IoT honeypots to collect and analyze malicious activities on IoT devices. Yin Minn Pa et al. [45] deploy a large-sale software IoT honeypot in IoTPot. Their work provides an insight into the compromise of IoT devices and divide the compromise into intrusion, infection, and monetization stages. HoneyCloud [15] investigates the fileless attacks on Linux-based IoT devices with the evidence collected from IoT honeypots. The above work also fails to dive deep into the remote infection process on Linux-based IoT devices. In contrast, our work conducts a comprehensive study of the remote infection process and share detailed measurement results specific to the remote infection on Linux-based IoT devices.

In addition to the work that is dedicated to the measurement of IoT malware, there is a body of work developing approaches to detect IoT malware. HADES-IoT [8] develops a host-based anomaly detection system for IoT devices by profiling the benign programs using SHA256 digest. CloudEyes [53] employs a cloud-based architecture for IoT malware detection, which stores signatures of malware on the cloud and implements a scanning agent in the IoT devices to collect signature fragments. Authors in [2] propose a low complexity signature-based method to detect malware for IoT devices. A classifier is developed in [52] to classify IoT malware based on image recognition techniques. All the above work focuses on modeling and classifying malicious payloads instead of remote infection processes on IoT devices, and thus can hardly unveil the malicious activities *early* in the infection stage. SHELLCORE [4] develops a deep learning model based on the term- and character-level features of shell commands. In our work, we model the infection process on the command level and provide a taxonomy of shell commands in terms of infection capability. Furthermore, we evaluate our approach with large-scale deployment in the wild. DeepPower [16] presents a detection approach for malware infection based on power side channels. The authors employ a deep-learning model trained with the power consumption of shell commands to imply malicious activities in the device. However, in DeepPower, the detection model trained for one type of device cannot be directly applied to detect malicious activities in other types of devices. In our work, we systematically investigate the remote infection process on Linux-based IoT devices. The model built based on our knowledge can be used to detect remote infections on different types of Linux-based IoT devices.

7 CONCLUSIONS

In this work, we have conducted an empirical study on a large-scale dataset of shell commands extracted from infection scripts, malicious ELF files, and IoT honeypot logs to gain a deep insight into the characteristics of IoT malware remote infection via shell commands. We shared our findings and provided a taxonomy of shell commands used in remote infections. Based on the knowledge gained from our study, we have developed a preliminary detector for remote infections to demonstrate the usefulness of our study. We have evaluated our detector on a large number of software IoT devices deployed across the world and three different real IoT platforms. The results indicate that our infection detector can achieve a high detection rate for remote infections in the wild and at the same time introduce very little performance overhead and functional impact on real IoT devices.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 2128107, 2128607, 2129164, 1700544 and 2037798, and the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award Title "S2OS: Enabling Infrastructure-Wide Programmable Security with SDI". It is also supported in part by ONR Grant No. N00014-20-1-2734. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, VMware and ONR.

REFERENCES

[1] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*. Springer, 86–103.

[2] Muhamed Fauzi Bin Abbas and Thambipillai Srikanthan. 2017. Low-complexity signature-based Malware detection for IoT devices. In *International Conference on Applications and Techniques in Information Security*. Springer, 181–189.

[3] Akamai. 2016. Akamai's State of the Internet / Security, Q3 2016 Report. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q3-2016-state-of-the-internet-security-report.pdf.

[4] Hisham Alasmary, Afsah Anwar, Ahmed Abusnaina, Abdulrahman Alabduljabbar, Mohammed Abuhamad, An Wang, Dae Hun Nyang, Amro Awad, and David Mohaisen. 2021. SHELLCORE: Automating Malicious IoT Software Detection Using Shell Commands Representation. *IEEE Internet of Things Journal* (2021).

[5] Omar Alrawi, Charles Lever, Kevin Valakuzhy, Kevin Snow, Fabian Monrose, Manos Antonakakis, et al. 2021. The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[6] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1093–1110.

[7] Bashlex 2021. Bashlex - Python parser for bash. https://github.com/idank/bashlex.

[8] Dominik Breitenbacher, Ivan Homoliak, Yan Lin Aung, Nils Ole Tippenhauer, and Yuval Elovici. 2019. HADES-IoT: A Practical Host-Based Anomaly Detection System for IoT Devices. In *ACM Asia Conference on Computer and Communications Security*. 479–484.

[9] Eric Brown. 2018. Linux Still Rules IoT, Says Survey, with Raspbian Leading the Way. https://circuitcellar.com/cc-blog/linux-still-rules-iot-says-survey-with-raspbian-leading-the-way/.

[10] Edmund K Burke and Yuri Bykov. 2017. The late acceptance hill-climbing heuristic. *European Journal of Operational Research* 258, 1 (2017), 70–78.

[11] Busybox man pages - user commands 2021. Busybox man pages - user commands. https://busybox.net/downloads/BusyBox.html.

[12] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.. In *NDSS*, Vol. 16. 1–16.

[13] Andrei Costin and Jonas Zaddach. 2018. Iot malware: Comprehensive survey, analysis framework and case studies. *BlackHat USA* (2018).

[14] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–175.

[15] Fan Dang, Zhenhua Li, Yunhao Liu, Ennan Zhai, Qi Alfred Chen, Tianyin Xu, Yan Chen, and Jingyu Yang. 2019. Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud. In *17th Annual International Conference on Mobile Systems, Applications, and Services*. 482–493.

[16] Fei Ding, Hongda Li, Feng Luo, Hongxin Hu, Long Cheng, Hai Xiao, and Rong Ge. 2020. DeepPower: Non-intrusive and Deep Learning-based Detection of IoT Malware Using Power Side Channels. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 33–46.

[17] Thomas Dube, Richard Raines, Gilbert Peterson, Kenneth Bauer, Michael Grimaila, and Steven Rogers. 2012. Malware target recognition via static heuristics. *Computers & Security* 31, 1 (2012), 137–147.

[18] Dyn. 2016. Dyn analysis summary of Friday October 21 attack. https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

[19] Peter Ferrie. 2007. Attacks on more virtual machine emulators. *Symantec Technology Exchange* 55 (2007).

[20] firmwalker 2017. firmwalker. https://github.com/danieluhricek/LiSa.

[21] firmware-mod-kit 2017. firmware-mod-kit. https://github.com/rampageX/firmware-mod-kit.

[22] Neil J. Gunther. 2010. UNIX Load Average Part 1: How It Works. https://www.helpsystems.com/resources/guides/unix-load-average-part-1-how-it-works.

[23] Michael Haag. 2013. Kaiten - Linux Backdoor. http://blog.michaelhaag.org/2013/12/kaiten-linux-backdoor.html.

[24] Danny Hendler, Shay Kels, and Amir Rubin. 2018. Detecting malicious PowerShell commands using deep neural networks. In *Asia Conference on Computer and Communications Security*. 187–197.

[25] Here Document 2021. Here Document. https://en.wikipedia.org/wiki/Here_document.

[26] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. 2019. Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet.. In *NDSS*.

[27] Ivan Homoliak, Martin Teknos, Martín Ochoa, Dominik Breitenbacher, Saeid Hosseini, and Petr Hanacek. 2018. Improving network intrusion detection classifiers by non-payload-based exploit-independent obfuscations: An adversarial approach. *arXiv preprint arXiv:1805.02684* (2018).

[28] Íñigo Íncer Romeo, Michael Theodorides, Sadia Afroz, and David Wagner. 2018. Adversarially robust malware detection using monotonic classification. In *ACM International Workshop on Security and Privacy Analytics*. 54–63.

[29] Rhena Inocencio. 2014. BASHLITE Affects Devices Running on BusyBox. http://blog.trendmicro.com/trendlabs-security-intelligence/bashlite-affects-devices-running-on-busybox/.

[30] Bo Li, Kevin Roundy, Chris Gates, and Yevgeniy Vorobeychik. 2017. Large-scale identification of malicious singleton files. In *ACM on Conference on Data and Application Security and Privacy*. 227–238.

[31] Libelium. 2014. Top 50 Internet of Things Application. http://www.libelium.com/resources/ top_50_iot_sensor_applications_ranking/.

[32] LightSensor-daemon 2021. LightSensor-daemon for OpenWrt. http://www.aboehler.at/hg/lightSensor-daemon.

[33] David Lindner. 2018. OWASP Internet Of Things Top 10 2018 Released. https://nvisium.com/blog/2019/01/02/internet-of-things-owasp-top-10-2018-released.html.

[34] Linux man pages - user commands 2021. Linux man pages - user commands. https://linux.die.net/man/1/.

[35] Robert Lyda and James Hamrock. 2007. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy* 5, 2 (2007), 40–45.

[36] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models. (2017).

[37] Vladimir Kuskov Mikhail Kuzin, Yaroslav Shmelev. 2018. New IoT-malware grew three-fold in H1 2018. https://securelist.com/new-trends-in-the-world-of-iot-threats/87991/.

[38] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. 2016. Reviewer integration and performance measurement for malware detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 122–141.

[39] Bradley Austin Miller. 2015. *Scalable platform for malicious content detection integrating machine learning and manual review*. Ph. D. Dissertation. UC Berkeley.

[40] Mirai Source code 2016. Leaked Mirai Source Code for Research/IoC Development Purposes. https://github.com/jgamblin/Mirai-Source-Code.

[41] MJPGStreamer 2021. MJPG Streamer for OpenWrt. https://openwrt.org/packages/pkgdata/mjpg-streamer.

[42] Motion 2021. The Motion program. https://motion-project.github.io/index.html.

[43] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Hossein Fereidooni, N Asokan, and Ahmad-Reza Sadeghi. 2019. DÏoT: A federated self-learning anomaly detection system for IoT. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 756–767.

[44] OpenWrt Project 2021. OpenWrt Porject. https://openwrt.org/.

[45] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. 2015. IoTPOT: analysing the rise of IoT compromises. In *9th USENIX Workshop on Offensive Technologies (WOOT'15)*.

[46] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. 2007. Detecting system emulators. In *International Conference on Information Security*. Springer, 1–18.

[47] Raspbian OS 2021. Raspberry Pi OS (previously called Raspbian). https://www.raspberrypi.org/downloads/raspberry-pi-os/.

[48] Rsyslog: rocket-fast system for log processing 2021. Rsyslog. https://en.wikipedia.org/wiki/Rsyslog.

[49] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. 2009. Pe-miner: Mining structural information to detect malicious executables in real-time. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 121–141.

[50] Miroslav Stampar. 2021. IPsum Threat Intelligence. https://github.com/stamparm/ipsum.

[51] Miroslav Stampar. 2021. Maltrail Malicious Traffic Detection System. https://github.com/stamparm/maltrail.

[52] Jiawei Su, Vargas Danilo Vasconcellos, Sanjiva Prasad, Sgandurra Daniele, Yaokai Feng, and Kouichi Sakurai. 2018. Lightweight classification of IoT malware based on image recognition. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. IEEE, 664–669.

[53] Hao Sun, Xiaofeng Wang, Rajkumar Buyya, and Jinshu Su. 2017. CloudEyes: Cloud-based malware detection with reversible sketch for resource-constrained internet of things IoT devices. *Software—Practice & Experience* 47, 3 (2017), 421–441.

[54] The networkx package 2021. NetworkX – Network Analysis Package in Python. https://networkx.org/.

[55] Pierre-Antoine Vervier and Yun Shen. 2018. Before toasters rise up: A view into the emerging iot threat landscape. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 556–576.

[56] 2021. Virustotal-free online virus, malware and url scanner. https://www.virustotal.com/en. (2021).

[57] VirusShare.com 2021. VirusShare.com. https://virusshare.com/.

[58] Wifatch source repository 2015. Linux.Wifatch. https://gitlab.com/rav7teif/linux.wifatch.

[59] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *ACM SIGSAC conference on computer and communications security*. 1105–1116.

[60] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*. 2361–2378.

Appendix A CDFS OF INFECTION AND BENIGN SCRIPTS

Fig. 12 shows the CDFs of risk scores of all benign scripts and all infection scripts in our training set.

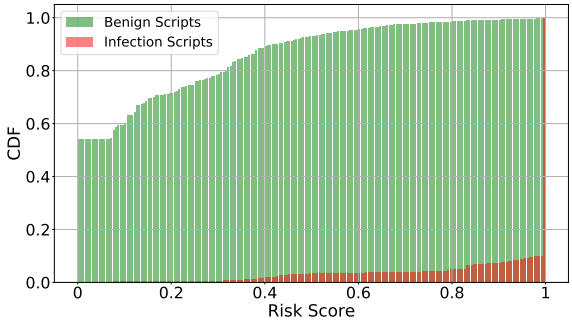

Figure 12: CDFs of risk scores of all benign scripts and all infection scripts.

Appendix B STATISTICS OF THREE SHELL COMMAND CATEGORIES

Table 5 lists the statistics of external, built-in, and hybrid shell commands found in our dataset.

		External	Built-in	Hybrid
Whole Dataset	All Commands			
Phases	Settlement	51.11%	48.89%	6.67%
	Environment Preparation	75.61%	24.39%	1.22%
	Payload Delivery	66.07%	33.93%	1.79%
	Payload Execution	67.14%	32.86%	1.43%
	Persistence & Covert	65.00%	35.00%	1.67%
Capabilities	Create and Write	66.67%	33.33%	33.33%
	Agnostic	83.33%	16.67%	8.33%
	Find Place	0.00%	100.00%	0.00%
	Cope File	88.89%	11.11%	0.00%
	Programming	5.88%	94.12%	0.00%
	Collect Information	100.00%	0.00%	0.00%
	Process Text	100.00%	0.00%	0.00%
	Network Probe	100.00%	0.00%	0.00%
	Manage Resource	0.00%	100.00%	0.00%
	Disable Security	100.00%	0.00%	0.00%
	Decode	100.00%	0.00%	0.00%
	Download	100.00%	0.00%	0.00%
	Decompress	100.00%	0.00%	0.00%
	Compile Code	100.00%	0.00%	0.00%
	Change Permission	83.33%	16.67%	0.00%
	Execute	88.89%	11.11%	0.00%
	Manage System	100.00%	0.00%	0.00%
	Manage Software	85.71%	14.29%	0.00%
	Implant Backdoor	100.00%	0.00%	0.00%
	Get Time	100.00%	0.00%	0.00%
	Change Time	100.00%	0.00%	0.00%
	Exclude Others	100.00%	0.00%	0.00%
	Kill Process	66.67%	33.33%	0.00%
	Remove History	50.00%	50.00%	0.00%

Table 5: Statistics of three categories of shell commands found in our dataset

Appendix C INFECTION CAPABILITIES EXPLOITED IN EACH INFECTION PHASE

Figure 13 demonstrates the what infection capabilities are exploited in each infection phase.

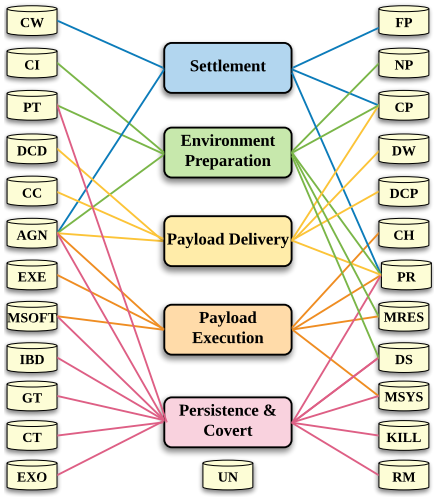

Figure 13: Infection capabilities exploited in each infection phase.

Appendix D INFECTION CAPABILITIES AND COMMANDS

A full list of the infection phases, capabilities, percentages, and corresponding shell commands is shown in Table 6

Phase	Capabilities	Percentage in each phase	Commands
Settlement	Create and Write	11.14%	echo* printf mkfifo open
	Agnostic	7.37%	sleep* yes lp find pwd md5sum clear fold kdialog logout wall
	Find Place	72.32%	cd*
	Cope File	8.42%	cp* tail head* mv* cat* read scp strings tee
	Programming	0.75%	expr test set unset declare local continue break unalias exit enable return let true trap readonly getopts
Environment Preparation	Collect Information	36.45%	uname* which* ls* ifconfig* lsof arp stat id* whoami* netstat* lsb_release ping* hostname* chkconfig df* file free* fuser getconf iptables-save nproc* pidof socklist uptime* who lspci* ps* du lastlog
	Cope File	29.29%	cp* tail head* mv* cat* read scp strings tee
	Process Text	4.15%	grep* awk* sed* cut tr* sort* egrep uniq* wc*
	Agnostic	25.77%	sleep* yes lp find pwd md5sum clear fold kdialog logout wall
	Network Probe	0.88%	nmap zmap dig
	Programming	2.61%	expr test set unset declare local continue break unalias exit enable return let true trap readonly getopts
	Manage Resource	0.83%	ulimit
	Disable Security	0.02%	accton ufw
Payload Delivery	Agnostic	15.31%	sleep* yes lp find pwd md5sum clear fold kdialog logout wall
	Decode	0.16%	base64*
	Cope File	17.49%	cp* tail head* mv* cat* read scp strings tee
	Download	65.00%	wget* tftp* curl* ftp* apt-get lynx git ftpget* mail
	Decompress	0.22%	tar gunzip gzip unzip
	Compile Code	0.27%	make gcc cc ldconfig
	Programming	1.55%	expr test set unset declare local continue break unalias exit enable return let true trap readonly getopts
Payload Execution	Agnostic	11.29%	sleep* yes lp find pwd md5sum clear fold kdialog logout wall
	Change Permission	28.35%	chmod* usermod chattr* umask chown chgrp
	Execute	53.44%	nohup* xargs crontab* perl sh* service nc sshpass bash* exec nice php python screen ssh busybox ./executable* env
	Manage System	3.28%	sysctl iptables* userdel su* mkisofs useradd defaults sudo reboot
	Programming	1.14%	expr test set unset declare local continue break unalias exit enable return let true trap readonly getopts
	Manage Resource	0.36%	ulimit
	Manage Software	2.13%	mktemp mkdir* export ln* apt-key dpkg yum
Persistence & Covert	Process Text	2.33%	grep* awk* sed* cut tr* sort* egrep uniq* wc*
	Agnostic	14.48%	sleep* yes lp find pwd md5sum clear fold kdialog logout wall
	Manage Software	2.73%	mktemp mkdir* export ln* apt-key dpkg yum
	Implant Backdoor	3.54%	httpd squid sshd*
	Get Time	0.15%	date
	Change Time	0.41%	touch*
	Exclude Others	0.70%	chpasswd iptables-restore passwd*
	Programming	1.47%	expr test set unset declare local continue break unalias exit enable return let true trap readonly getopts
	Disable Security	0.01%	accton ufw
	Kill Process	23.78%	pkill killall* kill*
	Remove History	50.40%	rm* history

Table 6: Statistics of 5 five infection phases, 25 infection capabilities, and all corresponding shell commands. (*) indicates the command appears in honeypot logs.