UNIVERSITY OF CALIFORNIA
RIVERSIDE

# Techniques for Detecting Intrusions

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Ali Davanian

September 2022

Dissertation Committee:

Prof. Michalis Faloutsos, Chairperson
Prof. Nael Abu-Ghazaleh
Prof. Mohsen Lesani
Prof. Chengyu Song

The Dissertation of Ali Davanian is approved:

---

---

---

---

Committee Chairperson

University of California, Riverside

## Acknowledgments

This thesis would not have been possible without the contribution and support of many great people. Firstly, I would like to thank my advisor Prof. Michalis Faloutsos who supports students not only technically and in their academic endeavour but also in their social life. I learned many things from Prof. Faloutsos both scientifically and socially, and I am grateful for all of his advice that certainly helped me become not only a better student but also a better person. This thesis might mark the end of my PHD but definitely not the end of my mentee-mentor relationship with Prof. Faloutsos.

Secondly, I would like to thank my committee members and collaborators. I was lucky to have this committee who provided me with valuable insights in the past five years. I would like to especially thanl Prof. Lesani who opened a new horizon to me when I attended his seminar course on software synthesis. Also, I would like to extend my gratitude to Prof. Martina Lindorfer who taught me a lot. She has a lovely character with an exemplary intelligence in academic research. Thank you Prof. Lindorfer for all you have taught me. There were many other people involved in this research that I would like to thank them all.

Finally, my journey in the last five years would not have been possible without the continuous support from my family and friends. I would like to thank Ahmad Darki, Nami Davoodzadeh, Joobin Gharibshah, Navid Gharavi, Amit Gupta, Leila Hashemi, Aria Hosseini, Nick Nickbakt, Ali Mohammadkhan and Hadi Zamani who have always been there when I needed them. I am also thankful to my girlfriend Sara Anbir who supported me despite her own problems. Last but not the least, I would like to sincerely express my gratitude to my mother Nasrin and my sister Mahsa who always unconditionally loved and

supported me. My mother has been my first mentor in life and I am always thankful of that. It goes unsaid that listing the names of all the individuals who supported me in the past five years would need more than 10 pages, and I apologize to all who have not been listed here.

To my mother and sister for all their supports.

# ABSTRACT OF THE DISSERTATION

Techniques for Detecting Intrusions

by

Ali Davanian

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2022
Prof. Michalis Faloutsos, Chairperson

Detecting intrusions is usually the first step in containing a security breach. In this dissertation, we focus on how we can understand and profile IoT malware behavior in order to find intrusions. Specifically, we want to develop a holistic view of how the malware behaves once installed in an IoT device, which will encompass: (a) how does the malware communicate with its Command and Control server, (b) how does the malware communication and infrastructure change over time, and (c) how does a network-affected data flows through the device (which is known as taint analysis).

In the first chapter of this thesis, we propose a novel solution for finding IP and port addresses of live Command and Control (CnC) servers of IoT botnets. Furthermore, we present a solution for detecting intrusions using only network level patterns given a malware binary. Our novelty lies in activating the malware binary and extracting patterns that can precisely indicate intrusions without the need of deep packet inspection.

In the second chapter, we conduct an extensive study of the behaviors of both bots and CnC servers over time. In more detail, we probe IP networks that show signs of CnC

hosting and search for CnC servers among neighbor hosts on a daily basis. This approach allows us to find CnC servers in an active manner; we find CnC servers even before their corresponding malware samples are collected by honeypots. In addition, the collected data provides insight on the network infrastructure of IoT botnets. Finally, we shed light on the non-CnC traffic of the IoT malware that mainly tries to spread the malware.

In the third chapter, we focus on taint analysis where the goal is to understand the flow of data within a system. In the Intrusion Detection context, the input data is the network traffic e.g. a command from the CnC. We discuss how dynamic taint analysis can be selectively applied for whole system analysis in intrusion detection systems, and show how this approach can speed up the system.

Overall, this thesis provides a fundamental step in understanding IoT malware behavior, which can lead to better defenses in terms of detecting and containing the associated damage.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Internet of Things (IoT) malware is the new battleground for security and it needs to be understood and analyzed with new approaches. The first major IoT malware based DDOS attack in 2016 disrupted the operations of many service providers such as Github, Twitter, Reddit, Netflix and Airbnb [83]. In 2020, the number of IoT malware attacks rose to 56.9 million, a 66% increase from the year before [28]. IoT devices range from a smart TV to an electric toothbrush. These devices have different requirements and operational capabilities, and can not afford expensive endpoint security solutions such as AntiViruses. Recent attacks of immense magnitude have clearly shown that the problem is not theoretical: security solutions need to be developed urgently.

In this dissertation, we focus on how we can understand and profile IoT malware behavior in order to find intrusions. This task can be broadly divided into two subtasks: (a) finding intrusions based on network behaviors observed from a network trace, or (b) finding intrusions based on the system level behaviors. In the first two chapters of this thesis, we address subtask (a). More specifically, we propose solutions for finding the live Command and Control (C2) servers and their network fingerprints. Identifying live C2

servers is a critical capability because then we can simply block traffic to/from these servers at the network perimeter and protect against the attacks. This is essentially important for IoT devices that can not afford to install endpoint security solutions such as an AntiVirus. In addition, robust (not regularly changing) network fingerprints allow intrusion detection systems to find C2 activity even if the C2 server quickly moves from an address to another. In the third chapter, we answer the question of how data flow analysis can be employed in intrusion detection systems designed based on subtask (b) efficiently? In particular, we provide a solution that can efficiently trace the flow of a command received from the C2 throughout the entire system.

The first chapter of this thesis presents the design and implementation of our holistic IoT malware analysis engine C2Miner[1]. As inputs, C2Miner, receives a set of IoT malware binaries, and an IP space that is suspected to host IoT malware C2 servers. C2Miner outputs the list of live C2 servers, and network fingerprints based on the observed C2 traffic. These addresses and the fingerprints can later be used for intrusion detection. C2Miner assumes no information is available about the input binaries and the IP space. More specifically, we neither know on what platform the binary can execute on nor what malware family it belongs to. Furthermore, we don't know about the malware network fingerprints and/or how the malware can communicate with its C2 server. Moreover, no information is available about the IP space a priori; we neither know what kind of C2 server it may host, nor any sample communicated traffic with that IP space. While these assumptions make the problem harder to solve, they will make the application more practical

[1]C2Miner is an extension of CnCHunter[35], and in this thesis we only focus on C2Miner.

as the analysis engine can function with zero a priori knowledge as long as the inputs are available.

There has been limited work addressing the problem in the way we have framed it here. We can classify related efforts in the following groups. First, several efforts study the behavior of IoT malware [11, 46, 50, 31, 32, 82]. None of this prior work addresses the problem of dissecting C2 communication protocol automatically and generally when the C2 server of a malware sample is not live that is a common case. Second, studying the network trace of malware has been the subject of much related work [37, 64, 48]. These efforts have visibility only to the sample network trace that they have access to. In other words, they can not find the C2 communications that have not been made yet to their sample network. Third group of work is the most relevant work to what we do where they do active probing [11, 42, 75, 97]. The main problem with this group of work is that they can not always handle the communication protocol when there is encryption. We discuss previous work in detail in their relevant chapters.

C2Miner workflow consists of four steps. First, C2Miner activates the binary in a sandbox environment that emulates IoT devices. This step includes statically finding the CPU architecture for the malware, and initializing an execution environment that emulates IoT devices. Next, C2Miner profiles the malware behavior by storing the system call traces and the network traffic. It then analyzes the network traffic and finds the C2 address that the malware tries to communicate with. In the next step, C2Miner Man-In-The-Middles (MitM) the C2 traffic to the addresses of choice (the input IP space). Finally, if the communication with the target addresses is successful, the communicated traffic is

dissected to a pattern that later can be used as a fingerprint for detection of the samples of the malware family.

We address significant challenges towards developing C2Miner. Firstly, we build our in-house sandbox environment for IoT malware analysis. The previous systems such as Cuckoo do not support IoT malware emulation. For successful activation of the IoT malware, not only the underlying CPU architectures should be supported but also the filesystem should emulate the common IoT devices filesystem. Second, finding the C2 address that the malware tries to communicate with from the generated traffic is not trivial; Megabytes of traffic are generated per second and often more than 99% of the traffic is non-C2. Third, weaponizing the malware so that it can be used for searching live C2 servers in an IP space is challenging. IoT malware families use different application layer (based on TCP/IP mode) protocols, and sometimes they use encryption. Henceforth, MitM should be done at the lower levels, and as far as we know we are the first to propose and develop a robust solution to MitM C2 traffic at the IP level. Finally, developing a generic fingerprinting approach that is resilient to the encryption and independent of the application layer is novel.

Our evaluation shows that C2Miner is accurate. We are able to activate 90% of the samples that we analyze. We have also a 90% precision in finding the sample's C2 address given that it is activated. In addition, our MitM based probing using malware samples can distinguish benign addresses from the C2 with a 86% F1 score (100% precision, and 75% recall). We further show that our fingerprints are unique, and can accurately indicate IoT malware C2 activity.

In the second chapter, we conduct a large-scale study of the live C2 servers. We deploy an automated system based on the C2Miner for a period of 1 year. We are interested in three main categories of IoT malware traffic: (a) its C2 server communication, (b) its proliferation techniques, and (c) its attacks as they are being launched. This comprehensive profile can help: (a) secure the network, through firewall rules, (b) harden the security of the device, and (c) provide intelligence of attacks as they launch.

We face a few challenges in conducting such a large scale study. Firstly, every task needs to be automated. The system needs to automatically collect samples for analysis, scan IP subnets and then extract communication information. Nevertheless, these are mainly engineering details. Secondly, we need to extract the communication payloads that aim to infect another device, a.k.a exploits, and then map them to the already disclosed vulnerabilities. Thirdly, we need to identify the commands from C2 servers that start a DDOS attack, and then profile the DDOS attack as it is being lunched.

In the third chapter of this thesis, we shift our focus from the network behaviors to system level behaviors. Chapter 3 tries to answer the question of how data flow analysis can be employed in intrusion detection systems efficiently? Given a network command, data flow analysis is interested in finding how the command flows through the system a.k.a taint analysis. This data flow analysis allows us to automatically find signatures that can indicate intrusions. A main barrier in doing so in Intrusion Detection Systems is the taint analysis overhead when performed at the system level. At minimum, this overhead for our analysis engine would be 6 times.

We propose an elastic whole-system dynamic taint analysis approach, and implement it in a prototype called DECAF++ [36]. Elastic whole-system dynamic taint analysis strives to perform taint analysis as least frequently as possible while maintaining the precision and accuracy. The idea is that the taint analysis does not need to be activated if the system is in a safe state; CPU registers do not contain a tainted value. Although this idea has been previously explored for process level taint analysis [84, 17, 55, 51] the challenges for application in system level taint analysis has not been addressed before. Furthermore, our solution is purely software based and correlates the performance degradation with the taint analysis load, a feature that we call elasticity. We believe this elasticity feature makes application of taint analysis in intrusion detection systems possible.

Overall, this thesis provides significantly novel approaches, tools and knowledge as to how IoT malware behaves. As a result, our work can be seen as a fundamental step in understanding IoT malware behavior, which can lead to better defenses in terms of detecting and containing the associated IoT malware damage.

# Chapter 2

# C2Miner: Tricking Malware Binaries into Revealing Live C2 Servers

Identifying Command and Control (C2) servers is a critical capability in the battle against botnets, and in particular against botnets targeting Internet of Things (IoT) devices. As the name suggests, C2 servers control their bots and orchestrate malicious activities, such as Denial of Service (DoS) attacks. Once the C2 servers of malware are known, we can mount a defense to contain its proliferation and damage. For example, the defense could include monitoring or blocking traffic to these destination addresses. This is an especially effective defense in the case of IoT devices, because they do not have enough computation power to have sophisticated on-device defenses like anti-virus (AV) software. In addition, we can identify infected devices within a network once we know the C2 addresses. Finally, Internet Service Providers (ISPs) and law enforcement can block and take down these C2 servers to disrupt the botnet [73, 63].

Given their crucial role in the operation of botnets, C2 domains and IPs have long been used as indicators of compromise (IoCs), and are part of different (commercial)

threat intelligence feeds. Nevertheless, the effectiveness of these feeds in terms of coverage, accuracy and timeliness is questionable [65, 18]. The issue for the domain-based feeds is more severe, and has been known for quite some time: for example a study in 2014 found quote "the union of 15 public blacklists includes less than 20% of malware domains" [61].

Finding C2 server addresses that are currently live is a challenging task. First, the current practice in industry for finding the C2 servers is based on manual effort and domain knowledge [102]. Even in academia, researchers rely on the patterns shared by industry experts for C2 address detection [61, 78]. This is because C2 server communications are using proprietary and increasingly sophisticated protocols, therefore automated reverse engineering of their application layer protocols is not straightforward. Second, malware C2 servers are short lived (usually less than three days), and in the case of IoT malware do not include fall-back mechanisms [91]. This means that by the time analysts find malicious binaries and reverse engineer them, botmasters have most likely moved their servers to new locations — essentially abandoning any existing binaries. This is because reviving an IoT botnet is relatively easy, and botmasters avoid maintaining them [91].

**Problem definition:** *How can we identify live C2 servers for a given IoT malware binary?* This is the question that lies at the heart of our work. The input is a malware binary, and a target IP:port (hereafter target for short) space of interest. The desired output is: (a) live C2 servers for that binary in the given IP:port space, and (b) traffic "fingerprints" of the C2 communication, which can be used to identify C2 servers in a network trace.

We make the overarching assumption of nearly *zero a priori knowledge*, which makes the problem harder but more relevant in practice. Specifically, we assume that we

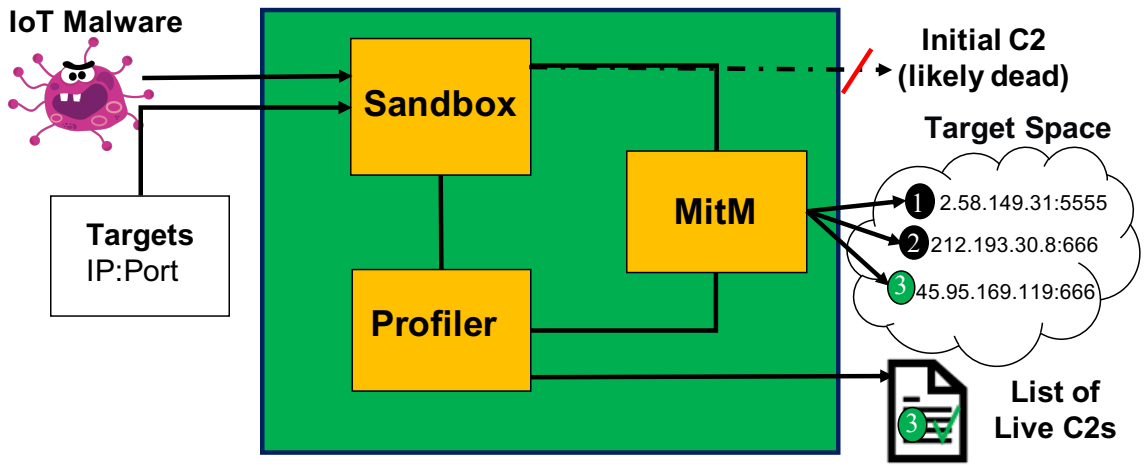


**Figure 2.1:** Overview of C2Miner and how it tricks malware binaries into revealing live C2 servers by activating them, and then redirecting the traffic to scan IP spaces of interest.

do not have any a priori knowledge about: (a) the binary (e.g., its family), (b) the targets within the given IP:port space, and (c) actual bot traffic patterns, i.e., no existing traffic traces. Here, the IP:port space of interest is part of the problem input and it could reflect the space that an ISP or an enterprise owns or is interested in.

**State of the art:** There has been limited work addressing the problem in the way we have framed here. We can classify related efforts in the following groups. First, several efforts analyze the behavior of IoT malware either at the operating system or network level [11, 50, 46, 31, 32, 68, 82, 94, 91]. These approaches develop behavior models but do not attempt to find live C2 servers, especially servers that are not contacted directly by the malware. Second, several efforts analyze network traces and develop methods to find botnet traffic [48, 47, 37, 64]. These are passive efforts and different from our active probing goal. Third, several approaches probe the Internet in pursuit of finding live targets [42, 11, 75, 97], but they follow a different approach to ours. They attempt to imitate a bot by replaying traffic or reverse engineering the protocol, and as a result they cannot handle

sophisticated and complex communication protocols, as we explained earlier. Finally, the most relevant related work tries to engage with the malware and reveal its alternative C2 servers [78, 74, 43]. Unfortunately, these methods are not as applicable to IoT botnets, which have a disposable nature and rarely use alternative C2 addresses [91]. We discuss previous work in detail in section 4.1.

**Contributions:** We propose C2Miner, a systematic probing approach for discovering live C2 server for given a malware binary with zero a priori knowledge. The novelty of our approach is that we fool the malware twice: (a) we get the malware to activate in a sandbox and start searching for its C2 server, and (b) we perform a Monkey-in-The-Middle (MitM) attack on the C2-bound traffic, which we redirect to C2 candidates within the target space of our choice. To substantiate this approach, we develop novel algorithmic solutions to: (a) disambiguate the C2-bound traffic initiated by the binary, (b) determine if a target IP is indeed a C2 server, and (c) fingerprint and cluster C2 communications, which is necessary for exploration at scale. In particular, we introduce a grammar-based method to model and fingerprint the C2 communication. We provide our code as a public GitHub repository with an open-source license (withheld for anonymity).

To summarize, the key contribution of our work is proposing and demonstrating the feasibility of our MitM-enabled active probing approach that "weaponizes" the malware binary turning it into a "spy". By contrast, most previous efforts that redirect malware traffic "sink" it into a fake server [78, 74] and do not use it for actively probing real potential candidates. In some cases [43], the malware is used to reveal its referred C2 server, but without combining it with the MitM approach that we use here.

We evaluate our approach by building a working prototype and using 1447 binaries to explore a space of 3M IP:port combinations. Our key results can be summarized in the following points:

- **We disambiguate C2-bound traffic accurately.** We propose an algorithm that can distinguish C2-bound traffic from other traffic (i.e., other malware activity, such as scanning and exploitation) with 92% precision.
- **We determine C2 servers accurately.** We develop an approach that can identify C2 servers among benign web servers with an F1 score of 86%. To achieve this, we introduce a grammar-based approach for characterizing the communication between bots and C2 servers.
- **We show that we can utilize old binaries to discover current live servers**. We provide initial indications of the promise of our approach by activating two six-month-old binaries to identify six live servers within a six day of probing 1,536 IP addresses on 12 ports. In addition, we find that 84% of pairs of malware in our dataset can interchangeably talk to the same C2 server.

**Open-source and data sharing.** We provide our code as a public GitHub repository with an open-source license (withheld for anonymity). We are committed to enabling and collaborating with the security community in the search of powerful tools against cyber-crime. In addition, we make our datasets and ground truth available to facilitate further research.

**Our work in perspective:** Our goal is to establish C2Miner as an impactful capability in practice. We ambitiously expect C2Miner to be deployed at scale and widely

by security companies and ISPs, although the purpose of this work is to show the feasibility and value of our approach. To ensure effective deployment at scale, several challenges need to be addressed that discuss in section 2.5.

**Ethical considerations:** We discuss our efforts to adhere to best practices and meet ethical standards in subsection 2.1.5.

## 2.1 Design & Implementation

We start by describing the architecture of C2Miner, which is shown in Figure 2.1. Recall that C2Miner takes as an input one or more malware binaries and a target IP:port space to explore. For ease of presentation, we discuss the operation of C2Miner in the following four steps: (a) activating the binary, (b) disambiguating C2-bound traffic from other traffic, (c) launching a MitM redirection of the C2-bound traffic to a target address, and (d) determining if the responding target is a C2 server. The *sandbox component* is in charge of the binary activation. The *profiler component* oversees the disambiguation and the determination functions above. The *MitM component* implements the redirection of the traffic to the target IP address. In the following subsections, we describe how each task is accomplished.

Note that, a significant contribution of our work is our grammar-based fingerprinting capability. In our context, fingerprinting refers to the detection of a remote network service based on its network footprint. We discuss this in section 2.2 in order to streamline the flow of the paper. We use this capability in two ways. First, it is used by the profiler for determining the existence of a C2 server. Second, it is used to cluster binaries based

on their type of interaction in order to optimize a large-scale deployment of C2Miner as we discuss in section 2.4.

### 2.1.1 Binary Activation

We need to overcome three challenges in order to activate an IoT malware binary. By activation, we mean driving the malware process execution to a point where it generates network traffic. First, the binary must be executed on the corresponding CPU architecture. We statically analyze the binary in order to find the right architecture for execution. If the binary is packed, we unpack it using well-known packers (like UPX). If we can not determine the CPU architecture after unpacking statically, we iteratively execute the binary on every CPU architecture that we support until the binary is activated, or we exhaust all options.

Second, we need to identify a virtual environment. Executing a malware on actual IoT devices is not feasible: there are many IoT devices and given our zero knowledge assumption, we would not know which device it targets. In contrast, a virtualized environment allows us to instrument the execution, and efficiently analyze the malware. However, failures could happen because sometimes an instruction is not supported by the virtualization engine. Our *sandbox* emulates the malware execution using QEMU [13], and as we report in subsection 2.3.3, QEMU performs very well.

Third, the execution platform should be configured properly for the malware to activate. By configuration, we mean the operating system and the filesystem. For instance, if the malware looks for a specific file, the absence of that file would result in the early termination of the execution. We use RiotMan [32], an open source tool, for this purpose, which provides us the appropriate configuration. We report our activation rate in subsection 2.3.3.

We highlight some implementation details of the binary activation process. C2Miner automatically copies the malware executable to the filesystem in the start-up script directory, and starts the input malware emulation. The profiler component stores the logs for analysis by other components. It logs network traffic, and the system call traces. The former provides us the data for C2 communication analysis, while the latter allows us verify the correctness of the emulation. For logging, we use *strace* and the QEMU traffic record functionality. Our emulator executes the malware with the *strace* logging enabled. For network traffic collection, we enable QEMU network bridging functionality, and record the traffic in pcap format. Finally, to communicate with the candidate addresses, the guest (the virtual machine that the malware runs on) needs an active Internet connection, so we activate IP forwarding on the Linux host. We also activate the ARP proxy configuration and NAT the traffic to the outside world.

### 2.1.2 Traffic Disambiguation

Traffic disambiguation aims to distinguish between the C2 and non-C2 traffic that the malware generates. The C2 traffic is only a small percentage of all the traffic that the malware generates. Worm-like malware like IoT malware tries to infect other devices, and hence it generates scanning and exploitation traffic. In addition, there might be random traffic, for instance, for checking Internet connectivity on the infected device. The profiler of C2Miner finds the C2 address (IP:port or DNS) that the analyzed malware tries to communicate with among all the traffic that it generates.

To identify C2 traffic, we developed the *Disambiguate-C2-Traffic* algorithm illustrated in Algorithm 1 (and implemented it using Python's pyshark library [56]) that

**Algorithm 1** Disambiguate-C2-Traffic

**Input:** Packets
**Output:** Scores ▷ for IP:ports

1: $TargetStats \leftarrow \{\}$ ▷ A hashtable tracking the number of connections to each target (ip:port or DNS).
2: $Ports \leftarrow \{\}$ ▷ A hashtable tracking the number of times a destination port is seen.
3: $Scores \leftarrow []$ ▷ A list of targets with their C2 likelihood score.
4: **for each** $pkt \in Packets$ **do**
5: **if** $Approved(pkt) == TRUE$ **then**
6: $target \leftarrow Get\_Target(pkt)$
7: $Update\_Target(target, TargetStats)$
8: $Update\_Ports(target, Ports)$
9: **for each** $target \in TargetStats$ **do**
10: **if** $is\_DNS(target)$ **and not** $White\_list(target)$ **then**
11: $Scores[target] \leftarrow Calc\_DNS\_Score(target)$
12: **if** $is\_IP(target)$ **then**
13: $Scores[target] \leftarrow Calc\_IP\_Score(target, Ports)$
14: $Sort\_Desc(Scores)$
15: **return** $Scores$

analyzes the generated traffic from the guest. The algorithm analyzes every target and assigns a score that shows the likelihood of a target being a C2 server. We use the term "target" to refer to either an IP:port tuple or a DNS address. The algorithm consists of two main parts: (a) quantifying the communication activity, and (b) assigning a likelihood score to the targets.

**Part 1. Quantifying activity.** As a first step (lines 4-8), we analyze each packet and count the number of times targets (IP:PORT or DNS) and ports are contacted. We assume the malware uses TCP for C2 communication, which seems to be the preferred protocol [74], although we note that extending our approach to UDP is a matter of engineering effort. We filter out the traffic from unrelated protocols: ICMP, DHCP, ARP and NTP at

line 5. Although there are records of using ICMP and NTP in covert channels, we do not find such instances in IoT malware and we ignore it for now.

For IP:port targets, we count the number of packets to those targets. Our insight here is that a binary will exchange a large number of packets with its C2 server. We anticipate that even if the C2 is not alive, the malware will most likely attempt to connect multiple times.

For DNS-based targets, we consider two cases. If the DNS resolution succeeds, then we focus on the resolved IP:port tuple. If it fails, we count the number of times the DNS queries fail. This is based on the observation that a blocked C2 DNS address would not resolve, and hence there will not be a connection to an IP address.

**Part 2. Calculating the C2 likelihood score.** Having completed the first part, we can calculate the likelihood score for each target (lines 9 to 13).

First, for DNS-based addresses, we check the reputation of the domain. To reduce false positives, we eliminate well-known domains by using the top X (X=1,000 in our case, but is configurable) number of domains based on the Alexa ranking. Note that we check the reputation of the entire DNS address, and not only the second level domain (e.g., Microsoft's reputation and ranking is different from its subdomains). This means that a cloud-based C2 address (e.g., hosted on Microsoft or Amazon) would not have necessarily equally high reputation. If the DNS address passes the reputation check, the score is the number of times it was queried.

Second, for IP:port-based targets, we assign a score that considers the activity for both the target and the port by relying on two insights. The first insight is that a bot will

have regular communication with the C2 server, so the higher the number of packets sent to an IP:port pair, the higher the likelihood it is a C2 server. The second insight is that a port number used for the C2 server is different from that used for other bot traffic, such as proliferation (scanning and attacking other devices). In this case, the same port will be used across many different IP addresses. So the higher the number of IP addresses that are being contacted at a specific port, the lower the likelihood that a communication at that port is towards a C2 server. These two insights can be quantified in many different ways. Here, we opted to start with the most straightforward way that does not require any additional parameters such as weights. We calculate the score of an IP:port pair as the number of packets to that IP:port divided by the number of times the port was used. This formula works well in practice as we show in subsection 2.3.3.

### 2.1.3 MitM-enabled Probing

The task of the MitM component is to redirect the malware C2 traffic to candidate targets which are given as inputs. The traffic redirection results in traffic exchange between the malware and the target that later can be used to determine whether the target is a C2 server. Note that this component assumes that Algorithm 1 has already identified the C2-bound traffic. We discuss below how the MitM component handles IP:port-based and DNS-based targets.

**a. IP:port-based targets.** Here, we want to replace the IP and port of the C2 server that the malware wants to reach with that of the candidate server. In our case, implementing this functionality is non-trivial: as using the readily available NAT functions at the network perimeter would not be sufficient, we moved the address manipulation to the

guest level. We used the Linux iptables, which we adapted to work in our case, as following. In the NAT mode, iptables allows altering the packets as soon as they come in and/or as they go out from the network proxy. In neither of these cases, redirecting traffic to another IP address is allowed on iptables installed at network perimeters. That said, iptables can change a destination IP and port address if the traffic originates from the proxy itself. For this reason, we rely on the guest's (infected machine) iptables to provide the redirection. Interested readers can refer to our GitHub repository for further technical details.

**b. DNS-based targets.** For DNS-based addresses, we take a different approach. Operating systems use different DNS resolution methods. Linux starts by looking up the DNS address in a host file that maps DNS names to IP addresses and uses other methods only if this method fails (although the order can be modified). We take advantage of this process, and modify the host file (/etc/hosts) of the guest machine that would map DNS addresses to IPs. We add an entry that maps the C2 DNS address to the candidate address, similar to the IP case, which we look up. Doing so, the traffic is redirected to the candidate address.

### 2.1.4 C2 Determination

One of the profiler component's tasks is to determine whether a target is a C2 server based on its traffic exchange with the malware. This problem can be solved if we have a priori knowledge about the application protocol used by the malware, however, we assume such knowledge is not available. If traffic samples of communication with the C2 server is available, some knowledge can be drawn through the analysis of the traffic and the problem can be solved differently.

We propose two methods: (a) SYN-DATA-aware, and (b) Fingerprinting-aware. In the next section, we present a fingerprinting method that, among other applications, can determine whether the target is a C2 server under the assumptions of having C2 traffic traces. We compare the two methods' accuracy in subsection 2.3.4.

**The SYN-DATA-aware method.** The solution that we present here is independent of the application layer protocol used by the malware and works even in the case of encryption at the application layer. Our solution is based on the insight that if a target is a C2 server, it should engage with the malware in a "meaningful" way. Establishing what communication is meaningful is a key task of the profiler component. A meaningful or successful connection could imply different behaviors for different application layer protocols. Assessing success based on the network level features derived from the transportation layer satisfies the assumption that no a priori knowledge about the application layer protocol is available.

Determining meaningful communication at the transport layer is a challenging task. On the TCP transport layer, a successful connection completes the handshake. However, a successful handshake does not always indicate a successful connection (at the application level). After the handshake, endpoints might immediately close the connection by sending RST/FIN flags or even by silently not responding back. Alternatively, an endpoint might respond with a packet informing the target about an error at the application level. For instance, in response to an invalid HTTP request, the server might respond with a "400 Bad Request" error code. It should be noted that these are all possible scenarios since

we redirect a malware sample's C2 traffic to a candidate that might or might not be a C2 server.

We reduce the problem of determining whether the target is a C2 server to whether the communication succeeds or fails by assessing transport layer features. If the communication fails, regardless of the reason for the failure, we find that malware insists on retrying to connect to the server. To illustrate this, consider an analogy to the human communication. If two individuals do not understand each other's language, they restart the conversation and repeat themselves. Similarly, if a candidate address listens to our contacted port, but it does not speak our malware application layer protocol, it restarts the connection. We can identify this behavior by looking at the number of times the SYN flag is set for a particular candidate. This approach is error prone in case malware constantly closes the connection to a port and re-opens it. However, such cases seem rather rare, as we discuss in subsection 2.3.4.

If the communication is successful based on the number of SYN flags, we check for exchange of data between the malware and the candidate to further assess the communication success. In general, the malware does more than completing the handshake; it will send data to the C2 server. This could be a simple "PING" command, but still it is a payload for the TCP layer. In response, the server also does more than acknowledging the packet; again, this might be only a few signature bytes, although in some cases the C2 server might not respond with data. Regardless, we find that looking for exchange of data increases the precision because false positives are rare (see subsection 2.3.4).

In summary, for C2 determination, we check whether the number of times the SYN flag is set is lower than a threshold (=1 by default), and if there is exchange of data packets between the malware and the C2 server. Interested readers can see our GitHub repository for further technical details.

### 2.1.5 Ethical Considerations

We adhere to an ethical code of conduct and best practices for executing malware and interacting with live servers [86]. First, we do not violate anyone's privacy, as we never deal with any personally identifiable information. We only measure properties of devices (publicly accessible servers) and the services that run on them. Second, our measurement study is light-weight and we does not increase in any significant way the load on servers or the traffic on the network. Third, we take all possible measures to limit any potential harm. We only let C2 traffic communicate with real servers, and we filter out traffic that goes to any destination other than the C2 server. In addition, C2 traffic initiated by the bot towards the server is focused on establishing a communication, so it is not harmful. Fourth, even as the malware running in our sandbox temporarily joins a botnet, we are vigilant and have filters in place to recognize: (a) C2 commands that could instigate an attack, and (b) outgoing filters to block any suspicious or potentially harmful traffic. More specifically, we use SNORT IDS to detect and prevent malicious traffic from leaving our network. Furthermore, we have techniques in place to contain malicious traffic in each of our experiments:

**a. Detection of C2s:** Using Algorithm 1, which we use to detect C2s, is done in isolation. Thus, this analysis does not interact with the Internet, as we "fake" connections to

the malware in the sandbox. For sophisticated binaries that check for Internet connectivity, we deploy InetSim to simulate services like DNS and http.

**b. Observing C2 traffic:**: Having identified the signatures of C2 communication in the previous step, we filter out any communication of the malware with the outside except connections to the C2 server. We record the C2 traffic, reverse engineer it, and do not allow malware to talk to any other target (except the C2).

**c. Probing IP subnets:** We only allow harmless C2 communications like "Call-Home" messages to interact with potential C2 servers. We have manually analyzed sample traffic traces and we have not found any cases of non-C2 communications. Our target subnets were small /24 subnets with a history of malicious activity. We do not send probes if the host does not listen on a port. On live ports, we filter out hosts that present a well-known banner (such as Apache or Nginx).

## 2.2 Grammar-based Fingerprinting

How can we profile the bot-to-C2 server interaction at the network level? This is the question that we are trying to answer in this section. We want to find communication patterns that are uniquely indicative of C2 communications, and at the same time robust to variations and noise. To do so, we propose to model the network flow as a dialogue and model it using a grammar. At the network level, if we look at the dialogue, we can see the following features that depend on the applications (vs. the operating system):

- *The number of send/receive pairs between the endpoints.* Similar to a dialogue, the communication is in the form of request/response pairs. For instance, just like "Hi" answers to a "Hi," a "PING" or a "PONG" is a response to a "PING."

- *The order within the send/receive pairs.* Is it the client who initiated the exchanged pair or the server? Please note that this is independent of who first initiated the flow. The order within each pair can change at any time. A dialogue can go quiet, and each of the participants can start a new send/receive pair.

- *The termination control flags (if any).* This helps us better understand how the connection terminated. Was it because the connection was interrupted by the network (or operating system)? Was there an error etc.?

At a high level, we summarize the flow by capturing: (a) its beginning and end, (b) its adherence to TCP flow rules, and (c) properties of its packets. We design a grammar $G$ that models the raw traffic to a *"phrase in this language"* based on the above observations.

Formally, we define $G = (N, \Sigma, P, Flow)$, where $N = \{Flow, FlowBody, DataPacket, ControlPacket, Flags, Sender, Attributes\}$ are the non-terminals symbols, and $\Sigma = \{handshake, client, server, ack, synfin, rst, attr^*\}$ are the terminals symbols (we explain $attr^*$ below). Note that $Flow$ is the start symbol. The production rules set $P$ consists of the following rules:

$$Flow \longrightarrow handshake\ FlowBody \tag{2.1}$$

$$FlowBody \longrightarrow DataPacket\ FlowBody \tag{2.2}$$

$$FlowBody \longrightarrow ControlPacket\ FlowBody \tag{2.3}$$

$$FlowBody \longrightarrow \epsilon \tag{2.4}$$

$$DataPacket \longrightarrow Sender"\_"Attributes \tag{2.5}$$

$$ControlPacket \longrightarrow Sender"\_"Flags \tag{2.6}$$

$$Sender \longrightarrow client|server \tag{2.7}$$

$$Flags \longrightarrow Flags\ Flags \tag{2.8}$$

$$Flags \longrightarrow ACK|SYN|RST|FIN \tag{2.9}$$

$$Flags \longrightarrow \epsilon \tag{2.10}$$

$$Attributes \longrightarrow attr* \tag{2.11}$$

**a. Summarizing a flow.** Any flow starts with the start symbol $Flow$ and expects to "see" a TCP handshake captured by our terminal symbol "handshake". This symbol corresponds to the three TCP packets: the client sends a packet with the $SYN$ flag, the server responds with $SYN$ and $ACK$ flags and finally client sending an $ACK$. The rest of the flow consists of control or data packets from the endpoints denoted by "$Sender$" non-terminal and the *client* and *server* terminals. In our malware analysis, *client* is the malware and the *server* is the C2 server. Please note that sometimes for better readability (and to save space), we use the symbol "—" (corresponding to logical or) to present two rules in one line.

**b. Summarizing each packet.** Our grammar provides the ability to profile each packet in the flow using the symbol Attribute and $attr*$. Defining what are the right properties of packets can adapt to the sophistication of the malware. For example, the attributes can be a vector of values that could include packet size, byte entropy, or the appearance of a string in the payload. The two latter attributes require deep packet inspection.

Our selection of attributes was driven by two considerations, we wanted to: (a) avoid deep packet inspection to increase its practicality, and (b) show the promise of the method. The former consideration suggests that only network headers are available. From a deployment point of view, this allows network devices to quickly process the packets, and also allows data sharing because of alleviated privacy issues. Another advantage is that packet headers are not affected by encryption at the application layer.

Consequently, we define $attr*$ in (11) to be simply the length of the packet. In section 4.5, we show that this attribute works sufficiently well with our dataset and the current sophistication level (or lack thereof) of the malware. However, in the future, we will consider additional attributes.

The best way to understand the grammar is through the examples illustrated in Figure 2.2. We use orange for control packets, yellow for data packets from the client and green for the server data packets. In these examples, "H" stands for the TCP handshake. The first example shows that after the handshake, the client (the malware) sends 11 bytes of data to which the server responds with 4 bytes. At the end, the communication terminates with 1 byte of data from the client. In the second example, the sever sends a packet with

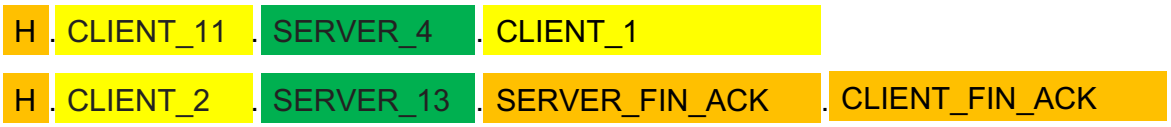


**Figure 2.2:** Two example phrases in our language: H is the TCP handshake, afterwards the client and server exchange packets of $x$ bytes (CLIENT_$X$ and SERVER_$X$, respectively).

the FIN flag set at the end, and the client acknowledges this and also sends a packet with the FIN flag set.

**c. Transforming flows into a "string" of the grammar.** We take as input a traffic pcap file, and our goal is to transform each network flow between the malware and a destination into a string in our grammar. Note that the malware is always one of the communicating entities.

The process still hides several subtleties that we discuss here briefly. First, the traffic does not need to be restricted to a single flow, and it might contain multiple flows to the C2 server. This is because the traffic might be captured as the result of executing the malware several times, or because the malware opens and closes its connection to the C2 periodically. Second, we remove erroneous packets from our trace, i.e., packets that are not "part of the dialogue." In practice, these are packets that are not acknowledged by the other endpoint or re-transmissions of the same packet. This could be in the handshake process, or after the connection establishment. Finally, based on the features we mentioned earlier, we generate a string in our grammar from the remaining packets. At the end of our transformation phase, we have one string per flow between the malware and each destination of interest.

**d. Comparing strings of the grammar.** Given two strings in our grammar, we can quantify how similar they are using a distance function. In fact, to increase the robustness of our approach, we use clusters of strings, which we discuss below. A cluster of similar fingerprint strings based on a distance function indicate recurring patterns.

**e. Clustering communication patterns.** Clustering the transformed traffic in form of strings in our language can help us find common patterns, which we refer to as $fingerprints$. These fingerprints can be used in several applications beyond the scope of this paper. Here we use it to: (a) determine if a target is a C2 server, and (b) optimize the deployment of C2Miner, as discussed in section 2.4. Our goal is to select a representative subset of binaries for probing a target space, such that we discover the same C2 servers as if we had used all the binaries. In essence, we can find malware clusters based on the similarity of their communication patterns.

For a given set of binaries $N$, we want to compare the similarity of their flows to their respective C2 servers. Formally, the input to the clustering algorithm is a set $S = \{S_1, S_2, S_3, ..., S_N\}$ where each $S_i$ is the set of network flows for binary $i$ to its C2 server represented by strings $f_i$ in our grammar, $S_i = \{f_1, f_2, f_3, ..., f_P\}$.

We use hierarchical clustering which gives us the ability to study the clusters at different levels of granularity. Here, we use the *hierarchical k-means* algorithm, which seems to work well for our applications. For the distance function, we use the Jaccard distance based on the output of the Longest Common Sequence (LCS) algorithm of two flows $f_i$ and $f_j$. $LCS(f_i, f_j)$ will be used to calculate the intersection length. In order to calculate the union size, we define $Length(f_i)$ as the total number of sender chunks in our grammar.

Since a sample's communication with the C2 could extend to multiple flows, the formula below accounts for these cases:

$$I(S_1, S_2) = \sum_{i=1}^{|S_1|} \sum_{j=1}^{|S_2|} LCS(f_i, f_j) \tag{2.12}$$

$$L(S_r) = \sum_{i=1}^{|S_r|} Length(f_i), \quad r = 1, ..., N \tag{2.13}$$

$$Jaccard(S_1, S_2) = \frac{I(S_1, S_2)}{L(S_1) + L(S_2) - I(S_1, S_2)} \tag{2.14}$$

**Usage and practical considerations.** In subsection 2.3.5, we discuss the choice of parameter $k$ and the seeds for the k-means algorithm. We also discuss how homogeneous the clusters are in practice. In subsection 2.3.4, we use our grammar-based method to fingerprint the communication behavior between the malware and likely C2 servers and we compare these string with that of known C2 servers. There, we report a fingerprint match under the condition that LCS returns a string that contains exchange of data from both endpoints.

## 2.3 Evaluation

In this section, we first describe our preliminary experiments with 10 IoT malware families characterize their communication protocols. We then evaluate our approach by answering the following questions:

**(Q1)** How accurately can we disambiguate C2-bound traffic from other malware traffic? (see subsection 2.3.3)

**(Q2)** How accurately can we determine that a probed target is indeed an active C2 server? (see subsection 2.3.4)

| Type | Breakdown | LOC |
|---|---|---|
| Language | Shell | 636 |
| | Python | 2,897 |
| Components | Sandbox | 1,239 |
| | MitM/Probing | 553 |
| | Profiler | 1,231 |
| | Other | 510 |

**Table 2.1:** LOC breakdown of the C2Miner implementation.

**(Q3)** Can our grammar-based fingerprinting help distinguish different families of malware? (see subsection 2.3.5)

**(Q4)** What is the likelihood that two binaries from the same malware family speak the same dialect, i.e., can communicate with the same C2 server? (see subsection 2.3.6)

This area of research is notorious for having limited to non-existent benchmarks and ground truth, despite several studies in this direction [94, 91]. Thus, we are left to create our own ground truth, which we will make available to the community. We start by discussing our general data collection strategy, and then detail the creation of ground truth in the respective section for each question separately.

**Implementation and experimental setup.** We implemented C2Miner in roughly 3,500 lines of code (LOC) in Python and Linux Shell scripts. More details about the effort breakdown can be found on Table 2.1. For the evaluation we use an x86-64 virtual server. Our machine has 4 Intel(R) Xeon(R) CPU E5-2686 v4 at 2.30GHz, and 16GB of RAM. It is running an Ubuntu bionic 18.04 AMD64 operating system.

| Malware | Communication | Details |
|---|---|---|
| Gafgyt | Custom | PONG command is communicated via IRC, and others are text commands. |
| Mirai | Custom | All C2 commands are custom binary based. |
| Lightaidra | IRC | All C2 commands are wrapped inside IRC PRIVMSG (private) messages. |
| Remaiten | IRC | Similar to Lightaidra but commands are different. |
| Lizkebab | Custom | Similar to Gafgyt but commands are different. |
| LuaBot | Encrypted payload | Uses MatrixSSL library for payload encryption. |
| Tsunami | IRC | All C2 commands are wrapped inside IRC NOTICE messages. |
| BASHLIFE | Custom | Similar to Gafgyt but commands are different. |

**Table 2.2:** Application layer communication protocol of reputable IoT malware families.

### 2.3.1 IoT Malware Communication Protocols

We conducted a small-scale manual study to gain a basic understanding of IoT malware communication protocols, i.e., the interaction between a bot and its C2 server. In this exploratory study, we analyzed the communication protocols of 10 IoT malware families based on their source code found on GitHub. Table 2.2 shows a summary of the application layer protocols for each family.

*a. The bad news.* We observe that communication protocols vary significantly between families. As a result, a binary of one family will most certainly fail to interact with a C2 server of another family.

*b. The good news.* The protocol tends to remain nearly identical for binary samples of the malware family. This implies that old malware samples could potentially be used to scan the Internet for new live C2 servers of the same family, which we we also corroborated in section 2.4.

These insights suggest that C2Miner could work reasonably well in practice as long as we have access to arbitrary binaries of a specific family of interest, even if these binaries are not the latest versions.

### 2.3.2 Malware Dataset

For our study of live IoT malware and evaluation of C2Miner we collect binary samples from MalwareBazaar [6] and VirusTotal [4] on a daily basis. We focus on the MIPS architecture since it is a common platform for IoT devices, and the least explored architecture by the security community [32]. MalwareBazaar tags binaries as MIPS, while for VirusTotal, we query *elf* samples and filter for "mips" keyword. Later, in our static analysis stage, we only analyze MIPS 32B Big Endian samples that have C2 based communications (not P2P malware). To further validate that the binary is malware, we expect at least 5 engines to identify the sample as malicious (using the query `fs:1d+ type:elf positives:5+ mips`), which is aligned with established best practices [105].

To achieve greater variability, we collected binaries during four different phases over the span of one year: Mar 29, 2021 to Apr 5, 2021 (P1), Jun 6, 2021 to Aug 11, 2021 (P2), Oct 25, 2021 to Nov 1, 2021 (P3), and Nov 10, 2021 to Mar 16, 2022 (P4). On average, we collect roughly 4 new MIPS binaries a day during our collection phases.

The union of all the samples that we collected is *DAll*. A summary of all our datasets is listed in Table 2.3. We initiated 3M exploration requests to 150K IP addresses over the course of our 24 weeks study period to augment and build these datasets. We identify 20K live services (IP:port), and tricked 149 malware binaries into connecting to

| Dataset | Description |
|---|---|
| *DAll* | 1447 binaries collected in total. |
| *Ground1* | 241 malware binaries used as ground truth. |
| *Ground2* | 1083 binaries and their IP:port C2 address cross verified by VT. |
| *Ground3* | 202 binaries of *Ground1* with a live C2 used as ground truth. |
| *Trace-1* | 230MB traffic of 49 binaries from *Ground3* redirected to "talk" to 32 C2 servers in Jan 2022. |
| *Trace-2* | 317MB traffic of 80 binaries from *Ground3* redirected to 34 C2 servers and 39 benign servers. |
| *DFinger* | 202 traffic fingerprints in our formal grammar of the 202 binaries from *Ground3*. |
| *DIP* | 367 IP:port pairs of C2 servers verified by VirusTotal used to filter subnets for probing. |

**Table 2.3:** Our datasets of a total of 1,447 MIPS 32BE IoT malware samples from MalwareBazaar and VirusTotal.

these addresses to prepare our ground truth. We manually checked the safety of our probes, i.e., that they are only "Call-Home" requests and do not modify the server state.

**Detection of live malware-specified C2 servers.** Due to the short-lived nature of C2 servers, we conduct the dynamic analysis on the same day we obtain the binary to maximize our chances of finding the malware-specified C2 servers alive.

Note that C2Miner activates 90% of the binaries, which is on par with success rates reported in earlier studies [32]. In the remainder of this work, we only focus on activated binaries. The failures to activate were mainly due to "illegal instruction error" within the QEMU operation, and in one instance, the malware killed itself, because it detected the emulation environment. Improving the activation rate is part of our future work.

**Obtaining C2 traffic for ground truth.** In our case, having a malware binary is the beginning: we still need to collect its C2 traffic, which is not straightforward. First, we need a live C2 server, but they are rare and ephemeral. In fact, the malware-specified C2

| | Not packed | UPX | Modified UPX | Total |
|---|---|---|---|---|
| Mirai | 449 | 273 | 31 | 753 |
| Gafgyt | 227 | 22 | 26 | 275 |
| Xored | 224 | 4 | 0 | 228 |
| P2P | 0 | 70 | 0 | 70 |
| Bash | 3 | 0 | 0 | 3 |
| Dakkatoni | 0 | 28 | 0 | 28 |
| Tsunami | 9 | 0 | 3 | 12 |
| Lightaidra | 53 | 0 | 0 | 53 |
| Daddyl33t | 10 | 0 | 0 | 10 |
| VPNFilter | 2 | 0 | 0 | 2 |
| Hajime | 13 | 0 | 0 | 13 |
| Total | 990 | 397 | 60 | 1447 |

**Table 2.4:** The 11 malware families and the number of binaries with different packing techniques in our *DAll* dataset.

server (which the malware attempts to contact initially on its own) are usually inactive: only 49% of the samples have a live C2 server by the time they appear in our two sources. Making things worse, many of the live servers become inactive before we finish our experiments.

In total, we were able to find 202 samples that had live C2 servers for this study, which we refer to as *Ground3*. These samples generate 230MB of traffic that includes C2 communication and scanning activity.

**Observation: C2 traffic is a small percentage of the overall malware-generated traffic.** In our ground-truth dataset, we find that C2 traffic is only 14.8KB or 0.06% of the total traffic. We were initially surprised to see this, as we expected that a newly activated malware would focus on connecting to its C2 server. This would have made the detection of the C2 traffic easier, but, unfortunately, this is not the case.

**A note on malware family coverage:** In our datasets, we find binaries of 11 different malware families. Classifying a binary according to family accurately is non-trivial, and we explain our approach below. We use AVClass2 [87] for labeling, but found that AV

engine labels for MIPS samples to be highly inaccurate. For example, all the instances of the Mozi family, a peer-to-peer (P2P) malware for which we observed the corresponding traffic in our analysis, are wrongly classified as Mirai. Thus, we use crowd-sourced YARA rules (provided by VirusTotal results) in addition to AVClass2 to identify the malware family labels. In particular, we use the YARA rules for P2P, XORed (variants of Mirai that use encryption, such as Fbot, Apep and Sora), Daddyl33t and VPNFilter. We do not analyze P2P binaries (mainly Mozi) in the remaining of the work, as we focus on C2 communications, which are typically absent in P2P malware.

### 2.3.3 Traffic Disambiguation Precision

To answer Q1, we want to evaluate the precision of the traffic disambiguation component, namely our ability to differentiate between C2-bound to other traffic generated by the malware. Among all the traffic generated by the binary, only a subset of it is sent to its C2 server. Other traffic could be towards, say a benign victim IP, as part of its proliferation attempt of the malware. In fact, the malware can even contact an IP address to mislead detection attempts. For example, some Mirai binaries contact 65.222.202.53, when the appropriate activation key is not provided at run time. Interestingly, the IP address obtained "notoriety," as it was arguably owned by the NSA [44], and even featured in an xkcd comic (`https://xkcd.com/1247`).

**Ground Truth:** In absence of a benchmark, we created the *Ground1* dataset by manually analyzing malware samples and the traffic they generate. We used network level message signatures to find the C2 servers of samples. First, we created network level signatures based on the source code of IoT malware (Table 2.2) and the reverse engineering

of binaries. We used the Ghidra [8] decompiler for statically reverse engineering of the unpacked samples. If the samples were packed (we only found UPX packing), we unpacked them first. Then, we analyzed the payload of communications of the samples in *Ground1* to find these signatures. We extract the target addresses of the found signatures as C2s. These addresses are either IP:port or a DNS name. We find that only 13% of the binaries use DNS-based C2 addresses, and the rest are IP:port-based.

**Result: We disambiguate C2-bound traffic with 90% precision.** We evaluate our algorithm using a fine-grained method and a coarse method.

In our fine-grained evaluation, for every binary in the *Ground1* dataset, the algorithm returns the target that is most likely the C2 server. We compare this result with the manually found C2s (explained above), and we report the precision of the algorithm in Table 2.5. C2Miner has a precision of 90% in correctly finding the C2-bound traffic on the *Ground1* dataset.

In our coarse evaluation, we find the C2s of binaries in *Ground2* using C2Miner and then query VirusTotal. If VirusTotal reports the C2 address as malicious, we count the finding as accurate. It turns out that the precision based on this method is the same as the one with the fine-grained method in our dataset.

We looked into the reasons why our traffic disambiguation might fail for some samples. We identified two reasons: (a) some samples employ evasion techniques that require a command line argument for activation and in absence of the command they mislead us by regularly communicating with a benign address, and (b) some samples do not start

| Dataset | Samples | Precision |
|---|---|---|
| *Ground1* | 241 | **90%** |
| *Ground2* | 1083 | **90%** |

**Table 2.5:** Precision of C2-bound traffic disambiguation using our fine-grained method with *Ground1* and our coarse-grained method with *Ground2* dataset.

communication with the C2 in our 3 minutes analysis time-span. A summary of our results is reported on Table 2.5.

Would a static analysis of the binary provide the C2 server? We answer this question to gauge the value that our dynamic analysis provides. It turns out that static analysis would provide the C2 server IP address only for 30% of the binaries in *Ground1*. In our static analysis, we first unpack the binaries using UPX (we only see UPX and modified UPX packing), and then we use Balbuzard [2] for string analysis. Balbuzard can crack malware obfuscation such as XOR and ROL.

**Bonus: Some of the malware-specified C2 servers were not known.** C2Miner is able to find malware-specified C2 servers that were not known by threat intelligence platforms. VirusTotal is a widely-used aggregator of security intelligence combining threat intelligence feeds of 91 vendors. For IP:port-based C2 targets, out of the 47 live C2 servers that C2Miner finds, 17% are reported as not malicious by VirusTotal. Intrigued, we kept querying VirusTotal and we found that this number dropped to 3% after 4 weeks. The identification of domain-based C2 servers is even worse as VirusTotal fails to detect 57.6% of domain-based C2 servers as malicious, and this number drops to 35% after 4 weeks. We speculate that a contributing factor is that IoT is an emerging threat, so tools and services around it may be less mature currently.

### 2.3.4 C2 Determination Accuracy

Now we turn to answer Q2 regarding our ability to determine that a probed target is indeed an active C2 server. Given that we have selected the probed target, it does not have to be a C2 server.

**Ground Truth:** The required ground truth here is quite unique. We need a collection of malware binary interactions with: (a) real C2 servers, and (b) benign web and application servers. The goal of the algorithm is to tell the two apart.

As far as we know such ground truth does not exist, therefore, we create our own *Trace-2* dataset as follows. First, we start from samples with live C2 server from the *Ground3* dataset. Second, we select /23 subnets from the C2 server of these samples; this is to simulate the real application of C2Miner in practice. Third, we perform a light-weight SYN stealth scanning to find the live services within those subnets. Fourth, we redirect the C2 traffic (only safe CALL-HOME) to all live services in those subnets. Finally, we manually analyze the contents of the traffic and decide what service the targets host.

In total, we communicate with 200 live services and receive responses containing a data payload from roughly 37% of these live services. The live services mainly host HTTP servers (Apache, Nginx, OrgaMon) but we find also OpenSSH, MySQL, ESMTP and IMAP servers. The rest are communications with live C2 servers. In total, we have malware redirected traffic to 34 C2 servers and 39 benign servers.

**We determine C2 servers with 86% F1 score.** We evaluate the effectiveness of our approach based on the Precision, Recall and the F1 score. We evaluate two C2 determination approaches: (a) *SYN-DATA-aware* (explained in subsection 2.3.4), and (b)

*Fingerprinting-aware* (explained in section 2.2). As reference, we use a simple baseline, which assumes that a target is a C2 server as long as we receive any data response from it. We report the results in Figure 2.3.

*a. Not everything that is "alive" is a C2 server.* The baseline approach is overly "aggressive" but it shows an interesting phenomenon. It finds all C2 server, but at the cost of poor Precision (44%): it also identifies benign servers as C2 servers. The 100% Recall is expected because, as we mentioned, C2 servers will respond with data packets to malware requests.

*b. Achieving 100% Precision with moderately reduced Recall.* Our SYN-DATA-aware method precision has 100% but its 62% recall is lower. The fingerprinting method, that is assessing the target based on an expected communicated pattern, has the same precision with an improved recall of 79%. In summary, our C2 determination based on fingerprinting has a 86% F1 score.

We analyze the errors (FNs and FPs) of the two approaches. For the SYN-DATA-aware, there are cases where the application wouldn't close the socket even in the case of error, and hence this will lead to a false positive. In the case of fingerprinting, there are cases where a legitimate service communication looks similar to the malware C2 communication. An example is a benign IRC server whose communication protocol is similar to that of some malware families (see Table 2.2).

### 2.3.5 Clustering and Fingerprinting

Evaluating the quality of our proposed clustering approach, and the clusters, ultimately depends on the application scenario. Here, we answer Q3 and evaluate the clustering

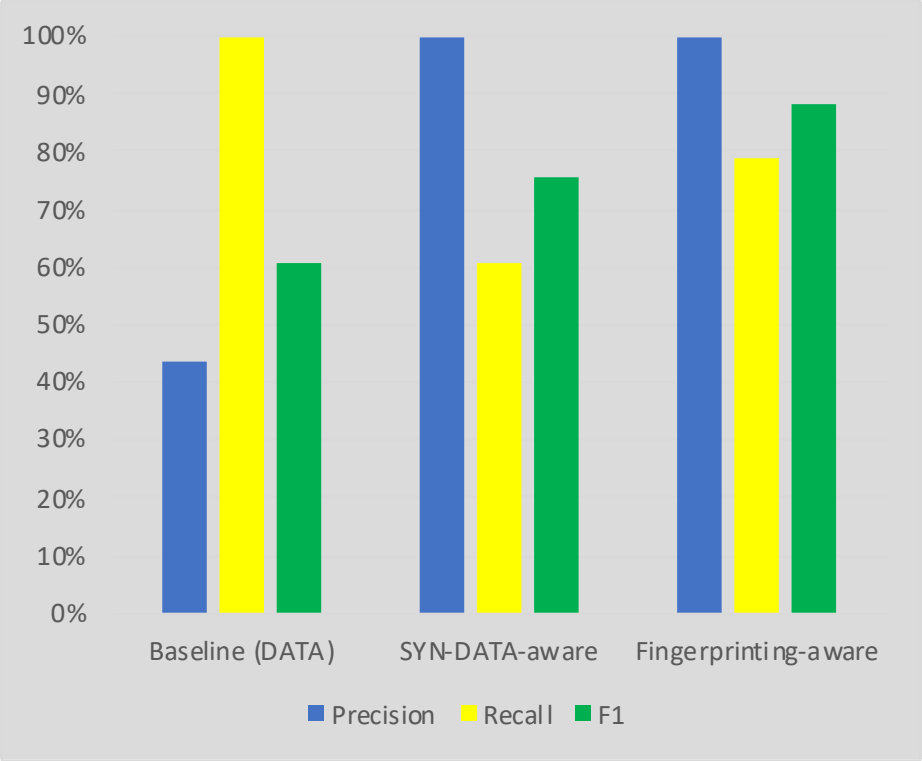


**Figure 2.3:** The precision of the C2 determination approaches based on the *Trace-2* dataset.

quality based on how well we can distinguish between different malware families. The idea is to first cluster the samples based on their communication patterns, and then see if clusters represent malware families.

**Ground Truth:** We require a traffic dataset of labeled malware samples' communications with their C2 servers. As we stated before, this dataset does not exist, and hence, we need to create it. We point out that an already existing labeled malware datasets (with malware family labels) would not be useful because the malware should have a live C2 server so we can collect the communicated traffic. The main challenge is obtaining communicated traffic with C2 servers for IoT malware samples.

We create the *DFinger* dataset which consists of 202 binaries and their fingerprints starting from the binaries in *Ground3* as follows; binaries in *Ground3* had a C2 live server and we were able to engage with their malware-specified C2 during the data collection. In the *DFinger* dataset, each binary is associated with: (a) the family label, as we explain below, and (b) a series of our grammar-based fingerprints of the communication with its C2 server that we captured by collecting samples on a daily basis, and storing the raw traffic that contains communication with live C2 servers. We then transform the traffic to strings in our grammar as shown in section 2.2. To determine the malware family, we query VirusTotal and combine the results with AVClass2 [87] and YARA rules, as discussed in subsection 2.3.2 and summarized in Table 2.4. We consider the family label of a YARA rule, if such a rule exists for the sample. If not, we rely on AVClass2: AVClass2 reports two labels for nearly half of the samples but we consider the one reported by the majority.

**Result: Clustering binaries into malware families based on their communication patterns.** We hierarchically cluster the C2 traffic based on the method that we explain in section 2.2. We choose the seeds randomly, and we observe that the choice of $k$ has minimal effect for $k > 4$; they always converge to the same four clusters; we select $k = 5$. The malware family labels of the four clusters is shown in Table 2.6. The clustering seems to capture the behavior of families of malware reasonably well, but not perfectly. For example, the majority of the binaries of Gafgyt, XORed, Lightaidra and Tsunami fall into clusters C1, C2, C3 and C4 respectively. At the same time, Mirai seems to be spread among all four clusters. Roughly half of the Mirai samples fall into C2 which is also the

dominant cluster for XORed. This makes sense as XORed consists of variants of Mirai with encryption.

**Result: Some families use multiple distinct communication protocols.** The match between clusters and families is not perfect. We were intrigued: why do communication protocols from the same malware fall into different clusters? It turns out that our profiling captures an interesting phenomenon: malware of the same family sometimes use different C2 protocols.

**Understanding Daddyl33t.** We briefly outline our investigation below for Daddyl33t. We study the fingerprints of the Daddyl33t binaries and observe three fairly distinct patterns of communication which correspond to C1, C2 and C3 clusters of Table 2.6.

**Pattern 1:** We observe the following pattern for Daddyl33t binaries in cluster C1. The malware connects to the server and sends 18 bytes of data. This is in plain text, and contains the message "VER:3:unknown". The server responds by a single byte followed by another 4 bytes of message. The binary closes the connection, and repeats the same thing again and again.

**Pattern 2:** The dominant pattern in the C2 cluster is different. Here, the Daddyl33t binaries connect to the server and send in plaintext "VER:0:/malware". The server responds with 516 bytes of data. Unlike the first pattern, here the binaries do not close the connection.

**Pattern 3:** In cluster C3, the malware connects to the server and sends information about the compromised device, such as the architecture and the Endianness. The binary just keeps the connection open.

| | Mirai | Gafgyt | Daddyl33t | Xored | Light/ra | Hajime | Tsunami |
|---|---|---|---|---|---|---|---|
| C1 | 25% | **68%** | 20% | 34% | 0% | 50% | 0% |
| C2 | **46%** | 0% | 40% | **66%** | 14% | 0% | 0% |
| C3 | 18% | 26% | 40% | 0% | **86%** | 0% | 0% |
| C4 | 11% | 6% | 0% | 0% | 0% | 50% | **100%** |

**Table 2.6:** Clustering effectiveness: the clusters are reasonably aligned with malware family labels in the *DFinger* dataset.

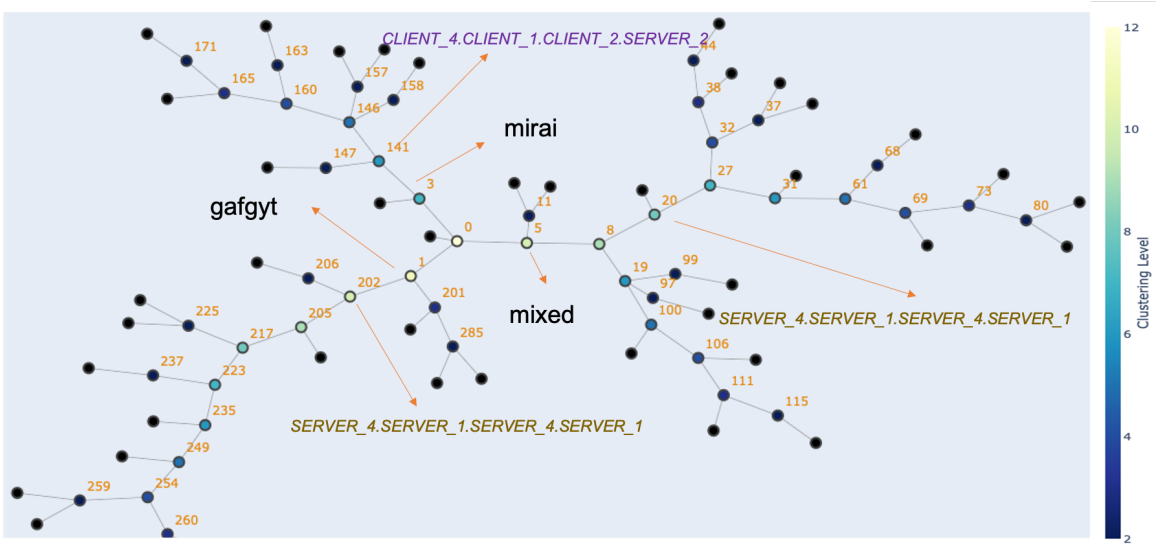


**Figure 2.4:** Hierarchical clustering of samples in *DFinger* based on their communicated patterns with C2. Clusters expand from cluster 0 (the initial dataset).

Interestingly, we find one C2 server that responds to communication patterns 1 and 3 which further indicates that these are communication patterns of the same malware.

With a similar investigation, we find that Hajime also exhibits two distinct patterns of communication which explains its presence in two different clusters.

Overall, our hierarchical clustering results in 215 clusters shown in Figure 2.4. The interesting observation is the common patterns that appear in communications. Most Mirai samples share the CLIENT_4.CLIENT_1.CLIENT_2.SERVER_2 pattern. On the other

hand, most Gafgyt samples share the SERVER_4.SERVER_1.SERVER_4.SERVER_1 pattern. With these two patterns alone we can detect 68% of C2 communication.

### 2.3.6 MitM Functionality Assessment

The answer to Q4 on whether two binaries of the same family talk to the same C2 server indicates whether we can trick arbitrary binaries of a family to reveal current C2 servers. Our initial analysis of publicly available IoT malware source code suggests this is true (see subsection 2.3.1), nevertheless, we empirically assess this hypothesis. Our assessment has two goals. First, we want to see whether the MitM component functions in practice. For example, reasons for failure could involve encryption at the IP layer, or active efforts to bypass the kernel-level networking. Second, we want to see whether old binaries can lead us to currently live C2 servers.

**Ground Truth:** For this evaluation, we need a dataset of live C2 servers and the samples with which they can communicate. In absence of such a dataset, we create the *Trace-1* dataset from our collected binaries as follows. In more detail, we start with the list of malware with live C2 addresses in *Ground3* as they are collected on a daily basis. We first find the malware-specified C2 target of the malware. Second, we want to find other binaries that can communicate with these new servers. To do this, we find binaries with similar C2 communication behavior using our fingerprinting which we described earlier. Third, we check if the binaries of the previous step communicate with the server successfully, which we validate with manual inspection. Our manual evaluation method is similar to what we described in subsection 2.3.4. The number of binaries in *Trace-1* was limited since many

| Parameter | Values |
|---|---|
| Subnets | 136.144.41/24, 195.133.40/24, 2.58.149/24, 212.193.30/24, 107.173.176/24, 45.95.169/24 |
| Ports | 1312, 666, 1791, 9506, 606, 6738, 5555, 1014, 3074, 6969, 42516, 81 |
| Sample(s) | Gafgyt 46501d723f368c22e5401f7c95d928ab |
| Sample(s) | Mirai 800af659256f0232a27f955a4430aed0 |

**Table 2.7:** The configuration parameters of the probing case study conducted in January 2022: (a) 6 /24 subnets, (b) 12 ports in order of "popularity", and (c) two malware samples.

C2 servers were short-lived, and it was not always possible to find compatible binaries to engage with them in time.

**Result: Our MitM capability works well in practice.** We measure how successful our MitM approach is in practice based on the number of sample pairs that can successfully talk to the designated candidate live C2 server for the malware cluster. A high rate of success indicates that one sample from a cluster is enough to find the C2 servers for the entire cluster. We use the fingerprinting-based method for determining C2 servers. We find that 84% of sample pairs can successfully cross-communicate with the C2 of the malware cluster.

## 2.4 Case Study: Hunting Live C2s

*How can we use C2Miner in practice to find live C2 servers?* This is the question that we answer in this case study.

**Our proof-of-concept deployment: limited effort, big results.** We showcase the value of C2Miner with a proof-of-concept deployment. For efficiency, we perform our probing in two phases. First, we use a light-weight SYN stealth scan using MASSCAN [45] to establish liveness. Once this is established, we deploy the more resource-consuming C2Miner

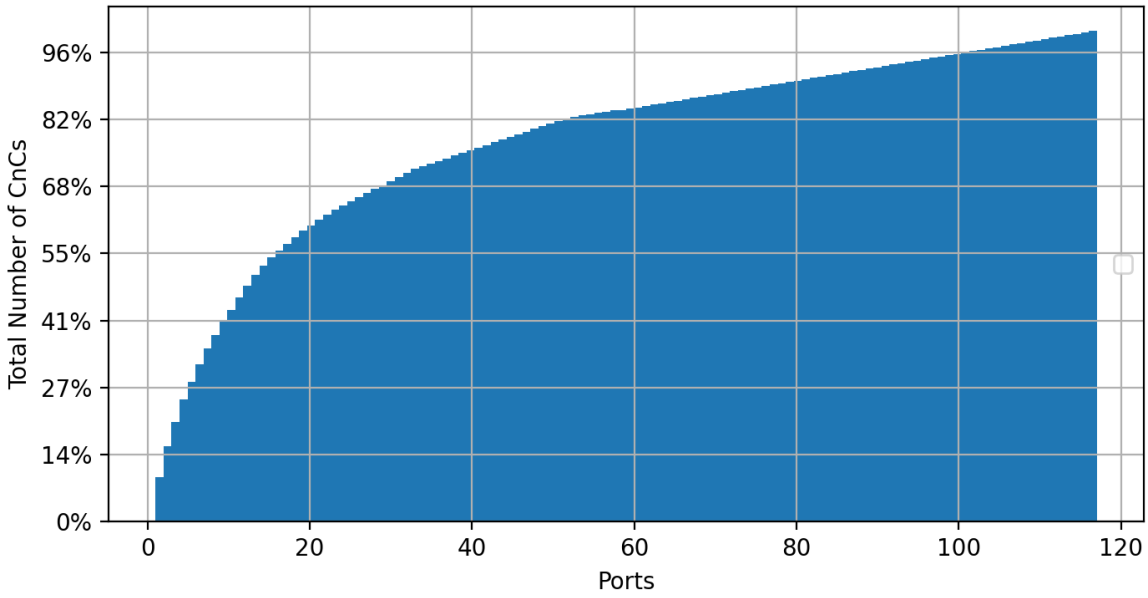


**Figure 2.5:** Cumulative distribution of C2 servers across port numbers ranked by popularity.

probing. We carefully select binaries and IP:port space for exploration to efficiently use the computation resources.

To show the promise of our approach, we focus on two old malware binaries: (a) one from the Mirai family, and (b) one from the Gafgyt family as shown in Table 2.7. We chose these two samples because their communication fingerprint is "close" to 68% of the samples, which is an application of our grammar-based clustering (see subsection 2.3.5). These samples are at least six month old.

We made the following choices for the IP:port space selection. We conduct roughly 331K probes across 18K IP:port combinations for 1,536 IP addresses and 12 ports with three probes per day for 6 days using the two binaries. We identified 6 /24 subnets and 12 "popular" ports as we explain below and shown in Table 2.7 based on the increased

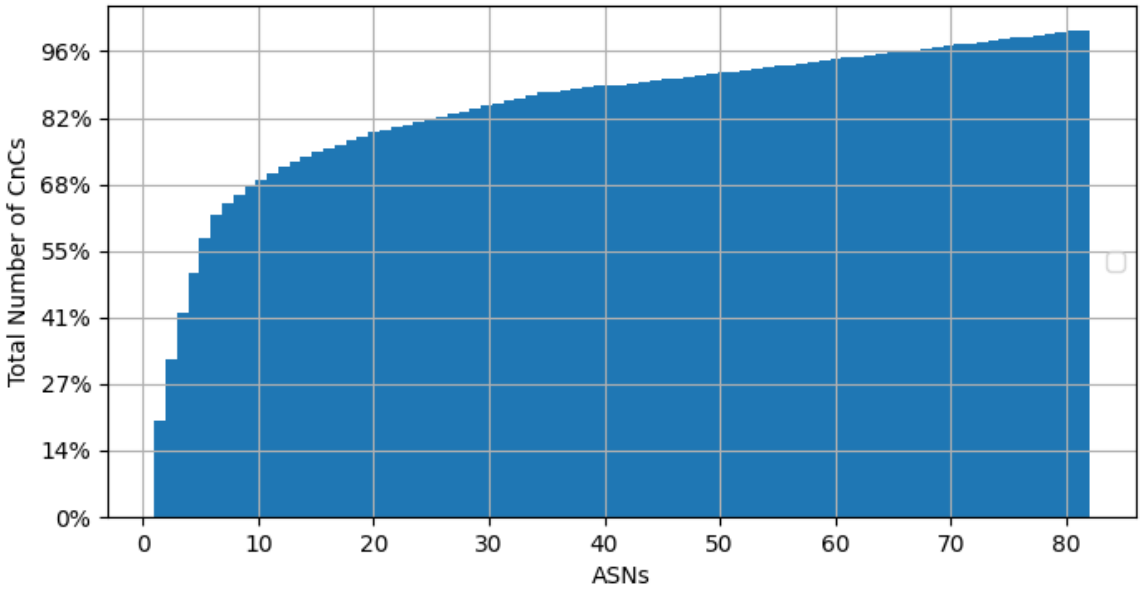


**Figure 2.6:** Cumulative distribution of C2 servers across Autonomous System Numbers (ASNs) ranked by popularity.

likelihood of observing malicious activity from past C2 hosts [25, 33]. This limited size case study shows the potential value that we can obtain, even with resource constraints.

**Spatial patterns of IoT C2 Servers.** Deviating from the problem formulation, we are required to identify our own promising target space here. We study the malware-specified C2 addresses in the *DIP* dataset. These are a subset of *Ground1* that we identified before Jan 2022: each row is a C2 IP:port pair associated with its hosting Autonomous System (AS) and country. We study the IP-space spatial properties of these malware-specified C2 servers in order to identify locality patterns.

First, we observe that 12 ports are used by half of the C2 servers. Figure 2.5 shows the distribution of ports. Using this insight, we focus our probing to these frequently-used ports. Second, to identify IP spaces with a higher likelihood of C2 hosting. We plot the

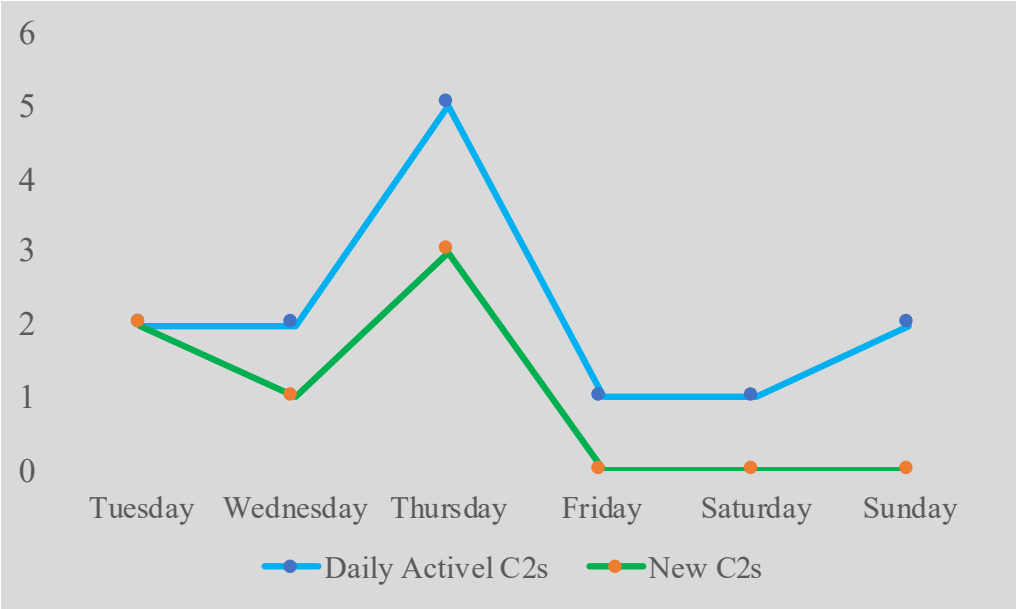


**Figure 2.7:** The number of responsive C2 servers per day and the number of distinct new C2 servers during our 6 day probing over only 1,536 IP addresses across 6 subnets and 12 ports.

distribution of the C2 servers across ASes in Figure 2.6. We find that the 7 most "popular" ASes host roughly 65% of the malware-specified C2 servers.

Based on the aforementioned observations, we create our target list of IP:port tuples as follows. From these 7 ASes servers, we select subnets with more than 10 reported C2s, and the top 12 popular ports as reported on Table 2.7.

**Temporal patterns: Daily report of active C2 servers.** We employ C2Miner's probing mode in practice for 6 days. Below are some highlights of the results.

**a. Probing success: 6 live C2 servers in 6 days.** The results are very promising: we find a total of 6 active C2 servers for our binaries: 5 Gafgyt and 1 Mirai. In Figure 2.7, we plot the number of active servers each day and the new distinct servers discovered that day.

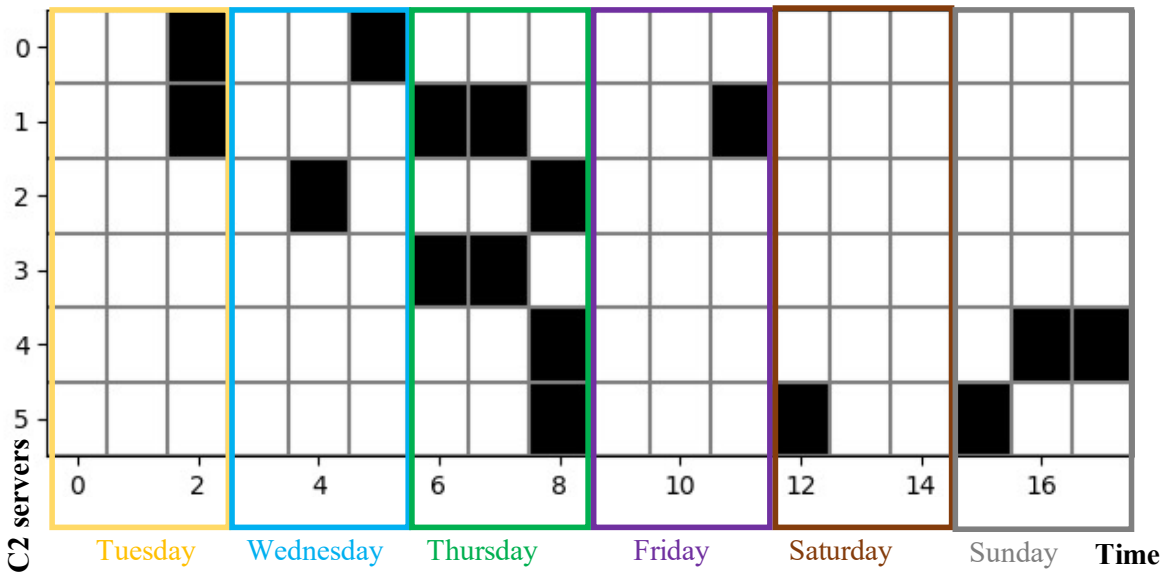


**Figure 2.8:** The responsiveness of the responsive C2 servers over time to our three daily probes which is 15% per probe assuming they are alive for the full duration. The exact IP:port values of these C2 servers are shown in Table 2.8.

We argue that the performance to cost ratio is high. Despite the limited scope of our study, we identified one live server a day. We consider this fairly remarkable given the limited IP space we explored, and the heuristic approach in finding IP space with likely C2 servers. Note that these are servers that without a MitM approach would not have been discovered even if one had activated these two malware in a sandbox, as the malware-specified C2 servers (i.e., the ones embedded in these particular binaries without any fallback mechanism) are no longer active.

**b. The temporal behavior of C2 servers.** Intrigued by the seemingly high temporal variability of the C2 servers, we investigate further the daily liveliness of the found C2 servers, which we show in Figure 2.8.

| C2 server ID | IP | port |
|---|---|---|
| 0 | 2.58.149.34 | :5555 |
| 1 | 212.193.30.91 | :666 |
| 2 | 45.95.169.119 | :666 |
| 3 | 136.144.41.240 | :666 |
| 4 | 212.193.30.123 | :5555 |
| 5 | 107.173.176.144 | :42516 |

**Table 2.8:** Live C2 servers (IP:ports) found during the 6 days of our case study of 331K probes targeting 1,536 IP addresses and 12 port.

**Key Observation: Live C2 servers response rate is low even for bots of their family.** Our preliminary statistics based on our limited study here seems to suggest that even a live server will not respond consistently to a bot request. We can count percentage of successful responses under two different assumptions. First, the success rate of a probe is 15% if we assume that the servers are alive for all six days. Second, the success rate is 31%, if we assume that a server is alive only during the days of its first and last response to our probes.

**Best practice recommendation:** Our initial results suggest that we need to send many probes to a server to validate that it is not live given the low success rate that we observed above.

## 2.5 Discussion and Limitations

We explain the opportunities and limitations of C2Miner in the form of questions below.

**a. What is the value of the MitM capability?** The MitM capability allows us to trick the *old* malware and turn it into an active probing tool. It should be clear by now

that without the MitM capability each binary can only reveal a small number of potentially obsolete C2 servers.

**b. How can C2Miner be deployed at a large-scale?** The current work provides the foundation for a large-scale study, and a proof of concept to show the promise of the approach. Although we addressed problems such as malware binary selection using our clustering, and IP space prioritization using locality patterns, there are still challenges that need to be addressed before a large-scale deployment. More specifically, we need to: (a) expand the supported architectures, (b) develop and use state of the art anti-evasion techniques, and (c) collaborate with ISPs and cloud providers for massive probing. While these challenges are important they are rather engineering and bureaucratic challenges.

**c. Are the evaluation results generalizable?** We believe our evaluation reflects the general performance of our approach, and the intrinsic nature of the problem. The size of our evaluation datasets is constrained by: (a) the scarcity of live C2 servers in our collection, (b) the lack of benchmark datasets, and (c) the manual effort required for establishing the ground truth. That said, our datasets are comparable or larger than the datasets used for similar evaluation studies [43, 74]. We continue to collect data and we will share the results with the community.

**d. How can the C2Miner output be used in Intrusion Detection Systems?** The fingerprinting strings that represent C2 clusters can also be used in Intrusion Detection Systems (IDS) like Snort and Surricata. For converting these strings into IDS rules, the *flowbits* functionality of these systems can be used to tag individual packets following our grammar rules. Then, the *isset* operator can be used to compare fingerprints of flows. The

Jaccard distance function can be used to calculate the similarity, and seeing if the traffic belongs to a cluster.

**e. What if the malware does not activate?** This is a concern for any work that relies on dynamic analysis. Our intention is to use the latest sandboxing technology, as this is not where the novelty of our work lies. If a "smart" malware refuses to activate, we cannot analyze it. It has been observed [32] that current IoT malware is not yet as sophisticated as PC-focused malware. Although this may change in the future, we can hope that novel sandbox techniques will also evolve. In addition, note that we do not need to activate all the binaries, as long as we activate enough binaries that "talk" to most types of C2 servers.

**f. Can C2Miner extend to non-IoT malware?** The overarching approach applies to any malware that can be activated in a sandbox. However, our current methods will need significant fine-tuning to adapt to "behaviors" of other malware. For example, the algorithmic solutions of our approach have been inspired by and trained on the worm-like behaviors of IoT malware. IoT malware often generates large traffic for scanning and exploitation that would hide C2 communication, and this might not be the case for other malware. However, with sufficient customization the overarching framework could be made to expand to additional types of malware and platforms.

**g. How difficult is it to evade C2Miner?** We consider two cases. First, if the malware does not use encryption and obfuscation at the TCP/IP layer, we argue that we can engage with the C2 server, as along as we can activate the binary. Our rational is that it is difficult for a C2 server to detect MitM-intervention by looking only at a bot request at the network layer and our initial results seem to corroborate this. Second, if the malware

uses encryption and obfuscation at the TCP/IP layer, our solution will not work. However, the use of encryption at this layer is computationally expensive and because of that not commonly used. In fact, many related security solutions will not work in the presence of encryption. Overall, security is an arms race, and a security approach is successful if it forces hackers to modify their behavior, especially, if the modification is non-trivial, as is the case with encryption at the network layer.

## 2.6 Related Work

There are several categories of related work to our research. Below, we discuss each category and explain how this research is different.

Overall, none of the related efforts has focused on the problem as framed here: finding live C2 servers for an unknown binary and within a target IP:port space as we do here. To the best of our knowledge, we are the first to redirect the communication of a binary using a MitM approach to "reconnect" old samples with currently live C2 servers.

**a. IoT malware analysis.** Studying the behavior of IoT malware has become a hot topic both for academia and industry. First, many efforts focus on characterizing the behavior of a single malware family [11, 50, 46]. These works characterize the infected IoT devices, the infection vectors, and the life-cycle of the malware. While these studies provide a deep insight into a single malware family and its life-cycle, they are less likely to lead to generalizable methods for detecting C2 servers dynamically.

Another group of prior efforts characterize the behaviors of several malware families at the same time [30, 31, 32, 68, 82, 9]. Finally, several studies analyze C2 server communication from a networking point of view [82, 94, 91]. Although interesting and

informative, these studies focus on understanding the infrastructure that supports its operation, and profiling aspects of the malware behavior, but do not engage in active probing like we do.

**b. Active probing of malware.** The most relevant studies to our work focus on active probing. Such efforts require an understanding of the malware communication protocol and the encryption algorithm, if any. Assuming that this can be accomplished effectively, searching for C2 servers of a single malware family becomes easier [42, 11]. In addition, there is prior work that took the first steps in automating the active probing for a more widespread group of malware families [75, 97]. The main problem with these approaches is that they can not handle the communication protocol when there is encryption. In contrast, we overcome this issue as we let the activated binary "speak" to the server with or without encryption, and we simply observe the communication.

**c. Engaging with malware.** Prior work has attempted to engage with malware to understand different aspects of its behavior [74, 78, 43]. Two earlier works [78, 74] try to reveal the malware's alternative methods of communicating with C2 servers. In contrast, IoT botnets are disposable and hence alternative addresses are not a broad phenomena for them [91]. A most recent work [43] develops techniques to detect and spoof bot-to-C2 communications, but their technique is applicable only to malware with over-permissioned protocols and they do not do active probing.

**d. Network traffic modeling and analysis.** Modeling network traffic enables the identification of particular protocols or activities. The first group of work in this area tries to infer the application layer protocol [95, 20, 27]. However, this is not effective in

presence of encryption, slow because of deep packet inspection, and requires access to the execution context. In comparison, we only need the network traffic for fingerprinting. The second group of work is more closely related and focuses on network flows [96, 103, 92, 15] and use features that include IP, port, timestamps, number of bytes, connection duration etc., which are more difficult to generalize. Instead, our approach creates an abstract representation of a flow by modeling using a formal language.

Studying the network behavior of malware has been the subject of another line of related work [48, 47, 37, 64] trying to find botnet traffic within a network trace. Thus, they need the network trace, and they can only find servers whose flows happen to be in that trace, in contrast to our ability to selectively target any potential device in the Internet.

## 2.7 Conclusion and Future Work

We propose C2Miner, a novel approach to trick arbitrary malware binaries to reveal their currently live C2 servers. The novelty of our approach is that we fool the malware twice: (a) we convince the malware to activate in a sandbox and start searching for its C2 server, and (b) we perform a MiTM attack on the C2-bound traffic and use it to search for C2 servers in an IP:port space of our choice. To substantiate this vision, we develop techniques to: (a) disambiguate the C2-bound traffic with 92% precision, (b) determine if a target IP:port is indeed a C2 server with an F1 score of 86%, and c) fingerprint and cluster C2 communications effectively.

In the future, we plan to extend our work in two ways. First, we plan to systematize and automate the task of finding candidate targets by combining external sources of information and using adaptive techniques. Second, we will conduct a large-scale longitudi-

nal study that will provide: (a) an extensive list of live C2 servers, and (b) spatio-temporal properties of their behavior and migration patterns.

Our approach is a fundamental step towards identifying live C2 servers on-demand given a binary. This capability is even more critical for IoT malware that is an emerging battleground for cybercrime. A large-scale deployment of our approach using a large number of binaries and scanning substantial swaths of the Internet can arguably become a game-changing capability in identifying C2 servers and containing the scourge of botnets.

# Chapter 3

# MalNet: A binary-centric network-level profiling of IoT Malware

A critical component in combating Internet of Things (IoT) malware is timeliness: Indicators of Compromise (IoC) and signatures need to be made available as fast as possible. Although this is true for malware in general, this is more urgent for IoT devices, which have significantly less built-in protection. For example, home sensors and industrial controllers are typically not protected by on-device defenses such as anti-virus software. Therefore, they rely evermore on firewalls and blacklists. In addition, if we identify the command and control (C2) servers or the exploits that the malware will use, we can improve our defense by hardening, spying and even subverting the botnets. For example, Internet Service Providers (ISPs) and law enforcement can block and take down these C2 servers to disrupt the botnet [73, 63].

**Problem:** How much information can we extract from a newly found IoT malware binary? This is the question that motivates our work. Our goal is to reveal a missed opportunity: malware samples are captured and made available through services, such as

| Dataset Name | Size | Methodology | Misc |
|---|---|---|---|
| *D-Samples* | 1447 | Daily Collection from VT and MalwareBazaar | MIPS samples for both C2 and P2P malware with the following YARA and AVCLass2 labels: Mirai, Gafgyt, Tsunami, Daddyl33t, VPNFilter, Mozi, Hajime |
| *D-C2s* | 1160 | C2Miner and VT | C2 addresses found by C2Miner and cross verified with VT and manually |
| *D-PC2* | 588 | Probing using C2Miner | Traffic of 7 C2s on a 4 hours interval recorded for 2 weeks |
| *D-Exploits* | 197 | Handshaker | Exploits found by completing the handshake with malware when targeting a victim |
| *D-DDOS* | 42 | Spying IoT C2 commands | Traffic of DDoS commands and the attacks launched by malware |

**Table 3.1:** The datasets used in this measurement study

MalwareBazaar, but are not really used to profile IoT malware network traffic in a systematic and timely way. Specifically, the input to the problem is a malware binary, and the desired output is a comprehensive profile that includes: (a) its C2 server communication, (b) its proliferation techniques, and (c) its attacks as they are being launched. This comprehensive profile can help: (a) secure the network, through firewall rules, (b) harden the security of the device, and (c) provide intelligence of attacks as they launch.

**Previous work:** we have key differences with the related work. In order to compare our work, we classify related efforts in the following groups. First, several efforts analyze the non-network behavior of the IoT malware [30, 31]; in contrast, we are only focused on the network behaviors. Second, several related work focus on only one of the three categories of IoT malware network traffic that we stated above. A few of the related work in this category analyze the C2 communication [78, 74, 43, 94, 91]. Another group within this category, explore the proliferation behaviors of the malware [68, 10, 54]. The last group in this category study the DDoS attacks [81, 60, 58]. We are different from these

work in that we provide a holistic view. In addition, we have subtle differences with each group within this category that we elaborate in section 3.6. Third, several related work analyze multiple aspects of the IoT malware network characteristics [50, 11, 46, 32, 82]. These work either are not binary-centric (they only look at traffic and network addresses), or do not provide a holistic view on all three aspects.

**Contribution:** We conduct an extensive and systematic daily analysis of the newly reported IoT malware binaries by VirusTotal and MalwareBazaar. Note that, unlike traffic analysis studies, a binary-centric study can create a holistic picture of the IoT malware with full attribution. In other words, we can connect a binary and its family, with a live C2 server, a set of proliferation techniques, and even actual launched DDoS attacks including their type of attack and the target. Our study is focused on timeliness in two ways. First, we collect binaries as soon as they become available from VirusTotal and MalwareBazaar for a year (March 2021 - March 2022). Second, we dynamically analyze these binaries *on the day* that we capture them. Overall, we collect and analyze 1447 malware binaries, which seem to cover seven major malware families as we will see in Table 3.1. We use two approaches to analyze the malware dynamically: (a) ***observational***, where we let the malware contact its own server, and (b) ***active probing***, where we redirect the C2 communication to potential targets in search of live servers.

We highlight key results from our study. A key goal is to provide a proof of concept for the value of a measurement study: binary-centric and focused on timeliness.

**a. Capturing the ephemeral and elusive behavior of C2 servers.** We study the spatiotemporal properties of live C2 servers. First, we find that the responsiveness of

live C2 servers is spotty: the servers never respond to all six probes within a day. In fact, 91% of the time a server does not respond to a second probe four hours after a successful probe. Second, we find that the *observed* lifespan for close to two thirds of the servers is one day. Third, we find that 15% of the live servers that we identify are not known to threat intelligence feeds, which could be partly caused by the elusive nature of the servers.

**b. Proliferation techniques use old vulnerabilities.** Our binaries attempt to exploit 12 different vulnerabilities with 9 of them more than 4 years old. Even the most recent vulnerability was 5 months old. On the one hand, this is an optimistic result: finding ways to defend against well-known and old vulnerabilities could safeguard our IoT devices. On the other hand, this could be an indication that IoT devices are so vulnerable that hackers don't have to try hard to compromise them.

**c. Eavesdropping on live DDoS attacks.** After connecting to live C2 servers, we capture the launch of 42 DDoS attacks, as evidenced by the command received from their C2 server. We obtain the target addresses and we identify 8 types of attacks with two types of attacks targeting gaming servers. Furthermore, we find that three target IP addresses were within networks owned by Google, Amazon and Roblox.

**Envisioning a large-scale deployment.** The key goal of this work is to show the significance of a binary-centric and timely dynamic analysis of malware. Our preliminary results show the type and value of the information that can be extracted. Our goal is to expand the scope of the study in the future into a large-scale continuous IoT malware monitoring infrastructure. Achieving this will require: (a) expanding the sandbox capability

to activate binaries efficiently by emulating different host devices, and (b) develop techniques to profile the collected information into easy to use rules for different firewall technologies.

**Open sourcing and sharing.** Our group is committed and has a track record of sharing tools and our data openly. In fact, this was a key reason for leveraging and expanding existing open-source tools. In addition, we will share all our datasets including: (a) captured malware binaries, (b) the discovered C2 servers, (c) the communication traffic, and (d) targets of DDoS attacks. Naturally, we will vet the credentials of users requesting sensitive information, especially the malware binaries.

## 3.1 Methodology and Datasets

In this section, we explain our experimental set up, and the methodology for establishing the datasets that we use in our study.

### 3.1.1 Experimental setup: our sandbox

In our dynamics analysis, we activate the malware binary in a sandbox. Among the various tools, we selected C2Miner [35], a powerful open-source tool for analyzing IoT malware binaries. Leveraging the capabilities of the tool, we conduct experiments in two different modes. In the first mode of execution, we find the referred C2 servers in the binary. In order to accomplish this, we emulate the execution of the malware binary using QEMU [13]. Then, we analyze the network communication of the C2 malware. As reported, we can detect C2-bound traffic with a 90% precision [35]. In the second mode of execution, we weaponize a binary and use it for probing a set of IP:port targets of interest. In this mode, we can identify the live C2 servers in the target address space of interest that are

able to communicate with the weaponized binary. In both modes of execution, the traffic generated by the malware can be captured in a pcap format.

### 3.1.2 Creating the malware binary dataset

We collect malware binaries with the following process. First, we collect malware on a daily basis. Every day between March 2021 and March 2022, we collect the new IoT malware binaries released by VirusTotal and MalwareBazaar. Then, we dynamically analyze the new malware binaries on the day that they become publicly known. In this study, we focus on MIPS 32B binaries as we focus on IoT malware. Considering additional types of malware would require extending the capabilities of our sandbox, which we intend to do in the future. We were able to collect 1447 MIPS 32B malware binaries, which we refer to as *D-Samples*. Our dataset seems to cover a wide range of malware families as shown in Table 3.1.

We briefly discuss our approach to verify the nature of the collected binaries. First, we ensure that each binary is malware by getting the corroboration of at least 5 malware detection engines. Note that the thershold of 5 engines is aligned with established best practices [105]. Second, we identify the family of the malware as follows. We use crowd-sourced YARA rules (provided by VirusTotal results) in addition to AVClass2 to identify the malware family labels. Note that the AVClass2 [87] seems to be often unreliable for for MIPS binaries. For example, all the instances of the Mozi family, a peer-to-peer (P2P) malware in our analysis, are wrongly classified as Mirai. In total, we have a total of 1447 binaries in our malware sample dataset.

### 3.1.3 Profiling IoT C2 addresses

We profile the IoT C2 addresses and create two datasets: *D-C2s* and *D-PC2*. First, we explain the creation of *D-C2s*. Our first step in creating *D-C2s* is filtering out the P2P samples (mostly Mozi malware family) from our *D-Samples*. Having done that, we have a set of malware samples that would have C2 communication. Next, we use our sandbox to analyze the binaries and find its referred C2 address. Then, we cross validate the result with Virus Total Intelligence feeds by checking whether the reported C2 address (IP or DNS) is malicious. In order to measure the miss rate of the threat intelligence feeds provided by VT, we query VT two times. Once on the day the binary is published, and once on May 7th 2022. If a C2 is reported as malicious by the second query, but not by the first one, it is a miss. Finally, we perform a manual verification of samples that have unverified (by the two queries) live C2 addresses. Our manual verification compares the captured traffic with Mirai, Gafgyt, Tsunami and Daddyl33t network protocols. We refer to this dataset of C2 addresses as *D-C2s*, which has 1160 addresses of C2 servers.

We complement our understanding of IoT C2 servers with active probing. The goal is to understand the IoT C2s temporal behavior. *D-C2s* is not timely because it depends on the latency of sharing the IoT malware binaries. Henceforth, *D-C2s* is not the best choice for evaluating temporal behaviors of C2 servers such as responsiveness and we rely on active probing for such an analysis. Below, we explain our active process.

The key idea to create *D-PC2* is to probe a target subnet and a set of ports, and then observe the C2 servers as they become online and go offline. To this end, we conduct an active probing study of 6 sample subnets and 12 ports with past history of

malicious activity. The next step to conduct this study is to select malware samples; we select two samples, one Gafgyt and another one Mirai. We probe the subnets and ports for two weeks on a 4 hours interval basis. In this period, we find 7 C2 servers. At the end of the measurement, we create *D-PC2* dataset that contains 64 traffic measurements per C2 address.

### 3.1.4 Observing exploits and vulnerabilities

In order to better understand the proliferation behaviors of the IoT malware, we extract the exploits from the malware using well-established methods [10]. Specifically, we trick the malware into sending over its exploits to fake victim targets that we control. Our process is as follows. First, we identify the ports that malware tries to scan and attack based on a threshold on the number of distinct IPs that are contacted for a particular destination port. We choose the value 20 for this parameter. Next, in a separate thread we create a socket on that port and redirect the future traffic to that local port for the next IP addresses. Having done that, the malware completes the TCP handshake with the fake target and collects the payload sent by the malware. We call this method handshaker, a term that was used before in the previous studies [91, 54].

We create dataset *D-Exploits* with the exploits we extract using the handshaker method. Overall, we could successfully extract exploits from 197 samples targeting 12 vulnerabilities. We will provide more details about the vulnerabilities that were exploited in section 3.3.

### 3.1.5 Observing DDoS attacks

As we mentioned earlier, we observe the launch of real DDoS attack, including the command from the C2 server, and the traffic generated by the malware in our sandbox. Note that this ability to listen in to the server commands and connect them with the actual attack is possible in our binary-centric study, but not possible in passive analysis of network traffic that other studies have done.

We describe our method in more detail. First, we find malware samples with live C2 servers. For this, we analyze the malware on the day they are first submitted, and watch the C2 communication. If the communication is successful, we let the malware run for 2 hours in a restricted mode (only C2 traffic is allowed). Next, we rely on two methods to find DDoS commands that we explain below.

**i. Extracting DDoS commands from known IoT C2 protocols:** We build a profile of three IoT malware application layer communication protocols: Mirai, Gafgyt and Daddyl33t. For Mirai, Gafgyt , we build the profiler based on the available source code of these malware families. For Daddyl33t, we reverse engineer the communicated traffic and create the profile. While Mirai employs a binary based protocol, Gafgyt and Daddyl33t use a text based protocol. Having prepared the profiles, we search the communicated C2 traffic for DDoS commands based on the profiles.

**ii. Extracting DDoS commands based on behavioral heuristics:** In order to cover other malware families, and new variants, we employ a heuristic detection method to find DDoS commands that is as follows. We count the number of packets sent to non C2 IP addresses, and measure the packets per second rate. If this rate is higher than a set

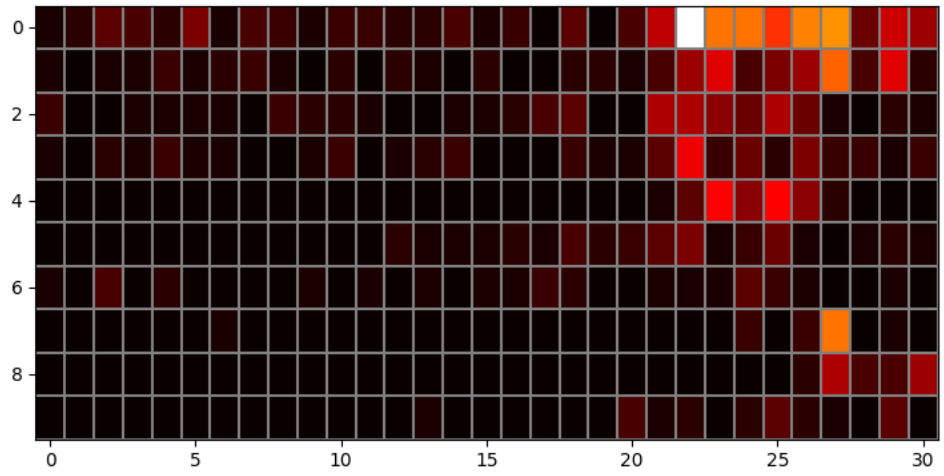


**Figure 3.1:** Heatmap showing the weekly (x-axis) activity of malware-specified C2 servers from our malware feed (MalwareBazaar and VirusTotal) across the ten most active ASes (y axis) between Mar 2021 and Mar 2022. Lighter color indicates more servers. The top four ASs are consistently more active with more dark red activity.

threshold, we consider the last issued C2 command as a C2 command and record it. By default, we set the threshold to 100 packets per second.

After collecting traffic that passes one of the above two filters, we further manually verify the correctness of the results. For the first method, we verify the command by evaluating whether the bot started to send traffic to that given DDoS target continuously. For the second method, we extract the target address and search for string and/or binary representation of the target IP in the last issued C2 command.

We refer to the dataset containing the DDoS commands, and the traffic as *D-DDOS*. This dataset contains the 42 commands issued to 20 different malware samples.

| AS Name | ASN | Country | Hosting | Anti DDoS? |
|---|---|---|---|---|
| ColoCrossing | 36352 | US | Yes | Yes |
| Delis LLC | 211252 | US | N/A | N/A |
| DigitalOcean | 14061 | US | Yes | Yes |
| FranTech Solutions | 53667 | LU | Yes | Yes |
| HOSTGLOBAL | 202306 | RU | Yes | Yes |
| Serverion LLC | 399471 | NL | Yes | Yes |
| OVH SAS | 16276 | FR | Yes | Yes |
| IP SERVER LLC | 44812 | RU | Yes | Yes |
| Apeiron Global | 139884 | IN | Yes | No |
| Serverius | 50673 | NL | Yes | Yes |

**Table 3.2:** Information about the top 10 Autonomous Systems that host the C2 IPs

### 3.1.6 Ethical Considerations

We are confident that our experiments have not caused any damage and took appropriate precautions in the four types of experiments we have. We use SNORT IDS to detect and prevent malicious traffic from leaving our network. Furthermore, we had techniques in place to contain malicious traffic in each of our experiments:

**a. Detection of C2s:** The tools that we use to detect C2s do not need Internet connection when the malware is analyzed. Thus, this analysis does not interact with the Internet, as we "fake" it to the sandbox. If a sophisticated binary detects that the Internet is not available, we deploy InetSim [5] to simulate services like DNS and http.

**b. Detection of Exploits:** This experiment did not need Internet connection as we fake victims for the IoT malware. We find the addresses that the malware tries to exploit and complete the handshake with the malware pretending to be those targets. We collect the following data packets that might contain the exploit code.

**c. Detection of DDoS targets:** Based on the results of (a), we filter out any other communication of the malware with the outside except to the C2 target. We record the C2 traffic, reverse engineer it and find the DDoS commands and their targets.

**c. Probing IP Subnets:** We only allow C2 communications like "Call-Home" messages to interact with potential C2 servers. We have manually analyzed sample traffic traces and we have not found any cases of non-C2 communications. Our target subnets were small /24 subnets with a history of malicious activity. We do not send probes if the host does not listen on a port. On live ports, we filter out hosts that present a well-known banner (such as Apache or Nginx).

## 3.2 Profiling C2 servers of IoT botnets

In this section, we answer the following questions:

*Q1: What is the distribution of the C2 servers across Autonomous Systems (ASes) and how this evolves over time?* (See subsection 3.2.1)

*Q2: Is there a common feature among popular Autonomous Systems (ASes)?* (See subsection 3.2.1)

*Q3: What is the lifespan of the IoT C2s based on recurrence in binary samples and in our probing?* (See subsection 3.2.2)

*Q4: How effective are the VT threat intelligence feeds in terms of comprehensiveness and timeliness?* (See subsection 3.2.3)

### 3.2.1 Hosting Environments of IoT C2s

**Some ASes are persistently more popular in hosting IoT C2s.** In order to better understand the hosting environments of IoT C2s, we look at the Autonomous Systems(AS) which the C2 IP address belongs to. In our one year observation period, we find that 10 ASes host 69.7% of the total number of all C2s servers in our *D-C2s* dataset. These C2s are listed on Table 3.2. Furthermore, 60% of these ASes consistently appear as top hosting ASes for IoT C2s during the one year of observation. The distribution of IoT C2s across the 10 most popular ASes based on the week number of the study is depicted on Figure 3.1. The distribution shows more C2s since January 2022. This is partially because we get more number of samples since that period, and partially because the tool we use (C2Miner) achieves a better activation rate. We see two interesting observations by analyzing Figure 3.1. First, IP SERVER LLC (AS-44812) and Aperion Global (AS-139884) become more active in the last 4 weeks of the study. On week 28, the highest number of C2s are from AS-44812, which is a Russian ISP. Interestingly, this is the week that Russia invaded Ukraine. Although it is tempting, a closer investigation will be needed to establish a connection between these two events. Second, we observe a peak of IoT malware samples on week 28, which leads to a peak of observed C2 addresses across all ASes.

**There are commonalities among the popular ASes for IoT c2.** Although we can not certainly claim why these Autonomous Systems (AS) are more popular for the IoT C2 servers we do see patterns among them. We use the information we find on these ASes website with the note that AS211252 do not provide any information on their website so it's excluded from further analysis. The first pattern that we observe about

these ASes is that they are all a hosting type and provide Dedicated or Virtual Private Servers (VPS) for customers. Second, all of them except one provide anti-DDOS services that is interesting given that the C2 servers are responsible for many DDoS attacks. Third, 70% of these service providers are in USA, Russia and The Netherlands. Fourth, 30% of these providers (AS53667, AS202306 and AS44812) accept cryptocurrency payments that can hide the identity of the payers.

We would like to point out that there is no correlation between the size of ASes or their service ranking and appearance of IoT C2 within. Neither of these ASes are among 100 top ASes based on the number of IPv4s they host [12]. In addition, they are not also among the top VPS and Dedicated providers [7]. In conclusion, the bot masters either like these providers and keep hosting their C2s on them or regularly find vulnerable targets within these ASes and host their C2 on compromised victim servers.

**Downloader and C2 servers are often on the same server:** We analyzed 47 distinct downloader addresses we find referred by the exploits in the *D-Exploits* dataset and only 12 downloader addresses are not identified as C2. All downloader servers host on http port 80.

### 3.2.2 Observed lifespan of C2 servers

This section reports on the temporal behavior of the C2 servers.

**IoT C2 servers seem to be short-lived and elusive.** We provide an analysis of the **observed lifespan** of the C2 servers, which we define as: the interval between the last and the first time we observe a C2 server referred by a sample. We measure the

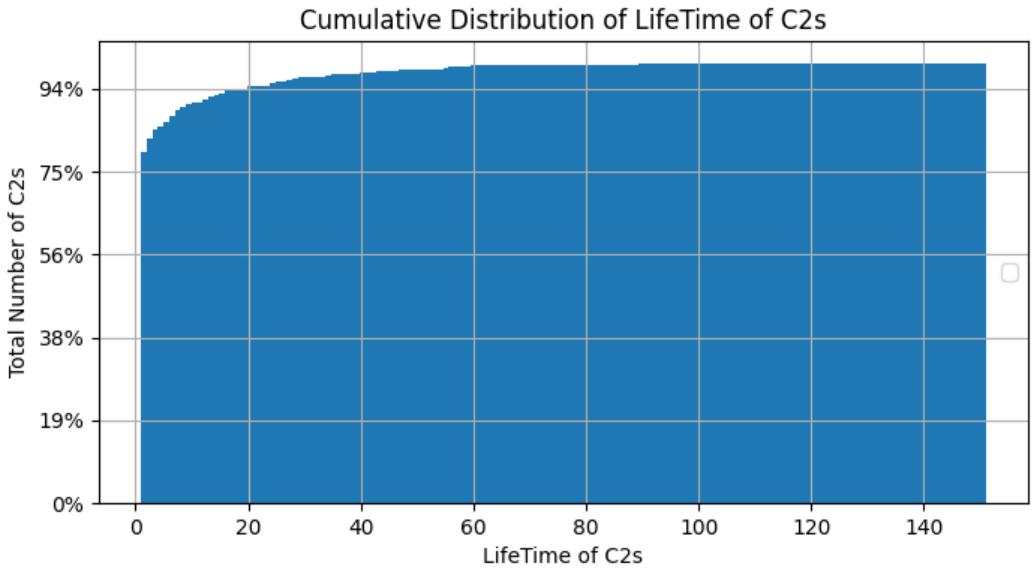


**Figure 3.2:** CDF of lifetime of C2 IPs

lifespan of C2 servers based on the *D-C2s* and *D-PC2* datasets. We analyze and report our two datasets separately below. The overarching observation is that C2 servers appear short-lived and elusive: servers are not always responding to our active probes.

**C2 servers are short-lived.** We support this assertion with two observations. First, we measure what percentage of the binaries in *D-C2s* have a live C2 server on the day they were reported to the malware repositories. We find that 60% of the samples have a dead C2 server on that day. This could be attributed to the latency in reporting malware binaries. Second, we plot the cumulative distribution of C2 IPs observed lifespan in Figure 3.2. We see that 80% of the binaries have an observed lifespan of one day with an average lifespan of 4 days. The results are qualitatively similar for DNS-based C2 addresses, which we show in Figure 3.3.

**IoT C2 servers are elusive in terms of responsiveness in our *D-PC2* dataset.** We observe an interesting behavior: C2 server responsiveness to our active probing

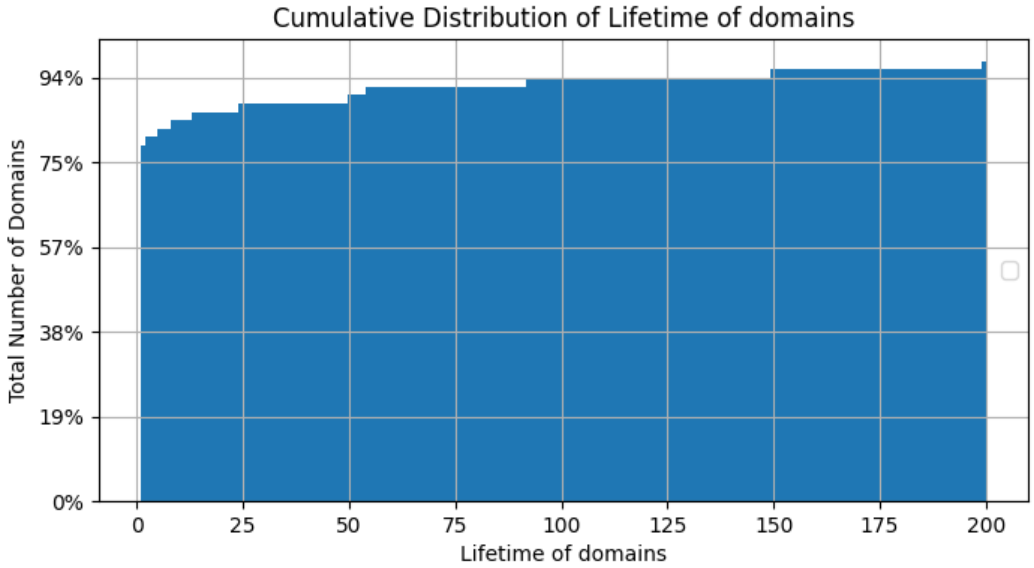


**Figure 3.3:** CDF of lifetime of C2 Domains

is spotty. In Figure 3.4, we show the responsiveness of our seven servers in our *D-PC2* dataset. Recall that we probe these servers daily with six probes and we mark as black a probe that receives a response from the C2 server. We see that C2 servers never responded to all six probes in one day. Furthermore, we see that 91% of the time a server does not respond to a second probe four hours after a successful probe. This is a strong indication that IoT C2 servers are not consistently responsive.

From a practical point of view, this observation suggests that any active probing study should be "persistent" and probe frequently to ensure accurate detection of live C2 servers.

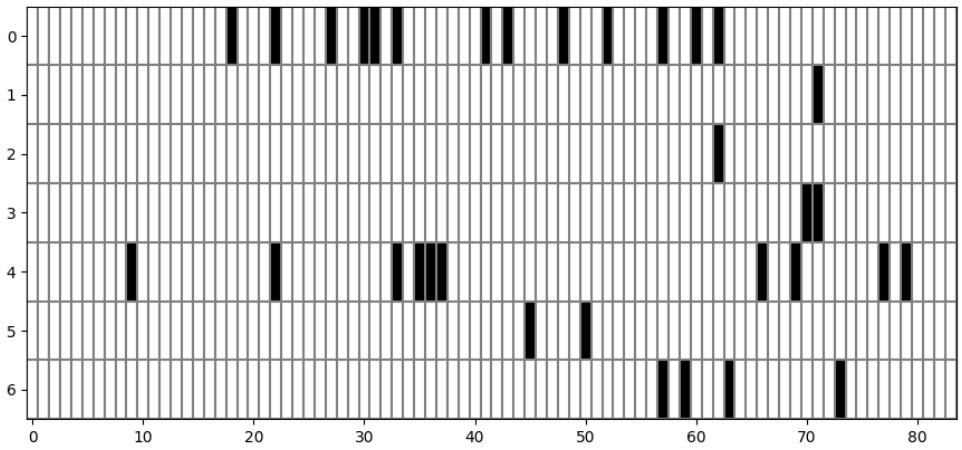


**Figure 3.4:** C2 servers are elusive: the responses of C2 servers are spotty to our 6 daily probes in the span of two weeks.

| Type | Same Day | May 7th 2022 |
|---|---|---|
| All | 15.3% | 3.3% |
| IP-based | 13.3% | 1.5% |
| DNS-based | 57.6% | 35.0% |

**Table 3.3:** The unreported C2 servers: The percentage of C2 servers that security vendors are not aware the day we discover them. Two months after the end of the experiment (May 7th 2022), these C2s are reported as malicious, which provides an indirect validation of our detection approach.

### 3.2.3 Threat intelligence effectiveness

In this section, we measure the effectiveness of the threat intelligence (TI) feeds provided by 89 vendors and shared by VT. We use the term effectiveness to refer to comprehensive and timeliness of information. Threat intelligence feeds could play an essential role in mitigating cyber-threats, if they are used in a blacklisting strategy. This is particularly true for IoT devices, which depend on the network perimeter safeguards, as many IoT devices do not have the computation resources to deploy sophisticated security solutions.

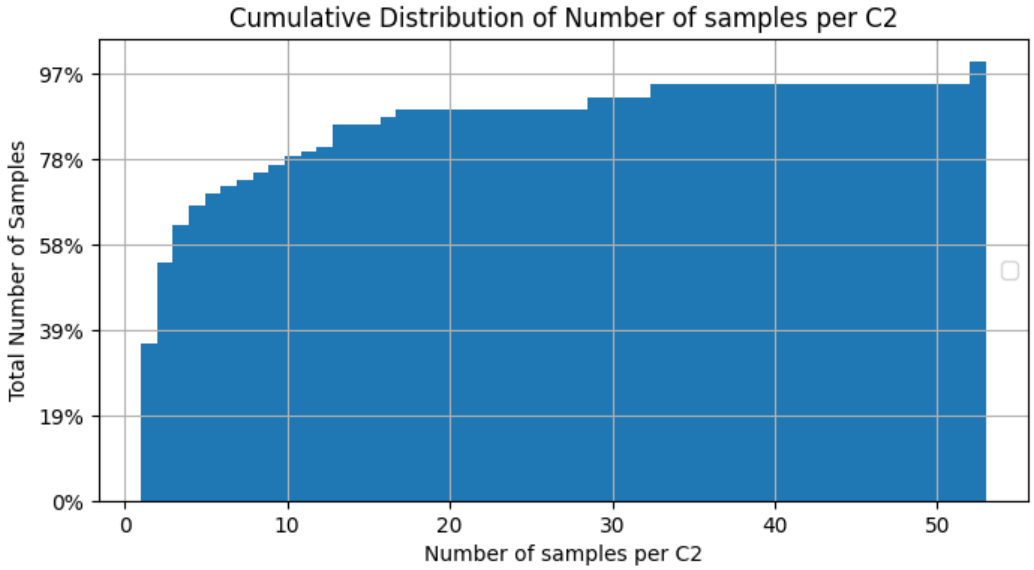


**Figure 3.5:** CDF of the number of distinct binaries that use a C2 IP address.

First, we find that **60% of C2 servers are contacted by more than one distinct binaries.** We show the cumulative distribution of number of times a C2 is contacted by different samples in *D-C2s* in Figure 3.5. We see that roughly 40% of C2 IPs are contacted by only one binary, while nearly 20% are contacted by more than 10 distinct binaries. The result for DNS names is similar in Figure 3.6. This observation shows that if we detect and "block" a C2 server based on one binary, it could help contain the effect of other binaries that use that server.

Second, we quantify the effectiveness of threat intelligence feeds using our *D-C2s* dataset. We want to measure how many of the C2 addresses we find are known to the intelligence feeds. Specifically, we count a C2 address as a miss, if it is reported not malicious by VirusTotal on the day that we discover it. We ensure that the validity of the C2 address if: (a) it is deemed malicious the second time we query VirusTotal or (b) its behavior matches that of known C2 communication patterns as we outlined in section 3.1.

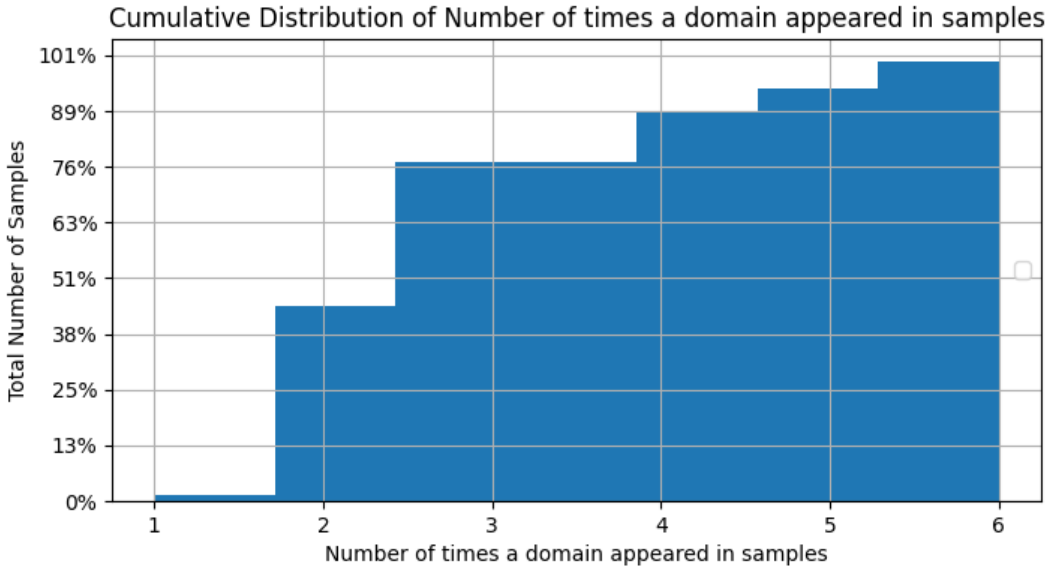


**Figure 3.6:** CDF of the number of distinct binaries that use a C2 domain

We point out that this way of measuring effectiveness is motivated by the ephemeral nature of the IoT C2s and the need for timely intelligence for containing bots in practice.

**Threat intelligence feeds fail to detect 15% of the C2 servers on the day of discovery of the binary.** The result of our measurement is reported in Table 3.3. Threat intelligence feeds are worse for DNS C2 servers defined by DNS addresses compared to servers with IP addresses. In addition, the results suggest that the reason for the miss is the lack of timeliness. Our measurement on May 7th shows that most of the missed C2 addresses will become reported malicious by threat intelligence feeds with a delay as we mentioned earlier. This is significant given the 1 day lifespan of the C2 addresses.

**Most threat intelligence feeds miss detecting even the known C2s.** The CDF of number of different vendor feeds that report a C2 address as malicious is illustrated in Figure 3.7. Out of 44 threat intelligence feeds for IoT C2 servers, 25% of the known C2 servers are reported by one or two feeds. This means that either intelligence sharing

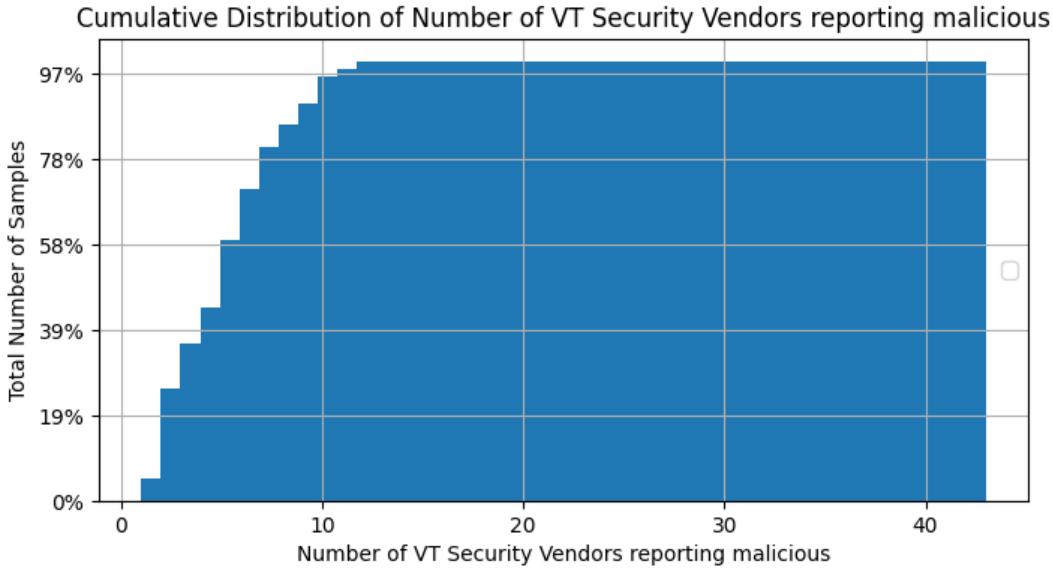


**Figure 3.7:** CDF of the number of vendors that report a known C2 server as malicious.

is absent, or it happens with a lag. Regardless of the reason, the result shows that for lower false negatives, an effective blacklist needs to aggregate data from multiple sources. However, this aggregation needs to be done carefully to avoid increasing the false positives as discussed in recent studies [105, 19].

## 3.3 Profiling IoT Malware Proliferation

In this section, we answer the following questions:

*Q5: Does any malware in our dataset exploit 0-day vulnerabilities?*

*Q6: What sources IoT device pen-testers should use to have a more accurate result?*

*Q7: Is there a recent exploitation of already disclosed vulnerabilities?*

*Q8: What are the most popular vulnerabilities based on the number of samples?*

| ID | Vulnerability | Exploit ID | Publication Date | Target Device | # Samples |
|---|---|---|---|---|---|
| 1 | CVE-2018-10561 | EDB-44576 | May 3, 2018 | GPON Routers | 139 |
| 1 | CVE-2018-10562 | EDB-44576 | May 3, 2018 | GPON Routers | 129 |
| 2 | CVE-2015-2051 | EDB-ID-37171 | February 23, 2015 | D-Link Devices | 132 |
| 3 | CVE-2017-18368 | N/A | May 2, 2019 | ZyXEL | 38 |
| 4 | Vacron NVR RCE | OPENVAS:1361412562310107187 | October 11, 2017 | Vacron NVR | 46 |
| 5 | CVE-2017-17215 | EDB-43414 | March 20, 2018 | Huawei Router HG532 | 1 |
| 6 | MVPower DVR Shell unauthenticated RCE | EDB-ID-41471 | February 27, 2017 | MVPower DVR TV-7104HE | 74 |
| 7 | CVE-2021-45382 | N/A | December 19, 2021 | D-Link DIR-820L command injection | 3 |
| 8 | Linksys unauthenticated RCE | EDB-ID-31683 | February 16, 2014 | Linksys E-series devices | 2 |
| 9 | WAN Side RCI | EDB-ID-40740 | November 8, 2016 | Eir D1000 Wireless Router | 9 |
| 10 | CVE-2018-20062 | EDB-45978 | December 11, 2018 | Devices that use ThinkPHP | 2 |
| 11 | CVE-2016-5680 | EDB-ID-40200 | August 31, 2016 | NUUO NVRmini2 / NVRsolo / Crystal Devices / NETGEAR ReadyNAS | 1 |
| 12 | Netlink GPON Router RCE | EDB-48225 | March 18, 2020 | Netlink GPON Routers | 2 |

**Table 3.4:** A description of the vulnerabilities that were exploited by the malware in our *D-Exploits* dataset.

**IoT malware authors rely on the exploitation of known and old vulnerabilities.** We find the exploitation of 14 vulnerabilities that are all known for a while. In more detail, these vulnerabilities and descriptions about them are listed on Table 3.4. These vulnerabilities are 3 years old on average. Five of these vulnerabilities do not have an assigned CVE number. although they have publicly available exploits. On the other hand, two of the vulnerabilities have CVEs assigned but do not have publicly available exploits.

From a practioners point of view, we suggest that **penetration testers should refer to multiple sources for IoT devices.** None of the popular vulnerability and exploit databases, NVD, EDB and OPENVAS, cover all the exploited vulnerabilities. Threfeore, one would need to consider all three sources to ensure the full vulnerability assessment of a device.

**IoT malware authors keep adding new exploits but not necessarily for new vulnerabilities.** One interesting observation is the additions of two new exploits for CVE-2016-5680 and CVE-2021-45382 compared to a study in 2020 [10]. While CVE-2021-45382 was disclosed after that study, CVE-2016-5680 has been known for 6 years and just recently has become exploited. This confirms a similar finding reported by that study [10].

**The most popular vulnerabilities are not the newest ones.** We rank vulnerabilities based on the number of binaries that use them in our datasets. The top four popular vulnerabilities (CVE-2015-2051, CVE-2018-10561, CVE-2018-10562 and MVPower DVR Shell RCE) are at least 4 years old. We point out that our dataset contains only the recently reported samples, and these are vulnerabilities that were first used by IoT malware in the year they were disclosed [10]. This suggests that despite their age, these vulnerabilities are preferred by hackers. The popularity of old vulnerabilities is an indirect challenge of the emphasis that security researchers place on the zero-day vulnerabilities

**The variance of popularity among vulnerabilities is high.** Not surprisingly, we observe that some vulnerabilities are more popular than others. Here we opted for a visual and temporal view of their pupolarity. We show the number of binaries in *D-Exploits* per day that exploit a vulnerability in Figure 3.8. We see four vulnerabilities that are

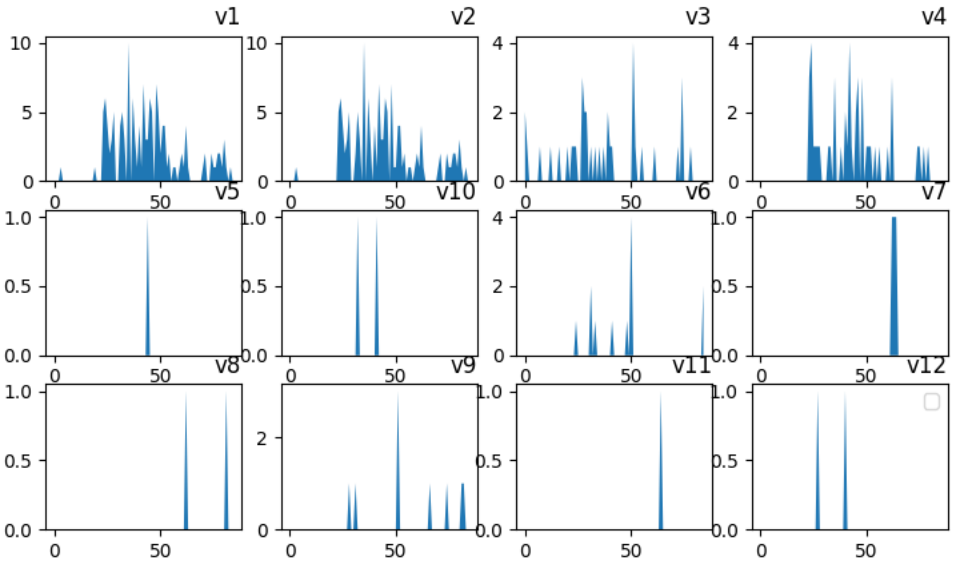


**Figure 3.8:** The number of binaries per day that target each one of the 12 vulnerabilities .

consistently and heavily used by binaries, while the rest of the vulnerabilities have shorter and less intense usage.

Additionally, we analyze the exploits of the vulnerabilities to discover any similarities. We observe two patterns. First, most of the exploits are similar, and seem to use the same template with variations on the downloader server address and the loader (the file that downloads the malware and executes on the victim) name. Second, the peculiarity of the loader names and their frequency suggests that authors use the same loader repeatedly in different exploits. To illustrate this, we plot the frequency of loader names from *D-Exploits* in Figure 3.9.

## 3.4 Profiling IoT malware attacks

In this section, we answer the following questions:

*Q9: What types of DDoS attacks are being launched by the IoT malware?* (See subsection 3.4.1)

*Q11: What protocols are the targets of IoT malware DDoS attacks?* (See subsection 3.4.2)

*Q10: Who are the targets of the IoT malware DDoS attacks?* (See subsection 3.4.3)

We use our *D-DDOS* dataset and track the issuance of DDoS commands from 6 malware variants across three malware families: Mirai (two variants), Gafgyt (two variants) and Daddy33t (two variants). In total, we observe 42 attacks issued by 17 distinct C2 servers to 20 of our malware binaries. The C2 servers are located in 6 different countries. We find that servers in USA, Netherlands and Czech Republic were responsible for 80% of the attacks. Two of the C2 servers (107.174.24.16 and 192.236.248.222) were not listed as malicious by VirusTotal on the day that the attacks were launched which is aligned with our observations in Table 3.3. Furthermore, this suggests that if our real-time eavesdropping had translated into actions, one could have mitigated or blocked these two attacks.

**Attack-launching C2 servers have longer observed lifespan.** We wanted to investigate further the C2 servers that launched attacks. Interestingly, the C2 servers with recorded DDoS attacks have a longer observed lifespan compared to the rest of C2s in our dataset. The attack-launching servers have an average lifetime of roughly 10 days which is longer than the overall lifespan average of 4 days (see subsection 3.2.2).

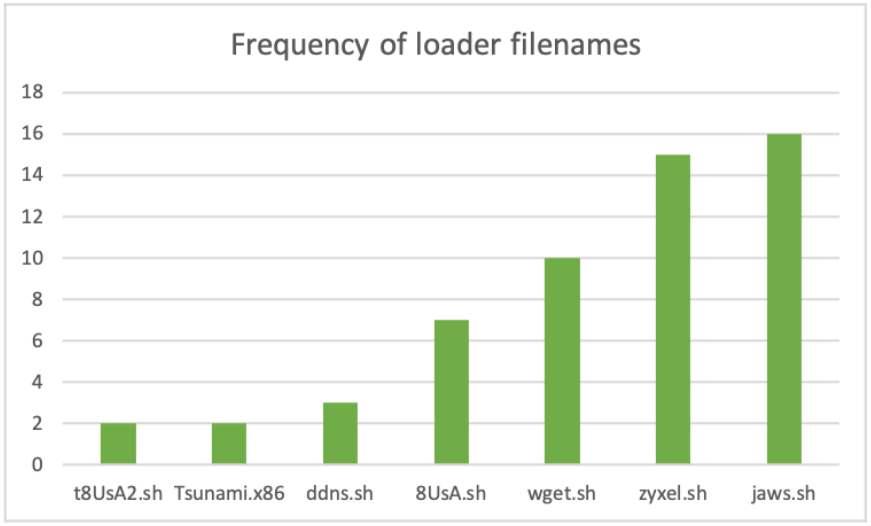


**Figure 3.9:** The number of binaries that use each loader file names in our *D-Exploits* dataset.

### 3.4.1 Types of observed attacks

We observe 8 types of DDoS attacks based on their issued commands. These attacks vary in the way they are mounted, and their target network protocol We review each attack based on their observed network behavior below.

**UDP DDoS Attack:** This is the most common type of DDoS attack and appears in all three malware families with different names. In this type of attack, the target is flooded with continuous packets at one or multiple UDP port(s). Although the implementation of this attack is similar in all three malware families, there are subtle differences that we describe next.

Mirai uses value "0" in the DDOS command to refer to this attack. Original implementation of this attack published with the source code of Mirai receives a target address, source port, destination address and the time length of the attack and floods the

target for the given time length. In our measurement, we saw implementations of Mirai that receive the target IP and port but show different behaviors regarding the choice of the source port. Some variants use the same port during the attack, while other variants use multiple source ports. The payload of the attack is the the null byte ($00h$).

On the other hand, Gafgyt uses the string $UDP$ and daddyl33t uses $UDPRAW$ to refer to this attack. Similar to one variant of Mirai, they receive a target address and a target port, and select a source port that remains the same throughout the attack. The payload of the attack is the same as Mirai.

**SYN Flood Attack:** In this type of attack, a target is flooded at one or multiple TCP port(s) repeatedly with the first packet of the TCP handshake (SYN flag set). We saw instances of this attack ordered by daddyl33t and Mirai botnets that we describe below.

In the case of daddyl33t, C2 sends a $HYDRASYN$ command that includes the target IP and port. The bots attack the target with multiple source ports. In case of Mirai, we saw two different implementations of this attack: (a) multiple source ports targeting the same destination port, (b) multiple source ports targeting multiple destination ports.

**TLS attack:** In this type of attack, a service that uses TLS is targeted. The computation on the server side is more resource intensive compared to the client, and hence it is possible to overwhelm the server. We see two implementation of this attack by daddyl33t and Mirai that we compare below.

Samples of daddyl33t family seems to target a UDP port possibly running DTLS, and send an encoded message repeatedly. On the other hand, Mirai completes the TCP

handshake with the target, sends a large message in different chunks and then sends a RST flag and starts over.

**BLACKNURSE attack:** This type of DDoS attack targets the ICMP protocol. The bots sends unsolicited IMCP type 3 (Destination Unreachable) packets to the target to overwhelm it. We only see daddyl33t employing this type of attack.

**STOMP attack:** This attack targets the application layer protocol STOMP that uses TCP transport. The attacker completes the TCP handshake and then floods the target with fake STOMP requests that contain junk data.

**VSE attack:** This DDoS attack targets the Valve Source Engine of the Steam game platform [93]. It is a UDP amplification attack where the bot sends TSource Engine Query requests to a gaming server. This attack first appeared with the release of Mirai source code but we see one instance of this attack launched by the Gafgyt malware.

**STD attack:** In this attack, the target is flooded with random strings. We saw one instance of this attack launched by gafgyt towards a UDP port. The random string is generated once, and then used repeatedly throughout the attack towards a target.

**NFO attack:** This attack specifically targets NFO servers hosting vendor that manufactures its own servers [80]. Our claim is based on two observations. First, the target of the attack is an IP address that belongs to this vendor AS. Second, the attack mentions the NFOV6. We are not sure what vulnerability the attack tries to exploit but we see a custom payoload for the attack that targets port 238 UDP on the target IP. This type of attack by IoT malware has been reported before [106]. In our dataset, we see a launch of this attack by daddyl33t.

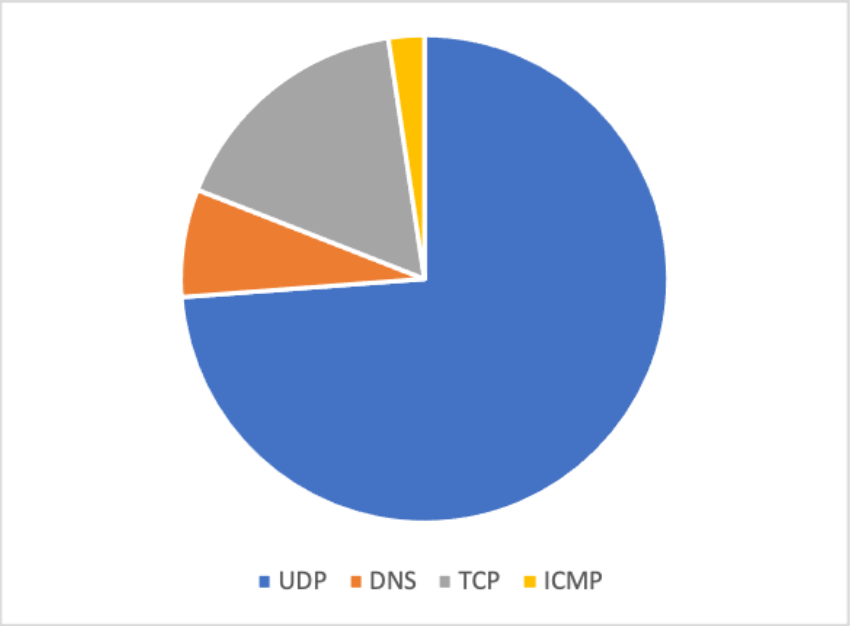


**Figure 3.10:** Distribution of DDoS attacks by target protocol: UDP-based attacks are dominant.

### 3.4.2 DDOS attack traffic

The DDoS attacks we observe target 4 protocols: UDP (excluding DNS), TCP, DNS and ICMP. The distribution of the attacks is illustrated in Figure 3.10. The vast majority (74%) of the attacks target a service (excluding DNS) on top of the UDP protocol. That said, because of the nature of UDP flood DDoS attacks, we can not certainly say whether a service has been targeted or the target IP. As we mentioned in the previous section, except one case, all attacks receive the target port as part of the attack command. As we don't have access to the C2 code, we speculate that the C2 splits all the target UDP ports and then distributes them to the bots. That said, 21% of the attacks target port 80 (mostly on UDP), and 7% target port 443 that are the default ports for the HTTP and HTTPS respectively.

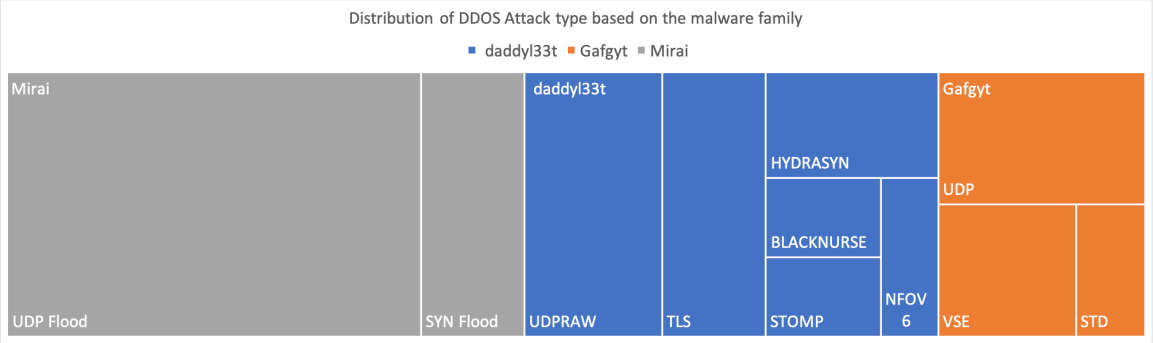


**Figure 3.11:** Distribution of DDoS attack type based on the malware family. Mirai (grey) has more attacks, Daddyl33t is second and is more diverse in the types of attacks, and Gafgyt (orange) has fewer attacks.

**One target hit by multiple attacks.** We analyzed the binaries and types of attacks One interesting observation is that, 25% of the targeted IP addresses are attacked using two different attack types in a single session. For instance, 142.x.x.109 on port 4567 UDP, was once attacked by the TLS, and then shortly after by HYDRASYN. Another example, is target 172.x.x.77, that was first attacked by TLS on port 443 UDP, and then BLACKNURSE targeting the ICMP protocol.

### 3.4.3 Targets of the attacks

We observe a few patterns in the targets of the DDoS attacks. We analyze the Autonomous Systems of the target victims to detect patterns of similarity. The first pattern is about the type of AS that the victim is located at. Targets are located in 23 Autonomous Systems that span 11 countries. 45% of these ASes are Internet Service Providers (ISP), and 36% are Hosting providers. The rest of the attacks target businesses: Google, Amazon and Roblox. These results are aligned with findings in a previous work[81]. An interesting observation is the game industry orientation of the Businesses and the hosting providers.

18% of the ASes are specialized in the computer gaming industry. Figure 3.12 shows the location of the targets and the type of AS hosts.

## 3.5 Discussion and Limitations

We discuss limitations, extensions, and practical issues.

**a. How does our approach fit in the cyber-security ecosystem?** Our ambitious plan is to turn our approach into a service with an ongoing monitoring effort. With this in mind, we see MalNet as a building block in the ecosystem of tools and services against the scourge of IoT malware and botnets. The role of our approach within this ecosystem could be as follows: (a) **IoT Honeypots and malware feeds** collect the binaries that are actually being propagated; (b) **Public and private sources** can exchange information with our service regarding the reputation of IP addresses; (c) **Firewall and NIDS** can incorporate rules and signatures regarding suspicious network behavior provided by our service, **ISPs and web hosters** can use our improved list of malicious actors (bots and C2 servers) in their networks, (d) **Traffic monitoring services** can use our malware signatures and identified compromised devices to identify the potential volumetric network traffic generated by botnets. All of those entities of course could be providing useful information to validate, expand and improve our capabilities.

**b. Are our malware binaries relevant and representative?** This is the typical question for any empirical study, that can mostly be answered indirectly. First, we collect the malware binaries *as soon as* they are made available by feeds that aggregate a large number of sources. Second, the fact that some of the C2 servers that we find are not already in threat intelligence feeds is an indication that we find reasonably fresh binaries

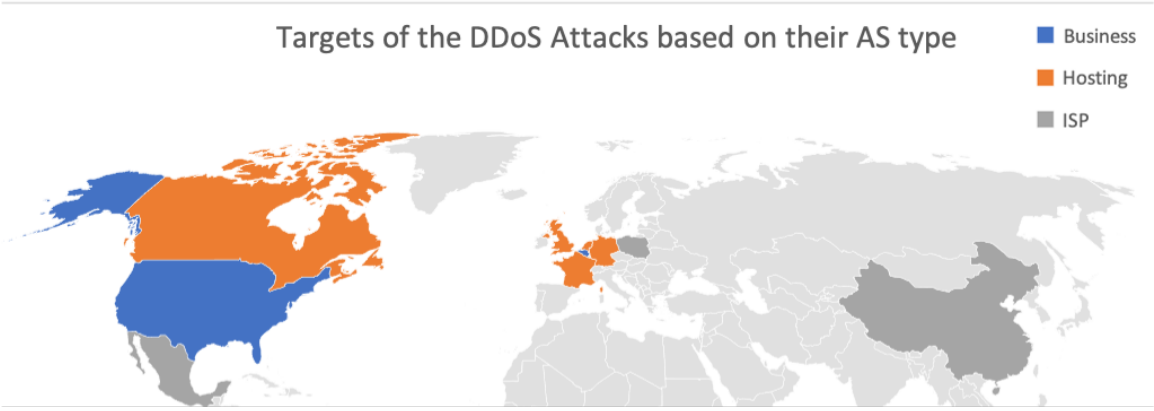


**Figure 3.12:** Targets of DDoS Attacks Based on the location and the Autonomous System type. A country is colored according to nature of the majority of its targets.

and unknown C2 servers. Third, our malware covers several major malware families, such as Gafgyt, Tsunami, and Mirai, as we saw in Table 3.1 and discussed in a few places earlier. In a future large scale study, we would like to experiment and compare not only the newly reported binaries, but conduct a longitudinal study of malware of different years.

**c. How statistically reliable and generalizable are our the observations?** This is a question that all measurement studies need to grapple with. First, all our observations are measurement driven and can only describe the observed behavior within the datasets that we collected. With that in mind, we have clearly explained how we collected and created our datasets. Given our interest to analyze newly reported malware, that limited the number of binaries that a study with a historical perspective may have had available. We find that examining the information that newly reported malware has to offer is an important study, which in fact, seems to not have been done in the past in the way we conduct it here.

**d. Could this approach be deployed in practice and at a large-scale?** The current work provides a proof of concept that shows the promise of our binary-centric approach. Our

ambition is to deploy it at large scale, which will have some challenges. Such a deployment would require us to: (a) expand the supported architectures, (b) adapt and continuously update state of the art anti-evasion techniques in our sandbox, and (c) collaborate with ISPs and cloud providers for massive probing. Note that these challenges include some that are active research problems in their own right, and some that are mostly engineering tasks.

**e. Can our approach extend to non-IoT malware?** The overarching approach applies to any malware that can be activated in a sandbox. In addition, some of our current methods for identifying the C2-bound traffic may need to be adapted in the case of different malware. However, with some customization, the overarching framework could be made to expand to additional types of malware and platforms. Note that our focus on IoT malware is motivated by the fact that IoT malware is newer, less studied but with an increasing presence and potential for harm.

## 3.6 Related Work

There are several categories of related work to our research. Below, we discuss each category and explain how this research is different. Overall, none of the related efforts has focused on a binary centric approach that provides a holistic view on the network behaviors of the IoT malware

**a. IoT malware non-network behavior analysis.** Studying the behavior of IoT malware has become a hot topic both for academia and industry [30, 31]. These work focus more on the system level behavior of the malware, namely the operating system layer, and the types of techniques that malware employees to evade detection, determine what type of device is the host, and make itself persistent. Understanding the system level behavior of

the malware is complementary, but significantly different from the network behavior which is our focus here.

**b. IoT malware network behavior analysis.** Some efforts focus on characterizing the behavior of a single malware family like Mirai [11, 46] or Hajime [50] while a few others characterize the behaviors of several malware families at the same time [32, 82]. We fall in the latter group. Although the previous efforts explore the network behavior of the IoT malware to some extent, neither they are binary-centric nor they provide a holistic view on all three types of IoT malware traffic as we do here. Many previous studies [46] [82] [32] do not profile the proliferation or the attack phase of a botnet. Another study [11] provides an analysis of the DDoS attacks, but they rely on ISP traffic, which is an intersting and challenging problem in its own right: given network traffic, one has to identify and characterize the attack traffic. By contrast, we follow a binary-centric approach, which allows attribution of the attacks to the C2 servers and the malware binaries that initiate and carry out the attack.

**c. IoT malware proliferation behavior analysis:** A few related work explore the proliferation behaviors of the IoT malware [68, 10, 54]. While we share many similarities with these work, we focus on the recent malware, and hence recent trends in terms of exploits and vulnerabilities. In addition, we provide a holistic view including the C2 behaviors and the DDoS attacks.

**d. C2 Communication Analysis:** Several studies analyze C2 server communication from a networking point of view [78, 74, 43, 94, 91, 35]. Although interesting and informative, these studies focus on understanding the infrastructure that supports the

botnet operation. By contrast, we provide a binary-centric measurement study with comprehensive profiling of proliferation and attack activity. In addition, we are the first to illustrate IoT C2 servers are elusive using active probing techniques.

**e. Studying DDoS Attacks:** Several studies focus on different aspects of DDoS attacks [81, 60, 58]. An earlier work [81] studies the victims of the DDoS attacks, and another work [60] studies the servers who issue the attack commands. Both studies use honeypots. On the other hand, a recent work [58] studies the effectiveness of booters takedown operations by analyzing network data from ISPs. Our fundamental difference is in our binary-centric approach. By only using malware binaries and spying into IoT malware botnets, we provide a similar analysis on the victims and origins of the DDoS attacks and get similar results as the previous work.

## 3.7 Conclusion

Our study can be seen as a proof of concept of different type of malware analysis that focuses on: (a) dynamic analysis, (b) timely collected malware, and (c) comprehensive profiling of all major bot activities. We collect daily and analyze on the same day the newly reported IoT malware from VirusTotal and MalwareBazaar. A binary-centric study can create a holistic picture of the IoT by connecting a binary and its family, with live C2 servers, a set of proliferation techniques, and even actual launched DDoS attacks.

First, we quantify the elusive behavior of C2 servers: 91% of the time a C2 server does not respond to a second probe sent four hours after a successful probe. We also find that 15% of the live servers that we find are not known by threat intelligence feeds available on VirusTotal. Second, we find that the IoT malware relies on fairly old vulnerabilities in

its proliferation. Our binaries attempt to exploit 12 different vulnerabilities with 9 of them more than 4 years old, while the most recent one was 5 months old. Third, we observe the launch of 42 DDoS attacks that span 8 types of attacks while we observe that a target is often hit by two different attacks.

The overarching goal is to show the potential of a binary-centric dynamic analysis of malware with a focus on newly discovered binaries. Our preliminary results show the information and insights that can be obtained. Our future goal is to expand the scope of the study into a large-scale continuous IoT malware monitoring infrastructure.

Finally, we want to reinforce our committed to sharing our tools and our data. In fact, we would attempt to create a set of reference datasets with continuous updates and with the help of willing members of the community.

# Chapter 4

# DECAF++: Elastic Whole-System Dynamic Taint Analysis

Dynamic taint analysis (also known as dynamic information flow tracking) marks certain values in CPU registers or memory locations as *tainted*, and keeps track of the tainted data propagation during the code execution. It has been applied to solving many program analysis problems, such as malware analysis [101, 59, 99], protocol reverse engineering [21], vulnerability signature generation [79], fuzz testing [85], etc.

Dynamic taint analysis can be implemented either at the process level, or at the whole system level. Based on process-level instrumentation frameworks such as Pin [67], Valgrind [76], and StarDBT [16], process-level taint analysis tools like LibDft [55], LIFT [84], Dytan [24] and Minemu [17] keep track of taint propagation within a process scope. Whole-system taint analysis tools (e.g., TaintBochs [23], DECAF [49] and PANDA [39]) are built upon system emulators (e.g., Bochs [70] and QEMU [14]), and as a result can keep track of taint propagation throughout the entire software stack, including the OS kernel and all the running processes. Moreover, whole-system dynamic taint analysis offers a better transparency and temper resistance because code instrumentation and analysis are completely

isolated from the guest system execution within a virtual machine; in contrast, process-level taint analysis tools share the same memory space with the instrumented process execution.

However, these benefits come at a price of a much higher performance penalty. For instance, the most efficient implementation of whole-system taint analysis to our knowledge, DECAF [49], incurs around 6 times overhead over QEMU [49], which itself has another 5-10 times slowdown over the bare-metal hardware. This overhead for tainting is paid constantly no matter how much tainted data is actually propagated in the software stack.

To mitigate such a performance degradation introduced by dynamic taint analysis, some systems dynamically alternate between the execution of program instructions and the taint tracking ones [84, 51]. For instance, LIFT [84] is based on the idea of alternating execution between an original target program (fast mode) and an instrumented version of the program containing the taint analysis logic. Ho et al. proposed the idea of *demand emulation* [51], that is, to perform taint analysis via emulation only when there is an unsafe input.

Despite the above, there are still some unsolved problems in this research direction. First, LIFT [84] works at the process level, which means LIFT has the aforementioned shortcomings of process level taint analysis. Moreover, LIFT still has to pay a considerable overhead for checking registers and the memory in the fast mode. Second, the demand emulation approach [51] has a very high overhead in switching between the virtualization mode and the emulation mode [51]. Third, some optimization approaches depend on specific hardware features for acceleration [84, 55, 17].

In this paper, we propose solutions to solve these problems, and provide a more flexible and generally applicable dynamic taint analysis approach. We present DECAF++, an enhancement of DECAF with respect to its taint analysis performance based on these solutions. The essence of DECAF++ is *elastic* whole-system taint analysis. Elasticity, here, means that the runtime performance of whole-system taint analysis degrades gracefully with the increase of tainted data and taint propagation. Unlike some prior solutions that rely on specific hardware features for acceleration, we take a pure software approach to improve the performance of whole-system dynamic taint analysis, and thus the proposed improvements are applicable to any hardware architecture and platform.

More specifically, we propose two independent optimizations to achieve the elasticity: elastic taint status checking and elastic taint propagation. DECAF++ elasticity is built upon the idea that if the system is in a safe state, i.e., there is no data from taint sources, there is no need for taint analysis as well. Henceforth, we access the shadow memory to read the taint statuses only when there is a chance that the data is tainted. Similarly, we propagate the taint statuses from the source to the destination operand only when any of the source operands are tainted.

We implemented a prototype dubbed DECAF++ on top of DECAF. Our introduced code is around 2.5 KLOC including both insertions and modifications to the DECAF code. We evaluated DECAF++ on nbench, SPEC CPU2006, and Apache bench. When there are tainted bytes, we achieve 202% (18% to 328%) improvement on nbench integer index, and on average 66% improvement on apache bench in comparison to DECAF. When

there are no tainted bytes, on SPEC CPU2006, our system is only 4% slower than the emulation without instrumentation.

**Contributions** In summary, we make the following contributions:

- We systematically analyze the overheads of whole system dynamic taint analysis. Our analysis identifies two main sources of slowdown for DECAF: taint status checking incurring 2.6 times overhead, and taint propagation incurring 1.8 times overhead.
- We propose an elastic whole-system dynamic taint analysis approach to reduce taint propagation and taint status checking overhead that imposes low constant and low transition overhead via pure software optimization.
- We implement a prototype based on elastic tainting dubbed DECAF++ and evaluate it with three benchmarks. Experimental results show that DECAF++ incurs nearly zero overhead over QEMU software emulation when no tainted data is involved, and has considerably lower overhead over DECAF when tainted data is involved. The taint analysis overhead of DECAF++ decreases gracefully with the amount of tainted data, providing the elasticity.

## 4.1 Related Work

### 4.1.1 Hardware Acceleration

Related works on taint analysis optimization focus on a single architecture [84, 55, 17]. Henceforth, they utilize the capabilities offered by the architecture and hardware to accelerate the taint analysis. LIFT uses x86 specific LAHF/SAHF instructions to accelerate the context switch between the original binary code and the instrumented code. Minemu

uses X86 SSE registers to store the taint status of the general purpose registers, and fails if the application itself uses these registers. libdft uses multiple page size feature on x86 architectures to reduce the Translation Lookaside Buffer (TLB) cache miss for the shadow memory. In this work, we stay away from these hardware-specific optimizations, and rely only on software techniques to make our solution architecture agnostic.

### 4.1.2 Shadow Memory Access Optimization

Minimizing the overhead of shadow memory access is crucial for the taint analysis performance. Most related works reduce this overhead by creating a direct memory mapping between the memory addresses and the shadow memory [55, 84, 17]. This kind of mapping removes the lookup time to find the taint status location of a given memory address. The implementation of the direct mapping requires a fixed size memory structure. This fixed size structure to store the taint status is practical only for 32-bit systems; to support every application, such an implementation requires 32 TB of memory space on 64 bits systems [55]. Even on 32-bit systems, the implementation usually incurs a constant memory overhead of around 12.5% [55]. Minemu furthers this optimization and implements a circular memory structure that rearranges the memory allocation of the analyzed application. The result is that it quickly crashes for applications that have a large memory usage [17]. In this work, we aim to follow a dynamically managed shadow memory that can work not only for applications with large memory usage but also for 64-bit applications.

In addition to the above, LIFT [84] coalesces the taint status checks to reduce the frequency of access to the shadow memory. To this end, LIFT needs to know ahead of time what memory accesses are nearby or to the same location. This requires a memory

reference analysis before executing a trace. LIFT scans the instructions in a trace and constructs a dependency graph to perform this analysis. LIFT reports that this optimization is application dependent (sometimes no improvement), and depends on what percentage of time the taint analysis is required for the application. LIFT is a process level taint analysis tool, and in case of whole system analysis, we expect the required taint analysis percentage for a program to be very low in comparison to the size of the system. Henceforth, we do not expect this optimization to be very useful for whole system analysis given the constant overhead of performing memory reference analysis for the entire system.

### 4.1.3 Decoupling

Several related works reduce the taint analysis overhead by decoupling taint analysis from the program execution [52, 72, 71, 62, 53]. ShadowReplica [52] decouples the taint analysis task and runs it in a separate thread. TaintPipe [72] parallelizes the taint analysis by pipelining the workload in multiple threads. StraightTaint [71] offloads the taint analysis to an offline process that reconstructs the execution trace and the control flow. LDX [62] performs taint analysis by mutating the source data and watching the change in the sink. If the sink is tainted, the change in the source would change the sink value. LDX reduces the overhead by spawning a child process and running the analysis in the spawned process on a separate CPU core. RAIN [53] performs on-demand dynamic information level tracking by replaying an execution trace when there is an anomaly in the system. RAIN reduces the overhead by limiting the replay and the analysis to a few processes in the system (within the information flow graph) based on a system call reachability analysis. Our work complements these works as we aim to separate the taint analysis from the original execution.

### 4.1.4 Elastic Tainting

**Elastic taint propagation** Similar ideas to elastic tainting have been explored in the previous works [84, 51]. Qin et al. introduced the idea of fast path optimization [84]. Fast path optimization is based on the notion of alternating execution between a target program and an instrumented version of the program including the taint analysis logic. The former is called check execution mode, and the latter is called track execution mode. Qin et al. presented this idea for process level taint analysis. We build upon this idea for whole system analysis. While the intuition behind both elastic tainting and fast optimization is the same, our work advances Qin et al.'s work for the whole system.

Our first novelty is that we reduce the overhead in the check mode. LIFT [84] checks registers and memory locations of every basic block regardless of the mode. Note that this overhead in system level analysis is a major issue because it affects every process (and the kernel) as we show in subsection 4.2.3. We reduce the overhead by releasing DECAF++ from checking registers in the check mode and instead monitor the taint sources, data from input devices or memory locations, directly. Combining this with our low overhead taint checking, we reduce the overhead in the check mode to nearly zero as we show in subsection 4.5.4. Our second novelty is that, unlike LIFT [84], we implement elastic tainting in an architecture agnostic way. Meeting this requirement while obtaining a low overhead is technically challenging. As an example, LIFT uses a simple jump to switch modes while such a jump in our case would panic the CPU since a single guest instruction might break into several host binary instructions that need to be executed atomically.

Finally, the effectiveness of elastic taint propagation for whole system taint analysis has not been investigated before. As we show in section 4.5, this optimization for whole-system taint analysis is application dependent and needs to be accompanied with a taint status checking optimization. Our work is the first that shows the elastic property through comprehensive evaluations, and provides a means to compare the elastic taint propagation with elastic taint status checking.

**Elastic taint status checking** Ho et al. present the idea of demand emulation [51] that has elements in common with our elastic taint status checking. The demand emulation idea is to perform taint analysis via emulation only when there is an unsafe input e.g. network input in their case. Otherwise, the system is virtually executed without extra overhead. Demand emulation idea looks enticing because the virtualization overhead is usually lower than emulation. However, as Ho et al. state, the transition cost between virtualization and emulation is quite high and possibly offsets the speedup gained through the virtualization. In contrast, as we show in section 4.5.4, our elastic tainting incurs almost zero transition overhead. Further, Ho et al. had to modify the underlying target operating system to provide efficient support for demand emulation. In contrast, in this work, we implement elastic tainting using only emulation without any modifications to the target systems, and show substantial improvements in real world applications of taint analysis.

## 4.2 Whole-System Dynamic Tainting

In this section, we introduce a basic background knowledge on DECAF, mainly focusing on its taint analysis functionality related to this work. For further details on QEMU, the underlying emulator, we refer the readers to **??**.

### 4.2.1 Taint Propagation

DECAF defines how instructions affect the taint status of their operands. Going to the details of the rules for every instruction is out of the scope of the current work; an interested reader can refer to [98]. Just to give an idea, *mov* instructions in x86 result in a corresponding *mov* of the taint statuses from the source to the destination (see Figure 4.1). What is important about the DECAF taint propagation is that DECAF inserts a few instructions before every Tiny Code Generator (*TCG*) IR instruction that do the following:

- Read the taint status of the source operand. The taint status depending on the operand type can be in the shadow registers, temporary variables or in the shadow memory.
- Decide the taint status of the destination operand based on the instruction tainting rule. To implement the tainting rule, a few TCG IRs are inserted before each IR to propagate the taint status.
- Write the taint status of the destination operand to its shadow variable.

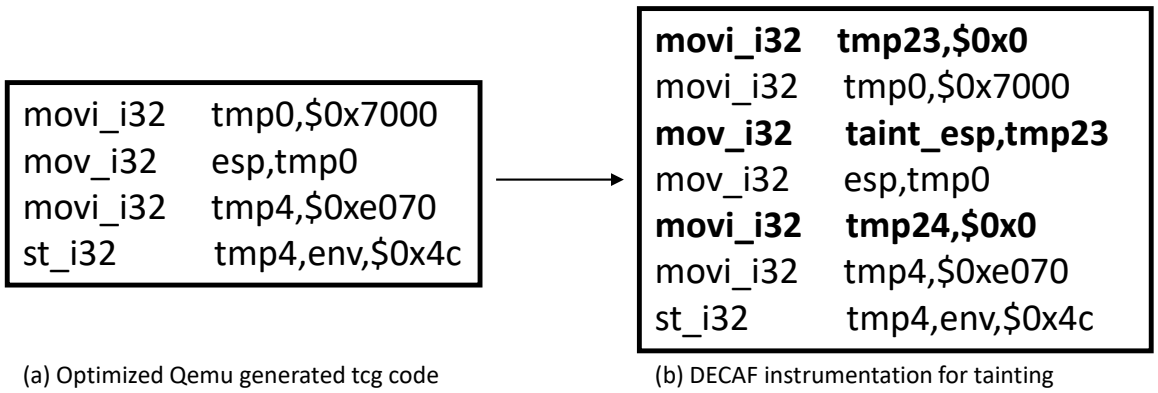


**Figure 4.1:** DECAF tcg instrumentation to apply tainting for the instruction *mov $0x7000, %esp*. (a) shows the tcg IR after translating the guest *mov* instruction, and (b) shows the tcg IR after applying the taint analysis instrumentation.

### 4.2.2 Shadow Memory

DECAF stores the taint statuses of the registers and the memory addresses respectively in global variables and in the shadow memory (allocated from heap). DECAF does not make any assumptions about the memory and can support any application with any memory requirements. The shadow memory associates the taint statuses with *guest physical addresses*. This is a key design choice because the taint analysis is done at the system level, and hence, virtual addresses point to different memory addresses in different processes.

DECAF stores the taint statuses in a two-level tree data structure. The first level points to a particular page. The second level stores the taint statuses for all the addresses within a physical page. This design is based on the natural cache design of the operating systems, and hence makes use of the temporal and spatial locality of memory accesses.

DECAF instruments QEMU memory operations to maintain the shadow memory. Instrumentation is in fact on the Tiny Code Generator (TCG) Intermediate Representation

(IR). DECAF instruments the IR instruction for memory load, *op_qemu_ld**, and the IR instruction for memory store, *op_qemu_st**. For load, DECAF loads the taint status of the source operand of the current instruction along with the memory load operation. For store, DECAF stores the taint status of the destination operand of the current instruction to its corresponding shadow memory along with the memory store operation. In both cases, the load (or store) is to (or from) a global variable named *temp_idx*.

### 4.2.3 Taint Analysis Overhead

We analyzed the current sources of slowdown in DECAF. There are three sources of slowdown in DECAF:

- The QEMU emulation overhead that is not inherent to the DECAF taint analysis approach but rather an inevitable overhead that enables dynamic whole system analysis. That said, QEMU is faster than other emulators like Bochs[70] by several orders of magnitude [14].
- The taint propagation overhead as explained in subsection 4.2.1.
- The taint status checking as explained in subsection 4.2.2.

After applying DECAF tainting instrumentation, the final binary code is on average *3 times* the original QEMU generated code according to Table 4.1. Clearly, the additional inserted instructions (after the instrumentation) impose an overhead.

We systematically analyzed the overheads of DECAF framework, i.e., taint propagation and taint status checking. To measure each overhead, we isolated the codes from DECAF that would cause the overhead by removing other parts. For taint propagation over-

**Table 4.1:** The statistical summary of the blownup rate after the instrumentations. The numbers show the ratio of the code size to a baseline after an instrumentation. QEMU baseline is the guest binary code, and DECAF baseline is QEMU IR code. DECAF increases the already inflated QEMU generated code size around 3 times on average.

| System | Component | Min | Median | *Mean* | Max |
|---|---|---|---|---|---|
| **QEMU** | lifting binary to tcg | 3.33 | 6.75 | 7.13 | 28.00 |
| **DECAF** | taint checking & propagation | 1.25 | 3.12 | 2.94 | 5.14 |

head measurement, we removed the shadow memory operations by disabling the memory load and store patching functionality of DECAF that adds the shadow memory operations. For taint status checking overhead measurement, we removed the taint propagation functionality from DECAF by deactivating the instrumentation that implements the taint propagation rules.

We measured the performance of the isolated versions of DECAF using nbench benchmark on a windows XP guest image with a given 1024MB of RAM. The experiment was performed on an Ubuntu 18 i686 host with a Core i7 3.5GH CPU and 8GB of RAM. Figure 4.2 illustrates the result of our analysis. Figure 4.2 reports the geometric mean of the nbench reported indexes normalized using a baseline. The baseline is the DECAF without the taint analysis functionality outright, that is, DECAF with only Virtual Machine Introspection (VMI). The result shows that on nbench, taint analysis slows down the system 400%. But more important than that, Figure 4.2 shows that taint propagation alone slows down the system about 1.8 times while taint status checking alone adds a 2.5 times slowdown.

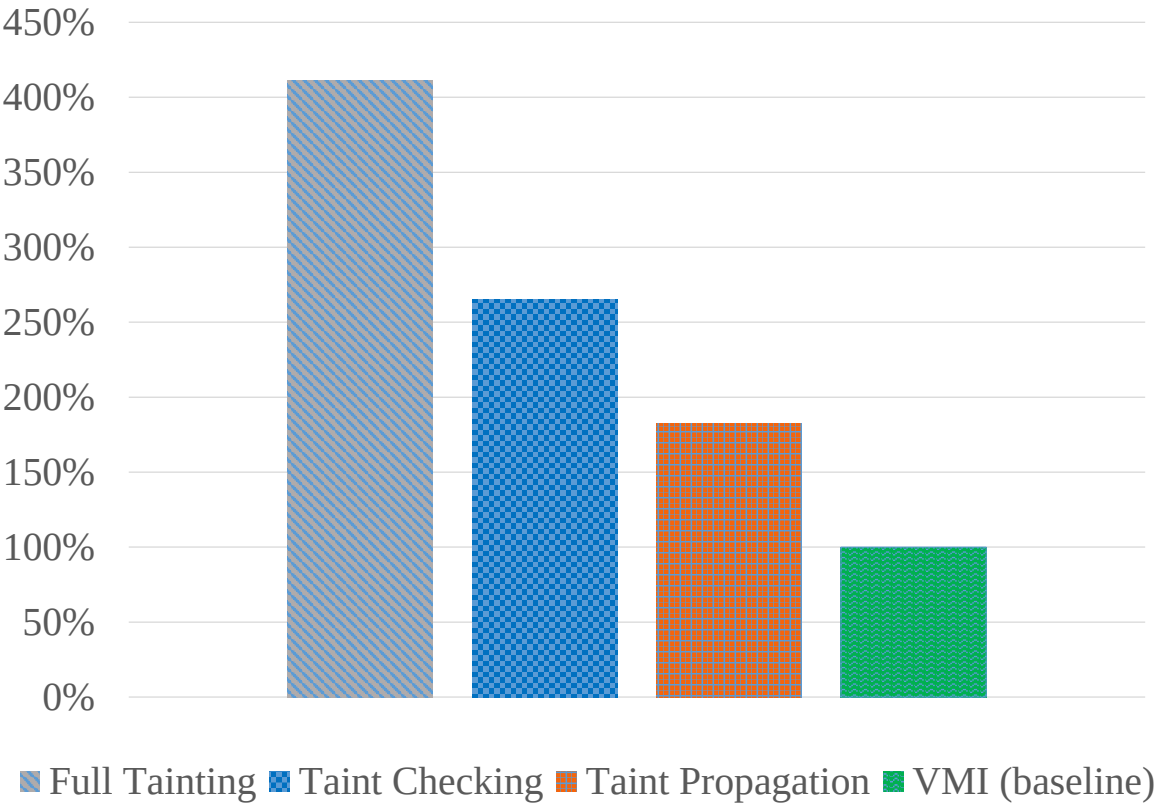


**Figure 4.2:** Breakdown of overhead given the DECAF VMI as baseline.

## 4.3 Elastic Taint Propagation

### 4.3.1 Overview

Elastic taint propagation aims to remove the taint propagation overhead whenever possible. It is based on the intuition that taint analysis can be skipped if the taint analysis operation does not change any taint status value. Taint analysis operations can possibly result in a change only when either of the source or the destination operand of an instruction is tainted.

**Two modes** Based on the above intuition, we define two modes with and without the taint propagation overhead. We name the mode with taint propagation operations *track*

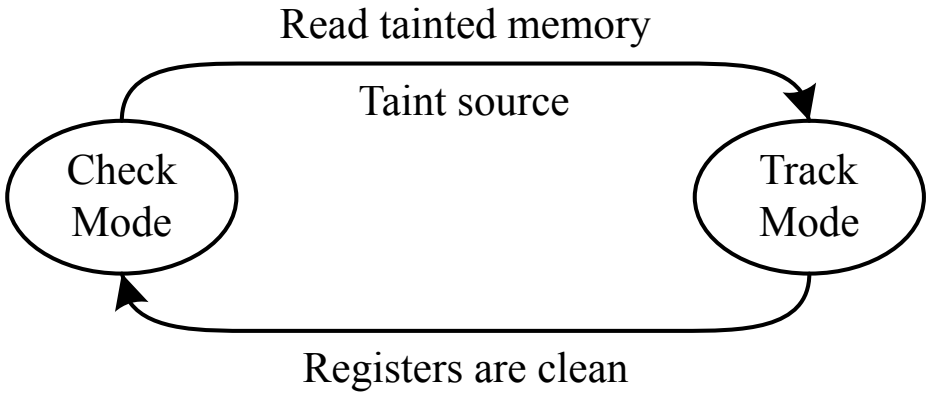


**Figure 4.3:** State transitions between check mode and track mode

*mode*, and the mode without taint propagation operations *check mode*. At any given time, the execution mode depends on whether any CPU register is tainted.

**Mode transition** When the system starts, no tainted data exists in the system, so the system runs in the check mode. The execution switches to the track mode when there is an input from a taint source. The taints propagate in the track mode until the propagation converges and the shadow registers are all zero (clear taint status). At this point, the execution switches back to the check mode. Finally, either based on an input or a data load from a tainted memory address the execution switches back to the track mode. Figure 4.3 shows when transition occurs between the track and the check mode.

### 4.3.2 Execution Modes

An execution mode determines the way a block should be instrumented. Each mode has its set of translation blocks. Further, each mode has its own cache tables. The execution in a mode can flow only within the same mode translation blocks. This means that blocks only from the same mode would be chained together. We determine the mode

using a flag variable. Based on the mode, the final code would be instrumented with or without the taint propagation instructions. The generated code will be reused for execution based on the execution mode unless invalidated. A cache invalidation request invalidates the generated codes regardless of the mode. The set of translation blocks and code caches virtually form an exclusive copy of the translated code for the execution mode.

**Check mode** The generated code in the check mode is the original guest code (program under analysis) plus the instrumentation code for memory load and memory store. For memory load, shadow memory is checked, and if any byte is tainted, the mode is switched to the track. In section 4.4, we further explain how we efficiently perform this checking. For memory store, the destination operand taint status will be cleaned because any propagation in this mode is safe.

**Track mode** Code generated in track mode is the same as the one generated in the original DECAF. Readers can refer to subsection 4.2.1 and subsection 4.2.2 for further details.

### 4.3.3 Transition

A key challenge after having two execution modes in place is to decide when and how the transition between the two modes should occur. The transition between the two modes should affect neither the emulation nor the taint analysis correctness. The execution should immediately stall in the check mode and resume in the track mode when there is a data load from a taint source. Further, this mode switch should happen smoothly without panicking the CPU.

We need to monitor data flow from the taint sources and the shadow registers for timely transition between the modes. In the check mode, we only monitor the taint sources. This is a key design choice for performance because monitoring registers for timely transition is very costly. Note that to implement register monitoring in the check mode, we would need to check the register taint statuses before *every instruction*. Thus, in the check mode, to reduce the overhead, we only monitor the taint sources without loosing the precision. In the track mode, we can check the registers less often because longer execution in this mode neither affects the taint analysis precision nor the emulation correctness.

**Input devices monitoring** The taint sources are generally memory addresses of the *input* devices like keyboards or network cards. For the input devices, DECAF++ relies on the monitoring functionality implemented in DECAF. However, we slightly modify the code to raise an exception whenever there is an input. This exception tells the system that it is in an unsafe state because of the user input and the track mode should be activated if not before.

**Memory monitoring** In addition to the input devices, we should also monitor the *memory load* operations. This is because after processing the data from an input device, the data might propagate to other memory locations and pollute them. We need to track the propagation in the check mode as soon as a tainted value is loaded from memory for further processing. Efficient design and implementation of memory monitoring is a key to our elastic instrumentation solution. We elaborate on how we do this efficiently in section 4.4.

**Registers monitoring** Monitoring the registers is a key to identifying when we can stop the taint propagation. If none of the registers carry a tainted value, no machine operations except the memory load can result in a tainted value. Henceforth, we can safely stop the execution in the track mode and resume the execution in the check mode while carefully monitoring the memory load operations as explained earlier.

We point out that monitoring registers in the track mode has low overhead. This is because in the track mode, we can tolerate missing the exact time that the registers are clean without affecting either the safety or the precision (we would propagate zero). Therefore, DECAF++ can check the cleanness of the registers in the track mode at block (instead of instruction) granularity.

We check the registers' taint status either after the execution of a chain of blocks, or when there is an execution exception (including the interrupts). Our experiments confirm that this is a fine granularity given its lower overhead comparing to an instruction level granularity approach. If all the registers have a clean taint status, we resume the execution for the next blocks in the check mode.

**Transition from check mode to track mode** Unlike the transition from the track mode to the check mode (always in the beginning of a block), the transition from the check to the track can happen anywhere in the block depending on the position of an I/O read or a load instruction. However, the execution of a single guest instruction should start from the beginning to the end, note that a single guest instruction might be translated to several TCG instructions [1]; otherwise, the result of the analysis would be both invalid and

[1]For instance, a single x86 "ADD m16, imm16" instruction will be translated to three TCG IR instructions; one load, one add and one store

unsafe. This is because the block code copies in each mode are different and the current instruction might have dependency on the former instructions that are not executed in the current mode. To have a smooth transition, the following steps should be followed:

1. Restore CPU state: before we switch the code caches, we need to restore the CPU state to the last successfully executed instruction. We need to restore the CPU state to avoid state inconsistency. Since the corresponding execution block in the other mode is different, we can not resume the execution from the same point; the same point CPU state is not consistent with the new mode block instructions. To restore the CPU state, we re-execute the instructions from the beginning of the block to the last successfully executed guest instruction. This will create a CPU state that can be resumed in the other mode.

2. Raise exception: after restoring the CPU state, we emulate a custom exception: *mem_tainted*. We set the exception number in the *exception_index* of the emulated CPU data structure. After that, we make a long jump to the QEMU main loop (*cpu-exec* loop).

3. Switch mode: in the QEMU main execution loop, we check the *exception_index* and change the execution mode if the exception is *mem_tainted*. We switch the mode by changing mode flag value that instructs us how we should instrument the guest code (track or check).

After switching the modes, QEMU safely resumes the corresponding block execution in the new mode because we restored the CPU state in the step 1.

## 4.4 Elastic Taint Status Checking

The main idea behind reducing the taint status checking overhead is to avoid unnecessary interactions with the shadow memory. In DECAF, the taint status checking happens for every memory operation. However, we can avoid the overhead per memory address if we perform the check for a larger set of memory addresses. Thus, if the larger set doesn't contain any tainted byte we can safely skip the check per address within that set. The natural sets within a system are physical memory pages.

DECAF++ scans physical pages while loading them in TLB, and decides whether or not to further inspect the individual memory addresses. We modified the TLB filling logic of QEMU according to Figure 4.4. The modifications are highlighted. The figure illustrates that if the page contains any tainted byte, DECAF++ sets a shadow memory handler for the page through some of the TLB entry control bits. Afterwards, whenever this page is accessed, DECAF++ redirects the requests to the shadow memory handler. In the following paragraphs, we explain how we handle memory load and store operations separately because of their subtle differences.

**Memory load** During memory load operations, we should load the taint status of the source memory address operands as well. We load the taint status value from the shadow memory only when the TLB entry for the page contains the shadow memory handler. In other cases, we can safely assume that the taint status is zero. Based on this notion, we modify the QEMU memory load operation logic as shown in Figure 4.5. In particular, two cases might occur:

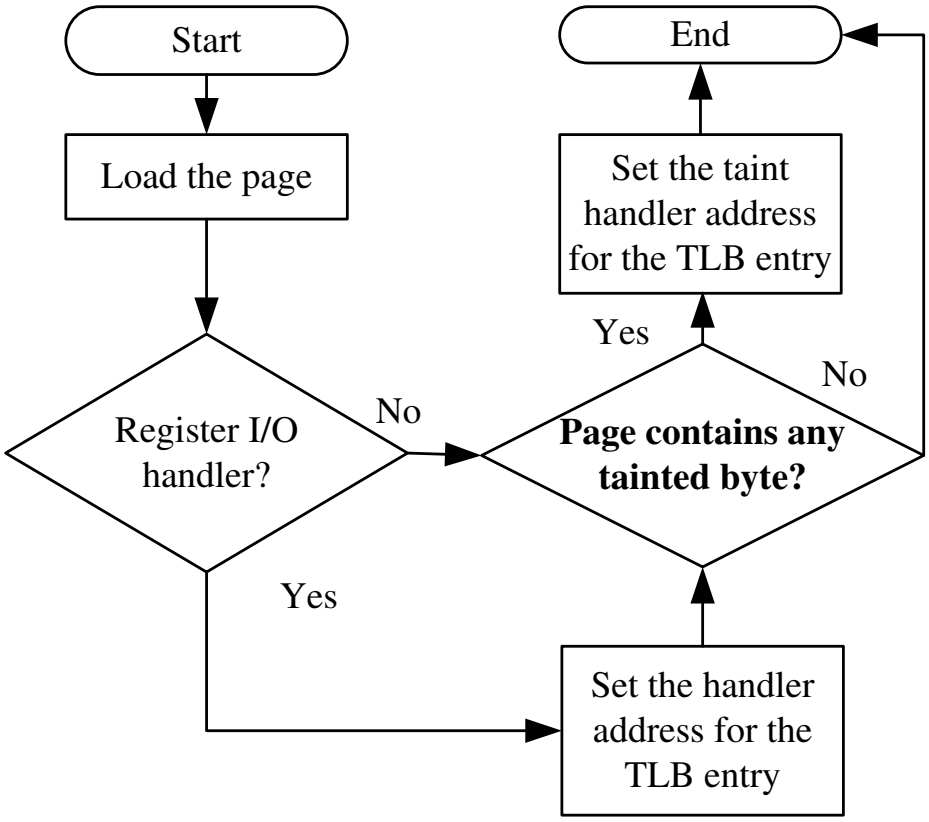


**Figure 4.4:** Fill TLB routine

- If the TLB entry control bits for the page contain the shadow memory handler, the address translation process for the memory load operations results in a TLB miss. In the TLB miss handler routine, if the control bits indicate that the shadow memory handler should be invoked we do so and load the taint status for the referenced address from the shadow memory. If the execution is not already in the track mode, and the loaded status is not zero we quickly switch to the track mode.
- If the address translation process for the load operation results in a TLB hit, we check whether we are in the track mode, and if so we load zero as the taint value status.

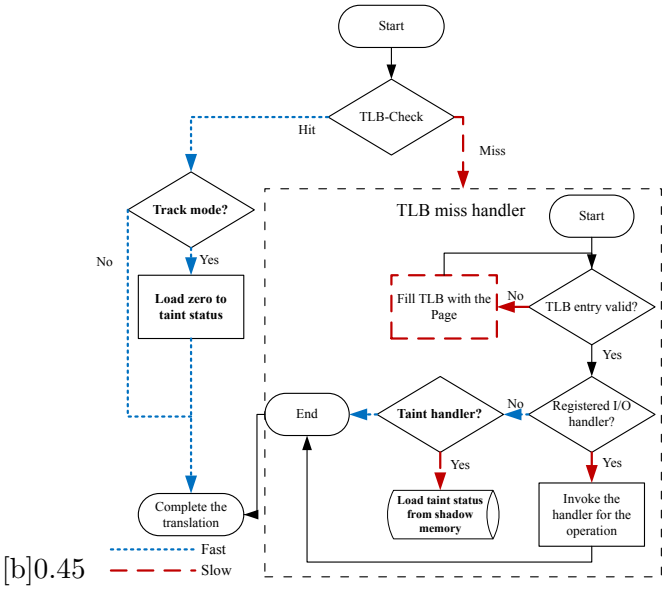


**Figure 4.5:** QEMU load memory operation

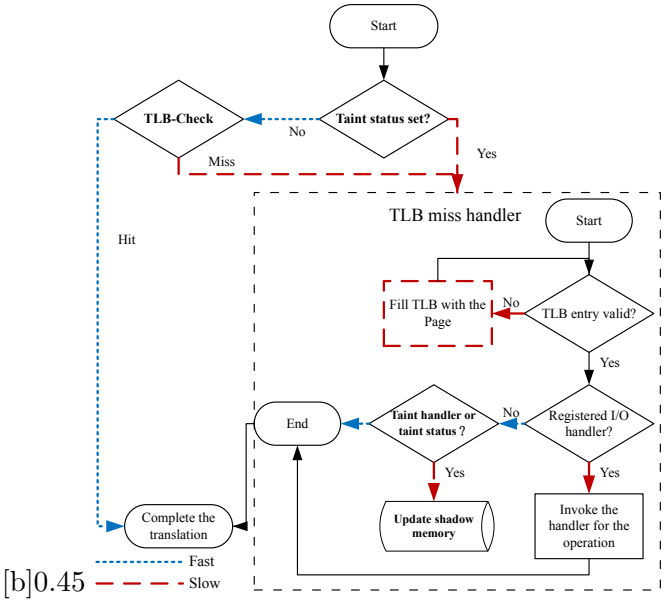


**Figure 4.6:** QEMU store memory operation

**Figure 4.7:** Elastic shadow memory access workflow

Otherwise, we don't need to load the taint status since it will not be used for taint propagation.

Based on the locality principle, a majority of accesses should go through the fast path shown in the Figure 4.5. Our elastic taint status checking is designed not to add overhead to this fast path, and hence a performance boost is expected.

**Memory store** During memory store operations, we should store the taint status of the source operand to the shadow memory address of the destination referenced memory address. Similar to memory loads, we perform the shadow memory store operation only when the TLB entry control bits for the referenced address page indicate so. That said, there is a subtle difference that makes memory stores costlier than memory loads. Figure 4.6 shows how we update the shadow memory alongside the memory store operations. In particular, there will be three cases:

- If the source operand for the store operation has a zero taint status, and the page TLB entry does not indicate a tainted page, we do nothing. This happens both in the check mode all the time, and in the track mode when the page to be processed does not contain any tainted byte.
- If the page TLB entry flags us to inspect the shadow memory, we check the TLB control bits in the TLB miss handler and update the shadow memory if the shadow memory handler is set (see Figure 4.6).
- If the taint status of the source operand is not zero, even if the page is not registered with a shadow memory handler, we still update the shadow memory. This can only

happen in the track mode, because in the check mode there is no taint propagation, and hence the source operand taint status is always zero. We update the TLB entry when this is the first time there is a non-zero taint value store to the page. Since the page now contains at least a tainted byte, all the next memory operations involving this page should go through the shadow memory handler. We update the TLB entry and register the shadow memory handler for the future operations.

**Propagation of non-tainted bytes** A special case of the taint status store is when a tainted memory address is overwritten with non-tainted data. This case happens when the TLB entry for the page flags shadow memory operation even when the source operand is not tainted. In such a case, the memory address taint status would be updated to zero but the page is still processed as unsafe; the memory operations still will go through the taint handler. For performance reason, we do not immediately reclaim the data structure containing the taint value, but rely on a garbage collection (see **??**) mechanism that would be activated based on an interval. The page remains unsafe until the garbage collector is called. The garbage collector walks through the shadow memory data structure and frees the allocated memory for a page if no byte within the page is tainted. After this point, Fill TLB routine will not set the taint handler for the page anymore, and any processing involving the page will take the fast path.

## 4.5 Evaluation

In this section, we evaluate DECAF++, a prototype based on the elastic whole system dynamic taint analysis idea. DECAF++ is a fork of DECAF project including the

introduced optimizations. Overall, our changes (insertion or deletion of code) to develop our prototype on top of DECAf does not exceed 2.5 KLOC. We defined two compilation options that selectively allows activating elastic taint propagation or elastic taint status checking. We evaluate DECAF++ to understand:

1. How effective each of our optimization, i.e., elastic taint propagation and elastic taint status checking and altogether is in terms of performance.

2. Whether our system achieves the elastic property for different taint analysis applications, that, is a gradual degradation of performance based on the increase in the number of tainted bytes.

3. What the current overheads of DECAF++ are and whether we can address the shortcomings of the previous works [84, 51], i.e., reducing the overhead in the check mode and in the transition between the two modes.

### 4.5.1 Methodology

We measure the performance metrics using standard benchmarks under two different taint analysis scenarios in subsection 4.5.2 and subsection 4.5.3. In both scenarios, a virtual machine image is loaded in DECAF++ and a benchmark measures the performance of the virtual machine while the taint analysis task is running.

To answer (1), we measure the performance of DECAF++ with different optimizations, i.e., with elastic taint propagation dubbed as *Propagation*, with elastic taint status checking dubbed as *Memory* and with the both dubbed as *Full* and compare them. To answer (2) and evaluate the elastic property, we introduce a parameter in our taint analysis

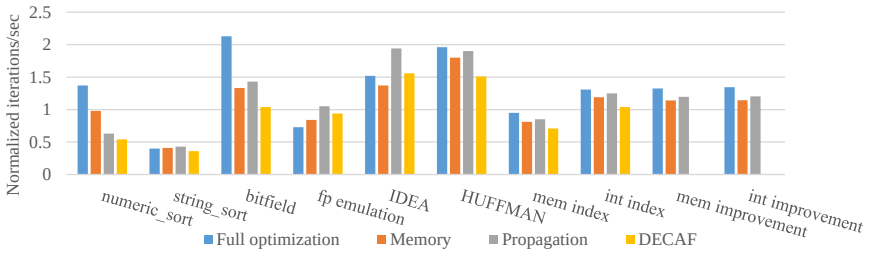


**Figure 4.8:** Comparing DECAF and DECAF++ performance.

plugin that adjusts the number or the percentage of tainted bytes. Plotting the performance trend based on this parameter values allows answering question (2). Finally to answer (3), we measure the overhead of the frequent or costly tasks in our implementation. In the the rest of this section, we describe the details and answer (1) and (2) in subsection 4.5.2 and subsection 4.5.3. In subsection 4.5.4, we answer (3).

### 4.5.2 Intra-Process Taint Analysis

In this scenario, we track the flow of information within a single process. For this experiment, we use nbench benchmark [40]. We track the flow of information within the nbench programs using a taint analysis plugin we developed for DECAF. The goal is to be able to report the performance indexes measured by nbnech while the taint analysis task is running. In the next paragraphs, we explain the configurations for the experiment, and at the end of this section we report the results.

**nbench** Understanding nbench is important since our taint analysis plugin instruments it. nbench has 10 different programs. These programs implement a popular algorithm and measure the execution time on their host. Although the underlying algorithms are different, they all follow the same pattern regarding loading the initial data. They all create an array

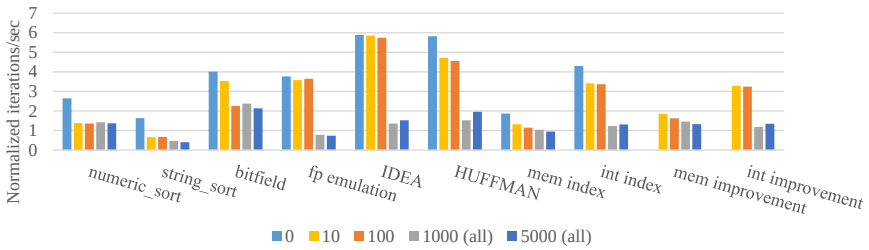


**Figure 4.9:** Evaluation of DECAF++ with full optimization under different number of tainted bytes; performance values are normalized by nbench based on a AMD k6/233 system.

of random values and then run the algorithm on that array. The arrays are allocated from the heap, and the random generator is a custom pseudo random generator.

**Taint analysis plugin** The taint analysis plugin instruments nbench programs and taints a portion of the initial input data based on a given parameter. Although this is not trivial, we do not go into the details. We just mention that we record the address of the allocated array from the heap, and taint a portion of the array right after the random initialization. The portion size depends on an input parameter that we call *taint_size*. For instance, *taint_size*=100 means that 100 bytes of the array (from the beginning) used in the running programs of nbench are tainted using the plugin. After the instrumentation, DECAF automatically tracks the propagation from the the taint sources to other memory locations.

**Experiments setup** We measure the performance of each solitary optimization feature (and together) of DECAF++ using nbench[2] on a loaded windows XP guest image. The image is given 1024MB of RAM. The experiment was performed on an Ubuntu18 i686 host with a Core i7 3.5GH CPU and 8GB of RAM. The reported indexes by nbench are the mean

[2]Three programs Fourier, NEURAL NET and LU DECOMPOSITION from the nbench did not reflect any change in their reported numbers so we removed them from analysis. Also due to cross compilation, Assignment test did not work on Windows XP.

result of many runs (depending on the system performance). Further, nbench controls the statistical reliability of the results and reports if otherwise. Finally, note that since all the measurements are conducted on the VM after it is loaded, the VM overhead would be a constant that is the same for all the measurements.

**Result** Figure 4.8 shows the performance of different optimization in DECAF++ in comparison to DECAF. The results in this figure answers question (1) for this scenario. Overall, when the entire program inputs are tainted, combining elastic taint propagation and elastic taint status checking (full optimization) achieves the best performance. However, the performance is application dependant and sometimes a single optimization can achieve better performance than both combined, e.g. HUFFMAN and IDEA.

Figure 4.9 shows the performance of the DECAF++ when both optimization are activated for varied taint_size values. This figure answers question (2): DECAF++ has the elastic property, that is, the performance degrades based on the number of tainted bytes. *On average, in comparison to DECAF, DECAF++ achieves on average 55% improvement (32% to 86%) on the nbench memory index and 202% (18% to 328%) on the nbench integer index.*

Further, we can see from Figure 4.9 that results for taint_size={10,100} are similar and differ from the result for taint_size={1000,5000}. For taint_size={10,100}, since the tainted bytes are adjacent, the track mode activates for a sequence of bytes and then quickly switches back to the check mode. Also since taint_size={10,100} is well below a page size, the shadow memory access penalty would be low because often the tainted bytes will be within a single page. However for taint_size={1000,5000}, almost the entire nbench

programs input array is tainted that results in frequent execution in the track mode and shadow memory access penalty.

In addition to the above, an interesting observation is the performance degradation for the sort algorithms. The performance degrades abruptly while taint_size changes from zero to greater values. This is because of the behavior of the sort algorithm, that is, frequently moving an element in the array. This behavior results in polluting the entire array quickly and hence degrading the performance abruptly.

### 4.5.3 Network Stack Taint Analysis

In this scenario, the taint analysis tracks the flow of information from the network throughout the entire system and every process that accesses the network data. Performing taint analysis in this level is only possible using whole-system taint analysis tools like DECAF. Since the taint analysis affects the entire system, the need of having an elastic property would be more necessary.

Honeypots are an instance of the applications that can greatly benefit from the elastic property. Previous studies show that the likelihood of the malice of a network traffic can be predetermined [34]. Therefore, a honeypot can adopt a policy to achieve taint analysis only for network traffic that are expected to be malicious. Elastic property helps such systems to boost their performance based on their policy.

We measure transfer rate and throughput based on a parameter called taint_perc (instead of taint_size in the previous experiment) that defines the percentage of network packets to be tainted. This is because percentage, here, better represents the real applica-

tions. For instance, for honeypots, the taint_perc can be easily derived based on the taint policy.

**Taint analysis plugin** Our taint analysis plugin taints the incoming network traffic based on the taint_perc parameter. We implemented this plugin using the callback functionality of the DECAF. Our plugin registers a callback that is invoked whenever the network receive API is called. Then, based on the taint_perc parameter, our plugin decides whether to taint the payload or not.

**Experiments setup** The experiments were performed on an Ubuntu16.04 LTS host with a Core i7 6700 3.40GHz×8 CPU and 16GB of RAM. The guest image was Ubuntu 11.10 and it was given 4GB of RAM. For throughput measurement, we use Apache 2.2.22. We isolate the network interface between the server (guest image running Apache) and the client (the host machine) to reduce the network traffic noise that might perturb the results. That said, there is still a large deviation in the throughput because of the non-deterministic interrupt processing behavior of the system. We rely on significantly different values considering the standard deviation to draw conclusions.

We use netcat to measure the transfer rate. Our measurement is based on the transfer rate for 200 netcat requests of size 100KB. We use apache bench [1] to measure the throughput of an apache web server on the guest image. We execute Apache bench remotely from the host system with a fixed 10000 request parameter. Apache bench sends 10000 requests and reports the average number of completed requests per second. For both

**Table 4.2:** Network transfer rate of solitary features of DECAF++ (and together as Full) on Netcat; the throughput is the mean of 5 measurements for a range of tainted bytes.

| Tainted Bytes | Implementation | Transfer Rate (MB/S) | Standard Deviation |
|---|---|---|---|
| 40KB - 50KB | Full | 3.57 | 18% |
| | Memory | 3.60 | 10% |
| | Propagation | 3.43 | 8% |
| 20KB - 30KB | Full | 5.70 | 3% |
| | Memory | 4.25 | 14% |
| | Propagation | 3.70 | 12% |
| 0KB | Full | 5.31 | 5% |
| | Memory | 5.24 | 7% |
| | Propagation | 4.12 | 8% |
| 0KB - 50KB | DECAF | 3.70 | 9% |
| 0KB - 50KB | QEMU | 6.00 | 3% |

transfer rate and the throughput, we repeat the experiments for each taint_perc parameter value 5 times and report the average and the relative standard deviation.

**Transfer rate result** Table 4.2 reports the result of our transfer rate measurement using netcat. The results show the transfer rate for three taint_perc parameter values: when every packet is tainted (40KB - 50KB tainted bytes), when half of the number of packets are tainted (20KB - 30KB) and finally when no packet is tainted. Note that although every request is 100KB, only a portion of the packet is payload, and not the entire 100KB payload would be live in the system at the same time; this is why eventually only around 50KB is tainted. *The results show a substantial 54% improvement when only half of the incoming packets are tainted.* This is only 5% less than the QEMU transfer rate that is the maximum we can achieve. There is no improvement when every packet is tainted but this is expected because taint propagation and taint status checking have to be constantly done.

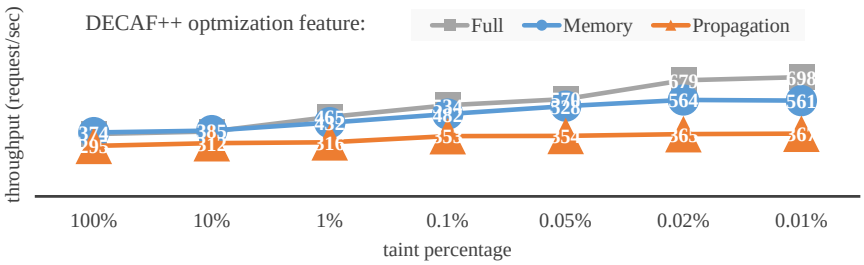


**Figure 4.10:** Throughput of solitary optimization features (and together) of DECAF++ in. The reported numbers are the mean of 5 measurements and the relative standard deviation are in range of [2%,18%] for Full, [1%,15%] for Mem and [1%,14%] for the Propagation. DECAF and QEMU 1.0 throughput are 320 and 815 request/sec.

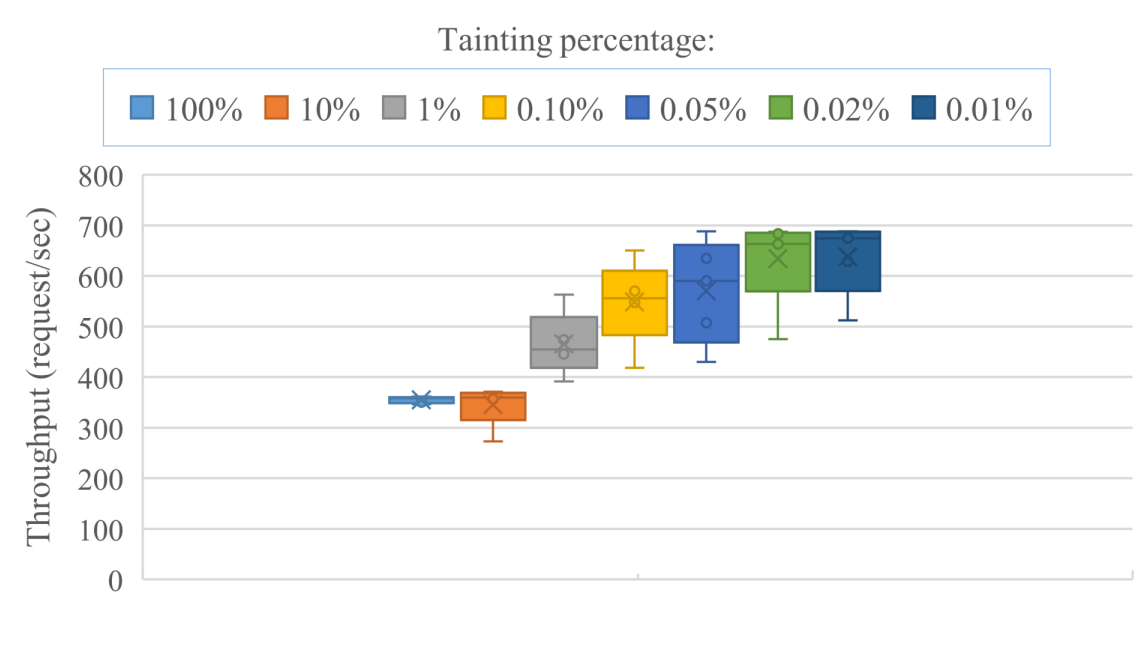


**Figure 4.11:** Evaluation of the DECAF++ on Apache bench. Each candlestick shows 5 measurements of throughput (request/sec) for a percentage of tainted packets.

**Throughput result** Figure 4.10 illustrates the result of our throughput measurement for apache server using apache bench. The figure shows that the full optimization achieves the best throughput. Answering (1), full optimization and elastic taint status checking outperform DECAF for all values of taint_perc, and elastic taint propagation outperforms DECAF when taint_perc is below 1%. Figure 4.11 shows the performance of the full optimization based on the percentage of the tainted bytes. Although DECAF++ has the elastic property and there is improvement in all cases (answering (2)), it is more tangible for taint_perc values less than 1%. We point out two points on why taint_perc values seem very small. First, the number of tainted bytes do not linearly decrease with taint_perc. Second, these even seemingly small taint_perc values represent the real world scenarios. For instance for security applications, the attacks are anomaly cases and the percentage of suspicious packets are well below 1%. Overall, DECAF++ achieves an average (geometric) 60% throughput improvement in comparison to DECAF. When there are no tainted bytes, our system is still around 18% slower than QEMU because of the network callbacks.

### 4.5.4 Elastic Instrumentation Overhead

In this section, we evaluate DECAF++ to answer (3) and understand whether we could address the shortcomings of the previous works that are high overhead in the check mode of LIFT [84] or high transition overhead of [51].

**Check mode overhead** Elastic taint analysis imposes an overhead even when there are no tainted bytes. In case of LIFT [84], it's the registers taint tag check at the beginning of every basic block and further memory tag checks before memory instructions. For DECAF++, our evaluation using SPEC CPU2006 and nbench illustrated in Figure 4.12 and

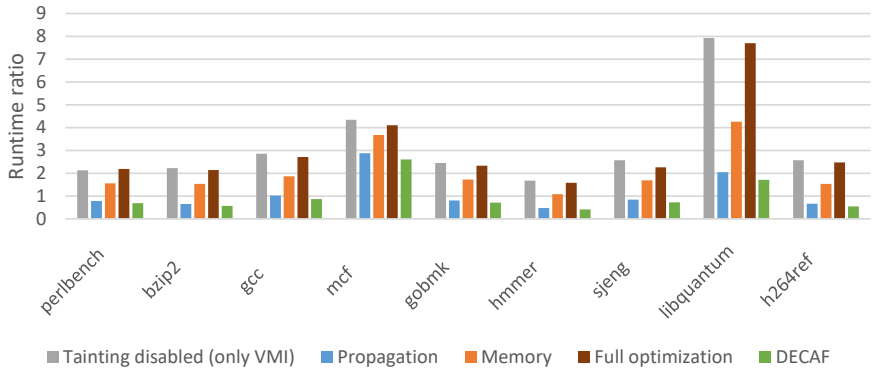


**Figure 4.12:** SPEC CPU2006 for different implementations

Figure 4.13 shows that DECAF++ imposes around 4% overhead even when there is no taint analysis task, that is, running only in the check mode. This overhead is in comparison to when tainting functionality is completely disabled. DECAF++ introduces a few overheads in comparison to this case. These overheads are:

- Checking the mode before the code translation and in the memory operations
- Patching the memory load operations in the track mode
- Checking the status of the page throughout the TLB filling process to register the taint status handler

We measured the effect of each of these overheads on indexes reported by nbench by removing the code snippets attributed to these functionalities. The results of these measurements are listed in Table 4.3. Removing none of the overheads except the mode checking has a substantial effect on the performance. This is because mode checking is frequently done along with every memory load and store operation. It goes unsaid that this overhead is inevitable.

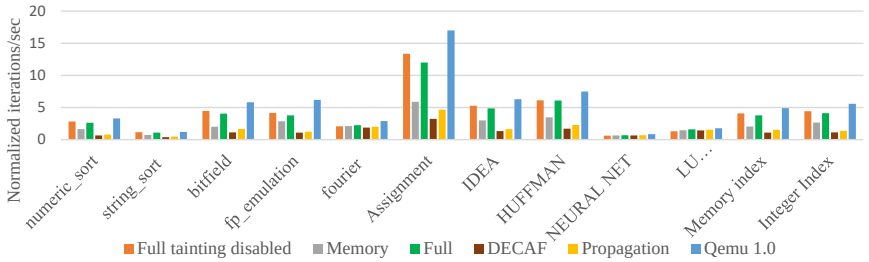


**Figure 4.13:** DECAF++ check mode overhead on nbench

**Table 4.3:** The nbench evaluation of DECAF++ by removing the potential overheads

| Procedure | Memory index | Relative STD | Integer Index | Relative STD |
|---|---|---|---|---|
| Baseline | 4.04 | 1% | 4.36 | 1% |
| Full | 3.86 | 2% | 4.17 | 2% |
| Mode checking | 4.04 | 2% | 4.34 | 3% |
| Load operations patching | 3.87 | 1% | 4.23 | 1% |
| Page check in TLB fill | 3.74 | 1% | 4.14 | 1% |

**Transition Overhead** The transition from the check mode to the track mode imposes an overhead as discussed in section 4.3.3. This overhead is the major issue with [51]. However, our measurement shows that this overhead is negligible for DECAF++. We measured the transition overhead by recording the time it takes to change the mode and execute the same instruction that was executing before the transition occurred. Our measurement was performed during nbench execution, and every input byte was tainted. We repeated the measurement 10 times. The average transition time is 0.031% of the overall benchmark execution time with 0.007% relative standard deviation.

## 4.6 Conclusion

In this work, we introduced elastic tainting for whole-system dynamic taint analysis. Elastic tainting is based on elastic taint propagation and elastic taint status checking that accordingly address DECAF taint propagation and taint status checking overhead by

removing unnecessary taint analysis computations when the system is in a safe state. We successfully designed and implemented this idea on top of DECAF in a prototype dubbed DECAF++ via pure software optimization. We showed that elastic tainting helps DECAF++ achieve a substantial better performance even when all inputs are tainted. Further, we showed how elastic taint propagation and elastic taint checking optimization each and together contribute to the performance improvement for different applications.

Our elastic tainting addresses the shortcomings of the previous works that are either high overhead when there's no tainted bytes or high transition cost when there is some. As a result, DECAF++ has an elastic property for both information flow within a process and information flow of a network input throughout the system. We believe whole-system dynamic taint analysis applications like intrusion detection systems and honeypots can greatly benefit from the elastic property. On one hand, this is because these systems are constantly online and taint analysis affects the entire system constantly. On the other hand, these systems can filter benign traffic and focus on the taint analysis of a small portion of the traffic that are likely to be malicious.

# Chapter 5

# Conclusions

In this thesis, we present novel techniques to detect and prevent intrusions concerning IoT malware. IoT is the new battleground for attackers and as we show, attackers rely on exploiting fairly old vulnerabilities to spread. On the other hand, the increase in the number of IoT threats suggest that the key players such as IoT vendors, security solution companies and network administrators fail to employ simple defenses that can significantly address the current issues.

This first chapter of this thesis laid the foundation for the defense against IoT threats at the network perimeter. More specifically, we presented C2Miner, a system that is able to detect the C2 server addresses passively by analyzing an IoT malware binary and actively via probing a target IP:port address space. We argue that a widespread deployment of C2Miner is a turning point in the battle against IoT malware threats.

In the second chapter of this thesis, we focus on a holistic analysis of the IoT malware traffic including the communications with C2 namely and the victims namely vulnerable IoT devices and the DDOS targets. Our findings in the chapter two provide

insights about the demographics of IoT malware C2 servers, vulnerabilities that the IoT malware tries to exploit and the details of the DDOS attacks.

Finally, in the third chapter of the thesis, we present a novel solution that can reduce the overhead of dynamic information flow analysis in applications that require whole system analysis such as intrusion detection systems. The improvement that we present allows a system to efficiently track the flow of network data e.g. an attack command from a c2 server throughout the system.

Overall, this thesis provides significantly novel approaches, tools and knowledge as to how IoT malware behaves. As a result, our work can be seen as a fundamental step in understanding IoT malware behavior, which can lead to better defenses in terms of detecting and containing the associated IoT malware damage.

# Bibliography


[1] apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Balbuzard - malware analysis tools to extract patterns of interest and crack obfuscation such as xor. https://github.com/decalage2/balbuzard.

[3] Radare2 - libre and portable reverse engineering framework. https://www.radare.org/n/.

[4] VirusTotal. https://www.virustotal.com.

[5] Hungenberg,Thomas and Eckert, Matthias. Internet services simulation suite. https://www.inetsim.org, 2022.

[6] Abuse.ch. MalwareBazaar. https://bazaar.abuse.ch/.

[7] Forbes Advisor. Best dedicated hosting services of 2022. https://www.forbes.com/advisor/business/software/best-dedicated-server-hosting.

[8] National Security Agency. Ghidra - software reverse engineering framework. https://www.nsa.gov/resources/everyone/ghidra/.

[9] Arwa Abdulkarim Al Alsadi, Kaichi Sameshima, Jakob Bleier, Katsunari Yoshioka, Martina Lindorfer, Michel van Eeten, and Carlos H Gañán. No Spring Chicken: Quantifying the Lifespan of Exploits in IoT Malware Using Static and Dynamic Analysis. In *Proceedings of the ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2022.

[10] Arwa Abdulkarim Al Alsadi, Kaichi Sameshima, Jakob Bleier, Katsunari Yoshioka, Martina Lindorfer, Michel van Eeten, and Carlos H Gañán. No spring chicken: Quantifying the lifespan of exploits in iot malware using static and dynamic analysis. 2022.

[11] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai Botnet. In *Proceedings of the USENIX Security Symposium*, 2017.

[12] Big Data Cloud API. Autonomous Systems (AS) advertised IPv4 space rank. https://www.bigdatacloud.com/insights/as-rank.

[13] Fabrice Bellard. QEMU, A Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference (ATC, FREENIX Track)*, 2005.

[14] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATC '05)*, pages 41–41, 2005.

[15] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[16] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the International Symposium on Code generation and Optimization (CGO '06)*, pages 333–345. IEEE Computer Society, March 2006.

[17] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Symposium On Recent Advances in Intrusion Detection (RAID'11)*, pages 1–20, Berlin, Heidelberg, September 2011. Springer.

[18] Xander Bouwman, Harm Griffioen, Jelle Egbers, Christian Doerr, Bram Klievink, and Michel van Eeten. A different cup of TI? the added value of commercial threat intelligence. In *Proceedings of the USENIX Security Symposium*, 2020.

[19] Xander Bouwman, Harm Griffioen, Jelle Egbers, Christian Doerr, Bram Klievink, and Michel van Eeten. A different cup of TI? the added value of commercial threat intelligence. In *Proceedings of the USENIX Security Symposium*, 2020.

[20] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[21] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[22] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: Detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*, pages 430–441. IEEE, 2018.

[23] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*, pages 321–336, August 2004.

[24] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 196–206, New York, NY, USA, July 2007. ACM.

[25] M Patrick Collins, Timothy J Shimeall, Sidney Faber, Jeff Janies, Rhiannon Weaver, Markus De Shon, and Joseph Kadane. Using uncleanliness to predict future botnet addresses. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2007.

[26] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying dormant functionality in malware programs. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2010.

[27] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2009.

[28] Bill Conner. 2021 sonicwall cyber threat report. `https://cdw-prod.adobecqms.net/content/dam/cdw/on-domain-cdw/brands/sonicwall/sonicwall-2021-cyber-threat-report.pdf`.

[29] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 133–147. ACM, 2005.

[30] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[31] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The Tangled Genealogy of IoT Malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[32] Ahmad Darki and Michalis Faloutsos. RIoTMAN: a systematic analysis of IoT malware behavior. In *Proceedings of International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2020.

[33] Ali Davanian. Effective granularity in internet badhood detection: Detection rate, precision and implementation performance. Master's thesis, University of Twente, 2017.

[34] Ali Davanian. Effective granularity in internet badhood detection: Detection rate, precision and implementation performance. Master's thesis, University of Twente, August 2017.

[35] Ali Davanian, Ahmad Darki, and Michalis Faloutsos. Cnchunter: An mitm-approach to identify live cnc servers. *Black Hat USA*, 2021.

[36] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. {DECAF++}: Elastic {Whole-System} dynamic taint analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 31–45, 2019.

[37] Lorenzo De Carli, Ruben Torres, Gaspar Modelo-Howard, Alok Tongaonkar, and Somesh Jha. Botnet protocol inference in the presence of encrypted traffic. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2017.

[38] Peter J Denning. The locality principle. In *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*, pages 43–67. World Scientific, 2006.

[39] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5)*, page 4. ACM, December 2015.

[40] Cornelia Cecilia Eglantine. Nbench. 2012.

[41] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[42] Brown Farinholt, Mohammad Rezaeirad, Paul Pearce, Hitesh Dharmdasani, Haikuo Yin, Stevens Le Blond, Damon McCoy, and Kirill Levchenko. To catch a ratter: Monitoring the behavior of amateur darkcomet rat operators in the wild. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[43] Jonathan Fuller, Ranjita Pai Kasturi, Amit Sikder, Haichuan Xu, Berat Arik, Vivek Verma, Ehsan Asdar, and Brendan Saltaformaggio. C3PO: Large-Scale Study Of Covert Monitoring of CC Servers via Over-Permissioned Protocol Infiltration. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[44] Sean Ghallager. In face of scrutiny, researchers back off NSA "Torsploit" claim. `https://arstechnica.com/tech-policy/2013/08/in-face-of-scrutiny-researchers-back-off-nsa-torsploit-claim/`, 2013.

[45] Robert David Graham. Masscan: Mass ip port scanner. `https://github.com/robertdavidgraham/masscan`.

[46] Harm Griffioen and Christian Doerr. Examining mirai's battle over the internet of things. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[47] Guofei Gu, Vinod Yegneswaran, Phillip Porras, Jennifer Stoll, and Wenke Lee. Active botnet probing to identify obscure command and control channels. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[48] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

[49] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*, pages 248–258. ACM, July 2014.

[50] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and analysis of hajime, a peer-to-peer iot botnet. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[51] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 29–41, New York, NY, USA, 2006. ACM.

[52] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. Shadowreplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 235–246. ACM, 2013.

[53] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 377–390, New York, NY, USA, October 2017. ACM.

[54] Seiya Kato, Rui Tanabe, Katsunari Yoshioka, and Tsutomu Matsumoto. Adaptive observation of emerging cyber attacks targeting various iot devices. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 143–151. IEEE, 2021.

[55] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12)*, volume 47, pages 121–132. ACM, 2012.

[56] KimiNewt. pyshark - python wrapper for tshark. `https://github.com/KimiNewt/pyshark/`.

[57] Jingfei Kong, Cliff C Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and System Support for Improving Software Dependability*, pages 18–24. ACM, 2006.

[58] Daniel Kopp, Matthias Wichtlhuber, Ingmar Poese, Jair Santanna, Oliver Hohlfeld, and Christoph Dietzel. Ddos hide & seek: on the effectiveness of a booter services takedown. In *Proceedings of the Internet Measurement Conference*, pages 65–72, 2019.

[59] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, pages 1691–1708, New York, NY, USA, October 2017. ACM.

[60] Johannes Krupp, Mohammad Karami, Christian Rossow, Damon McCoy, and Michael Backes. Linking amplification ddos attacks to booter services. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 427–449. Springer, 2017.

[61] Marc Kührer, Christian Rossow, and Thorsten Holz. Paint it Black: Evaluating the Effectiveness of Malware Blacklists. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, September 2014.

[62] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 503–515. ACM, March 2016.

[63] Victor Le Pochat, Sourena Maroofi, Tom Van Goethem, Davy Preuveneers, Andrzej Duda, Wouter Joosen, Maciej Korczyński, et al. A practical approach for taking down avalanche botnets under real-world constraints. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium*. Internet Society, 2020.

[64] Chaz Lever, Platon Kotzias, Davide Balzarotti, Juan Caballero, and Manos Antonakakis. A lustrum of malware network communication: Evolution and insights. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[65] Vector Guo Li, Matthew Dunn, Paul Pearce, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. Reading the tea leaves: A comparative analysis of threat intelligence. In *Proceedings of the USENIX Security Symposium*, 2019.

[66] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 349–358, 2012.

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200. ACM, June 2005.

[68] Tongbo Luo, Zhaoyan Xu, Xing Jin, Yanhui Jia, and Xin Ouyang. Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices. *Black Hat*, 2017.

[69] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'16)*, February 2016.

[70] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA'08, Beijing*, page 32, 2008.

[71] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, pages 308–319. IEEE, August 2016.

[72] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium (Security '15)*, pages 65–80, August 2015.

[73] Yacin Nadji, Manos Antonakakis, Roberto Perdisci, David Dagon, and Wenke Lee. Beheading hydras: performing effective botnet takedowns. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 121–132, 2013.

[74] Yacin Nadji, Manos Antonakakis, Roberto Perdisci, and Wenke Lee. Understanding the Prevalence and Use of Alternative Plans in Malware with Network Games. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[75] Antonio Nappa, Zhaoyan Xu, M Zubair Rafique, Juan Caballero, and Guofei Gu. Cyberprobe: Towards internet-scale active detection of malicious servers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[76] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, volume 42, pages 89–100. ACM, June 2007.

[77] NetLab. The mostly dead mozi and its lingering bots. `https://blog.netlab.360.com/the-mostly-dead-mozi-and-its-lingering-bots/`.

[78] Matthias Neugschwandtner, Paolo Milani Comparetti, and Christian Platzer. Detecting Malware's Failover C&C Strategies with Squeeze. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[79] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, volume 5, pages 3–4, 2005.

[80] NFOservers. NFOservers. `https://www.nfoservers.com`.

[81] Arman Noroozian, Maciej Korczyński, Carlos Hernandez Gañan, Daisuke Makita, Katsunari Yoshioka, and Michel van Eeten. Who gets the boot? analyzing victimization by ddos-as-a-service. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 368–389. Springer, 2016.

[82] Kevin Valakuzhy Ryan Court Kevin Snow Fabian Monrose Manos Antonakakis Omar Alrawi, Charles Lever. The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle. In *Procedings of the USENIX Security Symposium*, 2021.

[83] Nicole Perloth. Hackers used new weapons to disrupt major websites across u.s. `https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html`.

[84] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135–148. IEEE, 2006.

[85] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS'17)*, February 2017.

[86] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[87] Silvia Sebastián and Juan Caballero. Avclass2: Massive malware tag extraction from av labels. In *Procedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[88] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[89] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, volume 39, pages 85–96. ACM, 2004.

[90] Mingshen Sun, Tao Wei, and John Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. ACM, 2016.

[91] Rui Tanabe, Tatsuya Tamai, Akira Fujita, Ryoichi Isawa, Katsunari Yoshioka, Tsutomu Matsumoto, Carlos Gañán, and Michel Van Eeten. Disposable botnets: examining the anatomy of iot botnet infrastructure. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2020.

[92] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. Botfinder: Finding bots in network traffic without deep packet inspection. In *Proceedings of International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[93] Valve Developer Community. Forum. `https://developer.valvesoftware.com/wiki/Main_Page`, 2022.

[94] Pierre-Antoine Vervier and Yun Shen. Before toasters rise up: A view into the emerging iot threat landscape. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2018.

[95] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2009.

[96] Peter Wurzinger, Leyla Bilge, Thorsten Holz, Jan Goebel, Christopher Kruegel, and Engin Kirda. Automatically generating models for botnet detection. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2009.

[97] Zhaoyan Xu, Antonio Nappa, Robert Baykov, Guangliang Yang, Juan Caballero, and Guofei Gu. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[98] LK Yan, A Henderson, X Hu, H Yin, and S McCamant. On soundness and precision of dynamic taint analysis. *Dep. Elect. Eng. Comput. Sci., Syracuse Univ., Tech. Rep. SYR-EECS-2014–04*, 2014.

[99] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[100] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[101] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 116–127. ACM, 2007.

[102] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M. Blough, Elissa M. Redmiles, and Mustaque Ahamad. An Inside Look into the Practice of Malware Analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[103] Ali Zand, Giovanni Vigna, Xifeng Yan, and Christopher Kruegel. Extracting probable command and control signatures for detecting botnets. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, 2014.

[104] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, January 2011.

[105] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *Proceedings of the USENIX Security Symposium*, 2020.

[106] Albert Zsigovits. Mirai/Gafgyt Fork with New DDoS Modules Discovered. `https://cujo.com/mirai-gafgyt-with-new-ddos-modules-discovered/`, 2021.