UNIVERSITY OF CALIFORNIA,
IRVINE

# Privacy Implications of Smart TVs

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Networked Systems

by

Janus Varmarken

Dissertation Committee:
Professor Athina Markopoulou, Chair
Distinguished Professor Gene Tsudik
Associate Professor Zubair Shafiq

2023

# DEDICATION

In memory of my father, Jens-Erik Varmarken.

# TABLE OF CONTENTS

# LIST OF FIGURES

Page

# LIST OF TABLES

# ACKNOWLEDGMENTS

Throughout my journey as a Ph.D. student, many extraordinary people and organizations have supported me professionally and/or personally. I would like to express my deepest gratitude for this support. Without it, this dissertation would not have been possible.

First and foremost, I would like to thank my advisor, Professor Athina Markopoulou, for taking me under her wing. Ever since writing my undergraduate thesis under her supervision, Professor Markopoulou has been an exceptional mentor who has been instrumental in shaping my academic thinking. Her help in identifying important research problems and her constructive feedback on my work and writing have been priceless. The doors she has opened for me have been countless and have helped fast-track my career: not only did she welcome me into her research group as a visiting scholar and later a Ph.D. student, she also introduced me to multiple industry professionals, which resulted in intellectually rewarding internships and collaborations. It has been a privilege to work under Professor Markopoulou's guidance.

Thank you to Professor Gene Tsudik and Professor Zubair Shafiq for serving on my doctoral committee and for the advice they have provided throughout my graduate studies. A special thank you is due to Professor Shafiq for identifying the lack of research on advertising and tracking on smart TVs and for initiating a successful collaboration on this topic.

I am deeply grateful to my collaborators: Dr. Rahmadi Trimananda, Professor Brian Demsky, Hieu Le, Dr. Anastasia Shuba, Professor Zubair Shafiq, Jad Al Aaraj, Andrew Searles, Maganth Seetharaman, and Amogh Pradeep. A special thank you to Hieu Le for taking on a major workload when we studied advertising and tracking on smart TVs.

Thank you to past and current members of the UCI Networking Group for their help, insights, and friendship: Dr. Anastasia Shuba, Dr. Balint Tillman, Dr. Emmanouil Alimpertis, Dr. Evita Bakopoulou, Dr. Milad Asgari Mehrabadi, Hieu Le, Stelios Stavroulakis, Hao Cui, Jad Al Aaraj, Dr. Rahmadi Trimananda, Olivia Figueira, Mengwei Yang, Ismat Jarin, Yu Duan, and Marilyne Tamayo. I am especially thankful for the mentorship of Dr. Anastasia Shuba and Dr. Rahmadi Trimananda.

I am very grateful for the funding support I have received from (i) the University of California, Irvine, via a number of mechanisms, including, but not limited to, the Networked Systems Fellowship and Seed Funding by UCI VCR; and (ii) the National Science Foundation (awards 1649372, 1815666, 1900654 and 1956393).

I have been blessed with the love and support of my family: Inge-Merete Varmarken, Tine Kjøller Varmarken, Alicja Jagiełło, and Yentl Jagiełło-Varmarken. Thank you for always being there for me, and for keeping me motivated during times of struggle. Also thank you to all my friends for their support and for the joy they bring to my life.

Given how many magnificent individuals I have interacted with during my time as a Ph.D. student, I may have failed to name some. For that I apologize, and I hope these individuals know that I am grateful for their support nevertheless.

## Reprint Notice

Portions of this dissertation are reprints of the material in [159], used with permission from Sciendo. The co-authors listed in this publication are Hieu Le, Anastasia Shuba, Athina Markopoulou, and Zubair Shafiq. The authors retain copyright for [159].

Other portions of this dissertation are reprints of the material in [156], used with permission from the Proceedings on Privacy Enhancing Technologies (PoPETs). The co-authors listed in this publication are Jad Al Aaraj, Rahmadi Trimananda, and Athina Markopoulou. The authors retain copyright for [156].

The work presented in Chapter 5 is intended for publication in the proceedings of a conference, or in a journal. However, it has not been formally committed for publication at the time of writing.

# VITA

## Janus Varmarken

### EDUCATION

**Doctor of Philosophy in Networked Systems** — **2023**
University of California, Irvine — *Irvine, CA, USA*

**Master of Science in Networked Systems** — **2023**
University of California, Irvine — *Irvine, CA, USA*

**Master of Science in Information Technology** — **2017**
IT University of Copenhagen — *Copenhagen, Denmark*

**Bachelor of Science in Software Development** — **2013**
IT University of Copenhagen — *Copenhagen, Denmark*

### RESEARCH EXPERIENCE

**Graduate Research Assistant** — **September 2017 – June 2023**
University of California, Irvine — *Irvine, CA, USA*

**Graduate Research Assistant** — **December 2015 – January 2016**
IT University of Copenhagen — *Copenhagen, Denmark*

**Visiting Research Assistant** — **September 2014 – January 2015**
University of California, Irvine — *Irvine, CA, USA*

### TEACHING EXPERIENCE

**Teaching Assistant** — **Spring 2019, Spring 2020**
University of California, Irvine — *Irvine, CA, USA*

### PROFESSIONAL EXPERIENCE

**Software Engineering Intern** — **Summer 2021, Summer 2022**
Juniper Networks, Inc. — *Cupertino, CA, USA*

**Software Engineering Intern** — **Summer 2019**
Symantec Corporation — *Culver City, CA, USA*

**Software Engineer (part-time role)** — **October 2013 – July 2014**
Danish Maritime Authority — *Valby, Denmark*

**Software Engineer (part-time role)** — **March 2012 – September 2013**
Hammerstad A/S — *Herlev, Denmark*

## PUBLICATIONS

**Seqnature: Packet Sequences as Network Fingerprints** **2023**
In preparation

**FingerprinTV: Fingerprinting Smart TV Apps** **2022**
Privacy Enhancing Technologies Symposium

**The TV is Smart and Full of Trackers: Measuring Smart TV Advertising and Tracking** **2020**
Privacy Enhancing Technologies Symposium

**Packet-Level Signatures for Smart Home Devices** **2020**
Network and Distributed System Security Symposium

**AntMonitor: Network Traffic Monitoring and Real-Time Prevention of Privacy Leaks in Mobile Devices** **2015**
ACM S3 Workshop on Mobile Computing and Networking

**AntMonitor: A System for Monitoring from Mobile Devices** **2015**
ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big Internet Data

# ABSTRACT OF THE DISSERTATION

Privacy Implications of Smart TVs

By

Janus Varmarken

Doctor of Philosophy in Networked Systems

University of California, Irvine, 2023

Professor Athina Markopoulou, Chair

A smart TV is an Internet-connected TV with computational capabilities. These enhancements to the traditional TV set enable the smart TV to stream content from the Internet and run interactive applications (apps). While appealing from a functionality standpoint, smart TVs unfortunately also introduce new privacy risks. For example, unlike traditional TVs which can only receive TV channel broadcasts, the smart TV may use its Internet-connectivity to exfiltrate information about the user, including, but not limited to, viewing history. Despite massive user adoption of smart TVs, there is surprisingly little work on the privacy implications of smart TVs. Using a network measurement approach, this dissertation seeks to close this gap in the literature.

The first part of this dissertation presents a large-scale measurement study of the smart TV advertising and tracking ecosystem. Network traffic collected from smart TVs, when used by real users and when instrumented in the lab, reveal that smart TVs connect to well-known and platform-specific advertising and tracking services (ATSes). Automated tests of the top-1000 apps on two popular smart TV platforms unveil that (i) a subset of apps communicate with a large number of ATSes, and some ATS organizations only appear on certain platforms, showing a possible segmentation of the smart TV ATS ecosystem across platforms; and (ii) hundreds of apps exfiltrate personally identifiable information to third

parties and platform domains. Furthermore, an evaluation of DNS-based blocklists shows that even smart TV-specific blocklists miss ads and incur functionality breakage.

Next, the dissertation investigates if an in-network adversary can identify what smart TV app is launched. Automated tests are used to collect multiple samples of the network traffic generated by each of the top-1000 apps on the three most popular smart TV platforms. Network fingerprints are extracted from this dataset using three established fingerprinting techniques. The results show that smart TV app network fingerprinting is feasible and effective: even the least prevalent type of fingerprint manifests itself in at least 68% of apps of each smart TV platform, and up to 89% of fingerprints uniquely identify a specific app when two fingerprinting techniques are used together. It is also shown that apps that exhibit identical fingerprints often stem from the same developer or "no code" toolkit, and that apps that are present on all three smart TV platforms exhibit platform-specific fingerprints.

Finally, inspired by the observation that joint use of multiple fingerprinting techniques improves fingerprint distinctiveness, the dissertation proposes a general fingerprinting framework that can identify fingerprints that are based on any combination of several features, such as server identities and the sizes, directions, and/or order of packets. Through customizable parameters, the framework provides support for both joint and separate use of prior fingerprinting techniques, as well as any fingerprinting technique that can be formulated as a problem of identifying similar packet exchanges. To demonstrate its versatility, the framework is used to implement and evaluate two different fingerprinting techniques. Fingerprints for smart TV apps and for events on simple Internet of Things (IoT) devices, such as smart plugs and smart light bulbs, are extracted using the two fingerprinting techniques. The relative performance of the two fingerprinting techniques is established by comparing the number of fingerprints each fingerprinting technique identifies, as well as how distinct the extracted fingerprints are from other traffic.

# Chapter 1

# Introduction

## 1.1 Background

A *smart TV* (or *connected TV*) is an Internet-connected TV with computational capabilities. These enhancements to the traditional TV set enable the smart TV to stream content from the Internet and run interactive applications (apps).

Originally, most smart TV products could be categorized as either a built-in smart TV or an over-the-top (OTT) streaming device. A built-in smart TV is a traditional TV set where the TV manufacturer has integrated the hardware and software necessary to make the TV "smart." For example, Samsung offers built-in smart TVs powered by the Tizen operating system (which is developed in-house at Samsung). An OTT device is an external dongle or box that, when inserted into a port of a TV, transforms the TV into a smart TV. However, the distinction between built-in smart TVs and OTT devices is becoming increasingly blurred as some TV manufacturers now integrate OTT vendors' software/hardware platforms directly into their TV sets. For example, TCL and Sharp offer smart TVs that integrate the Roku smart TV platform (by Roku), while Insignia and Toshiba offer smart TVs that integrate

the Fire TV smart TV platform (by Amazon). This dissertation therefore uses "smart TV" as an umbrella term for both built-in smart TVs and OTT devices.

There is a diverse set of smart TV platforms, each with its own set of apps that users can install. Many smart TV platforms are based on existing mobile operating systems, in particular Android by Google. Google's own full-fledged smart TV platform was originally called Android TV, but was re-branded as Google TV in 2020 when Google introduced a new user interface and enhanced content recommendation [31]. Smart TVs that run stock Android TV/Google TV (e.g., smart TVs manufactured by Sony and Philips) have access to apps from the Google Play Store. Smart TV platforms that are derivatives of Android sometimes also have their own separate app store. For example, apps for Amazon's Fire TV are made available through the Amazon Appstore. Despite differences in app distribution channels, both Android TV/Google TV and Fire TV apps are built using technologies and tools that are similar to those used for building regular Android apps for smartphones. Likewise, Apple TV apps are built using technologies and tools that were originally designed for iOS apps, and Apple TV apps can be downloaded from Apple's App Store.

The technologies used for app development and distribution are different for smart TV platforms that are not derived from the two major mobile operating systems, Android and iOS. For example, apps for the Roku platform are built using BrightScript, which is a custom scripting language, and are distributed via the Roku Channel Store. Many other smart TV platforms follow a web-based ecosystem where applications are developed using HTML, CSS, and JavaScript. Examples include Samsung's Tizen, LG's webOS, and Hybrid broadcast broadband TV (HbbTV). Finally, some smart TV platforms, such as Chromecast, do not have app stores of their own, but are only meant to "cast" (i.e., mirror/project) content from other devices such as smartphones.

As with mobile apps, smart TV apps can integrate third-party libraries and services, often for advertising and tracking purposes. Serving advertisements (ads) is one of the main

ways for smart TV platforms and app developers to generate revenue [6], and the market is huge: according to Statista, smart TV ad spending exceeded $21 billion in 2022 in the U.S. alone, and is projected to double in 2026 [142]. Smart TV platform operators pocket a sizeable portion of this ad spending by requiring apps to hand over ad inventory (ad display slots) or ad revenue. For example, both Roku and Amazon (Fire TV) lay claim to 30% of an app's ad inventory [125, 140]. The smart TV advertising ecosystem mirrors many aspects of the web advertising ecosystem. Most importantly, smart TV advertising uses programmatic mechanisms that allow apps to sell their ad inventory in an automated fashion using behavioral targeting [129, 17].

## 1.2 Motivation

Smart TVs provide convenient access to streaming services such as Netflix and Hulu, a rich selection of games, and are also increasingly used as a central hub for entertainment in general, such as music streaming and social media [65]. At the same time, smart TVs are available at relatively affordable prices, with many of the external smart TV boxes/sticks priced less than $50, while built-in smart TVs now cost only a few hundreds dollars [8]. Consumers have responded positively to this value proposition: between 2011 and 2021, the percentage of U.S. TV households with at least one smart TV has increased from 30% to 82% [79], and smart TV sets now outnumber traditional TV sets in American homes [64].

While appealing from an entertainment standpoint, smart TVs unfortunately also introduce new privacy risks. The rapidly growing smart TV advertising business (see Section 1.1) incentivizes smart TV apps and platform operators to track and profile users in order to optimize ad targeting and thereby maximize profit. One way this can be achieved is via Automatic Content Recognition (ACR), which many smart TVs use to track what their users watch as this information can be used to serve targeted ads [86]. In one high-profile

case, Vizio paid $2.2 million to settle charges by the Federal Trade Commission (FTC) that Vizio used ACR to track users' viewing data without their knowledge or consent [43]. While smart TV platforms now allow users to opt out of such tracking, it is not straightforward for users to turn it off [54]. Furthermore, Consumer Reports found that even with ACR turned off, some smart TVs still require users to agree to a basic (but hard to understand) privacy policy that asks for the right to collect the user's location, choice of apps, etc. [33].

Despite their massive user base and the aforementioned examples of poor privacy practices, there exists remarkably little (academic) work on the privacy implications of smart TVs. For example, while the ecosystems of advertising and tracking services (ATSes) that facilitate delivery of targeted ads on the desktop [90, 51, 40] and mobile [113, 117, 136] platforms have been studied extensively, the ATS ecosystems of smart TV platforms have not been examined at scale until recently. Similarly, the literature on network fingerprints of websites and mobile apps is rich [59, 144, 27, 81, 58, 83, 105, 36, 163, 104, 57, 138, 103, 26, 139, 102, 141, 85, 60, 145, 146, 155], but no prior work has investigated if smart TV apps can be identified from their (encrypted) network traffic. Since viewing history may reveal sensitive information about the viewer (e.g., sexual orientation), and since the smart TV app in use can be synonymous with (the theme of) the content being watched, insight into whether smart TV apps can be fingerprinted is key to understanding the privacy implications of smart TVs. This dissertation seeks to close these gaps in the literature.

## 1.3 Contributions

This dissertation makes the following contributions.

### 1.3.1 Measurement of Advertising and Tracking on Smart TVs

In Chapter 3, we present a large-scale measurement study of the smart TV advertising and tracking ecosystem. First, we illuminate the network behavior of smart TVs as used in the wild by analyzing network traffic collected from residential gateways. We find that smart TVs connect to well-known and platform-specific advertising and tracking services (ATSes). Second, we design and implement software tools that systematically explore and collect traffic from the top-1000 apps on two popular smart TV platforms, Roku and Amazon Fire TV. We discover that a subset of apps communicate with a large number of ATSes, and that some ATS organizations only appear on certain platforms, showing a possible segmentation of the smart TV ATS ecosystem across platforms. Third, we evaluate the (in)effectiveness of DNS-based blocklists in preventing smart TVs from accessing ATSes. We highlight that even smart TV-specific blocklists suffer from missed ads and incur functionality breakage. Finally, we examine our Roku and Fire TV datasets for exposure of personally identifiable information (PII) and find that hundreds of apps exfiltrate PII to third parties and platform domains. We also find evidence that some apps send the advertising ID alongside static PII values, effectively eliminating the user's ability to opt out of ad personalization.

### 1.3.2 FingerprinTV: Fingerprinting Smart TV Apps

In Chapter 4, we propose FingerprinTV, a fully automated methodology for extracting fingerprints from the network traffic of smart TV apps and assessing their performance. FingerprinTV (1) installs, repeatedly launches, and collects network traffic from smart TV apps; (2) extracts three different types of network fingerprints for each app, i.e., domain-based fingerprints (DBFs), packet-pair-based fingerprints (PBFs), and TLS-based fingerprints (TBFs); and (3) analyzes the extracted fingerprints in terms of their prevalence,

distinctiveness, and sizes. From applying FingerprinTV to the top-1000 apps of the three most popular smart TV platforms, we find that smart TV app network fingerprinting is feasible and effective: even the least prevalent type of fingerprint manifests itself in at least 68% of apps of each platform, and up to 89% of fingerprints uniquely identify a specific app when two fingerprinting techniques are used together. By analyzing apps that exhibit identical fingerprints, we find that these apps often stem from the same developer or "no code" app generation toolkit. Furthermore, we show that many apps that are present on all three platforms exhibit platform-specific fingerprints.

### 1.3.3 Seqnature: Packet Sequences as Network Fingerprints

Finally, motivated by the observation made in Chapter 4 that joint use of multiple fingerprinting techniques improves fingerprint distinctiveness, specifically in the context of smart TV apps, Chapter 5 proposes a general fingerprinting framework, Seqnature, that can identify network fingerprints that are based on any combination of several features of (encrypted) network traffic, such as server identities and the sizes, directions, and/or order of packets. Seqnature is platform-agnostic, i.e., Seqnature can be used to identify fingerprints of any arbitrary event $e$ on any software/hardware platform, including, but not limited to, smart TVs. Through customizable parameters, Seqnature provides support for both joint and separate use of the fingerprinting techniques considered in Chapter 4, as well as any fingerprinting technique that can be formulated as a problem of identifying packet exchanges that consistently appear when the fingerprinted event $e$ is triggered.

Seqnature leaves the user in full control of what traffic features are used in a fingerprint, and what packet exchanges are considered identical. This makes it extremely simple to use Seqnature to implement, and compare the performance of, different fingerprinting techniques. We demonstrate this by using Seqnature to implement two fingerprinting

techniques, and then extract fingerprints for smart TV apps and for events on simple IoT devices. Both fingerprinting techniques identify fingerprints for almost all smart TV apps and for most events on IoT devices. Using SEQNATURE's ability to search traffic for manifestations of a fingerprint, we find that one fingerprinting technique produces fingerprints that have more discriminative power, as these fingerprints give rise to slightly fewer false positives than fingerprints extracted using the other fingerprinting technique. However, the difference is minor, and since the other fingerprinting technique is less compute-intensive, it may be preferable in scenarios where computing resources are limited and a few false positives can be tolerated.

## 1.4 Outline

The remainder of this dissertation is structured as follows. Chapter 2 discusses related work. Chapter 3 examines advertising and tracking on smart TVs using network measurements. Chapter 4 investigates if an in-network adversary can identify what smart TV app a user launches. Chapter 5 proposes a general network fingerprinting framework that can be used to experiment with new and existing fingerprinting techniques on any software/hardware platform, including, but not limited to, smart TVs. Chapter 6 concludes the dissertation.

# Chapter 2

# Related Work

## 2.1 Ads and Tracking on Smart TVs

While the desktop [90, 51, 40] and mobile [113, 117, 136] ATS ecosystems have been studied extensively, the smart TV ATS ecosystem has not been examined at scale until recently. Three papers [115, 63, 98] studied the network behavior of smart TVs concurrently with our work on the smart TV ATS ecosystem (see Chapter 3). Ren et al. [115] studied a large set of IoT devices spanning multiple device categories, including smart TVs. Their results showed that smart TVs were the category of devices that contacted the largest number of third parties, which further motivates our in-depth study of the smart TV ATS ecosystem. Huang et al. [63] used crowdsourcing to collect network traffic for IoT devices in the wild and showed that smart TVs contact many trackers by matching the contacted domains against the Disconnect blocklist. Finally, Moghaddam et al. [98] also instrumented the Roku and Fire TV smart TV platforms to map the ATS endpoints and the exposure of PII.

Our work independently confirms the findings of these works w.r.t. the smart TV ATS ecosystem, both by analyzing seven different smart TV platforms in the wild and by per-

forming systematic tests of two platforms (Roku and Fire TV) in the lab. In addition, we further contribute along two fronts. First, we show that even the same app across different smart TV platforms contact different ATSes, which shows the fragmentation of the smart TV ATS ecosystem. Second, we evaluate the effectiveness of different sets of blocklists, including smart TV specific blocklists, in terms of their ability to prevent ads and their adverse effects on app functionality. We also suggest ways to aid blocklist curation through analysis of domain usage across apps and PII exposures.

Earlier work in this space includes a series of papers by Ghiglieri et al. [48, 47, 50] who (i) showed how broadcasting stations and the user's neighbors could track the user's viewing behavior on the HbbTV platform; and (ii) presented a Pi-hole-like solution to prevent this. Related to our work, they found that HbbTV apps load third-party tracking scripts from Google Analytics. In contrast to the rich app-based platforms we study in this dissertation, the HbbTV platform simply overlays (interactive) HTML5 content on top of TV channels. Moreover, the number of apps studied in this dissertation is one order of magnitude larger than the combined number of channels across all these past studies (see Table 1 in [50]).

## 2.2 Other Work on Privacy Implications of Smart TVs

Other work that addresses the privacy implications of smart TVs includes a qualitative assessment of the privacy practices of five popular smart TVs and 10 popular streaming apps [72], and two surveys of consumers' understanding of the privacy risks and practices of smart TVs [49, 87].

In their evaluation of the privacy practices of smart TVs and streaming apps [72], Common Sense Media found that (i) among the top-5 smart TVs in the market (Apple TV, Google TV, Fire TV, Roku, and Nvidia Shield TV), all but Apple TV have privacy practices that

put consumers' privacy at considerable risk; and similarly (ii) except for Apple TV+, all of the top-10 streaming apps have privacy practices that put consumers' privacy at considerable risk. Common Sense Media highlight that these practices include selling data, sending third-party marketing communications, displaying targeted ads, tracking users across other sites and services, and creating advertising profiles for data brokers.

When surveying hundreds of users, both Ghiglieri et al. [49] and Malkin et al. [87] found widespread unawareness and confusion about the privacy implications of smart TVs, such as what data the smart TV collects and how that data is used. Malkin et al. [87] note that users find some smart TV features quite useful. These users therefore face a trade-off between the functional benefits of a smart TV and their privacy. When faced with this dilemma, some users (incorrectly) assume existing laws and regulations are in place to protect their data, while others grudgingly believe they never had a choice in the first place, an effect Malkin et al. refer to as "the resignation factor."

## 2.3 Network Fingerprints

A network *fingerprint* (or *signature*) is network traffic that is characteristic for certain software (or hardware) and that may thus be used to identify the presence of such software (hardware) on the network. Fingerprinting has been studied by academics and professionals for decades because it enables valuable services such as Network Intrusion Detection Systems and traffic prioritization schemes, yet at the same time also introduces privacy risks as it allows network operators to "spy" on individual users' computer usage.

The work of Moghaddam et al. [98] and our own work described in Chapter 4 were the first to study smart TV apps at scale, but focused on advertising and tracking on smart TVs. In Chapter 4, we continue the effort of understanding the privacy implications of smart TVs by

studying network fingerprints of smart TV apps. While we draw inspiration from [98] and Chapter 3 w.r.t. how we instrument the Fire TV and Roku smart TV platforms, large-scale assessment of network fingerprinting techniques' applicability to smart TV apps was not possible with prior work, and no dataset with multiple samples of apps' on-launch traffic previously existed. Additionally, whereas the instrumentation tools [157, 158] published alongside Chapter 3 only cover Fire TV and Roku and still require some manual intervention, FingerprinTV, a tool we develop for our study in Chapter 4, fully automates app testing, and adds support for Apple TV.

**Well-known Ports and Payload Analysis.** Early fingerprinting techniques relied on applications' use of well-known ports, but this approach had limited accuracy and granularity, which paved the way for proposals that relied on inspection of packet payload [99, 84]. Techniques based on payload inspection also initially proved applicable to recent emerging technologies, such as smartphones [96] and IoT [44], but their relevance is declining as the use of encryption is becoming more widespread in these technologies [112, 15, 63].

**Fingerprinting Encrypted Traffic.** With the introduction of SSL, the predecessor to TLS, researchers began exploring what information could (still) be inferred from network traffic, despite the payload being encrypted. For example, Bernaille and Teixeira [24] demonstrated that the application layer protocol used on top of SSL could be identified by analyzing the sizes and directions of the first few packets of an SSL session. Researchers have also reconstructed even more granular information from encrypted network traffic by fingerprinting HTTP User-Agent strings [66], websites [59, 144] (even in the presence of additional privacy enhancing technologies, such as encryption of domain names, tunneling, onion routing, and traffic morphing [27, 81, 58, 83, 105, 36, 163, 104, 57, 138, 103, 26, 139, 102, 141, 85, 60]), individual webpages on social media websites [162], desktop [19] and mobile [145, 146, 155] applications, and even individual user actions in mobile applications [34, 131] as well as voice commands on smart speakers [73, 161, 32, 67] and individual functionality on IoT

devices [35, 101, 115, 151, 13].

Some work has even shown that it is possible to fingerprint video content [114, 133]. Content fingerprinting provides more granular information than smart TV app fingerprinting, but is not as lightweight as the fingerprinting techniques we consider for smart TV apps in Chapter 4. For example, Schuster et al. [133] employ Convolutional Neural Networks, and also note that data collection is a bottleneck as content must be played back in real-time, multiple times, and the technique proposed in [114] needs access to at least 30 Application Data Units, which equates to two minutes of video playback, before any inference attempts can be made.

**Network Fingerprints in the Context of Smart TVs.** While the literature on fingerprinting is evidently rich, to the best of our knowledge, no prior work has studied smart TV app fingerprinting at scale. A few papers have investigated fingerprinting in the context of smart TVs (alongside other IoT devices) [44, 56, 115, 151, 130]. However, these papers are concerned with fingerprinting the smart TV "as a whole", i.e., identifying its presence on the network [44, 56, 130], or with fingerprinting basic events on smart TVs, e.g., returning to the menu screen [115, 151], but do not attempt to fingerprint individual smart TV apps. Furthermore, since the network traffic profiles of smart TV apps differ from those of simpler smart home devices, existing fingerprinting techniques, such as PingPong [151], may require modifications. We discuss this in further detail in Section 4.3.2.

# Chapter 3

# Measurement of Advertising and Tracking on Smart TVs

## 3.1 Overview

Despite massive consumer adoption of smart TVs, the ecosystem of advertising and tracking services (ATSes) that facilitate delivery of (targeted) ads on smart TVs is currently not well understood by users, researchers, and regulators. To that end, this chapter presents one of the first large-scale measurement studies of the emerging smart TV advertising and tracking ecosystem.

**In the Wild Measurements (Section 3.3).** First, we analyze the network traffic of smart TV devices in the wild. We instrument residential gateways of 41 homes and collect flow-level summary logs of the network traffic generated by 57 smart TVs from seven different platforms. The comparative analysis of network traffic from different smart TV platforms uncovers similarities and differences in their characteristics. As expected, we find that a substantial fraction of the traffic is related to popular video streaming services such as Netflix

and Hulu. More importantly, we find generic as well as platform-specific ATSes. Although realistic, the in the wild dataset does not provide app-level visibility, i.e., we cannot determine which apps generate traffic to ATSes. To address this limitation, we undertake the following major effort.

**Controlled Testbed Measurements (Section 3.4).** We design and implement two software tools, Rokustic for Roku and Firetastic for Amazon Fire TV, which systematically explore apps and collect their network traffic. We use Rokustic and Firetastic to exercise the top-1000 apps of their respective platforms, and refer to the collected network traffic as our *testbed* datasets. We analyze the testbed datasets w.r.t. the top Internet destinations the apps contact, at the granularity of fully qualified domain names (FQDNs), effective second-level domains (eSLDs), and organizations. We use "domain", "endpoint", and "destination" interchangeably in place of FQDN and eSLD when the distinction is clear from the context. We further separate destinations as first, third, and platform-specific party, w.r.t. to the app that contacts them.

First, we find that the majority of apps contact few ATSes, while about 10% of the apps contact a large number of ATSes. Interestingly, many of these more concerning apps come from a small set of developers. Second, we find what appears to be a segmentation of the smart TV ATS ecosystem across Roku and Fire TV as (1) the two datasets have little overlap in terms of ATS domains; (2) some third party ATSes are among the key players on one platform, but completely absent on the other; and (3) apps that are present on both platforms have little overlap in terms of the domains they contact. Third, we compare the top third party ATS domains of the testbed datasets to those of Android.

**Evaluation of DNS-Based Blocklists (Section 3.5).** Users typically rely on DNS-based blocking solutions such as Pi-hole [12] to prevent in-home devices such as smart TVs from accessing ATSes. Thus, we evaluate the effectiveness of DNS-based blocklists, selecting those that are most relevant to smart TVs. Specifically, we examine and test four popular

blocklists: (1) Pi-hole Default blocklist (PD) [12], (2) Firebog's recommended advertising and tracking lists (TF) [160], (3) Mother of all Ad-Blocking (MoaAB) [11], and (4) StopAd's smart TV specific blocklist (SATV) [76]. Our comparative analysis shows that block rates vary, with Firebog having the highest coverage across different platforms and StopAd blocking the least. We further investigate potential false negatives (FN) and false positives (FP). We discover that blocklists miss different ATSes (FN), some of which are missed by all blocklists, while more aggressive blocklists can suffer from false positives that result in breaking app functionality. We discuss two ways to discover false negatives, through observing domains contacted by multiple apps ("app prevalence") and keyword search (based on ATS related words like "ads" and "measure").

**PII Exposures (Section 3.6).** We further examine the network traces of our testbed datasets and find that hundreds of apps exfiltrate personally identifiable information (PII) to third parties and platform-specific parties, mostly for non-functional advertising and tracking purposes. Alarmingly, we find that many apps send the advertising ID alongside static PII values such as the device's serial number. This eliminates the user's ability to opt out of personalized advertisements by resetting the advertising ID, since the ATS can simply link an old advertising ID to its new value by joining on the serial number. We evaluate the blocklists' ability to prevent exposures of PII and find that they generally perform well for Roku, but struggle to prevent exfiltration of the device's serial number and the device ID to third parties and the platform-specific party for Fire TV.

**Contributions.** In this chapter, we analyze the network behavior of smart TVs, both in the wild and in the lab. Our contributions include the following: (1) providing an in-depth comparative analysis of the ATS ecosystems of Roku, Fire TV, and Android; (2) illuminating the key players within the Roku and Fire TV ATS ecosystems by mapping domains to eSLDs and parent organizations; (3) evaluating the effectiveness and adverse effects of an extensive set of blocklists, including smart TV specific blocklists; (4) instrumenting long experiments

per app to uncover approximately twice as many domains as [98]; and (5) making our tools, Rokustic and Firetastic, and our testbed datasets available [158, 157, 153].

## 3.2 Labeling Methodology

Throughout this chapter, we provide insight into the smart TV ATS ecosystems by labeling a domain according to (1) its purpose (ATS or non-ATS); (2) its parent organization, i.e., the domain owner; and (3) its relation to the app that uses it (first, third, or platform-specific party). We detail this methodology below.

**ATS Domains.** We identify ATS domains as follows. For figures that denote top domains, we check if the FQDN is labeled as ads or tracking by VirusTotal, McAfee, or OpenDNS [4, 91, 3], or if it is blocked by any of the blocklists considered in Section 3.5. For figures and tables that involve entire datasets, we only consider the blocklists due to the impracticality of manually labeling thousands of data points.

**Parent Organizations.** To understand the presence of different organizations on smart TV platforms, we map each FQDN to its effective second-level domain (eSLD) using Mozilla's Public Suffix List [100, 70], and use Crunchbase's [1] acquisition and sub-organization information to find the parent company of the eSLD. For example, `hulu.com` belongs to the Walt Disney Company and `youtube.com` belongs to Alphabet.

**App-Level Party Categorization.** The app-level visibility in our testbed experiments (Section 3.4) enables categorization of an Internet destination as a first party or a third party w.r.t. the app generating the traffic. We provide an overview of the technique here and defer details to Appendix A.1.

We adopt a technique similar to prior work [113], and we augment it to also include a

platform-specific party for traffic to platform-related destinations. We match tokenized eSLDs with tokenized package/app names and developer names. If the tokens match, we label the domain as *first party*. Otherwise, if the traffic originated from platform activity rather than app activity, we label it as *platform-specific party*: for Fire TV, AntMonitor [135] labels connections with the responsible process; for Roku, we check if the eSLD contains "roku." Otherwise, if the domain is contacted by at least two different apps from different developers, we label it as *third party*. Lastly, we resort to labeling it as *other* to capture domains that are only contacted by a single app.

## 3.3 Smart TV Traffic in the Wild

In this section, we study the network behavior of smart TV devices when used by real users by analyzing a dataset collected at residential gateways of tens of homes. This dataset is referred to as the *in the wild* dataset. We compare the number of flows and traffic volumes generated by smart TVs from seven different platforms. We also analyze the most frequently used domains of each platform by identifying ATS domains and mapping each domain to its parent organization.

**Data Collection.** To study smart TV traffic characteristics in the wild, we monitor network traffic of 41 homes in a major metropolitan area in the United States. We sniff network traffic of smart TV devices at the residential gateways using off-the-shelf OpenWRT-capable commodity routers. We collect flow-level summary information for network traffic. For each flow, we collect its start time, the FQDN of the external endpoint (using DNS), and the internal device identifier. We identify smart TVs using heuristics that rely on DNS, DHCP, and SSDP traffic and also manually verify the identified smart TVs by contacting the owner. Our data collection covers a total of 57 smart TVs across 41 homes over the duration of approximately 3 weeks in 2018. Note that we obtained written consent from

| Smart TV Platform | Device Count | Avg. Flow Count Per Device (x 1000) | Avg. Flow Volume Per Device (GB) | Avg. eSLD Count Per Device |
|---|---|---|---|---|
| Apple | 16 | 49.3 | 46.6 | 536 |
| Samsung | 11 | 62.6 | 33.2 | 369 |
| Chromecast | 10 | 201.9 | 26.3 | 354 |
| Roku | 9 | 48.1 | 83.0 | 543 |
| Vizio | 6 | 43.4 | 63.4 | 278 |
| LG | 4 | 10.9 | 0.9 | 1893 |
| Sony | 1 | 33.1 | 0.1 | 186 |

Table 3.1: Traffic statistics of 57 smart TV devices observed across 41 homes ("in the wild" dataset).

users, informing them of our data collection and research objectives, in accordance with our institution's IRB guidelines.

**Dataset Statistics.** Table 3.1 lists basic statistics of smart TV devices observed in our dataset. Overall, we note 57 smart TVs from 7 different vendors/platforms using a variety of technologies.

Devices can be built-in smart TVs such as Samsung and Sony, others like Chromecast and Apple TV can be external stick/box solutions, while devices like Roku can have both forms. For example, 7 out 9 Roku devices in our dataset were built-in Roku smart TVs, while the remaining two were external Roku sticks. Note that a smart TV platform such as Roku supports the same set of apps and a similar interface for both built-in and external smart TV devices. Thus, we do not differentiate between built-in vs. external Roku smart TV devices.

We expect smart TV devices to generate significant traffic because they are typically used for OTT video streaming [41]. Chromecast devices generate the highest number of flows (exceeding 200 thousand flows) on average, while Samsung, Apple, and Roku devices generate nearly 50 thousand flows on average. Roku devices generate the highest volume of flows (exceeding 80 GB) on average, with one Roku generating as much as 283 GB. Except for

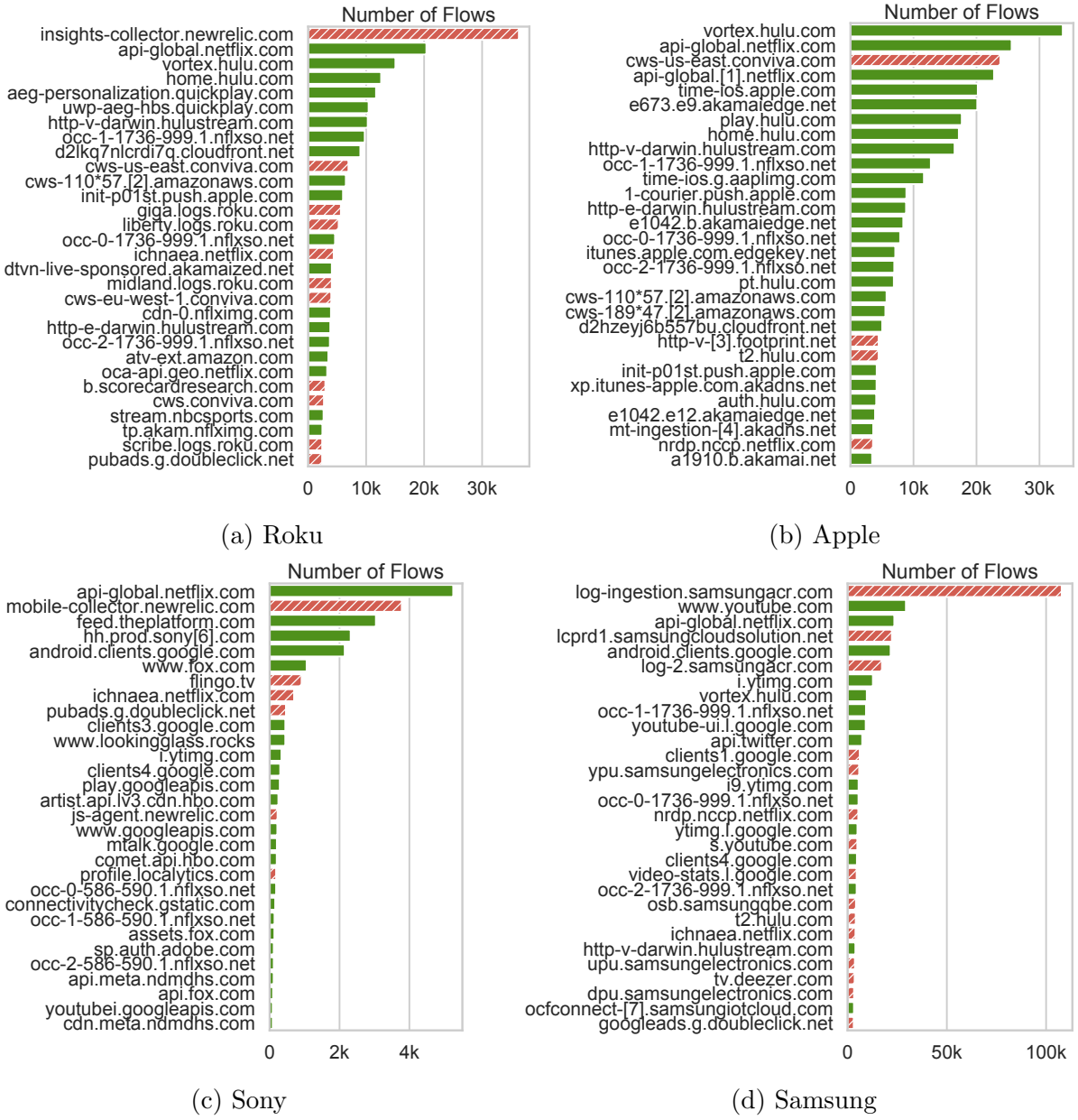
LG and Sony devices, all smart TV devices generate at least tens of GBs worth of traffic on average. Finally, we note that smart TV devices typically connect to hundreds of different endpoints on average.

**Endpoint Analysis.** Figure 3.1 plots the top-30 FQDNs in terms of flow counts for the Roku, Apple, Sony, and Samsung smart TV platforms. The plots for the remaining smart TV platforms are in Appendix A.5.1.

We note several similarities in the domains accessed by different smart TV devices. First, video streaming services such as Netflix and Hulu are popular across the board, as evident from domains such as api-global.netflix.com and vortex.hulu.com. Second, cloud/CDN services such as Akamai and AWS (Amazon) also appear for different smart TV platforms. Smart TVs likely connect to cloud/CDN services because popular video streaming services typically rely on third party CDNs [28, 14]. Third, we note the prevalence of well-known advertising and tracking services (ATSes). For example, *.scorecardresearch.com and *.newrelic.com are third party tracking services, and pubads.g.doubleclick.net is a third party advertising service.

We notice several platform-specific differences in the domains accessed by different smart TV platforms. Examples of domains that are unique to different smart TV platforms include: giga.logs.roku.com (Roku), time-ios.apple.com (Apple), hh.prod.sonyentertainmentnetwork.com (Sony), and log-ingestion.samsungacr.com (Samsung). In addition, we notice platform-specific ATSes. For example, the following advertising-related domains are not in the top-30 (and therefore not pictured in Figure 3.1), but are unique to different smart TV platforms: p.ads.roku.com (Roku), ads.samsungads.com (Samsung), and us.info.lgsmartad.com (LG).

**Organizational Analysis.** Figure 3.2 illustrates the mix of different parent organizations contacted by the seven smart TV platforms in our dataset. It shows the prevalence of Alphabet in smart TV platforms like Chromecast, Sony, and LG, while revealing competing




(a) Roku

(b) Apple

(c) Sony

(d) Samsung

Figure 3.1: Top-30 fully qualified domain names in terms of number of flows per device for a subset of the smart TVs in the "in the wild" dataset. See Appendix A.5.1 for the other brands. Domains identified as ATS are highlighted with red, dashed bars.

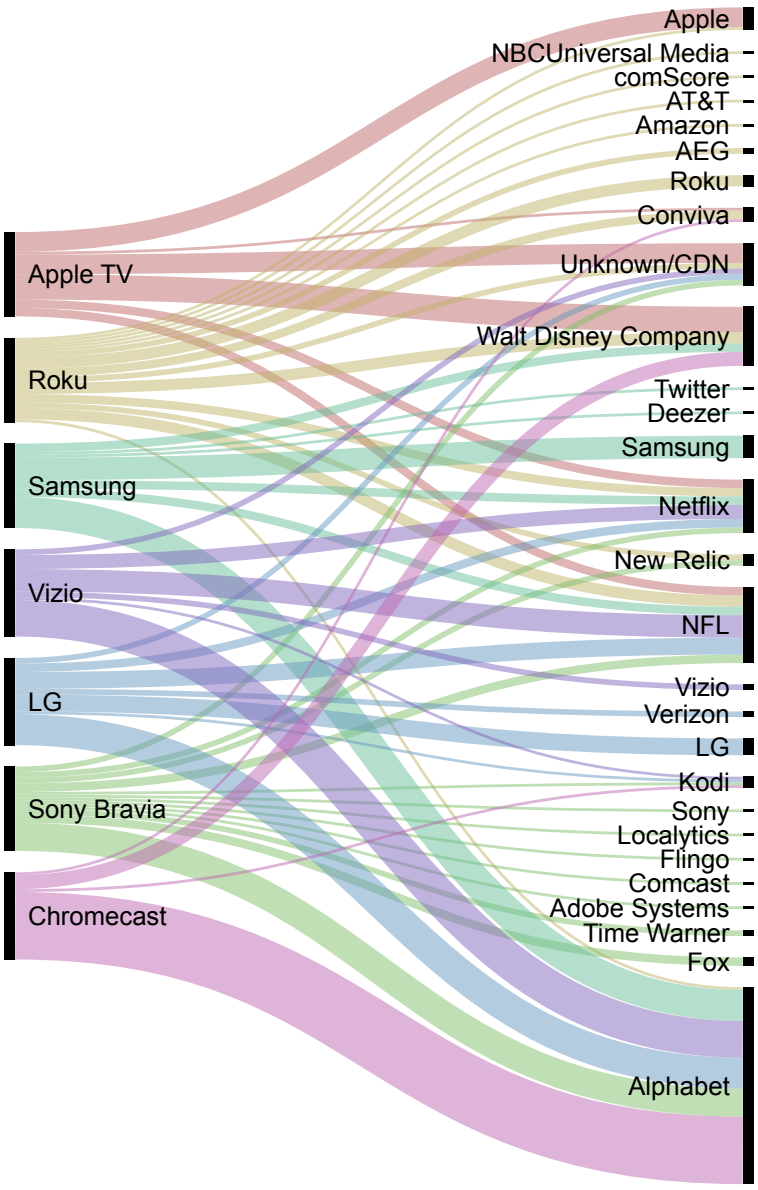



Figure 3.2: Mapping of platforms measured in the wild to the parent organizations of the endpoints they contact (for the top-30 FQDNs of each platform). The width of an edge indicates the number of distinct FQDNs within that organization that was accessed by the platform.

organizations such as Apple on the other end of the spectrum. Furthermore, it reveals the presence of organizations like Conviva, comScore, and Localytics, whose main business is in the advertising and tracking space. We note that Samsung, Deezer, Roku, LG, and Flingo have the majority of their domains labeled as ATSes, while Netflix and Walt Disney

Company have less than half of their domains labeled as ATSes.

**Takeaways and Limitations.** Traffic analysis of different smart TV platforms in the wild highlights interesting similarities and differences. As expected, all smart TVs generate traffic related to popular video streaming services. In addition, they also access ATSes, both well-known and platform-specific. While our vantage point at the residential gateway provides a real-world view of the behavior of smart TVs, it lacks granular information beyond flows (e.g., packet-level information) and does not tie traffic to the app that generates it. Another limitation of this analysis is that the findings may be biased by the viewing habits of the users in these 41 households. It is unclear how to normalize to provide a fair comparison of endpoints accessed by the different smart TV platforms. We address these limitations next by systematically analyzing two popular smart TV platforms in a controlled testbed.

## 3.4 Systematic Testing of Roku and Fire TV

In this section, we perform an in-depth, systematic study of two smart TV platforms, Roku and Amazon Fire TV, which we choose because they are popular [38], affordable ($25), and among the leading smart TV platforms in terms of number of ad requests [71]. Sections 3.4.1 and 3.4.2 present our measurement approach for systematically testing the top-1000 apps of each platform while collecting their network traffic. Since app exploration is automated, no real users are involved, thus no IRB is needed. Our measurement approach provides visibility into the behaviors of individual apps, which was not possible from the vantage point used for the in the wild dataset. In Section 3.4.3, we analyze the two testbed datasets, and compare them to each other and to the Android ATS ecosystem.

### 3.4.1 Roku Data Collection

In this section, we describe the Roku platform and our app selection methodology, and present an overview of Rokustic, our software tool that automatically explores Roku apps. We use Rokustic to explore and collect traffic from 1044 Roku apps. The resulting network traces are analyzed in Section 3.4.3.

**Roku Platform.** We start by describing the Roku platform, which has its own app store, the Roku Channel Store [124] (RCS), that offers more than 8500 apps, called "channels." For security purposes, the Roku operating system sandboxes each app (apps are not allowed to interact or access the data of other apps) and provides limited access to system resources [128]. Furthermore, Roku apps cannot run in the background. Specifically, app scripts are only executed when the user selects a particular app, and when the user exits the app, the script is halted, and the system resumes control [126].

To display ads, apps typically rely on the Roku Advertising Framework, which is integrated into the Roku SDK [127]. The framework allows developers to use ad servers of their preference and updates automatically without requiring the developer to rebuild the app. Even though such a framework eliminates the need for third party ATS libraries, the development and usage of such libraries is still possible. For example, the Ooyala IQ SDK [5] provides various analytics services that can be integrated into a Roku app. Thus, such libraries can help ATSes learn the viewing habits of users by collecting data from multiple apps. In terms of permissions, Roku only protects microphone access with a permission and does not require any permission to access the advertising ID. Users can choose to reset this ID and opt out of targeted advertising at any time [127]. However, apps and libraries can easily create other IDs or use fingerprinting techniques to continue tracking users even after opt-out. We further elaborate on this in Section 3.6.

**App Selection.** The RCS provides a web (and on-device) interface for browsing the avail-

able Roku apps. To the best of our knowledge, Roku does not provide public documentation on how to programmatically query the RCS. We therefore reverse-engineer the REST API backing the RCS web interface by inspecting the HTTP(S) requests sent while browsing the RCS, and use this insight to write a script that crawls the RCS for the metadata of all (8515 as of April 2019) apps.

To test the most relevant apps, we select the top-50 apps in 30 out of the 32 categories. We exclude "Themes" and "Screensavers" since these apps do not show up among the regular apps on the Roku and therefore cannot be operated using our automation software. We base our selection on the "star rating count," which we interpret as the review count. Roku apps can be labeled with multiple categories, thus some apps contribute to the top-50 of multiple categories. This places the final count of apps in our dataset at 1044.

**Automation (Rokustic).** To scale testing of apps, we implement a software tool, Rokustic, that automatically installs and exercises Roku apps. We provide a brief overview of Rokustic here, but defer a more detailed description to Appendix A.2.1.

We run Rokustic on a Raspberry Pi that acts as a router and hosts a local wireless network that the Roku is connected to. Rokustic utilizes ECP [123], a REST-like API exposed by the Roku device, to control the Roku. Given a set of apps to exercise, Rokustic installs each app by invoking the ECP endpoint that opens up the on-device version of the RCS page for the app, and then sends a virtual key press to click the "Add Channel" button. To exercise apps, Rokustic first invokes the ECP endpoint that returns the set of installed apps. For each app, Rokustic then (1) starts tcpdump on the Raspberry Pi's wireless interface; (2) uses ECP to launch the app and invoke a series of virtual key presses in an attempt to incur content playback; (3) pauses for five minutes to let the content play; (4) exits the app and repeats from step 2 an additional two times; (5) terminates tcpdump.

Since Roku apps cannot execute in the background (see "Roku Platform"), all captured

traffic will belong to the exercised app and the Roku system. The total interaction time with each app is approximately 16 minutes. We do not attempt to decrypt TLS traffic as we cannot install our own self-signed certificates on the Roku.

### 3.4.2 Fire TV Data Collection

In this section, we describe the Fire TV platform, our app selection methodology, and present an overview of Firetastic, our software tool that automatically explores Fire TV apps. By using Firetastic to control six Fire TV devices in parallel, we explore and collect traffic from 1010 Fire TV apps in one week. The resulting network traces are analyzed in Section 3.4.3.

**Fire TV Platform.** Although Fire TV is made by Amazon, its underlying operating system, Fire OS, is a modified version of Android. This allows apps for Fire TV to be developed in a similar fashion to Android apps. Therefore, all third-party libraries that are available for Android apps can also be integrated into Fire TV apps. Similarly, application sandboxing and permissions in Fire TV are analogous to those of Android, and any permission requested by the app is inherited by all libraries that the app includes. This allows third party libraries to track users across apps using a variety of identifiers, such as the Advertising ID, the Serial Number, the Device ID, etc. We further discuss tracking through PII exposure in Section 3.6.

**App Selection.** To test the most relevant apps, we pick the top-1000 apps from Amazon's curated list of "Top Featured" apps. We ignore some apps that use a local VPN (as they would conflict with AntMonitor), that could not be installed manually, and utility apps that can change the device settings (which would affect the test environment). As a result, we ignore around 200 apps while including 1010 testable applications. Amazon's app store offers around 4,000 free apps at the time of writing, thus our dataset covers approximately 25%.

**Automation (Firetastic).** We design and implement a software tool, Firetastic, that

integrates the capabilities of two open source tools for Android: an SDK for network traffic collection and a tool for input automation. We provide a brief overview of Firetastic here, and defer additional details to Appendix A.2.2.

We rely on AntMonitor [135, 7], an open-source VPN-based library, to intercept all outgoing network traffic from the Fire TV, and to label each packet with the package name of the application (or system process) that generated it. We enable AntMonitor's TLS decryption for added visibility into PII exposures. We analyze the success of TLS decryption in Appendix A.4. In summary, TLS decryption was generally successful with 10% or fewer failures for 55% of all apps, and 20% or fewer failures for 80% of all apps.

For app exploration, we utilize DroidBot [80], a Python tool that dynamically maps the UI and simulates user inputs such as button presses using the Android Debug Bridge (ADB). To increase the probability of content playback, we configure DroidBot to utilize its breadth first search algorithm to explore each app. The intuition is that the main content is often made available from top-level UI elements.

In summary, for each app, Firetastic: (1) starts AntMonitor; (2) explores the app for 15 minutes; (3) stops AntMonitor; and (4) extracts the `.pcapng` files that were generated during testing. We use Firetastic to explore apps on six Fire TV devices in parallel. Our test setup is resource-efficient and scalable, using only one computer to send commands to multiple Fire TVs.

### 3.4.3 Comparing Roku and Fire TV

In this section, we analyze the (ATS) domains accessed by the apps in the Roku and Fire TV testbed datasets. We first provide an overview of the datasets, and analyze how many (ATS) domains apps contact. We then look closer at the eSLDs and third party ATS domains

| Number of | Roku | Fire TV | Both |
|---|---|---|---|
| Apps exercised | 1044 | 1010 | 128 |
| Fully qualified domain names (FQDN) | 2191 | 1734 | 578 |
| FQDNs accessed by multiple apps | 669 | 603 | 199 |
| URL paths | 13899 | 240713 | 74 |

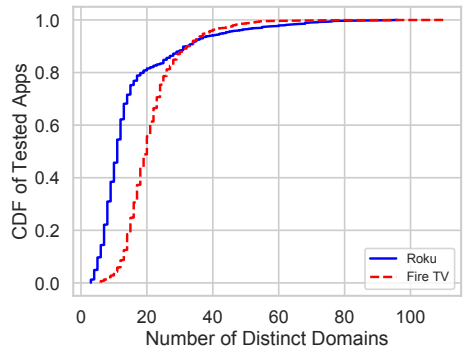
Table 3.2: Summary of the Roku and Fire TV testbed datasets. The rightmost column summarizes the intersection between the two testbed datasets. For example, there are 128 apps that are present both in the Roku dataset and in the Fire TV dataset.

that are contacted by the most apps, including which parent organizations they belong to. Furthermore, we compare the top third party ATS domains to those of Android. Finally, we compare the domains accessed by apps that are present on both testbed platforms.

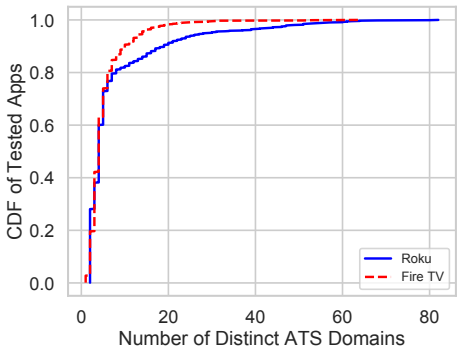
**Overview.** The datasets collected using Rokustic and Firetastic are summarized in Table 3.2. For Roku, we discover 2191 distinct FQDNs, 669 of which are contacted by multiple apps. For Fire TV, we discover 1734 distinct FQDNs, 603 of which are contacted by multiple apps. We also find 578 FQDNs that appear in both datasets, 199 of which are contacted by multiple apps. Our automation uncovers approximately twice as many FQDNs as [98], possibly due to longer experiments and different app exploration goals. We further detail this in Appendix A.3.

Leveraging the blocklists from Section 3.5, we identify 314 ATS domains that are unique to the Roku dataset, 285 that are unique to the Fire TV dataset, and an overlap of 227 between the two datasets. When considering eSLDs of the ATS domains, we find 68 eSLDs that are unique to the Roku dataset, 100 that are unique to the Fire TV dataset, and an overlap of 138 eSLDs. These numbers suggest that the ATS ecosystems of the two platforms have substantial differences, which we analyze in further detail later in this section.

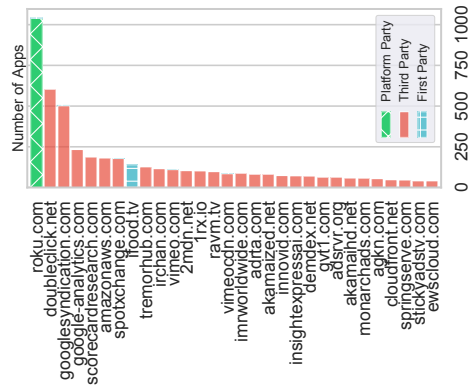
**Number of (ATS) Domains Contacted.** Figure 3.3a presents the empirical CDF of the number of distinct domains each app in the two testbed datasets contacts. Fire TV apps appear more "chatty": most Fire TV apps contact about twice as many domains as the Roku apps. However, when we consider the number of ATS domains contacted per app in



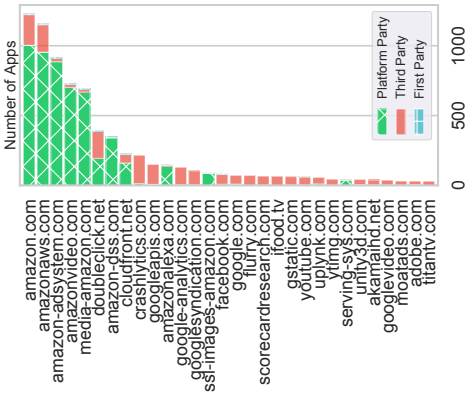

(a) **Roku & Fire TV**: Distinct domains per app.



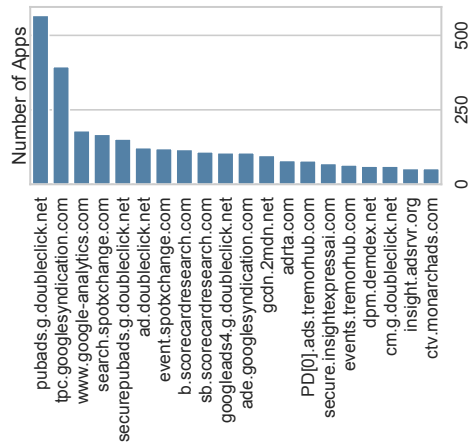

(b) **Roku & Fire TV**: Distinct ATS domains per app. A domain is considered an ATS if it is labeled as so by any of the blocklists considered in Section 3.5.



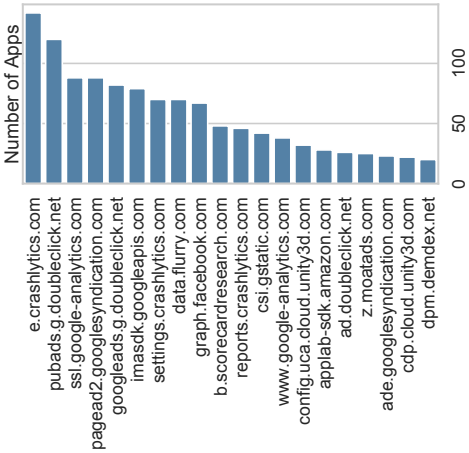

(c) **Roku**: Top-30 eSLDs.




(d) **Fire TV**: Top-30 eSLDs.




(e) **Roku**: Top-20 third party ATS domains.




(f) **Fire TV**: Top-20 third party ATS domains.

Figure 3.3: Analysis of domain usage per app and the top domains across all apps in the Roku and Fire TV testbed datasets.

Figure 3.3b, the vast majority (80%) of apps from the two platforms behave similarly.

On the positive side, for both platforms, around 60% of the apps contact only a small handful ATS domains. Yet, about 10% of the Roku and Fire TV apps contact 20+ and 10+ ATS domains, respectively. These concerning apps come from a small set of developers. For instance, for Roku, "Future Today Inc." [46], "8ctave ITV", and "StuffWeLike" [143] are responsible for 51%, 13%, and 11%, respectively, of these apps. On the Fire TV side, "HTVMA Solutions, Inc." [89] is responsible for 15% of the apps, and "Gray Television, Inc." [55] is responsible for 12% of the apps.

**Key Players.** Figures 3.3c (Roku) and 3.3d (Fire TV) present the top-30 eSLDs in terms of the number of apps that contact a subdomain of the eSLD. We define an eSLD's *app penetration* as the percentage of apps in the dataset that contact the eSLD.

*Platform.* The top eSLD of each platform has 100% app penetration and belongs to the platform operator. While these eSLDs also cover subdomains that provide functionality, we note that both platform operators are engaged in advertising and tracking, as shown in Figure A.3 in Appendix A.5.2, which separates traffic to the eSLDs in Figures 3.3c and 3.3d by ATS and non-ATS FQDNs.

*Third Parties.* As evident from Figures 3.3c and 3.3d, Alphabet has a strong presence in the ATS space of both platforms, with *.doubleclick.net, an ad delivery endpoint, achieving 58% and 35% app penetration for Roku and Fire TV, respectively. Its analytic services also rank high, with google-analytics.com in the top-20 for both platforms and crashlytics.com in the top-10 for Fire TV. To better understand additional key third party ATSes, we strip away the platform-specific endpoints and report the top-20 third party ATS FQDNs for Roku and Fire TV in Figures 3.3e and 3.3f, respectively. We note that both platforms use distinct third party ATSes. For example, SpotX (*.spotxchange.com), which serves video ads, is a significant player in the Roku ATS space with 17% app penetration, but only maintains 1%

app penetration for Fire TV. Even when considering the smaller players, we see little overlap between the two platforms, suggesting these players focus their efforts on a single platform. For example, Kantar Group's insightexpressai.com analytics service has 7% app penetration on the Roku platform, but only 0.01% on the Fire TV platform.

**Parent Organization Analysis.** We further analyze the parent organizations of Roku and Fire TV third party ATS endpoints in Figure 3.4, using the method described earlier in Section 3.2. Interestingly, the set of top third party organizations is rather diverse, with only a slight overlap in the shape of Adobe Systems and comScore, possibly suggesting that the remaining organizations focus their efforts on a single platform. Fire TV shows gaming and social media ATSes from Unity Tech and Facebook, whereas Roku exhibits more traffic to ATSes from companies that focus on video ads such as The Trade Desk, Telaria, and RTL Group. Similar to the in the wild organization analysis in Figure 3.2, we again note that Alphabet dominates the set of third party ATSes on both Roku and Fire TV.

**Comparison to Android ATS Ecosystem.** Next, we compare the top-20 third party ATS endpoints in our Roku and Fire TV datasets (Figures 3.3e and 3.3f) with those reported for Android [113].

*Roku vs. Android.* The key third party ATSes in Roku (Figure 3.3e) differ from the Android platform. For example, SpotX (*.spotxchange.com) and comScore (*.scorecardresearch.com) both have a strong presence on Roku, but are not among the key players for Android. In contrast, Facebook's graph.facebook.com is the second most popular ATS domain on Android, but insignificant on Roku. The set of top third party ATSes in Roku is also more diverse and includes smaller organizations such as Pixalate (adrta.com) and Telaria (*.tremorhub.com). While Alphabet has a strong foothold in both ATS ecosystems, it is less significant for Roku (9 out 20 ATS FQDNs are Alphabet-owned, vs. 16 out of 20 for Android).

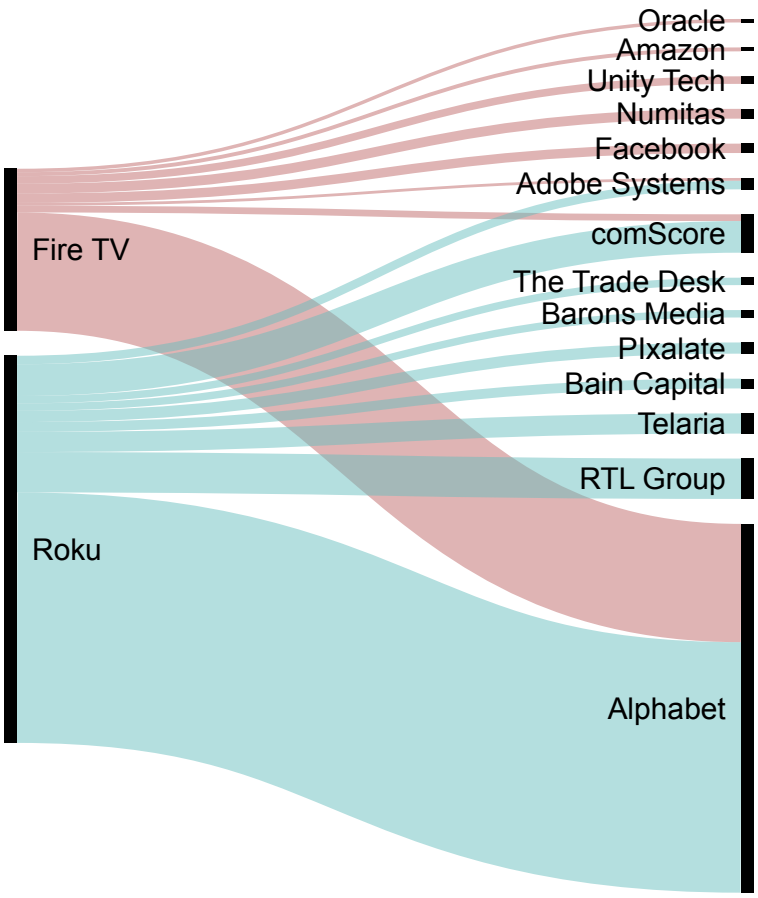



Figure 3.4: Mapping of platforms measured in our testbed environment to the parent organizations of the top-20 *third-party* ATS domains their apps contact. The width of an edge indicates the number of apps that contact each organization.

*Fire TV vs. Android.* In contrast to Roku, Fire TV is more similar to Android: we see an overlap of 9 FQDNs, 7 of which are owned by Alphabet. This is expected, given that Fire TV is based off of Android and thus natively supports the ATS services of Android. Facebook (graph.facebook.com) and Verizon (data.flurry.com) both have a strong presence on both Fire TV and Android. Some of the third party ATSes observed for Fire TV, which were not present for Android, include comScore, Adobe (dpm.demdex.net), and Amazon (applab-sdk.amazon.com).

**Common Apps in Roku and Fire TV.** Next, we compare the Roku and Fire TV datasets at the app-level by analyzing the FQDNs accessed by the set of apps that appear on both

platforms, referred to as *common apps*. Recall from Table 3.2 that the datasets collected using Rokustic and Firetastic contain a total of 128 common apps. We identified common apps by fuzzy matching app names since they sometimes vary slightly for each platform, e.g., "TechSmart.tv" on Roku vs. "TechSmart" on Fire TV. We further cross-referenced with the developer's name to validate that the apps were indeed the same, e.g., both TechSmart apps are created by "Future Today." The 128 common apps contact a total of 1248 distinct FQDNs. Out of these, 597 FQDNs are exclusively contacted by Roku apps, 496 are exclusively contacted by Fire TV apps, and only 155 FQDNs are contacted by both Roku and Fire TV apps.

Figure 3.5 reports the amount of overlapping and non-overlapping FQDNs for the top-60 common apps (in terms of the number of distinct FQDNs that each app contacts). In general, the set of FQDNs contacted by both the Roku and the Fire TV versions of the same app is much smaller than the set of platform-specific FQDNs. From inspecting the common FQDNs for some of the apps in Figure 3.5, we find that these generally include endpoints that serve content. For example, for Mediterranean Food, the only two common FQDNs are subdomains of ifood.tv, which belong to the parent organization behind the app. This makes intuitive sense as the same app presumably offers the same content on both platforms and must therefore access the same servers to download said content. On the other hand, the platform-specific domains contain obvious ATS endpoints such as ads.yahoo.com and ads.stickyadstv.com for the Roku version of the app, and aax-us-east.amazon-adsystem.com and mobileanalytics.us-east-1.amazonaws.com for the Fire TV version of the app. In conclusion, our analysis of common apps reveals (to our surprise) little overlap in the ATS endpoints they access, which further suggests that the smart TV ATS ecosystem is segmented across platforms.

**Takeaways.** The ATS ecosystems of the Roku and Fire TV platforms seem to differ substantially: (1) the full set of ATS domains contacted by apps in the two datasets have

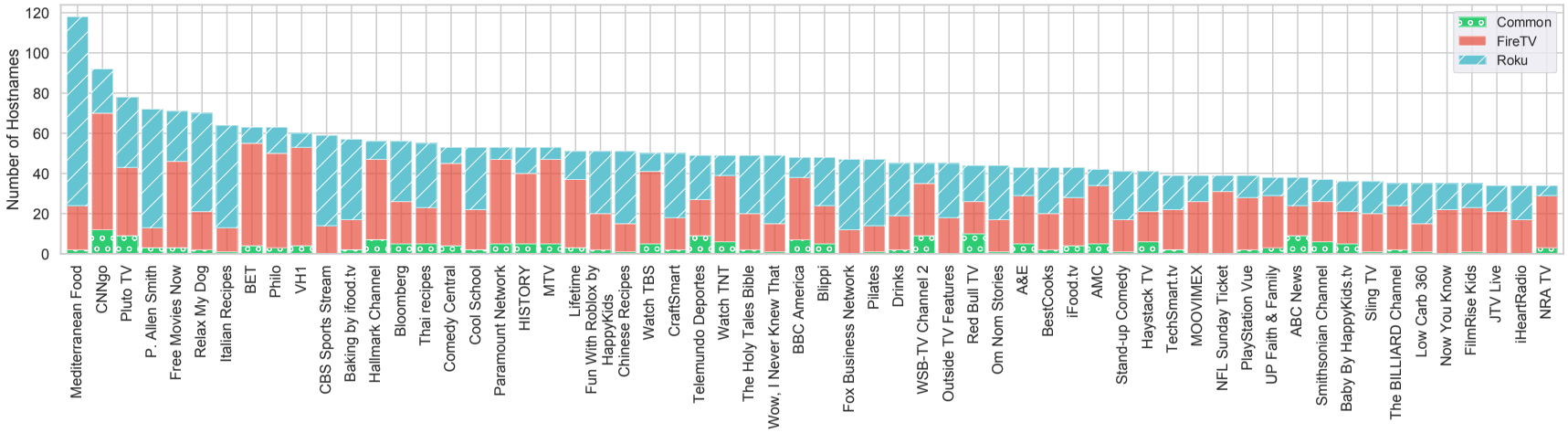



Figure 3.5: Top-60 common apps (apps present in both testbed datasets) ordered by the number of domains that each app contacts. Considering all 128 common apps, there are 597 domains which are exclusive to Roku apps, 496 domains which are exclusive to Fire TV apps, and 155 domains which are contacted by both the Roku and the Fire TV versions of the same app.

little overlap; (2) some organizations are key players on one platform, but almost absent on the other (e.g., SpotX has a significant presence on Roku, but is almost absent on Fire TV, whereas Facebook has a reasonable foothold on Fire TV, but almost zero presence on Roku); and (3) apps present in both datasets have little overlap in terms of the ATS domains they contact. Finally, we find that the key third party ATS players on Android have little overlap with Roku, but substantial overlap with Fire TV, which intuitively makes sense as Fire TV is built on top of Android.

## 3.5 Blocklists for Smart TVs

In this section, we evaluate four well-known DNS-based blocklists' ability to prevent smart TVs from accessing ATSes, and their adverse effects on app functionality. We further demonstrate how the datasets resulting from automated app exploration may aid in curating new candidate rules for blocklists.

### 3.5.1 Evaluating Popular DNS Blocklists

DNS-based blocking solutions such as Pi-hole [12] are used to prevent in-home devices, including smart TVs, from accessing ATS domains [147]. To block advertising and tracking traffic, they essentially "blackhole" DNS requests to known ATS domains. Specifically, they match the domain name in the DNS request against a set of blocklists, which are essentially curated hosts files that contain rules for well-known ATS domains. If the domain name is found in one of the blocklists, it is typically mapped to 0.0.0.0 or 127.0.0.1 to prevent outbound traffic to that domain [109].

**Setup.** We evaluate the following blocklists:

1. **Pi-hole Default** (PD): We test blocklists included in Pi-hole's default configuration [10] to imitate the experience of a typical Pi-hole user. This set has seven hosts files including Disconnect.me ads, Disconnect.me tracking, hpHosts, CAMELEON, MalwareDomains, StevenBlack, and Zeustracker. PD contains a total of about 133K entries.

2. **The Firebog** (TF): We test nine advertising and five tracking blocklists recommended by "The Big Blocklist Collection" [160], to emulate the experience of an advanced Pi-hole user. This includes: Disconnect.me ads, hpHosts, a dedicated blocklist targeting smart TVs, and hosts versions of EasyList and EasyPrivacy. TF contains 162K entries total.

3. **Mother of all Ad-Blocking** (MoaAB): We test this curated hosts file [11] that targets a wide-range of unwanted services including advertising, tracking (cookies, page counters, web bugs), and malware (phishing, spyware) to again imitate the experience of an advanced Pi-hole user. MoaAB contains a total of about 255K entries.

4. **StopAd** (SATV): We test a commercial smart TV focused blocklist by StopAd [76]. This list particularly targets Android based smart TV platforms such as Fire TV. We extract StopAd's list by analyzing its APK using Android Studio's APK Analyzer [53]. SATV contains a total of about 3K entries.

We applied these blocklists to both our in the wild and testbed datasets and we report the results next.

**Block Rates.** We start our analysis by comparing how many FQDNs are blocked by the different blocklists. We define a blocklist's *block rate* as the number of distinct FQDNs that are blocked by the list, over the total number of distinct FQDNs in the dataset. Table 3.3 compares the block rates of the aforementioned blocklists, when the blocklists are applied to the domains in our in the wild and testbed datasets. Overall, we note that TF, closely

| Platform | # Domains | Block Rate (%) PD | TF | MoaAB | SATV |
|---|---|---|---|---|---|
| **Dataset obtained "in the wild"** | | | | | |
| Apple | 3179 | 10% | 13% | 12% | 5% |
| Samsumg | 1765 | 14% | 19% | 15% | 8% |
| Chromecast | 1576 | 9% | 15% | 15% | 5% |
| Roku | 2312 | 15% | 19% | 18% | 7% |
| Vizio | 942 | 16% | 18% | 16% | 11% |
| LG | 627 | 45% | 54% | 50% | 27% |
| Sony | 119 | 16% | 24% | 16% | 7% |
| **Dataset obtained in our testbed** | | | | | |
| Roku | 2191 | 17% | 22% | 20% | 9% |
| Fire TV | 1734 | 22% | 27% | 22% | 9% |

Table 3.3: Block rates of the four blocklists when applied to the domains in our datasets.

followed by MoaAB and PD, blocks the highest fraction of domains across all of the platforms in both the in the wild and testbed datasets. SATV is the distant last in terms of block rate. It is noteworthy that TF blocks more domains than MoaAB despite being about one-third shorter. We surmise this is because TF includes a smart TV focused hosts file, and thus catches more relevant smart TV ATSes. This finding shows that the size of a blocklist does not necessarily translate to its coverage.

**Blocklist Mistakes.** Motivated by the differences in the block rates of the four blocklists, we next compare them in terms of false negatives (FN) and false positives (FP). False negatives occur when a blocklist does not block requests to ATSes and may result in (visually observable) ads or (visually unobservable) PII exfiltration. False positives occur when a blocklist blocks requests that enable app functionality and may result in (visually observable) app breakage.

We first systematically quantify visually observable false negatives and false positives of blocklists by interacting with a sample of apps from our testbed datasets while coding for ads and app breakage. We sample 10 Roku apps and 10 Fire TV apps, including the top-4 free apps, three apps that are present on both platforms, and an additional three randomly selected apps. We test each app five times: once without any blocklist and four times

where we individually deploy each of the aforementioned blocklists. During each experiment, we attempt to trigger ads by playing multiple videos and/or live TV channels and fast-forwarding through video content. We take note of any functionality breakage (due to false positives) and visually observable missed ads (due to false negatives). We differentiate between minor and major functionality breakage as follows: *minor breakage* is when the app's main content remains available, but the app suffers from minor user interface glitches or occasional freezes; and *major breakage* is when the app's content becomes completely unavailable or the app fails to launch.

**Missed Ads vs. Functionality Breakage.** Table 3.4 summarizes the results of our manual analysis for missed ads and functionality breakage. Overall, we find that none of the blocklists are able to block ads from all of the sampled apps while avoiding breakage. In particular, none of the blocklists are able to block ads in YouTube and Pluto TV (available on both Roku and Fire TV). Across different lists, PD seems to achieve the best balance between blocking ads and preserving functionality.

For Roku, PD and TF perform similarly. While TF is the only list that blocks ads in Sony Crackle, both lists miss ads in YouTube and Pluto TV. TF majorly breaks three apps, while PD only majorly breaks one app. MoaAB is unable to block ads in four apps and majorly breaks only one app. SATV does not cause any breakage, but is unable to block ads in six apps.

For Fire TV, PD again seems to be the most effective at blocking ads while avoiding breakage, but is still unable to block ads in one app (Pluto TV) and majorly breaks two apps. TF is also unable to block ads in Pluto TV, but majorly breaks four apps. MoaAB is unable to block ads in three apps and majorly breaks three apps (one minor). SATV is unable to block ads in four apps and majorly breaks one app (two minor).

**Takeaways.** All blocklists suffer from a non-trivial amount of visually observable FPs and

| | | App Name | No List | | PD | | TF | | MoaAB | | SATV | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | No Ads | No Breakage | No Ads | No Breakage | No Ads | No Breakage | No Ads | No Breakage | No Ads | No Breakage |
| **Roku** | Common | Pluto TV | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |
| | | iFood.tv | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ |
| | | Tubi | ✓ | ✓ | ✓ | ✓ | — | ✖ | ✓ | ✓ | ✓ | ✓ |
| | Top | YouTube | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |
| | | CBS News Live | ✗ | ✓ | ✓ | ✖ | ✓ | ✖ | ✓ | ✖ | ✓ | ✓ |
| | | The Roku Channel | ✗ | ✓ | ✓ | ✓ | ✓ | ✖ | ✓ | ✓ | ✗ | ✓ |
| | | Sony Crackle | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ |
| | Random | WatchFreeComedyFlix | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ |
| | | Live Past 100 Well | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | | SmartWoman | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Fire TV** | Common | Pluto TV | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✖ |
| | | iFood.tv | ✗ | ✓ | ✓ | ✓ | — | ✖ | — | ✖ | ✓ | ✓ |
| | | Tubi | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Top | Downloader | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | | The CW for Fire TV | ✗ | ✓ | — | ✖ | — | ✖ | — | ✖ | ✗ | ✓ |
| | | FoxNow | ✗ | ✓ | — | ✖ | — | ✖ | ✗ | ✓ | ✗ | ✓ |
| | | Watch TNT | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Random | KCRA3 Sacramento | ✗ | ✓ | ✓ | ✓ | — | ✖ | — | ✖ | ✗ | ✗ |
| | | Watch the Weather Channel | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| | | Jackpot Pokers by PokerStars | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |

Table 3.4: Missed ads and functionality breakage for different blocklists when employed during manual interaction with 10 Roku apps and 10 Fire TV apps. For "No Ads", a checkmark (✓) indicates that no ads were shown during the experiment, a cross (✗) indicates that some ad(s) appeared during the experiment, and a dash (—) indicates that breakage prevented interaction with the app altogether. For "No Breakage", a checkmark (✓) indicates that the app functioned correctly, a cross (✗) indicates minor breakage, and a bold cross (✖) indicates major breakage.

FNs. Some blocklists (e.g., PD and TF) are clearly more effective than others. Interestingly, SATV, which is curated specifically for smart TVs, did not perform well.

### 3.5.2 False Negatives

In this section, we demonstrate how datasets generated using automation tools such as Rokustic and Firetastic enable blocklist curators to identify potential false negatives in the blocklist. In particular, we observe that the more apps that contact an FQDN, the more likely

| **Fully Qualified Domain Name** | PD | TF | MoaAB | SATV |
|---|---|---|---|---|
| p.ads.roku.com | ✗ | ✗ | ✗ | ✗ |
| ads.aimitv.com | ✗ | ✗ | ✗ | ✗ |
| adtag.primetime.adobe.com | ✗ | ✗ | ✗ | ✗ |
| ads.adrise.tv | ✗ | ✓ | ✗ | ✗ |
| ads.samba.tv | ✗ | ✓ | ✗ | ✗ |
| tracking.sctv1.monarchads.com | ✗ | ✓ | ✗ | ✗ |
| ads.ewscloud.com | ✗ | ✗ | ✗ | ✗ |
| trackingrkx.com | ✗ | ✗ | ✗ | ✗ |
| us-east-1-ads.superawesome.tv | ✗ | ✗ | ✗ | ✗ |
| track.sr.roku.com | ✗ | ✗ | ✗ | ✗ |
| router.adstack.tv | ✗ | ✗ | ✗ | ✗ |
| metrics.claspws.tv | ✗ | ✗ | ✗ | ✗ |
| customerevents.netflix.com | ✗ | ✓ | ✗ | ✗ |
| event.altitude-arena.com | ✗ | ✓ | ✗ | ✗ |
| ads.altitude-arena.com | ✗ | ✓ | ✗ | ✗ |
| myhouseofads.firebaseio.com | ✗ | ✗ | ✗ | ✗ |
| mads.amazon.com | ✗ | ✗ | ✗ | ✗ |
| ads.aimitv.com.s3.amazonaws.com | ✗ | ✗ | ✗ | ✗ |
| analytics.mobitv.com | ✗ | ✗ | ✗ | ✗ |
| events.brightline.tv | ✗ | ✗ | ✗ | ✗ |
| ctv.monarchads.com | ✗ | ✓ | ✗ | ✗ |
| ads.superawesome.tv | ✗ | ✓ | ✗ | ✗ |
| adplatform-static.s3-us-west-1.amazonaws.com | ✗ | ✗ | ✗ | ✗ |
| kraken-measurements.s3-external-1.amazonaws.com | ✗ | ✗ | ✗ | ✗ |
| kinstruments-measurements.s3-external-1.amazonaws.com | ✗ | ✗ | ✗ | ✗ |
| venezia-measurements.s3-external-1.amazonaws.com | ✗ | ✗ | ✗ | ✗ |
| ad-playlistserver.aws.syncbak.com | ✗ | ✗ | ✗ | ✗ |

Table 3.5: Examples of potential false negatives for the four DNS-based blocklists found using app penetration analysis and keywords search ("ad", "ads", "analy", "track", "hb" (for heartbeat), "score", "event", "metrics", "measure").

it is that the FQDN is an ATS. This is intuitive and consistent with a similar observation previously made in the mobile ecosystem [113].

We first use simple keywords such as "ad", "ads", and "track" to shortlist obvious ATS domains in our datasets. While keyword search is not perfect, this simple approach identifies several obvious false negatives (we provide the full list in Table 3.5). For example, p.ads.roku.com and adtag.primetime.adobe.com are advertising/tracking related domains which are not blocked by any of the lists.

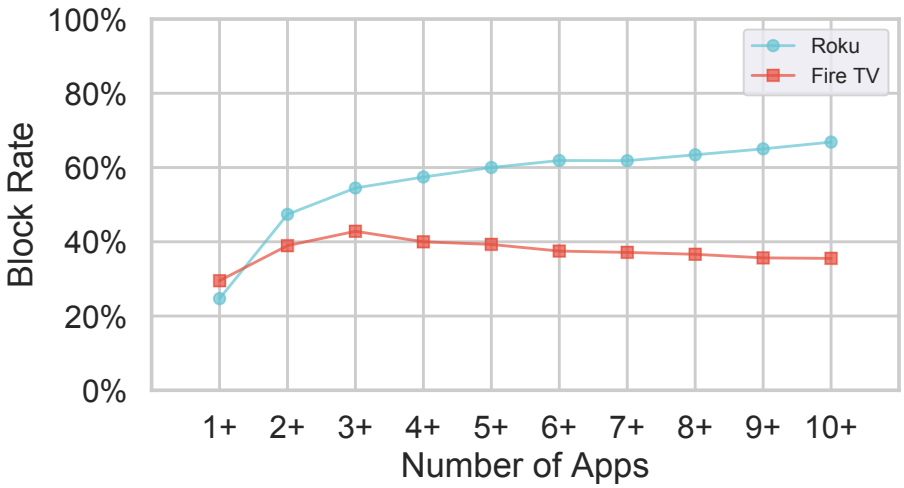



Figure 3.6: Block rates as a function of the number apps that contact an FQDN. For the horizontal axis, "2+" represents the set of FQDNs that are contacted by 2 or more apps. For Roku, the more apps that contact an FQDN, the more likely it is that the FQDN is an ATS, according to the blocklists. The same is not true for Fire TV because platform services start to dominate the set of FQDNs that are accessed by many apps, and platform services are often not blocked.

We observe that many of these false negatives (i.e., missed ATS domains) are contacted by multiple apps in our testbed datasets. For example, p.ads.roku.com is accessed by more than 100 apps in our Roku testbed dataset. To gain further insight into potential false negatives, we study whether the likelihood of being blocked is impacted by the number of apps that access a domain. Figure 3.6 plots the block rates for the union of the four blocklists as a function of FQDNs' occurrences across apps in our testbed datasets. We note that the block rate substantially increases for the domains that appear across multiple apps. For example, the block rate almost doubles for domains contacted by two or more apps as compared to domains contacted by a single or more apps. Domains that are contacted by multiple different apps are therefore more likely to belong to third party ATS libraries included by smart TV apps.

## 3.6 PII Exposures in Smart TVs

In this section, we examine our testbed datasets from Section 3.4 for exposure of personally identifiable information (PII) and evaluate the effectiveness of blocklists in preventing it. We define PII exposure as the transmission of any PII from the smart TV device to any Internet endpoint. We identify PII values (such as advertising ID and serial number) through the settings menus and packaging of each device. Since trackers are known to encode or hash PII [39], we compute the MD5 and SHA1 hashes for each of the PII values. We then search for these PII values in the HTTP header fields and URI path. Recall from Section 3.4 that we can analyze HTTP information even for encrypted flows in the Fire TV dataset due to AntMonitor's TLS decryption [135, 7], but can only analyze unencrypted flows in Roku. The number of PII exposures reported for Roku should therefore be considered a lower bound.

**Overview.** Table 3.6 reports the PII exposures for both testbed platforms. For Roku, the majority of the PII exposures are to third parties, whereas for Fire TV they are to the platform-specific party. The blocklists adequately prevent exfiltration of PII to third parties, blocking 74% or more of third party domains for all the PII values considered for Roku. For Fire TV, the blocklists mitigate the majority of advertising ID exposures, blocking 71% or more of the involved domains, but are not as effective in preventing exposures of serial number and device ID to third parties and platform destinations.

**Differentiating PII Exposures.** Inspired by [95], we adopt a simple approach for distinguishing between "good" and "bad" PII exposures that treats PII exposures to third parties as a higher threat to privacy than PII exposures to first parties. PII exposures to first parties are generally warranted as they likely have a functional purpose, such as personalization of content (e.g., to keep track of where the user paused a video). For example, the Roku app "Acacia Fitness & Yoga Channel" from "RLJ Entertainment", sends a request to the first party domain api.rlje.net with a URI path of "/cms/acacia/today/roku/content/browse.json"

| PII | Roku Testbed Dataset (Apps & eSLDs) 1st Party | 3rd Party | Other | Total | Fire TV Testbed Dataset (Apps & eSLDs) 1st Party | 3rd Party | Platform Party | Other | Total |
|---|---|---|---|---|---|---|---|---|---|
| Advertising ID | 4<br>4<br>25% | 263<br>36<br>88% | 6<br>3<br>0% | 269<br>42<br>81% | 17<br>7<br>25% | 53<br>31<br>78% | 715<br>4<br>71% | 5<br>5<br>40% | 725<br>39<br>71% |
| Serial Number | 48<br>17<br>5% | 128<br>16<br>74% | 2<br>2<br>0% | 174<br>34<br>36% | 10<br>3<br>0% | 51<br>4<br>33% | 867<br>4<br>9% | 2<br>2<br>0% | 881<br>9<br>12% |
| Device ID | - | - | - | - | 19<br>8<br>0% | 153<br>27<br>36% | 819<br>5<br>14% | 10<br>11<br>21% | 856<br>43<br>31% |
| Username | 4<br>4<br>0% | 1<br>1<br>100% | - | 5<br>5<br>20% | 1<br>2<br>0% | 2<br>2<br>100% | 1<br>1<br>100% | - | 4<br>5<br>40% |
| MAC | - | - | - | - | - | 2<br>2<br>100% | - | - | 2<br>2<br>100% |
| Location | - | 42<br>2<br>100% | - | 42<br>2<br>100% | - | 27<br>7<br>90% | 2<br>2<br>100% | - | 28<br>7<br>90% |

Table 3.6: **Applications / eSLDs / % Distinct FQDNs Blocked.** Number of apps that expose PII, number of distinct eSLDs that receive PII from these apps, and percentage of distinct subdomains of the eSLDs that are blocked by the blocklists. We further separate by party as defined in Section 3.2. Roku platform column omitted since we do not observe PII exposures to platform domains.

while including the device's serial number in an HTTP header field, suggesting that the serial number is used to personalize today's featured content.

On the other hand, PII exposures to third parties are generally unwarranted as they typically do not have a functional purpose. This extends to cases where the app retrieves its content through a third party CDN as the personalization could be achieved by first sending the PII to the first party server which could then respond to the app with the CDN URL for the content to be retrieved. For instance, the Roku app "ArmchairTourist" from "ArmchairTourist Video Inc." sends a request to the third party domain ads.adrise.tv with a URI path of "/track/impression..." that encodes the device's serial number, suggesting that the PII is used to track what ads have been shown to the user.

**Exposures to Platform Party.** For Fire TV, the majority of the exposures of serial number and device ID to platform destinations seem to be for advertising and tracking purposes. For example, 697 apps send the serial number and device ID (and advertising ID) to the platform endpoint aviary.amazon.com with a URI path of "/GetAds", and 53 apps send the serial number to dna.amazon.com, with a URI path of "/GetSponsoredTileAds." Judging from these paths, it would seem like the advertising ID alone would be sufficient and the more appropriate PII. On the other hand, some exposures seem to serve a functional purpose. For example, 67 apps send the serial number to atv-ext.amazon.com, with varying URI paths containing "/cdp/." We surmise that this domain serves as "Content Delivery Platform(s)" [20], allowing apps to personalize content without user login. Specifically, we see paths such as "/cdp/playback/GetDefaultSettings" coupled with an "x-atv-session-id" HTTP header field.

**Joint Exposure of Static and Dynamic PII.** We observe that some apps send the advertising ID *alongside* other static identifiers. This goes against recommended developer practices, where apps and ATSes should only rely on dynamic identifiers that can be refreshed like advertising ID to give users the ability to opt out of being tracked. Aside from the 697 Fire TV apps that expose the advertising ID alongside the serial number and the device ID discussed earlier, we observe 10 Roku apps (including prominent ones such as Pluto TV and PBS) that send both the advertising ID and the serial number to third parties (subdomains of scorecardreasearch.com and youboranqs01.com). Similarly, 12 Fire TV apps send the advertising ID alongside the device ID to third party destinations such as ads.adrise.tv and ctv.monarchads.com. Thus, these practices allow ATSes to link an old advertising ID to the new value by joining on the static identifiers.

**Leveraging Missed PII Exposures to Improve Blocklists.** The above indicate another direction for improving blocklist curation for smart TVs. By deploying tools such as Rokustic and Firetastic and searching the network traces for PII exposures, blocklist curators can

generate candidate rules that can then be examined manually. Using this approach, we identified 38 domains in the Roku dataset and 30 in the Fire TV dataset that receive PII, but were not blocked by any list. These numbers are conservative as we exclude location and account name, which are likely to be used for legitimate purposes, such as logging in or serving location-based content. The identified domains include obvious ATSes such as ads.aimitv.com and ads.ewscloud.com. Another noteworthy mention is hotlist.samba.tv: Samba TV uses Automatic Content Recognition to provide content suggestions on smart TVs, but this comes at the cost of targeted advertising that even propagates onto other devices in the home network [132].

**Takeaways.** Hundreds of Roku and Fire TV apps expose PII, mostly to third parties and the platform-specific party. For Fire TV, we observe that most of the exposures to the platform-specific party seem to be for advertising and tracking purposes. We observe that many Roku and Fire TV apps send the advertising ID *alongside* a static identifier (e.g., serial number), which enables the ATS to relink a user profile associated with an old advertising ID to a new advertising ID, thus eliminating the user's ability to opt out. The blocklists generally do well at preventing exposure of the advertising ID on both platforms, but are less successful at preventing exposures of serial number and device ID on Fire TV.

## 3.7 Summary, Limitations, and Directions

**Summary.** In this chapter, we performed one of the first comprehensive measurement studies of the emerging smart TV advertising and tracking service (ATS) ecosystem. To that end, we analyzed and compared: (i) a realistic but small in the wild dataset (57 smart TV devices from seven different platforms, with coarse flow-level information); and (ii) two large testbed datasets (top-1000 apps on Roku and Fire TV, tested systematically, with granular per app and packet-level information). Our work establishes that the smart TV

ATS ecosystem is fragmented across different smart TV platforms and is different from the mobile ATS ecosystem. We also evaluated popular DNS-based blocklists' ability to prevent smart TVs from accessing ATSes, and found that all lists suffer from missed ATSes and incur app breakage. Finally, we examined our testbed datasets for exposure of personally identifiable information (PII) and discovered that hundreds of apps send PII to third parties and the platform-specific party, mostly for advertising and tracking purposes.

**Limitations.** Our methodology has its limitations. First, the automated app exploration may not always result in content (video/audio) playback, which may impact the (ATS) domains discovered. We evaluate the extent of this limitation in Appendix A.3. We find that Rokustic and Firetastic perform on par with concurrent work [98] in terms of playback success, and that they manage to discover a large fraction of the number of domains discovered during manual interaction. Second, apps may prevent TLS interception through use of certificate pinning, which may prevent Firetastic from observing PII exposures in encrypted traffic. We assess the decryption failures in Appendix A.4: we find that for 80% of the apps in our Fire TV testbed dataset, TLS interception only fails for 20% or fewer of an app's TLS connections. Third, our analysis of FPs and FNs in DNS-based blocklists in Section 3.5.1 does not account for DNS over HTTPS (DoH), nor static advertisements, thus it may overcount blocklist FNs for these cases.

**Future Work.** Our findings motivate more research to further understand smart TVs and to develop privacy-enhancing solutions specifically designed for each smart TV platform. For example, more research is needed to curate accurate, fine-grained (as opposed to DNS-based), and platform-specific blocklists. To foster further research along this direction, we make our tools, Rokustic and Firetastic, and testbed datasets publicly available [158, 157, 153]. We intend to further improve our tools' ability to thoroughly explore smart TV apps along the directions discussed in Appendix A.3.

# Chapter 4

# FingerprinTV: Fingerprinting Smart TV Apps

## 4.1 Overview

In this chapter, we further characterize the privacy implications of smart TVs by studying if a passive, in-network observer can identify what app is in use on a smart TV from the network traffic it generates, even if the traffic is encrypted. This type of problem is commonly referred to as *network fingerprinting* and has been studied extensively in the context of websites [59, 144, 27, 81, 58, 83, 105, 36, 163, 104, 57, 138, 103, 26, 139, 102, 141, 85, 60], desktop and mobile applications [19, 145, 146, 155], and Internet of Things (IoT) devices with narrow functionality, such as smart light bulbs and smart plugs [35, 101, 115, 151, 13]. However, to the best of our knowledge, no work has explored the feasibility of smart TV app fingerprinting at scale. This constitutes a significant gap in the literature for the following reasons.

First, in the context of smartphones, app usage has been shown to be indicative of the

user's demographics, personality, interests, preferences, and habits [164]. Assuming this carries over to smart TVs, and considering that viewing history is regarded a cornerstone of programmatic TV advertising [88], smart TV app usage data is arguably a valuable asset for businesses engaged in targeted advertising. Since Internet Service Providers (ISP) are known to collect and use rich information about their customers for advertising purposes [45], it is important to quantify to what extent they can track their customers' smart TV app usage as well.

Second, recognizing that television viewing history may reveal sensitive information about the viewer, such as religion and sexual orientation, the U.S. Congress has enacted laws that obligate (cable) companies to obtain consent from consumers before they collect and/or disclose viewing history, e.g., the Cable Privacy Act and the Video Privacy Protection Act [45]. Although the smart TV app in use may not reveal the exact content the user is watching in that respective app, it may still reveal the content's theme (e.g., religious, political, adult etc.) as many smart TV apps limit their offerings to a certain genre. Furthermore, for smart TV apps that offer access to a single live stream, the smart TV app in use is synonymous with the content being watched. Smart TV app fingerprinting may thus potentially constitute a violation of said laws.

Third, privacy concerns aside, smart TV app fingerprinting may have potential security implications, and can also be used for quality-of-service optimization. For example, an attacker who is aware of a vulnerability in a certain smart TV app can, by observing the manifestation of its fingerprint in live traffic, time when to launch their attack. Additionally, since most smart TV app traffic is bandwidth intensive, as it often involves video streaming, network operators may be interested in the ability to dynamically prioritize traffic from smart TVs when certain apps are in use.

**Contributions.** In order to empirically assess the feasibility of fingerprinting smart TV apps, we take the following steps.

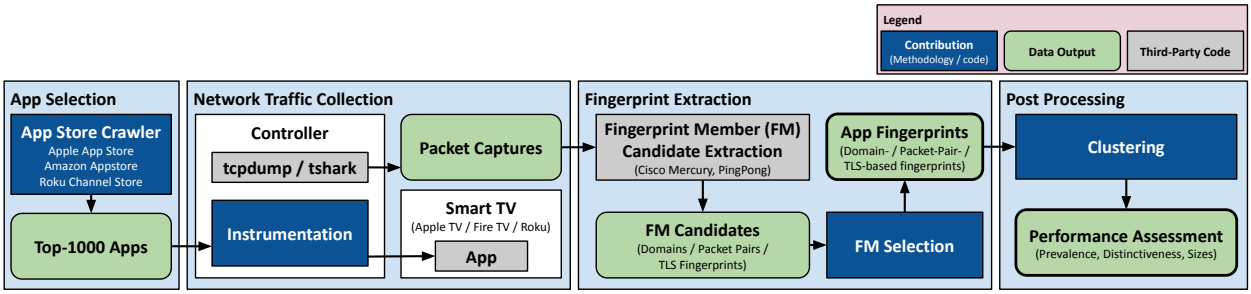



Figure 4.1: Overview of FingerprinTV, a system for assessing the feasibility of fingerprinting smart TV apps. The Controller is a computer with both a wired and a wireless network interface, which is configured as a wireless access point with NAT. The smart TV is associated with this wireless network. FingerprinTV first crawls the app store of the smart TV platform to determine a list of apps to test (see Section 4.2.1). Next, FingerprinTV collects multiple samples of the "on-launch" traffic of each app in this list (see Section 4.2.2). FingerprinTV then processes the collected traffic samples to identify consistently occurring traffic, referred to as fingerprints (see Section 4.3). Finally, FingerprinTV assesses the resulting fingerprints' discriminative power using a methodology we devise that is based on agglomerative (hierarchical) clustering (see Sections 4.4 and 4.5).

First, we design and implement FingerprinTV, a fully automated system for assessing the feasibility and effectiveness of fingerprinting smart TV apps. An overview of FingerprinTV is provided in Figure 4.1. FingerprinTV (1) automatically installs, and repeatedly launches apps, while collecting their network traffic; (2) extracts a domain-based fingerprint (DBF), a packet-pair-based fingerprint (PBF), and a TLS-based fingerprint (TBF) from the network traffic of each app; and (3) assesses the extracted fingerprints' performance, in terms of their prevalence, distinctiveness, and sizes. To that end, we propose a methodology based on agglomerative clustering that (i) provides flexibility to make a trade-off between a fingerprint's size and its reliability, and (ii) is general enough to be applicable across all three types of fingerprints. We consider DBFs, PBFs, and TBFs because they are lightweight and only rely on a few packets per network flow, yet remain applicable even if an app's traffic is encrypted using TLS.

Second, we deploy FingerprinTV to collect network traffic from the top-1000 most reviewed smart TV apps of the three most widely used smart TV platforms, namely Apple TV, Fire TV, and Roku [134]. To the best of our knowledge, this is the first large-scale smart

TV traffic dataset that also includes traffic from Apple TV apps. The dataset comprises 30K packet captures, 10K per platform.

Third, we analyze this dataset and provide the following findings and insights. We find that smart TV app fingerprinting is highly feasible and effective: even the least prevalent type of fingerprint manifests itself in at least 68% of apps of each platform. However, a fingerprint is only effective if it is distinct among other fingerprints. With this in mind, only DBFs and PBFs have merit, as up to 63% and 88% of the apps that exhibit DBFs and PBFs, respectively, have DBFs/PBFs that are distinct among those of other apps of the same platform. We also find that if DBFs and PBFs are used in conjunction, 78%, 89%, and 76% of Apple TV, Fire TV, and Roku apps, respectively, have distinct fingerprints. Furthermore, our results show that when multiple apps exhibit an identical fingerprint, a common explanation is that these apps stem from the same developer, or have been generated using the same "no code" toolkit. Finally, we find that among 80 apps that are made available on all three smart TV platforms, 76% exhibit a *different* fingerprint on each platform, thus making it possible to not only fingerprint the smart TV app itself, but also to identify which platform it is being used on.

Network traffic features, such as destinations, packet sizes, and TLS configuration parameters, have been used to create fingerprints in other contexts [75, 23, 56, 77, 108, 25, 24, 151, 121, 110], but our work is the first to consider them for smart TV apps. Our main contributions are the application and adaptation of existing families of techniques to smart TV apps, and a methodology that allows for automated extraction and evaluation of a range of fingerprints (i.e., DBFs, PBFs, and TBFs), in a uniform way and at scale. The implementation of our methodology can be used to repeat such assessments in the future, as smart TV apps and their fingerprints evolve. To that end, we plan to make the FingerprinTV code publicly available [152]. The FingerprinTV dataset has already been released and can be accessed through the link provided in [152].

**Outline.** The remainder of this chapter is structured as follows. Section 4.2 describes how we selected what apps to include in our study, and how FingerprinTV instruments the three smart TV platforms. Section 4.3 introduces the three fingerprinting techniques we consider in this chapter. Section 4.4 explains the methodology we devise for assessing the performance of the three fingerprinting techniques. Section 4.5 reports how many apps exhibit each type of fingerprint and how distinct those fingerprints are. Section 4.6 examines why some apps have identical fingerprints. Section 4.7 compares fingerprints across platforms. Section 4.8 examines the benefits of using different fingerprinting techniques in conjunction. Section 4.9 discusses possible defenses, limitations, and future directions. Section 4.10 concludes the chapter.

## 4.2 Data Collection

This section describes the design of FingerprinTV's data collection functionality (the App Selection and Network Traffic Collection boxes in Figure 4.1), and how we use FingerprinTV to collect network traffic from the top-1000 apps of each of the three most widely used smart TV platforms [134], namely Apple TV, Fire TV, and Roku. Section 4.2.1 explains how we use FingerprinTV to determine what apps to test. Section 4.2.2 explains how FingerprinTV automates interaction with the smart TVs. We summarize the dataset we collect using FingerprinTV in Section 4.2.3.

### 4.2.1 App Selection

To assess the feasibility of fingerprinting smart TV apps, we must test a large number of popular apps. To this end, we first use FingerprinTV to crawl the web interfaces of the app stores of the three smart TV platforms to obtain metadata for all available *free* apps of

each platform. Drawing inspiration from [98, 159], we then use the number of user ratings submitted for an app as a gauge for the number of users of that app, and pick the 1000 free apps with the most ratings for each platform. We refer to this selection as the respective platform's top-1000 apps. Below, we briefly discuss the crawlers and, to add context to the top-1000s, report how many apps they discovered.

**Apple TV.** The Apple TV platform was not covered in [98, 159]. We implement our own crawler that traverses Apple's "iTunes Preview" website [21], which lists metadata for all apps (across all Apple platforms) that are made available on Apple's App Store. The crawl was performed in January 2021 and returned a total of 3,841 free Apple TV apps.

**Fire TV.** We also implement our own crawler for Fire TV since [159] does not provide a crawler, and since the crawler [97] provided alongside [98] does not log all app metadata necessary for our purposes. Our crawler determines the free apps that are compatible with our Fire TV model ("Fire TV Cube 2nd Generation"). This crawl was performed in March 2021 and returned a total of 6,870 free Fire TV apps.

**Roku.** For Roku, we use the scripts [158] provided alongside [159] to crawl the Roku Channel Store. The crawl was performed in May 2021 and returned a total of 14,246 free Roku apps.

### 4.2.2 Automation

To enable collection of apps' on-launch network traffic at scale, FingerprinTV instruments the Apple TV, Fire TV, and Roku platforms to repeatedly launch each app while collecting the smart TV's network traffic. In this section, we first provide a platform-agnostic overview of the hardware setup and the instrumentation procedure, followed by platform-specific implementation details.

**Hardware Setup.** The general setup for the instrumentation is depicted in the Network

Traffic Collection box of Figure 4.1. The Controller is a computer with a wired and a wireless network interface that runs a Unix-based operating system. It is configured as a wireless access point (with NAT), and the smart TV under test is associated with this wireless network. The Controller is also responsible for executing the instrumentation code and logging the network traffic from/to the smart TV.

**Instrumentation Procedure.** Each app is subjected to the same four-step instrumentation procedure:

1. The instrumentation first installs the app.

2. Next, it performs three "warm-up" launches of the app, without logging any network traffic. In each warm-up launch, the instrumentation emulates one of three sequences of key presses on the smart TV's physical remote. The purpose of these warm-up launches is to dismiss any terms of service and/or initial setup screens (e.g., for selecting viewing preferences) that only appear at first launch or until the user has made their choice(s); this way, we fingerprint how the app behaves during daily use, and not during first use. Drawing inspiration from [98], we pick the three most common key press sequences for dealing with terms of service and initial setup screens among the top-100 apps.

3. The instrumentation then enters its main phase where it collects $L$ samples of the on-launch traffic for the app. For each sample, the instrumentation (1) kills the app to ensure it is not already running; (2) starts capturing traffic on the Controller's wireless interface; (3) launches the app; (4) waits for approximately 45 seconds; and (5) terminates the traffic capture. This produces $L$ traffic captures (PCAP files), each of which contains all the traffic that occurred during a single launch of the app. We use the term *launch sample* to refer to such a capture.

4. Finally, the instrumentation uninstalls the app to free up space on the smart TV.

**Apple TV Implementation Details.** We use a MacBook Pro Retina (15 inch, Mid 2012), running macOS Catalina, as the Controller in Figure 4.1. We use an Apple Thunderbolt-to-Ethernet adapter to transform one of the MacBook's Thunderbolt ports into an Ethernet port, and connect this interface to the WAN. The MacBook's wireless interface is configured as a wireless access point, to which the Apple TV is connected. The instrumentation procedure is implemented using Apple's testing framework, XCUITest [22], (for controlling the Apple TV programmatically) and tshark (for capturing all network traffic passing through the wireless interface of the MacBook). We note that for Apple TV, the Controller must be a macOS-based system as the XCUITest APIs are only supported on macOS.

**Fire TV Implementation Details.** We use a Raspberry Pi 4 (8GB RAM) as the Controller in Figure 4.1. During data collection, the Raspberry Pi's onboard wireless radio proved unreliable, so we replaced it with a TP-Link Archer T3U Plus Wi-Fi adapter. The instrumentation is realized through a Python implementation [68] of the Android Debug Bridge [52] (for controlling the Fire TV programmatically) and tcpdump (for capturing all network traffic passing through the wireless interface of the Raspberry Pi).

**Roku Implementation Details.** We use a Raspberry Pi 4 (8GB RAM) with a TP-Link Archer T3U Plus Wi-Fi adapter, as the Controller in Figure 4.1. The instrumentation is realized using the Roku External Control Protocol [123] (for controlling the Roku device programmatically) and tcpdump (for capturing all network traffic at the wireless interface of the Raspberry Pi).

### 4.2.3 Dataset Summary

We deploy FingerprinTV and collect $L = 10$ samples of the network traffic generated at launch time by each of the 1000 most widely used apps on Apple TV, Fire TV, and Roku. The resulting dataset comprises 30K packet captures, 10K per platform.

## 4.3 Fingerprinting Techniques

In the context of this chapter, a *fingerprinting technique* is any algorithm that identifies what network traffic (if any) consistently occurs whenever a specific smart TV app is launched. In other words, a fingerprinting technique *extracts* fingerprints from a training network traffic dataset, and the extracted fingerprints can then later be used to identify the corresponding apps in live traffic. This section introduces the three fingerprinting techniques we consider in this chapter and implement support for in FingerprinTV (each fingerprinting technique is responsible for carrying out the steps in the Fingerprint Extraction box of Figure 4.1). We pick these particular techniques because they are lightweight, as they only rely on a few packets per flow, yet applicable even if an app's communication is encrypted using TLS. While the research community has explored these fingerprinting techniques in other contexts [75, 23, 56, 77, 108, 25, 24, 151, 121, 110], our work is the first to apply them to smart TV apps at scale.

### 4.3.1 Domain-Based Fingerprints (DBF)

The first fingerprinting technique we consider identifies an app based on the set of domains the app consistently contacts when it is launched. The intuition is that apps are likely to contact the same set of servers at launch time, for example to fetch featured content. Moreover, past work [98, 159] has shown that smart TV apps contact a large number of distinct domains, suggesting that domain access patterns may be a useful fingerprint for smart TV apps. We refer to this type of fingerprint as a *domain-based fingerprint* (DBF):

**Definition 4.1.** *The* domain-based fingerprint *(DBF) of app $A$, $F_D(A)$, is the set $S$ of domains s.t. for every domain $d$ in $S$, $d$ appears at least once in at least $U$ launch samples of $A$. The* size *of $F_D(A)$ is the number of domains in $S$.*

Throughout this chapter, we refer to the parameter $U$ (in Definitions 4.1, 4.2, and 4.3) as the *usage threshold*. Let $F(A)$ denote the fingerprint of app $A$ (of any type, i.e., DBF, PBF, or TBF). $U$ controls the trade-off between the size of $F(A)$ and its *reliability*, i.e., the likelihood that $F(A)$ will manifest itself in live traffic of $A$.

For example, the size of $F_D(A)$ decreases as $U$ increases, as domains that are only contacted occasionally when $A$ is launched (e.g., if the app has cached some of its resources) will then not become part of $F_D(A)$. Intuitively, and as shown in Section 4.5, the discriminative power of $F_D(A)$ decreases with the size of $F_D(A)$, as a smaller DBF is less likely to be distinct from other DBFs. On the other hand, a large $U$ implies that $F_D(A)$ manifests itself in most live traffic of $A$, making it more likely that $A$ can be consistently detected.

Throughout the remainder of this chapter, and for all three fingerprinting techniques, we choose the most conservative approach: we set $U = L = 10$, where $L$ is the number of launch samples, i.e., we report results for fingerprints that are *always* present.

**Domain Extraction.** As all launch samples for the same app $A$ are collected back-to-back (see Section 4.2.2), any on-device DNS caching will significantly reduce the size of $S$ if domains are only extracted from DNS traffic. FingerprinTV therefore constructs $S$ based on domains found in (1) the answers section of DNS responses, (2) the `Host` header field of HTTP requests (if sent as plaintext), and (3) the TLS SNI extension.

### 4.3.2 Packet-Pair-Based Fingerprints (PBF)

The second fingerprinting technique we consider identifies an app based on packet sizes and packet directions in packet exchanges that consistently occur when the app is launched. The motivation for this technique came from an observation initially made through visual inspection of the packet captures for Roku apps: the Roku communicates with scribe.logs.roku.com

over TLS *every time any app is launched*, and the size of one client-to-server packet in this communication appears to be correlated with the number of digits in the launched app's ID (e.g., the packet's size is 881 bytes if the app ID is a two-digit number, 882 bytes if the app ID is a three-digit number etc.). While the immediate implication of this observation is that Roku is likely tracking the user's app usage, this packet exchange also enables an in-network observer to infer the number of digits in the ID of the app that is launched. If an app exhibits other such consistently occurring packet exchanges, it may be possible to identify the app from observing these packet exchanges happening jointly.

A similar observation was made in our prior work [151], where we introduced the concept of *packet-level signatures* (PLS) for smart home devices. A smart home device exhibits a PLS if the invocation of some specific functionality consistently results in packet exchanges between the device and some endpoint(s), where the packets' sizes (with slight variations) and directions stay consistent across all invocations. In [151], we detail a multi-step methodology for extracting a PLS, where the first step separates packets in TCP connections into *packet pairs*, and later steps reassemble adjacent, consistently occurring packet pairs into longer packet sequences and enforce inter-sequence temporal ordering. Informally, a packet pair is two sequential packets that go in opposite directions, or a single packet paired with a nil value, if the subsequent packet goes in the same direction; see [151] for the formal definition.

In the technique considered here, we essentially terminate the PLS methodology early, namely when it has identified the consistently occurring packet pairs. This set of packet pairs then constitutes the fingerprint, referred to as a *packet-pair-based fingerprint* (PBF):

**Definition 4.2.** *The* packet-pair-based fingerprint *(PBF) of app $A$, $F_P(A)$, is the set $S$ of packet pairs such that for every packet pair $p$ in $S$, $p$ appears $U$ times across $L$ launch samples of $A$. The* size *of $F_P(A)$ is the number of packet pairs in $S$.*

We use the PingPong tool [151] to extract the consistently occurring packet pairs from

our dataset; see Figure 4.1. We treat the $L$ launch samples of each smart TV app $A$ as corresponding to the smart home events that are triggered $L$ times in [151]. To convert our dataset to the format expected by PingPong, we concatenate the $L$ launch samples of $A$ into one. Then, given that trace as input, PingPong produces clusters of packet pairs of similar sizes and matching directions.

We use the default parameters given in [151], with two modifications. First, we only consider packet pairs with identical packet sizes as candidates for inclusion in $F_P(A)$ (i.e., clusters without any variability in the packet pair sizes), while PingPong allows for small variations in packet sizes in the same cluster. We make this conservative choice to align design choices for PBFs with DBFs: domains used in DBFs have no variation. However, less conservative choices can also be accommodated by our methodology, as discussed in Appendix B.1. Second, we do not attempt to temporally order packet pairs to create longer packet sequences, as the traffic profiles of smart TV apps are more complex than those of the simpler smart home devices studied in [151]. In particular, there is often a causal explanation for the temporal order of packet sequences in PLS: the device first receives a control command, e.g., "turn off", in one packet sequence, and then updates the cloud with its new state in another packet sequence [151]. On the other hand, a smart TV app may parallelize resource downloads, which makes the temporal order of packet pairs on different connections less predictable.

### 4.3.3 TLS-Based Fingerprints (TBF)

The third fingerprinting technique we consider attempts to identify an app based on the set of TLS fingerprints the app consistently exhibits when it is launched. We refer to this type of fingerprint as a *TLS-based fingerprint* (TBF). A TBF is conceptually identical to a DBF (see Definition 4.1), but where individual TLS fingerprints assume the role of domains. A TLS fingerprint, originally due to Ristić [121, 110], is the concatenation of a subset of

the information that is contained in the TLS Client Hello message that the client sends to the server in order to initiate a TLS session. Since Ristić's work, various implementations have surfaced [29, 16, 93]. These mainly differ in terms of what components of the Client Hello they include in the TLS fingerprint. We opt for Mercury [93] because it considers the most comprehensive set of Client Hello components (see [19] for the formal definition), which, presumably, increases the TLS fingerprints' discriminative power. For consistency, we formalize TBFs in Definition 4.3.

**Definition 4.3.** *The* TLS-based fingerprint *(TBF) of app $A$, $F_T(A)$, is the maximal set $S$ of TLS fingerprints such that for every TLS fingerprint $s$ in $S$, $s$ appears at least once in at least $U$ launch samples of $A$. The* size *of $F_T(A)$ is the number of TLS fingerprints in $S$.*

## 4.4 Fingerprint Performance Assessment Methodology

In this section, we define a methodology that forms the basis for how FingerprinTV assesses the performance of a fingerprinting technique $F$ (see the Post Processing box in Figure 4.1). For $F$ to enable identification of app $A$, the fingerprint $F(A)$ that $F$ extracts for $A$ must be unique among all other apps' fingerprints. Now, recall from Definitions 4.1, 4.2, and 4.3 in Section 4.3 that at their core, the three types of fingerprints we consider in this chapter are essentially just sets with different types of members, namely domains, packet pairs, and TLS fingerprints. Thus, performance assessment of $F$ is fundamentally a set difference problem.

We tackle this problem using agglomerative clustering [122], as it enables us to compute the (dis)similarity of individual apps' fingerprints (i.e., fingerprint member sets) in a structured, yet extensible, way. More precisely, when performed as described below, agglomerative clustering enables us to (1) identify apps that have *distinct* fingerprints, i.e., when the set of fingerprint members that make up $F(A)$ is different from *all* the sets of fingerprint members

of all other apps; and (2) identify apps that share the same fingerprint, i.e., when the set of fingerprint members that make up $F(A)$ is identical to the set of fingerprint members that make up the fingerprint, $F(B)$, of some other app $B$.

**Clustering Procedure.** The agglomerative clustering is performed as follows. We first form a fingerprint-member-by-app matrix $M$ such that $M[m_i, A_j]$ holds the number of launch samples of app $A_j$ that fingerprint member $m_i$ was observed in. Next, $M$ is pruned by dropping all rows (fingerprint members) that are not present in at least $U$ (the usage threshold, see Definitions 4.1, 4.2, and 4.3) launch samples for at least one app, i.e., row $i$ is removed *iff* $M[m_i, A_j] < U$ for every $j$. $M$ is then converted to a binary matrix by setting $M[m_i, A_j] = 0$ if $M[m_i, A_j] < U$ and $M[m_i, A_j] = 1$ if $M[m_i, A_j] \geq U$, for all combinations of $i$ and $j$. In Sections 4.5, 4.6, 4.7, and 4.8, we use $U = L = 10$ to conservatively report results for fingerprints that are *always* present, i.e., we set $U$ to be equal to the number of launch samples $L$ we perform (see Section 4.2.3).

Figure 4.2 shows an example of the matrix $M$ for the DBFs of 10 popular Fire TV apps. A blue cell indicates that the cell's value is 1, which means that the domain appeared in all $U = 10$ launch samples of the respective app and is therefore part of that app's DBF. A white cell indicates that the cell's value is 0, i.e., the domain appeared in less than 10 launches of the respective app and is therefore *not* part of that app's DBF. The DBF of an app is the binary vector of the corresponding column. For example, the "Facebook" app contacts three domains and we say it has a DBF of size 3; see Definition 4.1. The example also illustrates that DBFs can vary significantly in size: the DBF of "ES File Explorer File Manager" contains a single domain, while the DBF of "NBC" contains 24 domains.

We then compute the agglomerative clustering of the columns (i.e., apps) in $M$ using SciPy [149]. We use cosine distance, defined as $1 - \frac{a \cdot b}{\|a\|_2 \|b\|_2}$, where $a \cdot b$ is the dot product of $a$ and $b$, the fingerprint member vectors for apps $A$ and $B$ respectively, and $\|x\|_2$ is the 2-norm of $x$, to compute the distance between two apps [150]. For example, we can see in

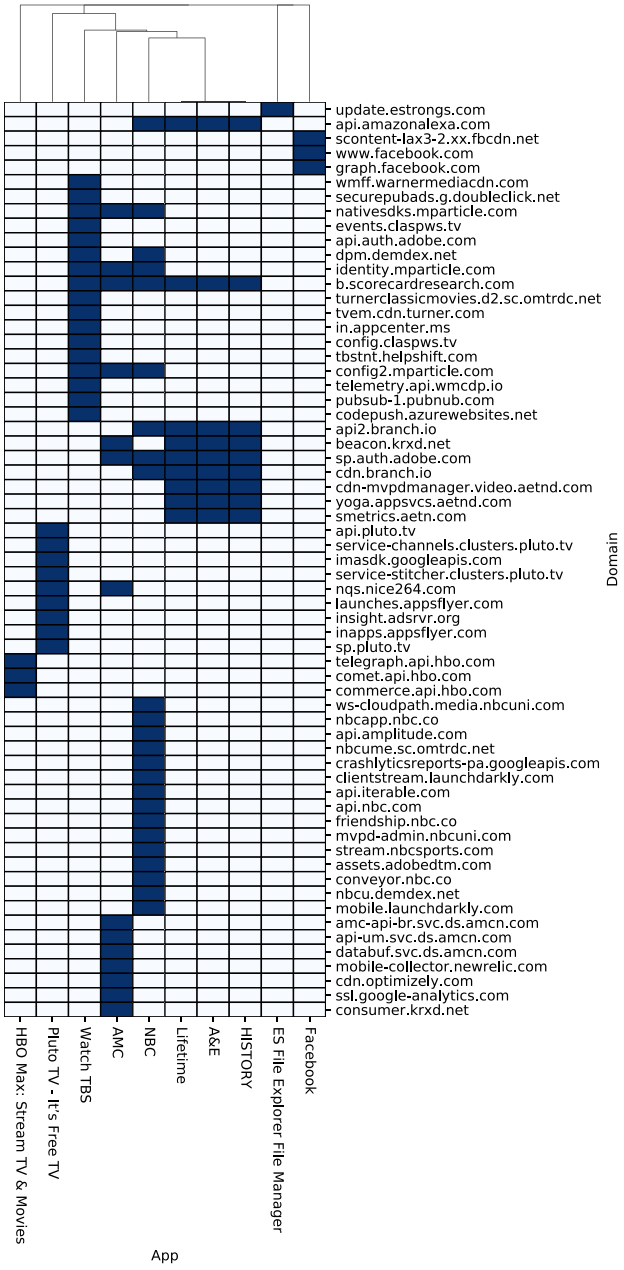



Figure 4.2: Example: DBFs of 10 popular Fire TV apps. Rows correspond to domains, columns correspond to apps, and the dendrogram on top corresponds to the clustering of apps based on the similarity of their DBF. A blue cell indicates that the domain is contacted $U = 10$ times and, thus, part of the respective app's DBF; a white cell indicates otherwise. The DBF of an app is the binary vector of the corresponding column. For example, the "Facebook" app has a DBF of size 3 and it is part of a cluster with size 1. The "HISTORY", "A&E", and "Lifetime" apps contact the same nine domains. This means that they have the exact same DBF of size 9, they have distance 0 from each other, and are together in a cluster of size 3.

Figure 4.2 that the "Lifetime", "A&E", and "HISTORY" apps exhibit the same DBF and, thus, the distance between these apps is 0, and they will be in the same cluster. Since the values in the fingerprint member vectors for apps $A$ and $B$ are binary, a fingerprint member $m_i$ decreases the cosine distance *iff* $m_i$ is part of *both* $F(A)$ *and* $F(B)$. Notice that this implies that the cosine distance will be $> 0$ when one fingerprint is a subset of another. We consider such fingerprints distinct from one another, since fingerprint subsumption can be accounted for by enforcing timing constraints when examining live traffic.

We use the Nearest Point Algorithm for computing the inter-cluster distances when merging clusters [149] (but the Farthest Point and UPGMA Algorithms produce similar results). When extracting clusters from the agglomerative clustering, we use a distance threshold $t = 0$ [148] such that $F(A)$ and $F(B)$ have to be identical to end up in the same cluster. The choice of $t = 0$ also ensures that if $F(A)$ is distinct among all other fingerprints, it will end up in a singleton cluster, and the number of distinct fingerprints is thus simply the number of singleton clusters formed.

We note that we make conservative choices for the parameters in our methodology (e.g., $U$ and $t$). Even under these strict choices, fingerprints are highly prevalent and have significant discriminative power. Furthermore, the methodology is flexible enough to accommodate less conservative choices, e.g., one can use $U < L = 10$ when extracting DBFs to make a trade-off between its size and reliability, as described in Section 4.3.1.

## 4.5 Fingerprint Prevalence, Distinctiveness, and Size

This section reports the *prevalence*, *distinctiveness*, and *sizes* of the fingerprints extracted from the dataset described in Section 4.2.3 using the fingerprinting techniques defined in Section 4.3. We first introduce the three terms and how FingerprinTV computes them,

| Platform | DBF | | PBF | | TBF | | DBF or PBF | | DBF and PBF | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Prevalence (P) | Distinctiveness (D) | P | D | P | D | P | D | P | D |
| Apple TV | 96% | 59% | 68% | 77% | 95% | 3% | 96% | 78% | 68% | 89% |
| Fire TV | 88% | 63% | 95% | 88% | 86% | 7% | 99% | 89% | 85% | 95% |
| Roku | 100% | 46% | 100% | 72% | 100% | 1% | 100% | 76% | 100% | 76% |

Table 4.1: Summary of the three fingerprinting techniques' performance on the top-1000 apps of the three smart TV platforms. Prevalence is the percentage of apps among the top-1000 that exhibit a fingerprint. Distinctiveness is the percentage of apps that exhibit a fingerprint that is distinct from all other apps' fingerprints of the same type, among the total number of apps that exhibit a fingerprint of that type (i.e., each distinctiveness column is computed using the raw numbers behind the prevalence percentage values immediately to its left as the baseline).

and then proceed to report the numbers for each smart TV platform. The results are summarized in the left part of Table 4.1.

**Prevalence.** The *prevalence* of a fingerprint type is the percentage of smart TV apps (from a single platform) that exhibit that type of fingerprint. Recall from Section 4.4 that apps with distinct fingerprints end up in singleton clusters, that apps that have identical fingerprints end up in the same cluster, and that apps that do not exhibit a fingerprint are discarded during clustering. The number $N$ of apps that exhibit a fingerprint is thus the sum of the number of members (i.e., apps) of all clusters in the clustering. The prevalence $P$ is then $P = \frac{N}{1000} \times 100\%$ (as there are 1000 apps per platform).

**Distinctiveness.** The percentage of apps with distinct fingerprints is arguably the most important metric for assessing how well a fingerprinting technique works. We use the term *distinctiveness* to refer to this metric. As explained in Section 4.4, if the fingerprint, $F(A)$, of app $A$ is distinct, $A$ will end up in a singleton cluster, and the number $M$ of apps with distinct fingerprints is thus equal to the number of clusters with size $x = 1$. The distinctiveness $D$ is then $D = \frac{M}{N} \times 100\%$, i.e., the percentage of apps that exhibit a distinct fingerprint of a given type, taken among all apps that exhibit a fingerprint of that type. A large $D$ thus means

that the fingerprinting technique produces fingerprints that are generally able to uniquely identify apps without ambiguity, but is only meaningful if the prevalence is also high.

**Sizes.** Recall from Definitions 4.1, 4.2, and 4.3 that the size of a fingerprint is the number of fingerprint members it contains, e.g., the number of domains in a DBF. We report the fingerprint sizes to give insights as to how many fingerprint members the general fingerprint contains, and to test the intuition that a larger fingerprint is more likely to be distinct (see Section 4.3.1).

### 4.5.1 Domain-Based Fingerprints (DBF)

**Prevalence.** Figures 4.3a, 4.3b, and 4.3c show the number of clusters, grouped by cluster size, for DBFs for the three smart TV platforms (Figure 4.3d is discussed in Section 4.7.2). We find that 96% ($N = 961$) of the top-1000 Apple TV apps exhibit a DBF; 88% ($N = 884$) of the top-1000 Fire TV apps exhibit a DBF; and 100% ($N = 1000$) of the top-1000 Roku apps exhibit a DBF.

**Distinctiveness.** The number of apps with distinct DBFs per platform is the number of clusters with size $x = 1$ in Figures 4.3a, 4.3b, and 4.3c. On all three platforms, the DBF is distinct for about half of the apps that exhibit a DBF: the DBF is distinct for 564 (59%) of the 961 Apple TV apps that exhibit a DBF, 555 (63%) of the 884 Fire TV apps that exhibit a DBF, and 462 (46%) of the 1000 Roku apps that exhibit a DBF.

**Sizes.** Figure 4.4 shows the distribution of DBF sizes per cluster size (the label on top of each point is the app count) for the three smart TV platforms. In summary, we find that the median DBF size is four on all three platforms. We also observe that DBF sizes are generally larger for clusters with fewer members, thus the intuition that larger fingerprints are generally more distinct seems to hold true for DBFs.

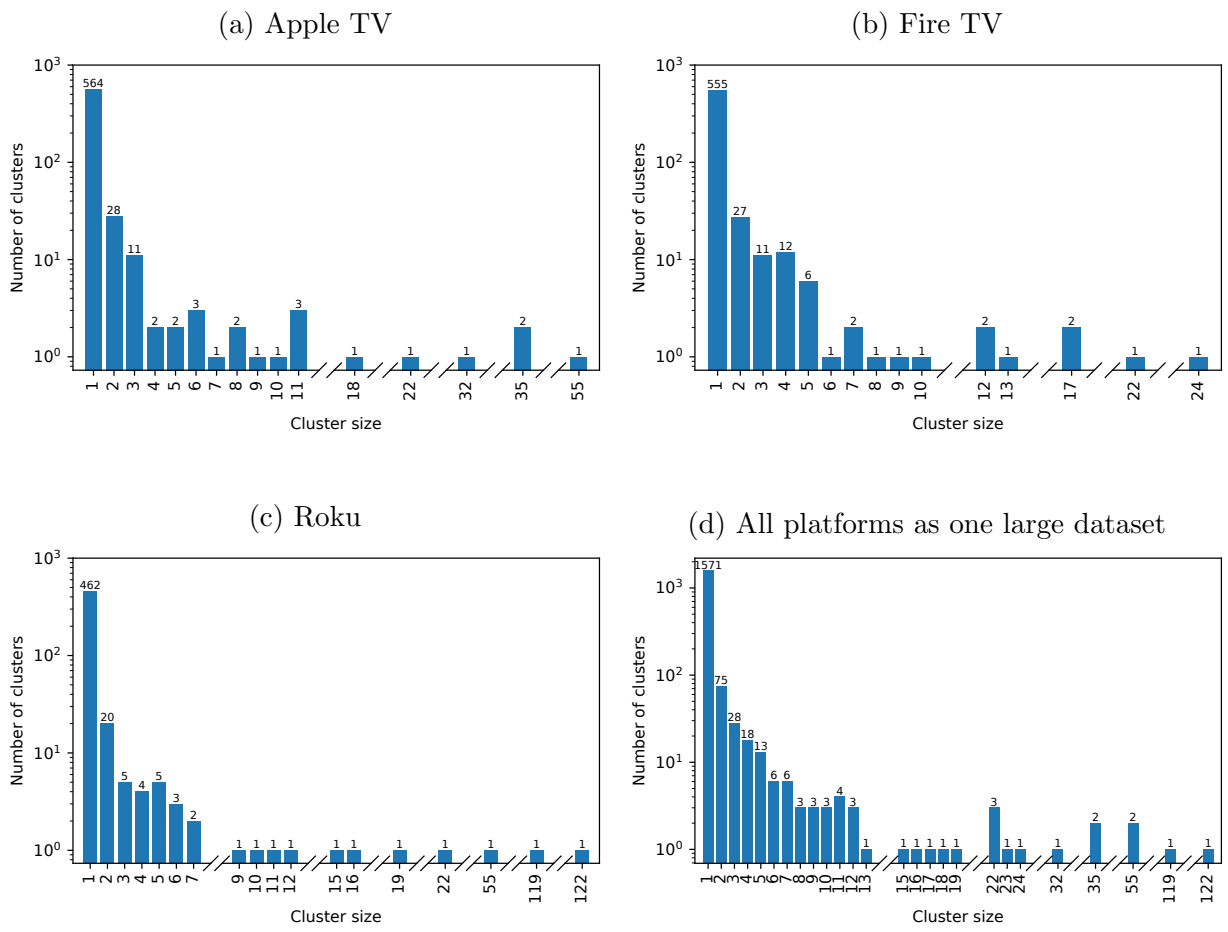



Figure 4.3: Distribution of clusters by cluster size for DBFs. The cluster size is the number of apps in a cluster (i.e., apps with the exact same DBF; see Section 4.4). For instance, the bar at $x = 2$ in Figure 4.3a indicates that there are 28 clusters that each contains 2 apps, for a total of 56 apps.

The three most common DBF sizes are are 4, 5, and 3 for Apple TV; 3, 4, and 2 for Fire TV; and 3, 5, and 4 (tied with 2) for Roku. The median DBF size is 4 across the board. For all three platforms, there appears to be some correlation between a DBF's size and its distinctiveness as the majority of DBFs that are larger than the median DBF are distinct DBFs: 296 of the 463 (64%) Apple TV DBFs, 248 of the 311 (80%) Fire TV DBFs, and 231 of the 420 (55%) Roku DBFs that are larger than the median DBF are distinct.

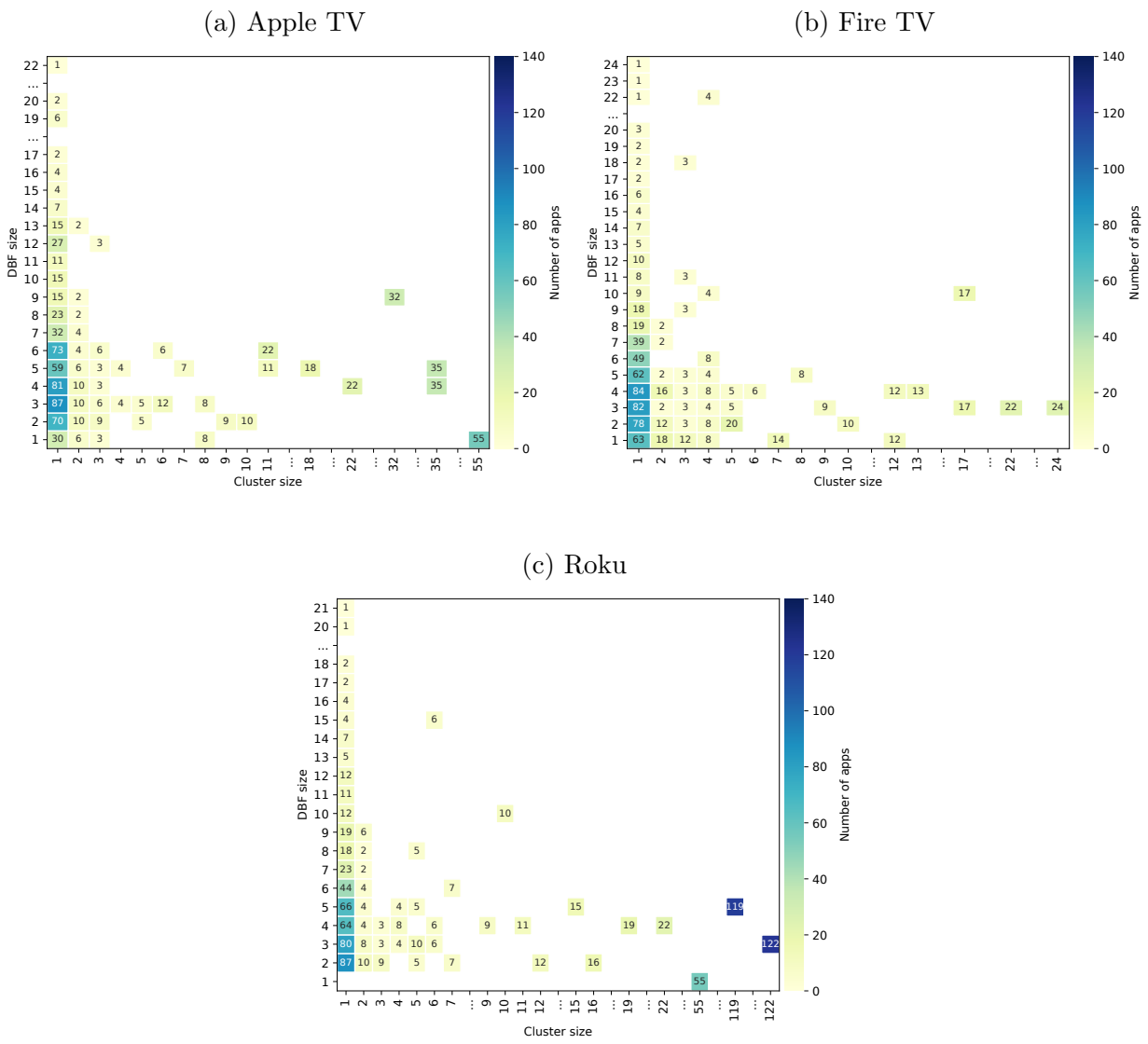



Figure 4.4: Distribution of DBF sizes per cluster size. The DBF size is the number of domains in a DBF. The cluster size is the number of apps in a cluster. App counts are shown for each point. For instance, the point at (2, 2) in Figure 4.4a indicates that there are 10 apps that each has a DBF that contains 2 domains, and these 10 apps reside in clusters that each contains 2 apps.

### 4.5.2 Packet-Pair-Based Fingerprints (PBF)

**Prevalence.** Figures 4.5a, 4.5b, and 4.5c show the number of clusters, grouped by cluster size, for PBFs for the three smart TV platforms (Figure 4.5d is discussed in Section 4.7.2). We find that 68% ($N = 678$) of the top-1000 Apple TV apps exhibit a PBF; 95% ($N = 952$)

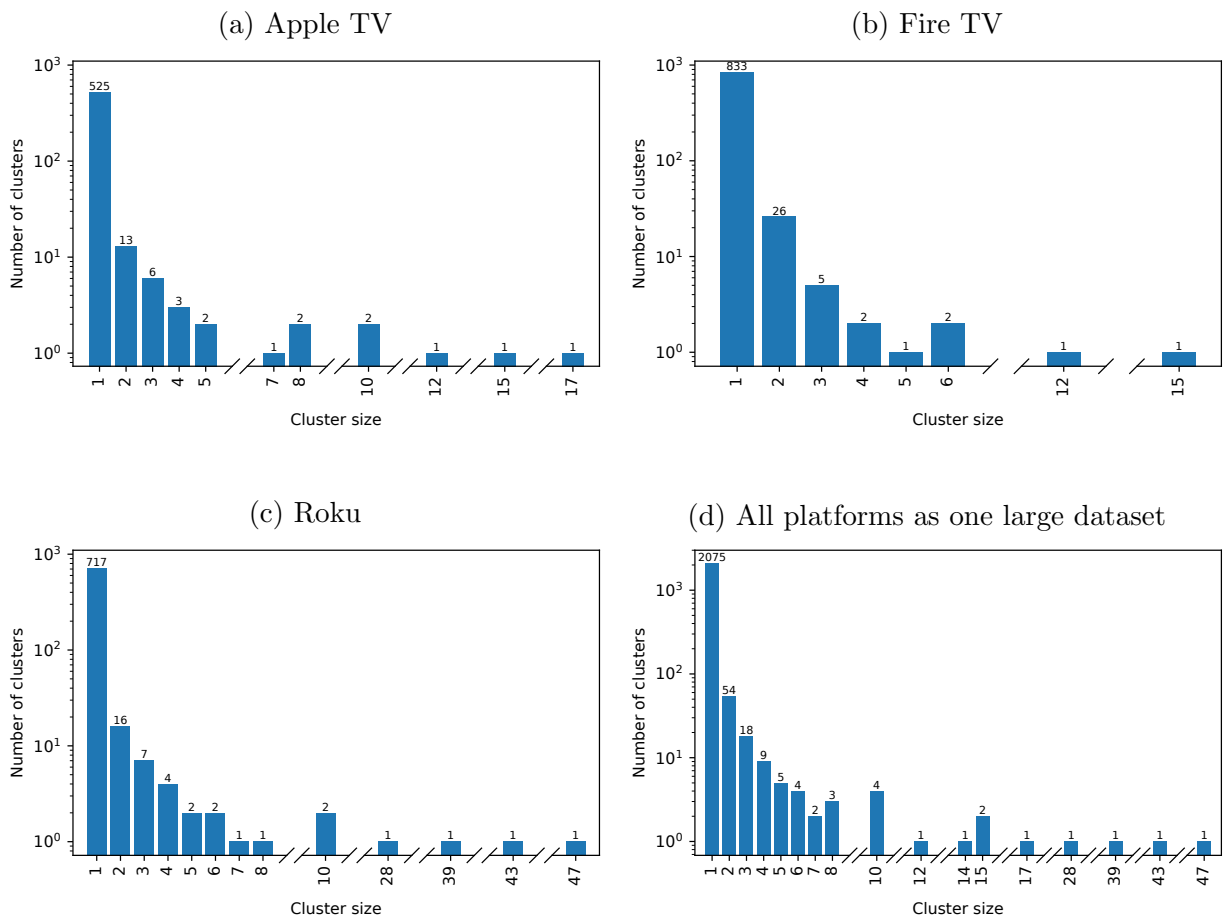



Figure 4.5: Distribution of clusters by cluster size for PBFs. The cluster size is the number of apps in a cluster (i.e., apps with the exact same PBF; see Section 4.4). For instance, the bar at $x = 2$ in Figure 4.5a indicates that there are 13 clusters that each contains 2 apps, for a total of 26 apps.

of the top-1000 Fire TV apps exhibit a PBF; and 100% ($N = 1000$) of the top-1000 Roku apps exhibit a PBF.

**Distinctiveness.** The number of apps with distinct PBFs per platform is the number of clusters with size $x = 1$ in Figures 4.5a, 4.5b, and 4.5c. PBFs have more discriminative power than DBFs: among the apps that exhibit PBFs, 77% of Apple TV apps, 88% of Fire TV apps, and 72% of Roku apps exhibit distinct PBFs.

**Sizes.** Figure 4.6 shows the distribution of PBF sizes per cluster size (the label on top of

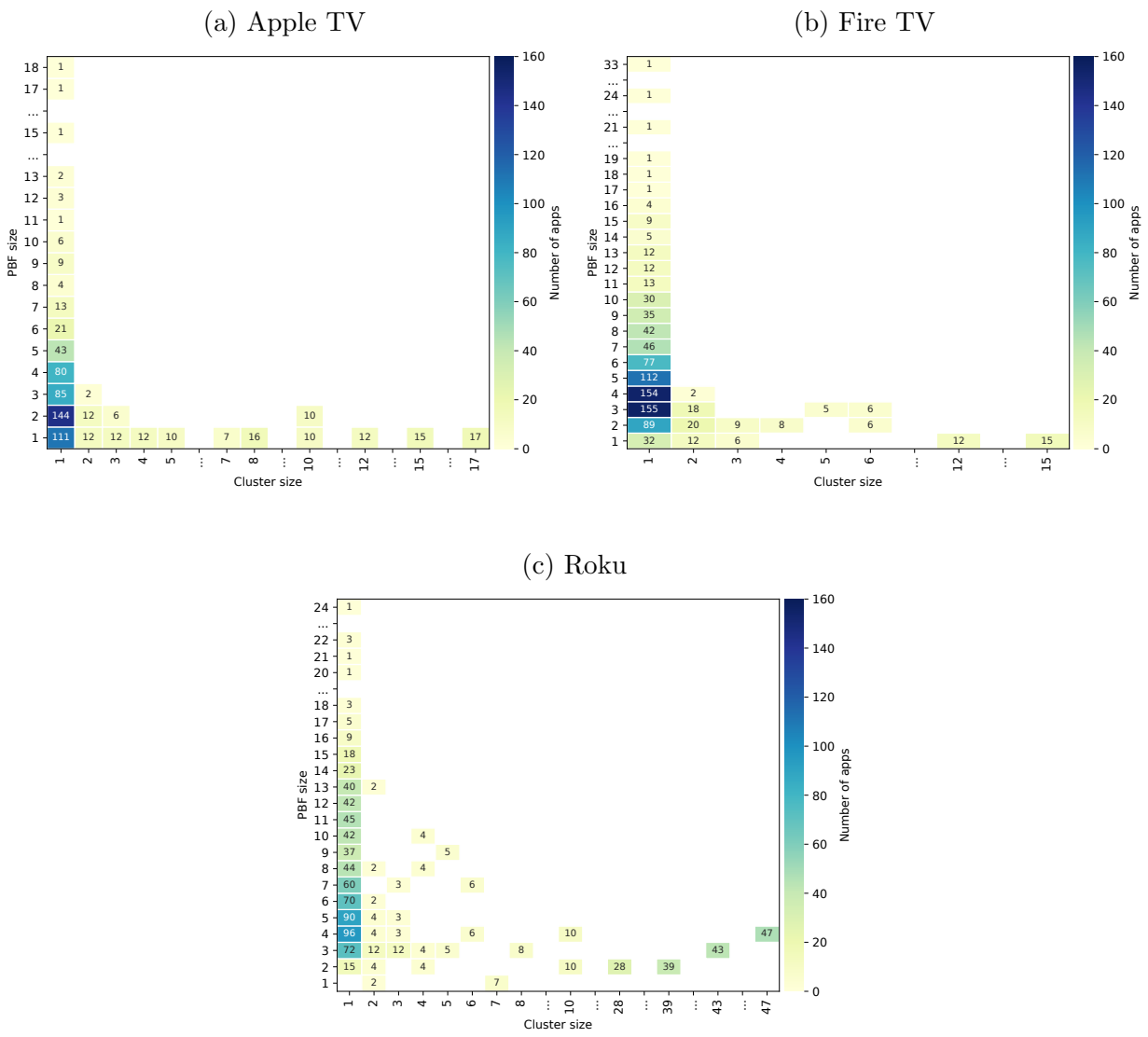



Figure 4.6: Distribution of PBF sizes per cluster size. The PBF size is the number of packet pairs in a PBF. The cluster size is the number of apps in a cluster. App counts are shown for each point. For instance, the point at (2,2) in Figure 4.6a indicates that there are 12 apps that each has a PBF that contains 2 packet pairs and that reside in clusters that each contains 2 apps.

each point is the app count) for the three smart TV platforms. The median PBF sizes are 2, 4, and 5 for Apple TV, Fire TV, and Roku, respectively. Like for DBFs, we also observe a strong correlation between a PBF's size and its distinctiveness: 270 of the 272 (99%) Apple TV PBFs, all of the 403 (100%) Fire TV PBFs, and 444 of the 472 (94%) Roku PBFs that are larger than the median PBF are distinct.

### 4.5.3 TLS-Based Fingerprints (TBF)

We find that TBFs have very little discriminative power and therefore only briefly summarize the results and omit the diagrams. In total, our findings for smart TVs align with those of other work on TLS fingerprinting [19]: (sets of) TLS fingerprints are not sufficiently unique on their own to fingerprint apps.

**Prevalence and Sizes.** TBFs are about as widespread as DBFs: 95%, 86%, and 100% of the top-1000 Apple TV, Fire TV, and Roku apps, respectively, exhibit a PBF. The median TBF size is 2 across all platforms. While prevalence is large for TBFs, this also bears the positive message that most smart TV apps encrypt (part of) their communication. The TBF prevalence observed for Roku is in line with what is reported in the appendix of [98], but we observe slightly fewer TBFs for Fire TV, possibly because we only consider "on-launch" traffic, whereas [98] also inject user actions post launch.

**Distinctiveness.** While TBFs are prevelant on all three platforms, they have little discriminative power: only 3%, 7%, and 1% of the Apple TV, Fire TV, and Roku apps that exhibit TBFs, exhibit distinct TBFs. Furthermore, a few TBFs are shared by a large number of apps; e.g., for both Apple TV and Roku, the clustering outputs two clusters with over 300 apps. As TBFs evidently have little discriminative power, we omit them from the discussion going forward.

### 4.5.4 Distinctiveness and Dataset Size

In Table 4.1, and throughout this chapter, we report the distinctiveness of DBFs, PBFs, and TBFs for the top-1000 Apple TV, Fire TV, and Roku apps. To shed further light on the potential for fingerprint collisions, in Figure 4.7, we split each of the three datasets into increasingly larger subsets and show how the distinctiveness evolves for DBFs and PBFs as

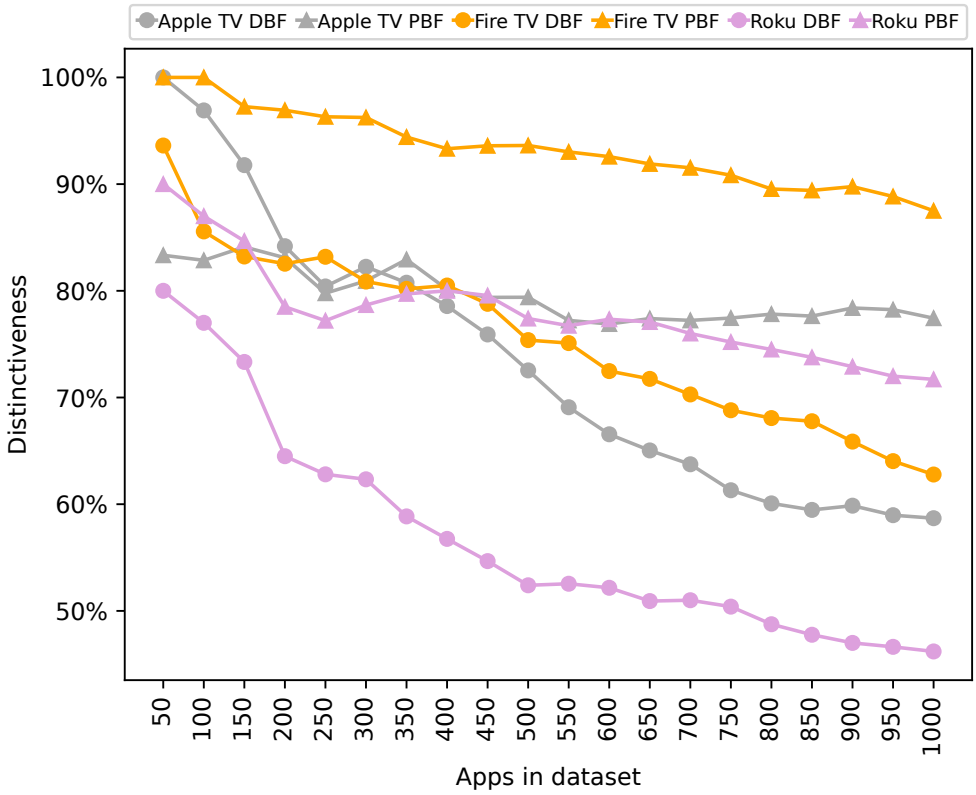



Figure 4.7: Distinctiveness of DBFs and PBFs as a function of the number of apps in the dataset. Apps are added to the dataset based on the number of reviews submitted for each app. That is, $x = 50$ is a dataset comprised of the 50 most reviewed apps, $x = 100$ is a dataset comprised of the 100 most reviewed apps, and so forth.

the number of apps considered increases. Apps are included in the subsets based on the number of user reviews submitted for each app. That is, in Figure 4.7, $x = 50$ is a dataset comprised of the 50 most reviewed apps, $x = 100$ is a dataset comprised of the 100 most reviewed apps, etc.

Figure 4.7 shows that the distinctiveness decreases slowly as the number of apps considered increases beyond 500 apps, especially for PBFs. In particular, the distinctiveness of PBFs of Apple TV apps appear to plateau. Moreover, the distinctiveness of PBFs generally declines slower than the distinctiveness of DBFs. The distinctiveness of PBFs is also generally greater than the distinctiveness of DBFs, which is in line with the observation made for the full dataset.

### 4.5.5 Takeaways

DBFs, PBFs, and TBFs are all prevalent among Apple TV, Fire TV, and Roku apps. However, only DBFs and PBFs have enough discriminative power to reliably identify apps, and we therefore omit TBFs from the discussion going forward. We also observe a correlation between a fingerprint's size and its discriminative power. Overall, PBFs seem to have more discriminative power than DBFs, but they are also more likely to change over time: even small changes to an app's communication protocol(s) directly affect packet sizes [151].

## 4.6 Identical Fingerprints

This section examines to what extent apps with identical fingerprints stem from the same developer. If a developer releases multiple apps for the same platform, they may opt to use some of the same backend servers to deliver content, which could make these apps exhibit identical fingerprints. We investigate this by examining the number of distinct developers present in clusters with size $x > 1$, i.e., clusters of apps that share the same fingerprint. We consider developers to be identical if they are part of the same parent organization. For example, the Fire TV app developers "Scripps Networks, LLC", "Discovery Communications" and "OWN, LLC" are identical, since Discovery, Inc. owns a majority stake in these companies.

### 4.6.1 Domain-Based Fingerprints

**Summary.** Figure 4.8 shows the number of clusters of size $x$ (for DBFs) that contain apps from $Q$ distinct developers for the three smart TV platforms. In summary, we find that a sizeable fraction of identical DBFs are indeed attributable to apps from the same developer.

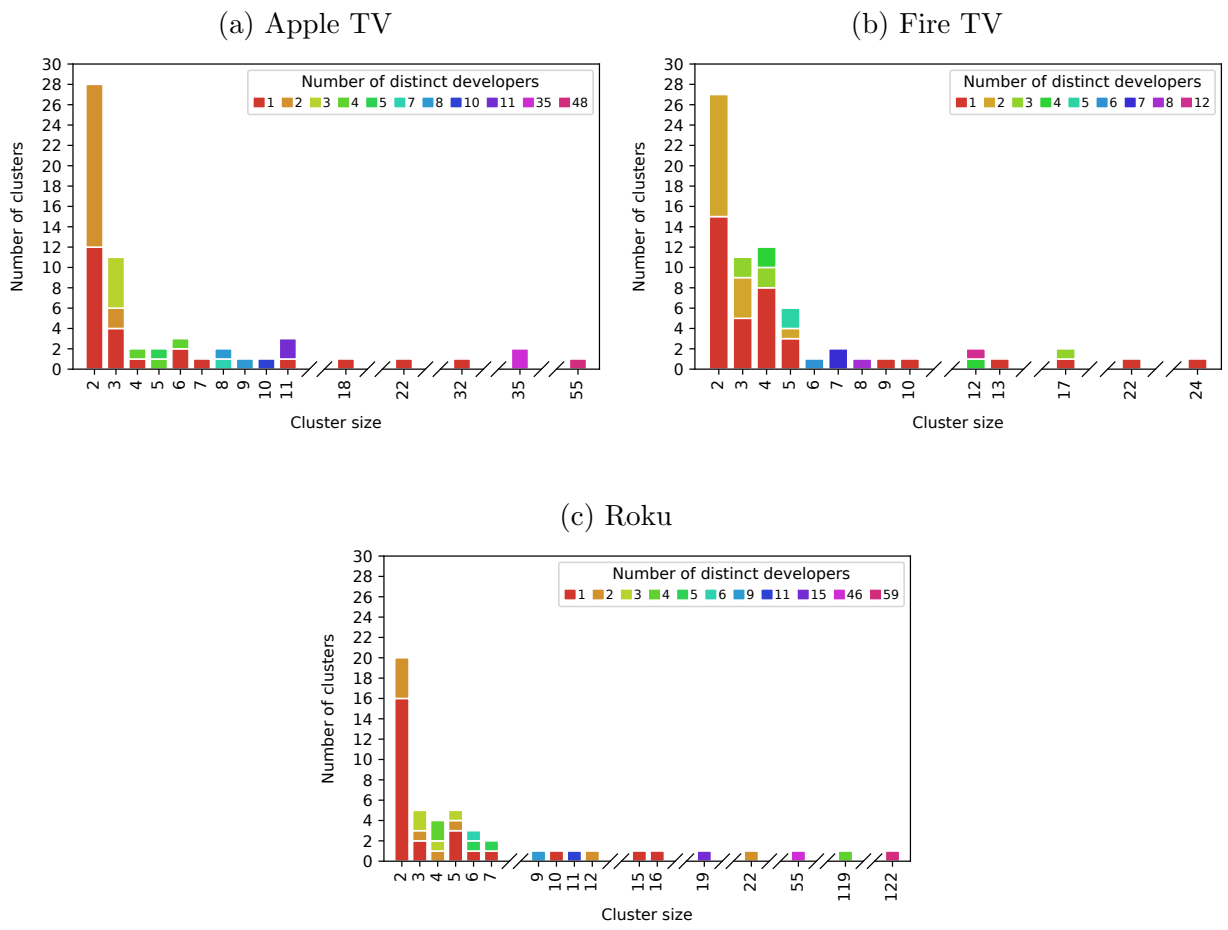



Figure 4.8: Distribution of the number of developers responsible for apps in clusters of size $x > 1$ for DBFs. For instance, the bar at $x = 2$ in Figure 4.8a indicates that 12 of the 28 clusters of size $x = 2$ (see Figure 4.3a) only contain apps from the same developer, while the remaining 16 contain apps from 2 developers.

We also find many examples of apps that appear to be generated using "no code" toolkits provided by consulting firms, and apps that are generated using the same toolkit tend to have identical DBFs.

**Apple TV Details.** For Apple TV, 142 of the 397 apps (37%) that share their DBF with other app(s) only share it with other apps from the same developer. We note that while the apps in the two clusters of size $x = 35$ in Figure 4.8a are officially published by completely different developers ($Q = 35$ for these clusters), they all appear to be due to the same consulting firm, Subsplash, Inc., (who offers a "no code" toolkit for generating

and publishing apps) since their DBFs contain subdomains of subsplash.com and (most of) their bundle IDs begin with "com.subsplash". Further, they are all religious apps, which is Subsplash's specialty. Similarly, all apps in one of the three clusters of size $x = 11$, in one of the two clusters of size $x = 4$, and in one of the 28 clusters of size $x = 2$ appear to be due to MAZ Systems Inc.; all apps in one of the two clusters of size $x = 8$ and in one of the 28 clusters of size $x = 2$ appear to be due to UscreenTV, LLC; and RadioKing and Streann Media Inc each appear to be behind all apps in one of the 28 clusters of size $x = 2$. If we treat these cases as the same developer, we instead get that 243 of the 397 apps (61%) that share their DBF with other app(s) only share it with other apps from the same developer.

**Fire TV Details.** For Fire TV, 187 of the 329 apps (57%) that share their DBF with other app(s) only share it with other apps from the same developer. We observe that when a DBF is shared among many apps, those apps generally stem from the same developer: notice that $Q = 1$ for many of the clusters of size $x > 8$ in Figure 4.8b. We note that we suspect that the $Q = 3$ distinct developers responsible for the apps in one of the two clusters of size $x = 17$ are the same (or closely related) entities as these apps (1) have custom made, yet similar, privacy policies; (2) use the same pattern for their contact emails; and (3) are all ambience apps that use the same app naming scheme.

As for Apple TV, we again find many clusters of apps that are officially published by multiple developers (and shown as such in Figure 4.8b), but where the domains in their DBFs and/or their package names suggest that they are from the same consulting firms. If we treat these cases as the same developer, 227 of the 329 apps (69%) that share their DBF with other app(s) only share it with other apps from the same developer.

**Roku Details.** For Roku, 107 of the 538 apps (20%) that share their DBF with other app(s) only share it with other apps from the same developer. We again find many clusters of apps that are officially published by multiple developers (and shown as such in Figure 4.8c), but where the domains in the DBFs suggest that they are due to the same consulting firms. If

we treat these cases as the same developer, 143 of the 538 apps (27%) that share their DBF with other app(s) only share it with other apps from the same developer.

### 4.6.2 Packet-Pair-Based Fingerprints

We do not find evidence that identical PBFs primarily stem from apps from the same developer. For Apple TV, 10 of the 153 apps (16%) that share their PBF with other app(s) only share it with other apps from the same developer. This also only applies to 7 of 119 (6%) Fire TV apps, and 42 of 283 (15%) Roku apps.

### 4.6.3 Takeaways

When multiple apps exhibit identical fingerprints, an in-network observer can only make ambiguous inferences about app usage. However, apps with identical DBFs primarily stem from the same developer. As some developers focus their efforts on building apps with a certain theme (e.g., religious apps), it may thus still be possible to infer what *type of content* the user is consuming. The same does not hold true for PBFs, but identical PBFs are less widespread (see Section 4.5.2).

## 4.7 Fingerprints Across Platforms

This section compares how fingerprints of apps that are present on all three platforms differ across platforms and extends the evaluation in Section 4.5 of the extracted fingerprints' distinctiveness by considering the datasets of the three platforms as a single, large dataset.

### 4.7.1 Multi-Platform Apps

We first compare the DBFs and PBFs of apps in our dataset that are present on all three platforms, referred to as *multi-platform apps*. Drawing inspiration from [159], we identify 80 multi-platform apps through fuzzy matching on the app and developer names, as the same app can have slightly different names on each platform, e.g., "YouTube" on Fire TV and Roku, but "YouTube: Watch, Listen, Stream" on Apple TV.

**DBFs.** Figure 4.9 shows, using one color-coded bar per platform per app, the sizes of the DBFs of 60 of the 80 multi-platform apps; due to space constraints, we only show the 60 multi-platform apps with the largest DBFs in descending order. The textured part of each bar indicates domains in the DBF that are unique to the corresponding platform, i.e., domains that are not present in the DBFs of the same app on the other two platforms. Most multi-platform apps have at least one platform-specific domain in their DBFs for all three versions of the app (Apple TV, Fire TV, and Roku). In fact, only 19 of the 80 multi-platform apps (24%) have version(s) with no unique domains in their DBFs, and in 18 cases this only

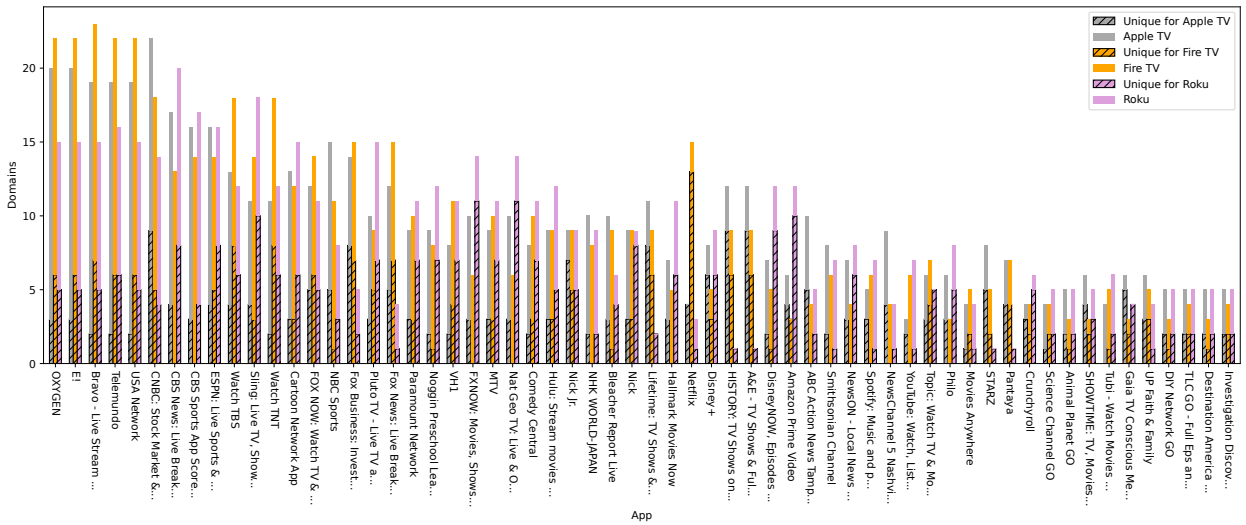



Figure 4.9: DBF sizes for the 60 multi-platform apps that exhibit the largest DBFs in descending order. The size of the DBF for each version (Apple TV, Fire TV, and Roku) of an app is indicated using color-coded bars. The textured part of each bar indicates domains in the DBF that are unique to the corresponding platform.

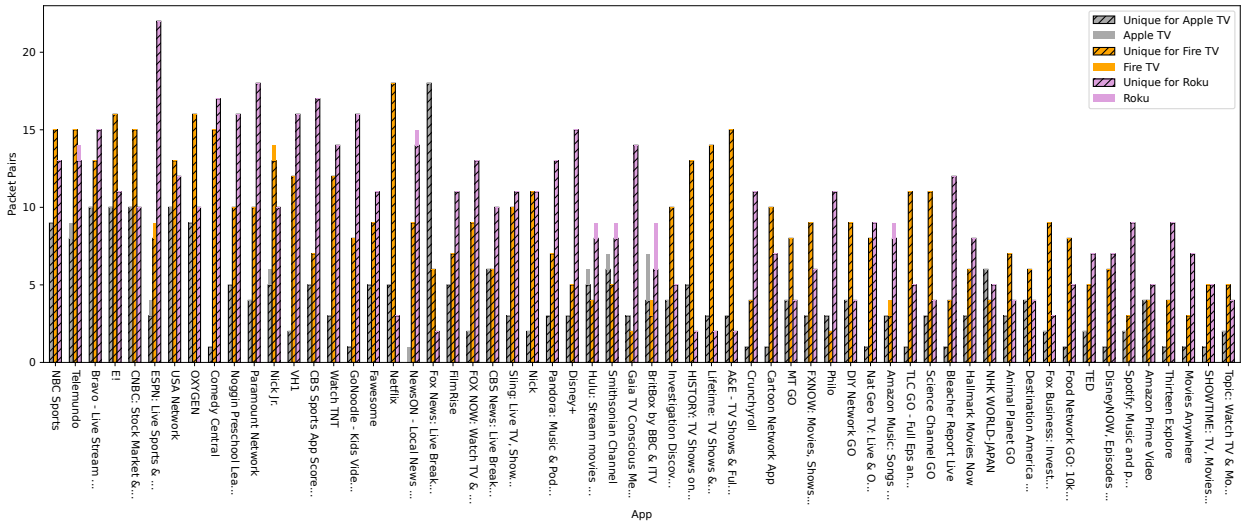



Figure 4.10: PBF sizes for the 60 multi-platform apps that exhibit the largest PBFs in descending order. The size of the PBF for each version (Apple TV, Fire TV, and Roku) of an app is indicated using color-coded bars. The textured part of each bar indicates packet pairs in the PBF that are unique to the corresponding platform.

applies to a single version of the app. Interestingly, it is primarily the Fire TV version that lacks platform-specific domains in its DBF. The Roku version *always* has at least one unique domain in its DBF, as *all* Roku apps communicate with scribe.logs.roku.com when they are launched (see Section 4.3.2).

**PBFs.** In similar style as Figure 4.9, Figure 4.10 shows the sizes of (i.e., the number of packet pairs in) the PBFs of 60 of the 80 multi-platform apps. All apps, except the Apple TV version of two apps, have platform-specific packet pairs in their PBFs for all versions of the app. In fact, the vast majority—and in many cases *all*—packet pairs in the PBFs are platform-specific.

### 4.7.2 Distinctiveness of Fingerprints Across Platforms

To further assess the distinctiveness of fingerprints across platforms, we perform a similar analysis as in Section 4.5, but where we consider fingerprints of apps on all three platforms

as a single, large dataset. We refer to apps with the same fingerprint as having a "collision".

**DBFs.** Figure 4.3d shows the number of clusters, grouped by cluster size, when the DBFs of apps of all three platforms are considered as a single, large dataset. The impact this merging of datasets has on the DBFs' distinctiveness is understood by comparing Figure 4.3d and the sum of Figures 4.3a, 4.3b, and 4.3c. If the DBFs are less distinct in the merged dataset, the bar corresponding to the number of clusters with size $x = 1$ in Figure 4.3d will be smaller than the sum of the corresponding bars at $x = 1$ in Figures 4.3a, 4.3b, 4.3c. Similarly, increases in DBF collisions would be reflected as larger values in Figure 4.3d compared to the sum of the values at the respective $x$ in Figures 4.3a, 4.3b, and 4.3c, for $x > 1$.

We observe a very slight decrease in the number of distinct DBFs: there are $564+555+462 = 1581$ distinct DBFs when the platforms are considered individually (Figures 4.3a, 4.3b, and 4.3c), and 1571 distinct DBFs when the three platforms are considered together (Figure 4.3d). DBF collisions also only change slightly: the number of clusters with size $x > 1$ are mostly the same across Figure 4.3d and the sum of Figures 4.3a, 4.3b, and 4.3c. We note that *all* additional DBF collisions that arise in the merged dataset are among Apple TV and Fire TV apps. This is because *all* Roku apps' DBFs include one Roku domain (see Section 4.3.2). Furthermore, the collisions are attributable to DBFs that are smaller than or equal to the median DBF size, which further confirms the intuition that larger DBFs have more discriminative power (see Sections 4.3.1 and 4.5.1).

**PBFs.** Figure 4.5d shows the number of clusters, grouped by cluster size, when the PBFs of apps of all three platforms are considered as a single, large dataset. We observe *no change* in the number of distinct PBFs when the three individual datasets are considered as one: the sum of distinct fingerprints across Figures 4.5a, 4.5b, and 4.5c is 2075, which is also the number of distinct fingerprints in Figure 4.5d. There is also hardly any change to PBF collisions as the number of clusters with size $x > 1$ are almost identical across Figure 4.5d and the sum of Figures 4.5a, 4.5b, and 4.5c. In fact, the only change is that a cluster of two

Fire TV apps is merged with a cluster that contains 12 Apple TV apps. The PBFs of these apps are comprised of a single MTU-sized client-to-server packet, likely because the client is sending data in bulk. This results in single-packet pairs (see Section 4.3.2 and [151]), which have little discriminative power.

### 4.7.3 Takeaways

The DBFs and PBFs of apps that are made available on all three platforms are often platform-specific. Thus, it is generally possible to not only fingerprint smart TV apps themselves, but also (as a side-effect) to identify which smart TV platform they are being used on. Additionally, the distinctiveness of DBFs and PBFs appears to hold steady when the three datasets are considered as one large dataset. This provides some indication that the fingerprints are likely to retain their discriminative power in an open world setting.

## 4.8 Combining Fingerprints

This section considers the benefits of combining DBFs and PBFs. To establish to what extent smart TV apps can be fingerprinted in general, using any technique, Section 4.8.1 examines how many apps exhibit a DBF or a PBF (or both). Since larger fingerprints have more discriminative power (as confirmed in Section 4.5.5), Section 4.8.2 examines how many apps exhibit *both* a DBF *and* a PBF.

### 4.8.1 DBF or PBF

The "DBF or PBF" column of Table 4.1 lists the prevalence and distinctiveness of fingerprints that are comprised of a DBF or a PBF (at least one, or both). The reported distinctiveness

is based on the distinctiveness of the individual components considered separately. That is, a fingerprint is *not* considered distinct if neither the DBF nor the PBF are distinct on their own, even if the combination of them is exclusive to one app. This is to remain conservative in our reporting, as confusion could arise if another app that exhibits the same DBF is launched at the same time as a third app that exhibits the same PBF, e.g., on different smart TVs in the same household.

The results show that nearly all apps on all three platforms exhibit either a DBF or a PBF (or both): the prevalence is $\geq$ 96% across the board. More importantly, most of these fingerprints are distinct: the fingerprint is distinct for 78% (750) of the Apple TV apps, 89% (884) of the Fire TV apps, and 76% (760) of the Roku apps that exhibit a DBF or a PBF (or both). It is worth noting that even for Apple TV, where DBFs contribute significantly to the prevalence (PBFs on their own only achieve 68% prevalence, whereas this joint fingerprint achieves 96%), the distinctiveness is in line with that of PBFs on their own (recall from Section 4.5.5 that PBFs have more discriminative power than DBFs). Thus, the extra coverage that can be achieved by also considering DBFs appear to primarily stem from distinct DBFs.

### 4.8.2 DBF and PBF

The "DBF and PBF" column of Table 4.1 lists the prevalence and distinctiveness of fingerprints that are comprised of *both* a DBF *and* a PBF. The reported distinctiveness is computed in the same way as for "DBF or PBF", described above.

The results show that DBFs and PBFs generally co-occur: notice that the prevalence for this joint fingerprint almost equals its upper bound, i.e., the minimum prevalence of DBFs and PBFs individually. This implies that almost all Apple TV apps that exhibit a PBF must also exhibit a DBF, and vice versa for Fire TV (*all* Roku apps exhibit both DBFs and

PBFs). The fingerprints also have high discriminative power: the fingerprint is distinct for 89% (599) of the Apple TV apps, 95% (802) of the Fire TV apps, and 76% (760) of the Roku apps that exhibit *both* a DBF *and* a PBF.

### 4.8.3 Takeaways

Nearly *all* apps across all three smart TV platforms can be fingerprinted if one uses either a DBF or a PBF (or both) on a per-app basis. Moreover, these joint fingerprints have high discriminative power, but fingerprints for Roku apps fall slightly short of those of Apple TV apps and Fire TV apps in this respect.

## 4.9 Discussion

This section briefly discusses how to mitigate the inferences that can be made using DBFs and PBFs, as well as the limitations of our work, and future directions.

**Possible Defenses.** As DBFs rely on access to domain names in cleartext, DNS-over-HTTPS (DoH) [61] and DNS-over-TLS (DoT) [62] are perhaps the most obvious potential defenses. However, as evident from the results reported in Section 4.5.3, most smart TV apps consistently communicate over TLS at every launch. If the smart TV app uses the TLS Server Name Indication (SNI) [37] extension for these sessions, the domain can be recovered from the TLS session alone, which negates the defense provided by DoH/DoT. To test this, we ran a modified version of FingerprinTV that would disregard all DNS data, and confirmed that the results were almost identical to those reported in Section 4.5.1. In essence, to be effective against DBFs, DoH/DoT must be paired with Encrypted SNI (ESNI), but this requires that the apps' backend servers add support for ESNI, which likely lies years ahead.

Network-level blockers, such as Pi-hole, that block traffic to select domains can prevent identification of an app via its DBF, if the DBF includes one or more domains from the blocklist, as the adversary will only observe a partial DBF match. The adversary can easily counter such protection by removing domains that are present in popular blocklists from the DBF they extract for the app during training, and then (also) examine live traffic for the manifestation of this reduced DBF. Since smaller DBFs have less discriminative power (see Sections 4.3.1 and 4.5.5), network-level blockers may add uncertainty to the adversary's inferences, but may also introduce app breakage.

For complete protection against DBFs, the smart TV's traffic can be tunneled through a VPN. Assuming the adversary is somewhere on the path from the smart TV to the VPN server, this defense nullifies the effectiveness of DBFs entirely, as the tunnel encrypts the three protocol fields that domains can be extracted from (see Section 4.3.1). The downsides are that tunneling adds additional network overhead, and that the VPN server may become a bottleneck (e.g., if it enforces per-user rate limits that the smart TV can saturate).

For defenses against PBFs, we refer to our prior work [151]. There, we argue that packet padding is effective against fingerprints that rely on packet sizes, at the cost of some overhead, which also applies to the special case of PBFs considered in this chapter. In summary, a VPN that pads packets provides an effective defense against inference attacks based on DBFs (by hiding the destination) and/or PBFs (by obfuscating packet sizes).

**Limitations.** While our assessment of the feasibility of fingerprinting smart TV apps is made against the largest smart TV dataset to date, it is not without limitations. In particular, some smart TV apps require that the user logs in, e.g., subscription-based apps such as Netflix. As FingerprinTV does not attempt to create accounts and log in, the fingerprints it extracts for these apps may be different from what can be observed in the wild as the on-launch traffic may differ depending on the login state. This is a common limitation of automated measurement studies, e.g., [98, 159, 69, 74], as automating account creation

and login is a difficult problem. Some studies approach this limitation by manually logging in to a subset of the tested apps and/or by relying on third-party authentication such as Google [117, 118, 69], while others record the key presses involved in the login procedure s.t. future versions of the same app can be tested automatically [116]. An adversary intent on extracting (more) precise fingerprints for select smart TV apps can adopt a similar strategy using FingerprinTV.

**Future Directions.** In this chapter, we purposely opted for the most conservative design choices to extract reliable fingerprints. For example, we require that a domain must be present at least once in *all* launch samples of some app $A$ for it to be included in $A$'s DBF, and, unlike in [151], we do *not* allow for small variations in packet sizes when considering packet pairs for inclusion in a PBF. It may be possible to increase fingerprint prevalence and sizes by relaxing these strict requirements, possibly at the cost of fingerprint reliability and/or distinctiveness. Future work can study these trade-offs by adjusting existing parameters of our FingerprinTV framework (both in pre-processing and in the clustering algorithm).

Since network traffic fingerprints may change over time, it would be interesting to perform a longitudinal analysis to see how the extracted fingerprints evolve, and how often fingerprints need to be extracted. FingerprinTV facilitates such studies through its automated data collection, fingerprint extraction, and fingerprint assessment features: the clustering that is inherently unsupervised in our methodology will identify and extract any updated fingerprints.

## 4.10 Summary

We presented FingerprinTV, a methodology and implementation we devised for automatically extracting and evaluating network fingerprints of smart TV apps. By deploying

FingerprinTV to the top-1000 Apple TV, Fire TV, and Roku apps, we showed that smart TV app fingerprinting is highly feasible and effective on all three platforms, as most apps exhibit a fingerprint, and most fingerprints are distinct. The FingerprinTV dataset has been made publicly available, and we plan to release the FingerprinTV code as well [152].

# Chapter 5

# Seqnature: Packet Sequences as Network Fingerprints

## 5.1 Overview

Motivated by the observation made in Section 4.8 that joint use of multiple fingerprinting techniques improves fingerprint distinctiveness in the context of smart TV apps, this chapter proposes a general fingerprinting framework, SEQNATURE, that can identify fingerprints that are based on any combination of several features extracted from (encrypted) network traffic, such as server identities and the sizes, directions, and/or order of packets. SEQNATURE is platform-agnostic, i.e., SEQNATURE can be used to identify fingerprints of arbitrary events on any software/hardware platform, including, but not limited to, smart TVs and IoT.

A fingerprint for event $e$, as extracted by SEQNATURE, is the set of packet sequences that consistently occur when $e$ is triggered. A *packet sequence*, formally defined in Definition 5.1, is an $n$-gram of packets in a TCP stream. Alternatively, a packet sequence can be thought of as analogous to a substring, but where packets assume the role of characters. SEQNATURE's

configuration parameters allow the user to define when packet sequences are considered identical, and to constrain the lengths of packet sequences considered for inclusion in the fingerprint. This makes it simple to use SEQNATURE to implement (and combine) existing fingerprinting techniques (e.g., DBFs and PBFs, see Sections 4.3.1 and 4.3.2), and to experiment with new fingerprinting techniques.

**Definition 5.1.** *A* packet sequence *of length $n$ is $n$ consecutive packets within a TCP stream, after retransmissions and zero-payload packets have been dropped and the packets in the TCP stream have been ordered by timestamp.*

**Fingerprinting Framework.** An overview of SEQNATURE is provided in Figure 5.1. To fingerprint event $e$ on computing device $d$, SEQNATURE is provided with $T$ samples of the network traffic that occurred on $d$ immediately after $e$ was triggered. These raw packet captures are referred to as *traffic samples*. The traffic samples are first reduced to only the TCP packets to/from $d$ that carry payload. The filtered traffic samples are then converted to tabulated traffic samples by transforming each remaining packet into a vector of features, where the features are the server the packet was exchanged with, the packet's size, the packet's direction etc.

The core of SEQNATURE is an iterative procedure, referred to as *fingerprint refinement*, which relies on clustering to identify packet sequences that consistently co-occur with $e$. In order to identify consistently occurring packet sequences of any possible length $n$, fingerprint refinement iteratively considers increasingly shorter packet sequences. Fingerprint refinement can be customized using a range of parameters. This allows for flexibility in terms of when two packet sequences observed in different traffic samples are considered the same. For example, the parameters can be used to specify the allowed variance in packet sizes.

When fingerprint refinement concludes, it outputs a set of clusters of packet sequences which constitutes the final fingerprint. This fingerprint is also referred to as the *seqnature* of $e$,

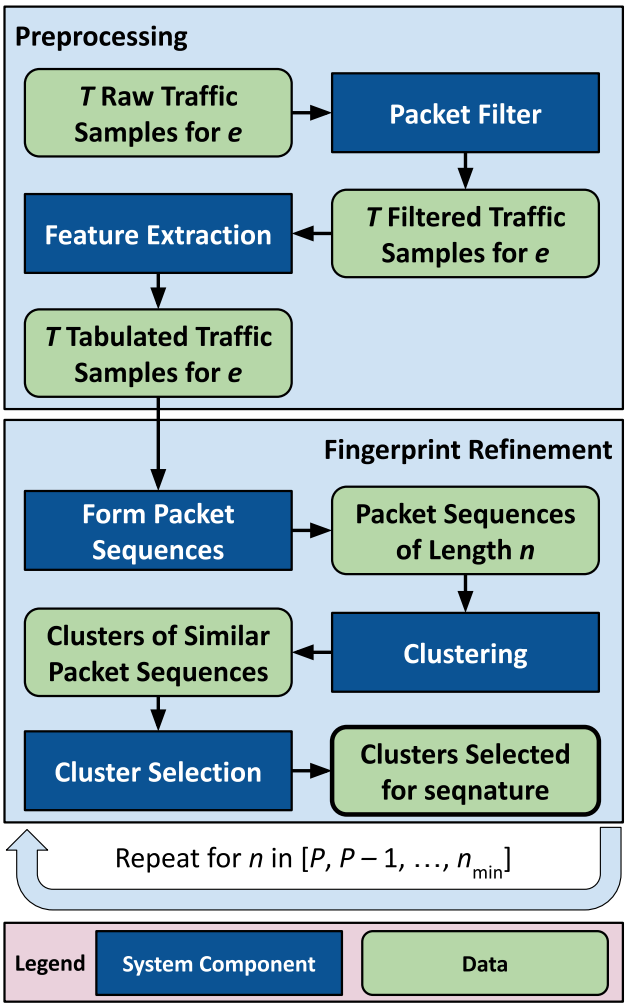



Figure 5.1: Overview of SEQNATURE. To fingerprint event $e$, SEQNATURE is provided with $T$ samples of the network traffic that occurred immediately after $e$ was triggered. SEQNATURE has two phases: a preprocessing phase (Section 5.2.1) that extracts TCP stream information from the raw traffic samples, and an iterative fingerprint refinement phase (Section 5.2.2) that identifies packet sequences that co-occur with $e$.

a play on the words "sequence" and "signature", the latter of which is sometimes used as a synonym for fingerprint. Throughout the remainder of this chapter, we therefore use fingerprint and seqnature interchangeably when referring to the fingerprint of some event $e$. Each cluster in the resulting seqnature represents a group of packet sequences that are considered identical, and which appear across a user-defined minimum number of different traffic samples. The seqnature can be used to identify the occurrence of $e$ in unseen traffic by checking if, for every cluster in the seqnature, there exists at least one packet sequence

in the candidate traffic that would fall in that cluster.

**Contributions.** The primary contribution of this chapter is SEQNATURE, a fingerprinting framework that makes it simple to implement, and evaluate the performance of, arbitrary fingerprinting techniques that extract fingerprints that are based on packet sequences. To demonstrate the versatility of SEQNATURE, we use it to implement two fingerprinting techniques that extract two different types of fingerprints, namely endpoint-specific packet-sequence-based fingerprints (EPBFs) and endpoint-based fingerprints (EBFs).

As a secondary contribution, we provide a comparison of the performance of the aforementioned fingerprinting techniques, not only in the context of smart TV apps, but also in the context of events on simple IoT devices, such as smart plugs and smart light bulbs. The results show that almost all smart TV apps, and a large fraction of events on simple IoT devices, can be fingerprinted using either technique. The extracted fingerprints are also distinct from other network traffic: the median number of false positives is 0 for EPBFs, and the general EBF only produces false positives in less than 1% of traffic.

**Outline.** The remainder of this chapter is structured as follows. Section 5.2 describes the design and implementation of the SEQNATURE framework. Section 5.3 defines the two fingerprinting techniques we consider and explains how they are implemented using SEQNATURE. Section 5.4 introduces the datasets we use in our evaluation of the two fingerprinting techniques. Section 5.5 compares the performance of the two fingerprinting techniques. Section 5.6 discusses future directions. Section 5.7 concludes the chapter.

## 5.2 Fingerprinting Framework

In this section, we describe the design and implementation of SEQNATURE, a framework for extracting a network fingerprint for any arbitrary event $e$ that triggers network activity

on a computing device $d$. SEQNATURE has two phases (see Figure 5.1): (i) a *preprocessing* phase, described in Section 5.2.1, that extracts all TCP streams from $e$'s traffic samples and transforms packets into feature vectors; and (ii) an iterative *fingerprint refinement* phase, described in Section 5.2.2, that, by clustering increasingly shorter packet sequences, identifies packet sequences that consistently appear when $e$ occurs. The final fingerprint of $e$, also referred to as the seqnature of $e$, is the set of these packet sequences, accompanied by information about how they may vary, as described in Section 5.2.3. Section 5.2.4 describes a basic algorithm we implement to examine a tabulated traffic sample for the manifestation of a given seqnature. We use this algorithm to examine our datasets for false positives, i.e., when the seqnature of $e$ manifests itself in traffic that does not stem from $e$.

### 5.2.1 Preprocessing

The preprocessing phase of SEQNATURE is summarized in the "Preprocessing" box in Figure 5.1. This phase consists of two steps: (i) a step that filters packets in the raw traffic samples; and (ii) a step that vectorizes packets in the raw traffic samples to create a format suitable for cluster analysis. Below, we detail each step in the context of a single traffic sample, but all $T$ traffic samples are subjected to the same procedure. It is assumed that the traffic samples are collected on $d$'s gateway router and that $d$ is assigned an IP address from one of the private IP address spaces.

**Filtering.** The first preprocessing step filters packets in the raw traffic sample so as to only retain those packets that are relevant for the fingerprinting objective. As we are interested in identifying fingerprints that are observable upstream from the home router's WAN port, the filter only retains IP datagrams to/from $d$, where either the source or the destination is a public IP address. However, as most local networks are configured to use a local DNS resolver by default (the home gateway often assumes this role), DNS messages to/from $d$ are

not subject to the public IP address requirement.

For TCP streams, retransmissions and zero-payload packets are dropped prior to cluster analysis to ensure that the identified seqnature (if any) only comprises packet exchanges that carry data relevant to the application, but *not* packet exchanges that are a result of the intricacies of TCP's operation under certain network conditions. Ideally, this should make the resulting seqnature network-agnostic, enabling a seqnature derived from training data collected in one network to be used for event detection in another network.

The filtering logic may be changed to accommodate different fingerprinting objectives. For example, to extract fingerprints that are only observable on the LAN, the filter should only retain IP datagrams to/from $d$ where the other endpoint's IP address is also in one of the private address spaces.

**Feature Extraction.** The second preprocessing step selects all TCP streams from the filtered traffic sample and transforms each packet into a feature vector. An example of the output and a description of each feature is provided in Table 5.1. We refer to this format as a *tabulated traffic sample* as it lends itself to presentation in table form and is suitable as input for cluster analysis (Section 5.2.2). The extracted features (i) tie the packet to the TCP stream it appeared in (the combination of features Event ID, Sample ID, and Stream ID); (ii) label the packet with the domain name of the server it was exchanged with (Domain); (iii) order the packet relative to other packets in the same stream (Position in Stream); (iv) log the packet's size (Size); and (v) log the packet's direction relative to $d$ (Direction).

The value for the Domain feature is extracted from the TLS Server Name Indication (SNI), if the TCP stream wraps a TLS stream that includes a Client Hello message where the SNI extension is present. Otherwise, the server's IP address is matched against IP addresses returned in preceding DNS responses, and Domain is set to the name queried in the most recent DNS response in which there is an IP address match. If the domain name also cannot

| **Event ID** | **Sample ID** | **Stream ID** | **Domain** | **Position in Stream** | **Size** | **Direction** |
|---|---|---|---|---|---|---|
| Identifies the event $e$ this packet was observed for | Identifies the traffic sample this packet appeared in | Identifies the TCP stream this packet pertains to | The domain name of the server | This packet's position within the TCP stream it pertains to | This packet's size (bytes) | This packet's direction w.r.t. $d$ |
| 28 | 1 | 2 | scribe.logs.roku.com | 1 | 583 | upstream |
| 28 | 1 | 2 | scribe.logs.roku.com | 2 | 1514 | downstream |
| 28 | 1 | 2 | scribe.logs.roku.com | ... | ... | ... |
| 28 | 1 | 2 | scribe.logs.roku.com | 9 | 97 | downstream |
| 28 | 1 | ... | ... | ... | ... | ... |
| 28 | 1 | 6 | tuner.pandora.com | 1 | 293 | upstream |
| 28 | 1 | 6 | tuner.pandora.com | 2 | 188 | upstream |
| 28 | 1 | 6 | tuner.pandora.com | 3 | 362 | downstream |
| 28 | 1 | ... | ... | ... | ... | ... |

Table 5.1: Snippet of a tabulated traffic sample for the Roku app with ID=28 ("Pandora") in the FingerprinTV dataset that was collected as part of the work presented in Chapter 4. Each row represents a single packet.

be determined from DNS, the server's IP address is used for Domain as a last resort.

The value for the Position in Stream feature is the packet's position relative to the other packets in the stream, after retransmissions and packets without TCP payload have been discarded. The position is derived from the packet's timestamp, rather than its sequence number, to form a bidirectional ordering as opposed to two direction-specific orderings. Timestamp-based ordering should also make it easier to add support for UDP later on.

### 5.2.2 Fingerprint Refinement

The fingerprint refinement phase of SEQNATURE is summarized in the "Fingerprint Refinement" box in Figure 5.1. This phase selects packet sequences for inclusion in the seqnature by identifying identical (or similar) packet sequences that consistently (or often) occur when $e$ is triggered. The input to fingerprint refinement is the $T$ tabulated traffic samples resulting from the preprocessing phase described in Section 5.2.1. In summary, fingerprint refinement (i) forms all possible packet sequences of length $n$ from the TCP streams in the $T$ tabulated

traffic samples; (ii) clusters said packet sequences to identify identical (or similar) ones; and (iii) post-processes the resulting clusters to select clusters for inclusion in the seqnature based on criteria such as how many of the $T$ traffic samples the packet sequences in a cluster stem from. Fingerprint refinement is an iterative process that considers increasingly shorter packet sequences, i.e., the aforementioned steps are repeated for all $n \in [P, P-1, ..., n_{\min}]$, where $P \geq n_{\min} \geq 1$. The user controls $P$ and $n_{\min}$ using SEQNATURE's command-line interface. $P$ specifies how many packets of each TCP stream should be considered during fingerprint refinement, and $n_{\min}$ specifies the minimum length of any packet sequence considered for inclusion in the seqnature. Below, we describe each of the three fingerprint refinement steps in detail.

**Forming Packet Sequences.** The first step of each fingerprint refinement iteration forms all possible packet sequences of length $n$ from the first $P$ packets of each TCP stream in the $T$ tabulated traffic samples. As a packet sequence is essentially an $n$-gram of packets in a TCP stream (see Definition 5.1), SEQNATURE forms all possible packet sequences of length $n$ by sliding a window of size $n$ across the first $P$ packets of each TCP stream. An example of how packet sequences of length $n = 4$ are formed from the first $P = 20$ packets of a TCP stream is provided in Figure 5.2.

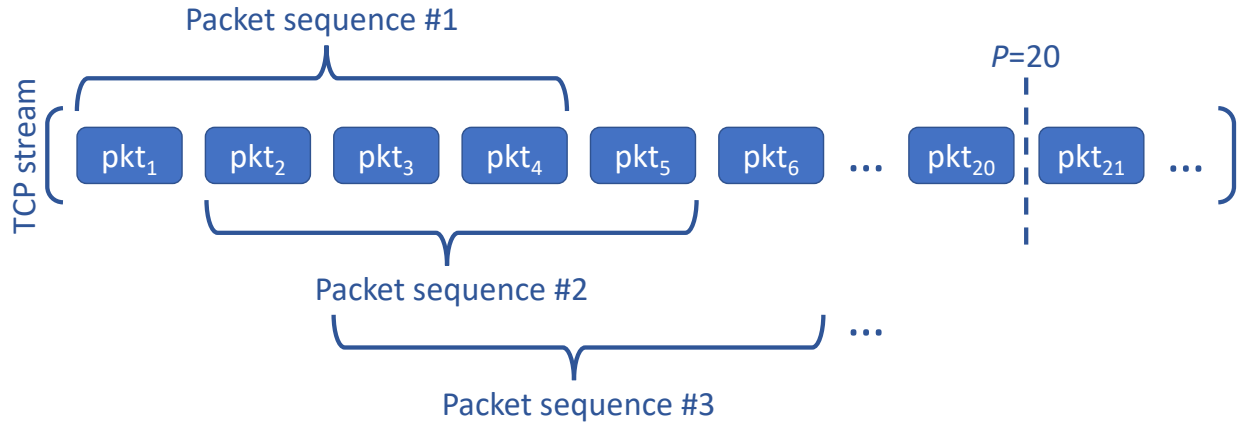



Figure 5.2: Example of how packet sequences of length $n = 4$ are formed from the first $P = 20$ packets of a TCP stream.

The parameter $P$ ensures that fingerprint refinement remains computationally tractable for TCP streams with many packets, e.g., TCP streams carrying large data volumes. Throughout this chapter, we set $P = 20$ unless otherwise stated. This choice is grounded in prior work which has demonstrated that a small prefix of packets of each stream suffices for classifying IoT traffic, mobile app traffic, and DoT/DoH traffic [120, 119, 137, 82, 18]. Using a small value for $P$ also makes online fingerprint detection more feasible in comparison to approaches that need access to all packets in the stream, e.g., those that rely on statistical measures of packet-level features such as packet size.

**Clustering.** The second step of each fingerprint refinement iteration clusters the packet sequences formed in the preceding step to group identical (or similar) packet sequences together based on a user-defined notion of packet sequence similarity. The resulting clusters become candidates for inclusion in the fingerprint as they *may* represent packet exchanges that co-occur with $e$. An example clustering of packet sequences of length $n = 4$ extracted from $T = 3$ different traffic samples is provided in Figure 5.3.

Clustering in SEQNATURE is fully customizable: support for any clustering algorithm avail-

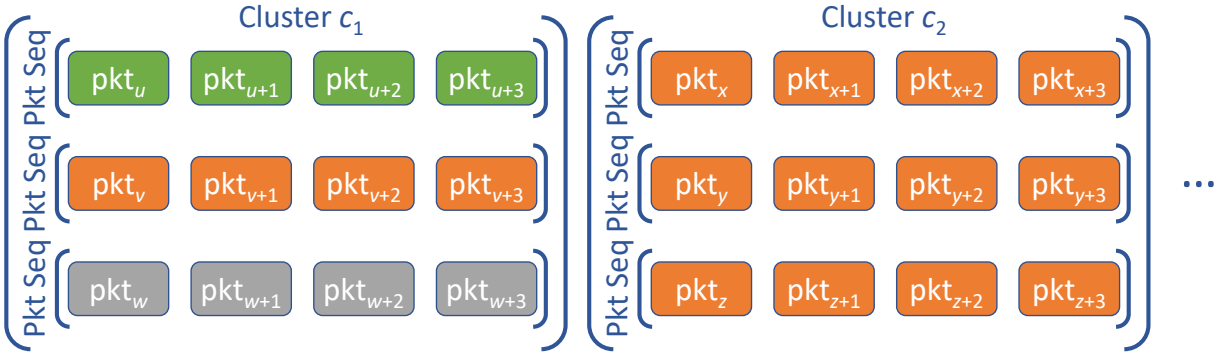



Figure 5.3: Example clustering of packet sequences of length $n = 4$ extracted from $T = 3$ different traffic samples. The color of packets in a packet sequence denotes what traffic sample the packet sequence stems from. For example, all packet sequences with orange packets stem from the same traffic sample. The number of packet sequences in a cluster may vary across clusters (and can be greater than $T$), but all packet sequences across all clusters will always be of the same length $n$, as $n$ only changes between each fingerprint refinement iteration.

able in scikit-learn [106] (or any Python module that adopts its API [30], e.g., hdbscan [94]) can be added with two lines of code; parameters for configuring the algorithm of choice can be specified using SEQNATURE's command-line interface; and any custom distance metric can be introduced by defining a function in a dedicated module. This configurability leaves the user in full control of what packet sequences end up in the same cluster, which in turn allows for the implementation of many different fingerprinting techniques, as demonstrated in Section 5.3.

**Cluster Selection.** The third fingerprint refinement step post-processes the clusters formed in the previous step to determine which, if any, should be included in the seqnature of $e$. A cluster of identical (or similar) packet sequences is not necessarily useful as one part of the seqnature of $e$ as there is no guarantee that all $T$ invocations of $e$ are represented in the cluster. For example, if the same packet sequence occurs $T$ times in a single traffic sample, but not in any of the remaining $T-1$ traffic samples, a cluster of $T$ packet sequences will be formed, but such a cluster is obviously not useful for fingerprinting purposes (cluster $c_2$ in Figure 5.3 illustrates this scenario). For this reason, SEQNATURE defines a parameter, $T_{\text{min}}$, that is used to specify how many different traffic samples packet sequences in a cluster must stem from in order for the cluster to be considered valid for inclusion in the seqnature of $e$. Throughout this chapter, we set $T_{\text{min}} = T$, i.e., the strictest case where a packet sequence must always co-occur with $e$ for the packet sequence to become part of the seqnature of $e$.

Finally, each valid cluster is compared against clusters already included in the seqnature of $e$ to prevent duplicates. Specifically, if a cluster of packet sequences of length $n_j$ has been identified as valid and included in the seqnature, all subsequent fingerprint refinement iterations operating on packet sequences of length $n_i$, where $n_i < n_j$, will give rise to at least one cluster of packet sequences of length $n_i$ that are shorter versions of the packet sequences of length $n_j$ (see Figure 5.4 for an example). Naturally, such clusters should not be included in the seqnature. Therefore, before including a valid cluster $c$ in the seqnature, SEQNATURE

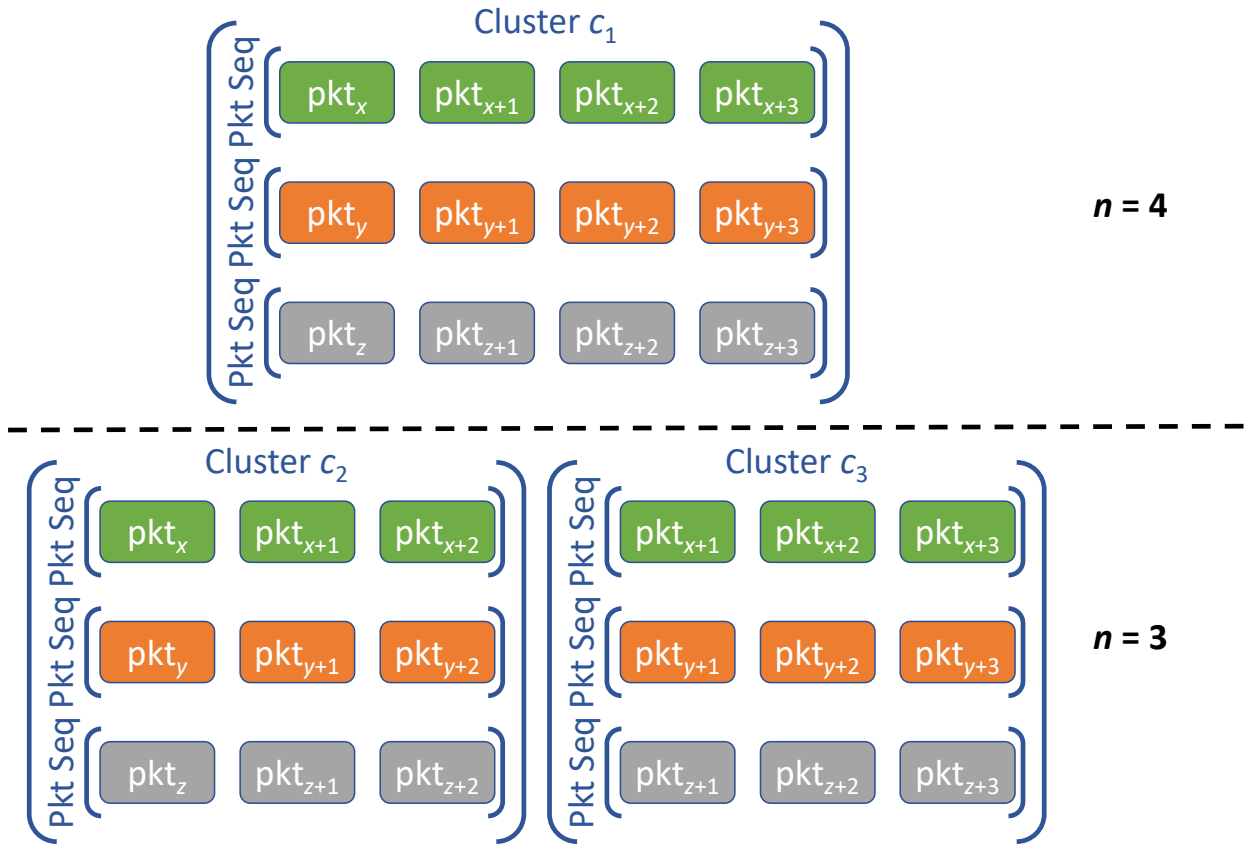
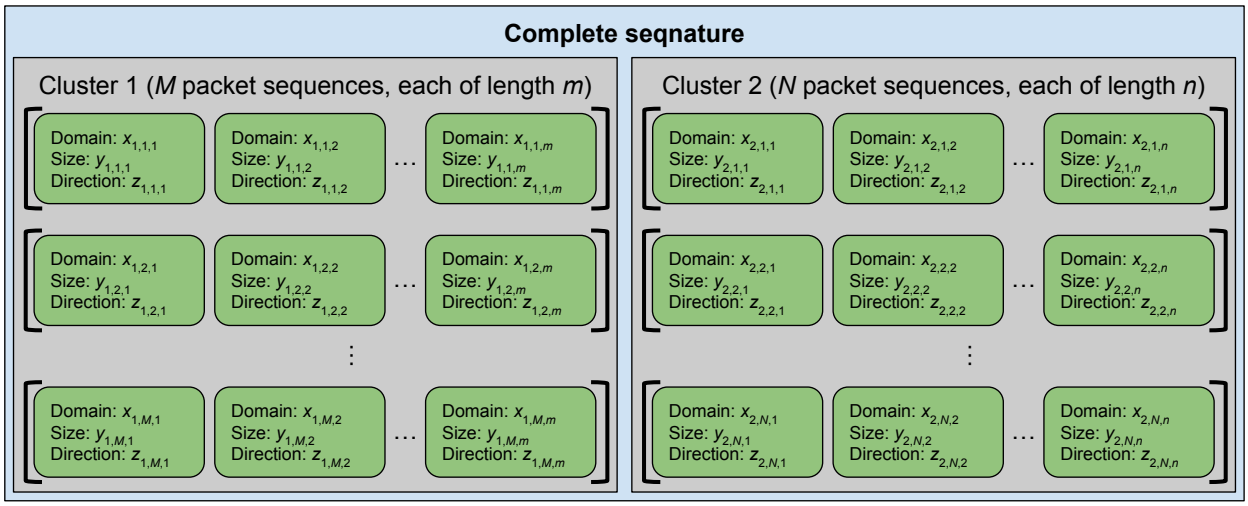



Figure 5.4: Example clusterings for two successive fingerprint refinement iterations ($n = 4$ and $n = 3$). When $n$ decreases, at least one cluster containing shorter versions of packet sequences that have already been included in the seqnature will be formed: in this example, clusters $c_2$ and $c_3$ both (exclusively) consist of shorter versions of the packet sequences in cluster $c_1$.

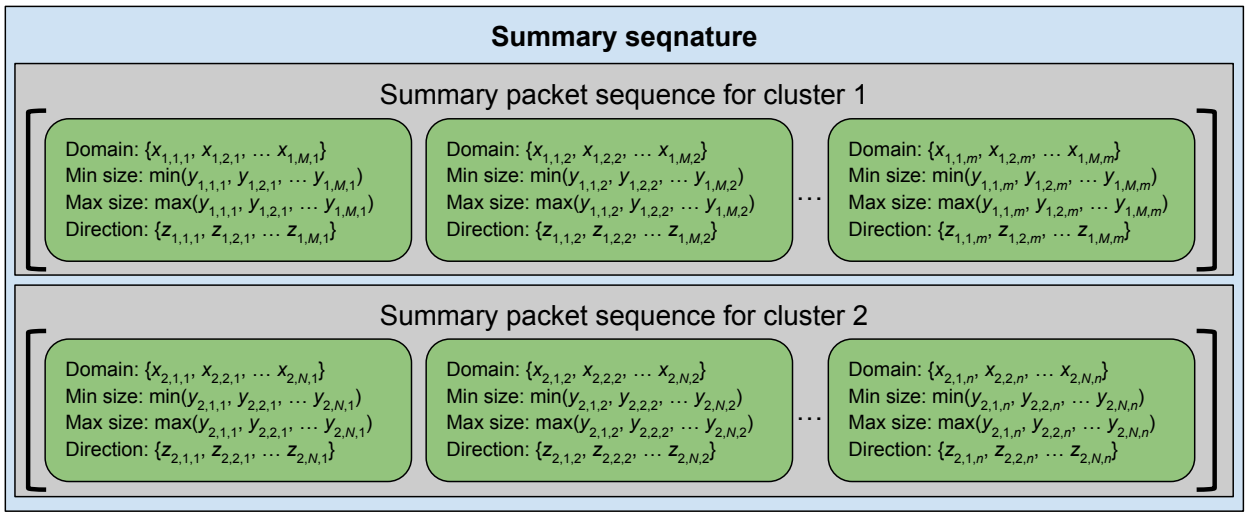
first compares all packet sequences in $c$ against the packet sequences in the clusters that are already in the seqnature. If any packet sequence $s$ in $c$ is already accounted for in the seqnature, in the sense that a longer version of $s$ exists in one of the clusters in the seqnature, $c$ is discarded. To make packet sequence lookup more efficient, each cluster in the seqnature is stored as a generalized suffix tree of the packet sequences it contains [92, 154, 107].

### 5.2.3 Representations of the Resulting Fingerprint

The final seqnature of $e$ is the set of clusters selected for inclusion in the seqnature when fingerprint refinement concludes, i.e., when $n < n_{\min}$. The seqnature can be used to identify the occurrence of $e$ in unseen traffic by checking if, for every cluster in the seqnature, there exists at least one packet sequence in the candidate traffic that would fall in that cluster. To simplify the implementation of such an algorithm, and to make visual inspection of




(a) Complete form.




(b) Summary form.

Figure 5.5: Example of a seqnature that comprises two clusters, represented in complete form and in summary form.

seqnatures easier, we condense the resulting seqnature by replacing each cluster of packet sequences with a single summary packet sequence. We describe this summarization below and provide an example of the same seqnature before and after summarization in Figure 5.5.

Since $n$ only changes between each fingerprint refinement iteration, each cluster in the final seqnature will only contain packet sequences of the same length. Therefore, we can summarize each cluster $c$ as a single packet sequence $p$ in which the packet at index $i$, $p[i]$, encodes, for each feature $f$, the range of values $f$ assumes for packet $i$ across all packet sequences in $c$. For some features, this can simplify the representation significantly. For example, for packet size, we only store the minimum and the maximum size observed across all packets at index $i$. For other features, such as the domain and the packet's direction, we must store the set of values observed across all packets at index $i$ in order to accommodate user-defined distance metrics that do not require these features to assume identical values, e.g., a distance metric that only requires the eSLDs, but not the FQDNs, to match.

### 5.2.4 Fingerprint Matching

The SEQNATURE framework includes functionality for examining a tabulated traffic sample for the manifestation of a seqnature, as represented in summary form (see Section 5.2.3). This functionality is used in Section 5.5.2, where we evaluate the distinctiveness of different types of seqnatures. The seqnature matching algorithm declares a match of seqnature $s$ in tabulated traffic sample $t$ if every summary packet sequence in $s$ has at least one match in $t$. We briefly describe the algorithm below.

To limit memory usage, the algorithm reads and processes $t$ packet-by-packet, as opposed to reading all packets in $t$ into memory at once. For each TCP stream in $t$, the algorithm maintains a buffer of the $n_{\max}$ most recent packets, where $n_{\max}$ is the length of the longest summary packet sequence in $s$. Whenever a new packet is read from $t$ and added to its

respective TCP stream's buffer $b$, every summary packet sequence $p$ in $s$, for which no match has been found in $t$, is compared to the $n$ most recent packets in $b$, where $n$ is the number of packets in $p$ and $n \leq n_{\max}$. In order to facilitate detection of any arbitrary, user-defined type of seqnature, SEQNATURE allows the user the option to define their own custom logic for when the $n$ most recent packets in $b$ is considered a match of $p$. The algorithm terminates when either (i) a match has been found in $t$ for all packet sequences in $s$, in which case there is a match of $s$ in $t$; or (ii) all packets in $t$ have been processed, in which case there is no match of $s$ in $t$.

## 5.3 Fingerprinting Techniques

In this section, we demonstrate the versatility of SEQNATURE by using it to implement two different fingerprinting techniques. This section describes the two fingerprinting techniques and how SEQNATURE is configured to implement them. An evaluation of the performance of each fingerprinting technique is deferred to Section 5.5.

### 5.3.1 Endpoint-Specific Packet-Sequence-Based Fingerprints

The first fingerprinting technique we implement illustrates how SEQNATURE can be used to combine existing fingerprinting techniques. This fingerprinting technique identifies fingerprints that can be thought as a combination of DBFs and PBFs (see Sections 4.3.1 and 4.3.2), with a few additional enhancements, namely that Internet endpoints are identified by their IP addresses when no domain name information is available, and packet exchanges are not limited to just packet pairs, but can be packet sequences of any length between $P$ and 2. We refer to a fingerprint identified by this fingerprinting technique as an endpoint-specific packet-sequence-based fingerprint, or EPBF for short. EPBFs are rooted in the hypothesis

that whenever the event $e$ occurs, some subset of data is always exchanged with the same server(s): an EPBF for the event $e$, formalized in Definition 5.2, is the set of packet sequences that are consistently exchanged with the same endpoints whenever $e$ occurs.

**Definition 5.2.** *The* endpoint-specific packet-sequence-based fingerprint *(EPBF) of event $e$ is the set $S$ of packet sequences s.t. for every packet sequence $p$ in $S$, $p$ appears at least once in at least $T_{\min}$ of the $T$ traffic samples for $e$. Two packet sequences $p_1$ and $p_2$ are considered identical iff (i) $|p_1| = |p_2|$, i.e., $p_1$ and $p_2$ are the same length; (ii) $p_1$ and $p_2$ are exchanged with the same endpoint, where the endpoint is identified by its FQDN, when available, or otherwise by its IP address; (iii) the directions of packets at corresponding indices in $p_1$ and $p_2$ are identical; and (iv) $\sum_{i=1}^{n} ||p_1[i]| - |p_2[i]|| \leq h$, i.e., the sum of absolute differences in sizes of packets at corresponding indices in $p_1$ and $p_2$ is below a user-defined threshold $h$.*

To enable extraction of EPBFs using SEQNATURE, we define a distance metric that considers the distance between packet sequences $p_1$ and $p_2$ to be *maximal* if either (i) $p_1$ and $p_2$ are exchanged with different endpoints (endpoints are considered different if they are identified by different FQDNs, different IP addresses, or if one endpoint is identified by an FQDN while the other is identified by an IP address); or (ii) any pair of packets at corresponding indices in $p_1$ and $p_2$ go in opposite directions. Otherwise, the distance is $\sum_{i=1}^{n} ||p_1[i]| - |p_2[i]||$, i.e., the sum of absolute differences in sizes of packets at corresponding indices in $p_1$ and $p_2$.

In Section 5.5, we extract, and evaluate the performance of, the strictest possible case of EPBFs, i.e., when $T_{\min} = T$ and $h = 0$ (see Definition 5.2). In other words, a cluster must contain packet sequences from all $T$ traffic samples to be considered for inclusion in the EPBF, and packet sequences must be identical to end up in the same cluster. This strict case of EPBFs is achieved by configuring SEQNATURE to use the DBSCAN clustering algorithm [42] with parameters $\epsilon = 0$ and `MinPts` $= T = 10$.

### 5.3.2 Endpoint-Based Fingerprints

The second fingerprinting technique we implement illustrates how SEQNATURE can also be used to implement fingerprinting techniques that are not focused on identifying similar packet sequences, but instead focus on stream-wide features. A fingerprint extracted using this fingerprinting technique is the set of endpoints that are consistently contacted whenever the event $e$ occurs. We refer to such a fingerprint as an endpoint-based fingerprint (EBF) and formalize it in Definition 5.3. EBFs can be thought of as slightly enhanced DBFs (see Section 4.3.1), as they may include endpoints where the domain name cannot be determined.

**Definition 5.3.** *The* endpoint-based fingerprint *(EBF) of event $e$ is the set $S$ of endpoints s.t. for every endpoint $d$ in $S$, $d$ is contacted in at least $T_{\min}$ of the $T$ traffic samples for $e$. An endpoint is identified by its FQDN, or, when no domain name information is available, by its IP address.*

To enable extraction of EBFs using SEQNATURE, we define a distance metric that considers the distance between packet sequences $p_1$ and $p_2$ to be 0 if $p_1$ and $p_2$ are exchanged with the same endpoint, i.e., if $p_1$ and $p_2$ are exchanged with the same FQDN or the same IP address. Otherwise, the distance is maximal. As the endpoint is the same for all packets in a stream, there is no need to analyze more than one packet per stream during fingerprint refinement. Therefore, we use $P = n_{\min} = 1$ when extracting EBFs.

In Section 5.5, we extract, and evaluate the performance of, the strictest possible case of EBFs, i.e., when $T_{\min} = T$. In other words, we extract EBFs that only consist of the endpoints that are contacted in all traffic samples for the event $e$. This strict case of EBFs is achieved by configuring SEQNATURE to use the DBSCAN clustering algorithm [42] with parameters $\epsilon = 0$ and `MinPts` $= T = 10$.

## 5.4 Datasets

To illustrate how SEQNATURE facilitates comparisons of different fingerprinting techniques, we use SEQNATURE's implementations of the fingerprinting techniques described in Section 5.3 to extract fingerprints from two datasets. Taken together, these datasets span different software and hardware categories: smart TV apps and events on simple IoT devices, such as smart plugs and smart light bulbs. This section introduces the datasets and describes how we preprocess them. Section 5.5 provides the results.

### 5.4.1 FingerprinTV: Smart TV Apps

As our primary objective in this dissertation is to shed light on the privacy implications of smart TVs, our main dataset is the FingerprinTV dataset [156], which was collected as part of the work described in Chapter 4. In summary, the dataset consists of 10 samples of the network traffic that each of the top-1000 apps on Apple TV, Fire TV, and Roku generate at launch time, for a total of 30K traffic samples (10K per smart TV platform). See Section 4.2 for details on how the apps were selected and how network traffic was collected. Aside from the preprocessing performed as part of the fingerprinting procedure (see Section 5.2.1), no preprocessing is necessary as this dataset is already in the format expected by SEQNATURE.

### 5.4.2 PingPong: Events on IoT Devices

The second dataset we consider consists of network traffic that simple IoT devices, such as smart plugs and smart light bulbs, generate when an event is triggered, e.g., when the user toggles a smart plug ON using the companion app of the smart plug. We refer to this dataset as the PingPong dataset as it was published alongside [151], which proposed a fingerprinting technique of the same name.

The PingPong dataset is made available as one large packet capture (PCAP file) for each combination of IoT device, event, and scenario. As SEQNATURE expects $T$ separate traffic samples, we split the full PCAP file using the event timestamps provided as part of the PingPong dataset. For each split, we include 15 seconds of traffic following the event timestamps in the resulting traffic sample. This duration is chosen to stay consistent with [151].

The PingPong dataset spans 19 IoT devices, and there are between two and six different events per device. Some devices are tested in different scenarios: local phone, remote phone, and/or IFTTT. In the local (remote) phone scenario, the event is triggered using the companion app of the IoT device from a smartphone that is (not) part of the same LAN as the IoT device. In the IFTTT scenario, the event is triggered using IFTTT [2]. When extracting fingerprints from the PingPong dataset using SEQNATURE, we attempt to extract a fingerprint for each combination of IoT device, event, and scenario, i.e., each such combination is analogous to the notion of an event $e$ used throughout Sections 5.2 and 5.3. There are 140 such combinations. While the PingPong dataset contains 50 invocations of each event, we only consider the first 10 invocations in order to keep the number of traffic samples consistent across datasets. The PingPong dataset, as considered here, therefore encompasses 1400 traffic samples.

## 5.5 Fingerprinting Results

In this section, we compare the performance of the two fingerprinting techniques we implement using SEQNATURE (see Section 5.3). We start by reporting the number of fingerprints discovered by each fingerprinting technique in Section 5.5.1. Then, in Section 5.5.2, we examine if the extracted fingerprints are distinct among other traffic by analyzing how many false positives they give rise to.

| Subdivision | Events | Prevalence | |
|---|---|---|---|
| | | EPBFs | EBFs |
| **FingerprinTV** | | | |
| Apple TV | 1000 | 96% (958) | 96% (959) |
| Fire TV | 1000 | 99% (993) | 100% (997) |
| Roku | 1000 | 100% (1000) | 100% (1000) |
| **PingPong** | | | |
| Full dataset | 140 | 66% (92) | 81% (113) |

Table 5.2: Prevalence of EPBFs and EBFs in the FingerprinTV and PingPong datasets. The prevalence is the percentage of events in the dataset that exhibit a fingerprint.

### 5.5.1 Prevalence

Table 5.2 reports the prevalence of EPBFs and EBFs in the two datasets from Section 5.4. The prevalence is the percentage of events in the respective dataset that exhibit the respective type of fingerprint.

**Smart TV Apps.** The two fingerprinting techniques from Section 5.3 perform similarly in terms of the number of smart TV apps they are able to fingerprint, as nearly all smart TV apps exhibit both EPBFs and EBFs. This is somewhat surprising as EPBFs are essentially stricter special cases of EBFs, where not only the endpoints, but also the packet sizes and the packet directions, need to stay consistent across traffic samples. Thus, if a smart TV app consistently contacts one or more endpoints, in most cases, a subset of the app's communication with at least one of those endpoints will be deterministic, i.e., there will be at least one sequence of two or more packets where the packet sizes and the packet directions stay consistent across all traffic samples.

When we compare the prevalence of EBFs to the prevalence of DBFs (see Section 4.5.1), we observe that EBFs' use of the endpoint's IP address as a fallback, when no domain name information is available, results in a slight increase in the number of Fire TV apps that can be fingerprinted.

The astute reader may question why EPBFs are more prevalent than PBFs (see Section 4.5.2), as EPBFs require the endpoint, the packet sizes, and the packet directions to stay consistent across traffic samples, whereas PBFs only require the packet sizes and the packet directions to stay consistent. The explanation for this is that each of the PBFs considered in Section 4.5.2 only consists of the packet-pairs that appear $T$ times in total across the $T$ traffic samples (see Definition 4.2 and the end of Section 4.3.1). This upper bound on the number of occurrences of a packet-pair stem from the fact that PBFs were extracted using PingPong [151], which discards packet-pairs that occur multiple times per event. In contrast, an EPBF only requires a packet sequence to occur in at least $T_{\text{min}}$ traffic samples, but does not constrain how many times the packet sequence is allowed to occur within each traffic sample or in total across the $T$ traffic samples.

**Events on IoT Devices.** The prevalence of EPBFs and EBFs in the PingPong dataset reflects that EPBFs are stricter special cases of EBFs: 81% of events on IoT devices exhibit EBFs, but only 66% exhibit EPBFs.

The prevalence of EPBFs and EBFs is significantly lower in the PingPong dataset than in the FingerprinTV dataset. However, some devices in the PingPong dataset do not generate external TCP traffic for some event and scenario combinations (recall from Section 5.2.1 that our focus in this work is on fingerprints that are observable upstream from the home router's WAN port). Naturally, these events are impossible to fingerprint using any fingerprinting technique that operates on external TCP traffic. If we only consider events where the target device exchanges at least one TCP packet with an Internet endpoint in every traffic sample, then the total number of events in the PingPong dataset decreases from 140 to 116. With this baseline, the prevalence becomes 97% and 79% for EBFs and EPBFs, respectively.

### 5.5.2 Distinctiveness

In this section, we demonstrate how SEQNATURE can be used to compare the discriminative power of different types of fingerprints. The relative discriminative power of different types of fingerprints can be compared by examining how distinct fingerprints of each type are among other traffic. SEQNATURE facilitates such comparisons by (i) making it easy to define and extract different types of fingerprints; and (ii) providing functionality that searches a dataset for manifestations of each fingerprint.

We refer to a manifestation of the fingerprint, $F(e_1)$, of event $e_1$ in a traffic sample for event $e_2$, where $e_1 \neq e_2$, as a false positive. We compare the discriminative power of EPBFs and EBFs by (i) searching all traffic samples of each dataset for manifestations of each and every EPBF and EBF; and then (ii) summarizing, using statistical measures, how many false positives the general EPBF and the general EBF give rise to. This false positive analysis is a more comprehensive evaluation of the fingerprints' discriminative power than the notion of distinctiveness used throughout Chapter 4. In Chapter 4, we only compare each app's fingerprint to all other apps' fingerprints, i.e., an app's fingerprint is only compared to the subset of traffic that is selected for inclusion in other apps' fingerprints. Here, we compare each fingerprint against *all* traffic in each dataset, including traffic that does not find its way into any fingerprint.

When analyzing the discriminative power of each type of fingerprint, we report the total number of false positives the general fingerprint gives rise to, but also the number of different events these false positives are spread across. Recall from Section 5.4 that each dataset contains $T = 10$ traffic samples for each event. Now, assume that we identify 10 false positives for the fingerprint, $F(e)$, of some event $e$. These 10 false positives could stem from manifestations of $F(e)$ in 10 traffic samples from 10 different events, or from $F(e)$ consistently appearing in all traffic samples for a single event—or somewhere in between

these two extremes. Therefore, if we only consider the total number of false positives for each fingerprint, when we report aggregate results for all fingerprints in each dataset, it becomes difficult to discern if the false positives are spread across traffic samples for many different events, or if they repeatedly occur in most/all traffic samples for a single/few event(s). In our view, the number of different events the false positives are spread across is therefore a better measure of how distinct a fingerprint is among a wide variety of other traffic.

**Comparison of False Positives for EPBFs and EBFs.** Tables 5.3 and 5.4 summarize, using statistical measures, how many false positives a single EPBF and a single EBF, respectively, produce, when (subdivisions of) the datasets introduced in Section 5.4 are searched for false positives. Each "Total" column reports the number of false positives across all traffic samples in the respective dataset, and each "Spread" column reports the number of different events in the respective dataset that give rise to one or more false positives.

As evident from Table 5.3, the mode and the median number of false positives is 0 across the board, which means that the general EPBF never manifests itself in traffic from other events. The within-platform median number of false positives is slightly larger for Apple TV, Roku, and PingPong EBFs (see the top-left to bottom-right diagonal of Table 5.4). Nevertheless, these medians are still well below 1% of the total number of traffic samples and events in each dataset (see Section 5.4), and the general EBF is thus distinct from the vast majority of other events' traffic.

In total, this comparison of EPBFs and EBFs suggests that the additional complexity of EPBFs may not be worthwhile if the use case for the fingerprints can tolerate a few false positives. However, a limitation of this closed-world false positive test is that each fingerprint is only compared against traffic from a single event at a time. Most gateways in real-world home networks use Network Address Translation (NAT), which makes traffic from multiple devices appear as traffic from a single device. This increases the potential for false positives, as a false positive will arise whenever the union of traffic from all devices in the home

| | Target dataset | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Apple TV (FingerprinTV) | | Fire TV (FingerprinTV) | | Roku (FingerprinTV) | | PingPong | |
| **Fingerprints** | Total | Spread | Total | Spread | Total | Spread | Total | Spread |
| Apple TV (FingerprinTV) 958 EPBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 5958.00 | 694.00 | 30.00 | 3.00 | 10.00 | 1.00 | 0.00 | 0.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | 357.95 | 47.05 | 0.03 | 0.00 | 0.01 | 0.00 | 0.00 | 0.00 |
| | 1206.09 | 156.66 | 0.97 | 0.10 | 0.32 | 0.03 | 0.00 | 0.00 |
| Fire TV (FingerprinTV) 993 EPBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 931.00 | 118.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | 0.00 | 0.00 | 14.18 | 1.99 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 77.87 | 10.21 | 0.00 | 0.00 | 0.00 | 0.00 |
| Roku (FingerprinTV) 1000 EPBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 4956.00 | 496.00 | 0.00 | 0.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | 0.00 | 0.00 | 0.00 | 0.00 | 184.49 | 21.94 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 737.40 | 80.73 | 0.00 | 0.00 |
| PingPong 92 EPBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 30.00 | 3.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 5.38 | 0.59 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 8.96 | 0.92 |

Table 5.3: Number of false positives per EPBF (minimum / maximum / **mode** / **median** / mean / standard deviation). Total: total number of false positives across all traffic samples in the dataset. Spread: number of different events in the dataset that give rise to one or more false positives.

network contains a manifestation of the fingerprint, even when no single device generates traffic that matches the fingerprint. As EBFs only require the set of endpoints to match, but pay no attention to the data exchanged with said endpoints, it is plausible that EBFs may generate (significantly) more false positives than EPBFs in real-world settings. We leave a more exhaustive evaluation of false positives to future work as our objective here is not to definitively rank EPBFs and EBFs w.r.t. their respective costs and benefits, but rather to

| | Target dataset | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Apple TV (FingerprinTV) | | Fire TV (FingerprinTV) | | Roku (FingerprinTV) | | PingPong | |
| **Fingerprints** | Total | Spread | Total | Spread | Total | Spread | Total | Spread |
| Apple TV (FingerprinTV) 959 EBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 6238.00 | 696.00 | 1350.00 | 135.00 | 1696.00 | 173.00 | 10.00 | 2.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | **10.00** | **2.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | 590.24 | 65.39 | 8.25 | 0.86 | 4.11 | 0.52 | 0.01 | 0.01 |
| | 1557.30 | 173.31 | 65.68 | 6.83 | 69.29 | 7.73 | 0.33 | 0.09 |
| Fire TV (FingerprinTV) 997 EBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 380.00 | 38.00 | 1799.00 | 184.00 | 6.00 | 2.00 | 0.00 | 0.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | 0.38 | 0.38 | 48.32 | 5.21 | 0.01 | 0.00 | 0.00 | 0.00 |
| | 12.03 | 1.20 | 190.37 | 20.23 | 0.19 | 0.06 | 0.00 | 0.00 |
| Roku (FingerprinTV) 1000 EBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 9990.00 | 999.00 | 0.00 | 0.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | **0.00** | **0.00** | **0.00** | **0.00** | **40.00** | **4.00** | **0.00** | **0.00** |
| | 0.00 | 0.00 | 0.00 | 0.00 | 941.56 | 96.50 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 2273.63 | 227.50 | 0.00 | 0.00 |
| PingPong 113 EBFs | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 128.00 | 13.00 |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **10.00** | **1.00** |
| | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** | **10.00** | **1.00** |
| | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 27.54 | 2.90 |
| | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 29.93 | 3.07 |

Table 5.4: Number of false positives per EBF (minimum / maximum / **mode** / **median** / mean / standard deviation). Total: total number of false positives across all traffic samples in the dataset. Spread: number of different events in the dataset that give rise to one or more false positives.

demonstrate how SEQNATURE facilitates comparisons of different fingerprinting techniques.

## 5.6 Future Directions

The work presented in this chapter is ongoing. Going forward, we plan to (i) analyze the extracted EPBFs and EBFs in further detail; and (ii) use SEQNATURE to implement and

evaluate additional fingerprinting techniques. We outline these plans below.

### 5.6.1 Further Analysis of EPBFs and EBFs

**EPBFs and EBFs with Many False Positives.** While the general EPBF and the general EBF produce very few false positives, a small subset of EPBFs and EBFs give rise to many false positives, as evident from the "maximum" entries in cells across the top-left to bottom-right diagonals of Tables 5.3 and 5.4. We plan to examine these fingerprints in more detail to understand why they have little discriminative power.

Preliminary observations indicate that if an EPBF of a smart TV app gives rise to many false positives, it is generally because the EPBF only comprises a few packet sequences that are exchanged with platform-specific endpoints. These packet exchanges likely stem from the app's use of operating system (OS) services that incur network activity. If an EPBF only consists of these packet exchanges, it will manifest itself in traffic from all other apps that also utilize these OS services. This also helps explain why most false positives for EPBFs are within-platform (i.e., fall along the top-left to bottom-right diagonal of Table 5.3): if most apps consistently utilize these OS services, then the resulting packet exchanges will become part of most apps' EPBFs, which will make the EPBFs distinct from traffic of apps on other platforms, as apps on other platforms will not utilize these OS services and therefore not produce the corresponding packet exchanges.

**False Positives in NAT'ed Traffic from Multiple Devices.** Recall from Section 5.5.2 that NAT increases the potential for false positives as NAT makes traffic from multiple devices appear as traffic from a single device, and a false positive will therefore arise if the combination of traffic from all devices in the network contains the fingerprint. To enable an evaluation of the number of false positives in such scenarios, one option is to simulate NAT'ed traffic by combining traffic samples from a set of events $E$ into a single traffic sample

and then examining the resulting traffic sample for false positives of the fingerprint, $F(e)$, of event $e$, where $e \notin E$.

### 5.6.2 Additional Fingerprinting Techniques

To enable a comparison of the relative importance of different traffic features, we plan to use SEQNATURE to implement and evaluate additional fingerprinting techniques that consider different features. For example, we plan to implement a fingerprinting technique that identifies fingerprints that consist of packet sequences where the packet sizes and packet directions stay consistent across traffic samples, irrespective of what endpoints these packet sequences are exchanged with. By comparing the number of false positives for this fingerprinting technique to the number of false positives for EBFs, we can provide intuition about whether the endpoint is the most important feature in EPBFs (as the results presented in Section 5.5.2 suggest), or if the combination of packet size and packet direction result in similar discriminative power.

We also plan to experiment with the granularity of individual features to investigate if the fingerprints retain their discriminative power when the granularity of a feature is reduced. For example, we intend to use SEQNATURE to both implement fingerprints that comprise the set of consistently contacted FQDNs (i.e., DBFs, see Section 4.5.1), as well as fingerprints that comprise the set of consistently contacted eSLDs. If the latter has comparable discriminative power, it will generally be the preferred type of fingerprint as it will often be more widely applicable than DBFs: it is not uncommon for subdomains to contain regional identifiers, and some DBFs can therefore only be used in a specific region.

## 5.7 Summary

In this chapter, we presented SEQNATURE, a general fingerprinting framework that makes it simple to implement, and evaluate the performance of, arbitrary fingerprinting techniques that extract network fingerprints that are based on packet sequences. We demonstrated the versatility of SEQNATURE by implementing two different fingerprinting techniques. We used these two fingerprinting techniques to extract fingerprints for smart TV apps, but also for events on simple IoT devices. We examined the discriminative power of the extracted fingerprints by relying on SEQNATURE's ability to search network traffic for false positives and found that most fingerprints, extracted using either of the two fingerprinting techniques, are distinct from other traffic. By taking similar steps, future work can easily compare the performance of multiple different fingerprinting techniques. To this end, we plan to make SEQNATURE publicly available.

# Chapter 6

# Conclusion

## 6.1 Summary

In this dissertation, we studied the privacy implications of smart TVs using a network measurement approach. In Chapter 3, we conducted a large-scale measurement study of the advertising and tracking ecosystems of different smart TV platforms. Network traffic from smart TVs used in the wild revealed that smart TVs connect to well-known and platform-specific advertising and tracking services (ATSes). To understand if this behavior was attributable to select few apps, or a general theme across smart TV apps, we developed software tools that automatically test Roku and Fire TV apps. Using these tools, we collected network traffic from the top-1000 Roku and Fire TV apps. Our analysis of this traffic revealed that most apps contact between one and five different ATSes, while about 10% of Roku and Fire TV apps contact 20+ and 10+ ATSes, respectively. We observed that some ATS organizations only operate on one platform, suggesting that the ATS ecosystems of different smart TV platforms have substantial differences. We also evaluated DNS-based blocklists' ability to prevent smart TVs from accessing ATSes and found that even smart

TV-specific blocklists miss ads and incur functionality breakage. Finally, we found (i) that hundreds of apps exfiltrate personally identifiable information (PII) to third parties and platform domains; and (ii) alarming evidence of joint exposure of the advertising ID and static PII values, which effectively eliminates the user's ability to opt out of ad personalization.

In Chapter 4, we examined if an in-network observer, e.g., an Internet Service Provider (ISP), could infer what smart TV app is launched. To facilitate this study, we designed and implemented FingerprinTV, which is a fully automated methodology for extracting three different kinds of network fingerprints from the network traffic of smart TV apps and assessing the resulting network fingerprints' performance. We applied FingerprinTV to the top-1000 apps of the three most popular smart TV platforms, namely Roku, Fire TV, and Apple TV. The results revealed that smart TV app network fingerprinting is feasible and effective: even the least prevalent type of fingerprint manifested itself in at least 68% of apps of each platform, and up to 89% of fingerprints uniquely identified a specific app when two fingerprinting techniques were used together. We analyzed apps that exhibited identical fingerprints in further detail and found that these apps would often stem from the same developer, or had been built using the same "no code" toolkit. We also found that many apps that were present on all three platforms exhibited platform-specific fingerprints.

In Chapter 5, we expanded our scope beyond smart TVs and devised a general fingerprinting framework, Seqnature, which can be used to identify network fingerprints of any arbitrary event $e$ on any software/hardware platform, including, but not limited to, smart TVs. Through a range of customizable parameters, Seqnature enables both joint and separate use of the fingerprinting techniques considered in Chapter 4, and makes it simple to implement any fingerprinting technique that can be formulated as a problem of identifying packet exchanges that consistently appear when the fingerprinted event $e$ is triggered. We demonstrated how Seqnature facilitates comparisons of different fingerprinting techniques by using it to implement two different fingerprinting techniques. We used the two finger-

printing techniques to extract fingerprints for smart TV apps and for events on simple IoT devices. Both fingerprinting techniques were capable of fingerprinting almost all smart TV apps as well as the majority of events on IoT devices. We found that one fingerprinting technique produced fingerprints that had more discriminative power, as these fingerprints gave rise to slightly fewer false positives than the fingerprints extracted using the other fingerprinting technique. However, the other fingerprinting technique is less compute-intensive and may therefore be preferable in scenarios where computing resources are limited and a few false positives can be tolerated.

## 6.2 Perspective

Smart TVs offer convenient access to rich entertainment, but this comes at the cost of privacy as most smart TV apps are monetized through behavioral advertising, which relies on tracking to profile users. Our analysis of advertising and tracking on smart TVs has revealed that tracking is pervasive in smart TV apps when smart TVs are used in their default configurations. As opting out of tracking is not straightforward [54], and since some smart TVs require the user to agree to some minimum level of data collection [33], privacy-conscious use of smart TVs is a luxury reserved for advanced users. There is thus a need for legislative action to cause a shift from the current consent-based model, to a model in which settings are required to be privacy-friendly by default.

Due to the closed nature of most smart TV platforms, privacy enhancing tools are generally limited to DNS-based blocklists. Since ATS ecosystems of different smart TV platforms appear to have substantial differences, blocklists must be tailored to each smart TV platform. Unfortunately, this is a daunting task, as most blocklists are manually curated. There is therefore a need for additional research on how to automate blocklist generation for smart TVs. Such research may draw inspiration from recent inroads made on automatic blocklist

generation for websites [78]. To facilitate research on automatic blocklist generation for Roku and Fire TV, we have made our software tools that automate interaction with Roku and Fire TV apps available [158, 157].

As discussed in Section 4.9, users can rely on a VPN that pads packets to prevent in-network observers, e.g., their ISPs, from inferring their smart TV app usage. However, this adds additional network overhead, which is not ideal as most smart TV apps are already bandwidth-intensive as they are used to stream video content. Moreover, the average user cannot be expected to posses the technical know-how necessary to configure a VPN, especially if the smart TV platform does not support on-device VPNs, in which case the only option is to configure the home router to tunnel all WAN traffic through a VPN. Smart TV app developers and operators can, with relatively little effort, help protect their users from being profiled using simple fingerprinting techniques, such as DBFs (see Section 4.3.1), by updating their apps and servers to use DNS-over-HTTPS/DNS-over-TLS (DoH/DoT) in combination with Encrypted SNI (ESNI). However, this only offers limited protection, as most smart TV apps connect to third-party servers, which the smart TV app developer/operator does not control and therefore cannot configure to use ESNI. The smart TV app developer/operator should therefore consider proxying their app's requests to third-party servers through their own infrastructure to eliminate these servers from their app's DBF.

# Bibliography


[1] Crunchbase. https://www.crunchbase.com/. [Online; accessed 2019-08-29].

[2] IFTTT. https://ifttt.com.

[3] OpenDNS Domain Tagging. https://community.opendns.com/domaintagging/. [Online; accessed 2019-08-24].

[4] VirusTotal. https://www.virustotal.com/. [Online; accessed 2019-08-24].

[5] Ooyala IQ SDK for Roku. https://github.com/ooyala/iq-sdk-roku, 2015. [Online; accessed 2019-04-22].

[6] Connected TV Advertising is Surging. https://www.videonuze.com/article/connected-tv-advertising-is-surging, 2017. [Online; accessed 2019-05-10].

[7] AntMonitor open-source. https://github.com/UCI-Networking-Group/AntMonitor, 2018. [Online; accessed 2019-05-10].

[8] Amazon: Smart TVs. https://www.amazon.com/smart-tv-store/b?ie=UTF8&node=5969290011, 2019. [Online; accessed 2019-05-10].

[9] Android tcpdump. https://www.androidtcpdump.com/, 2019. [Online; accessed 2019-04-11].

[10] Customising Sources for Ad Lists. https://github.com/pi-hole/pi-hole/wiki/Customising-Sources-for-Ad-Lists, 2019.

[11] MoaAB: Mother of All AD-BLOCKING. https://forum.xda-developers.com/showthread.php?t=1916098, 2019. [Online; accessed 2019-04-22].

[12] Pi-Hole: A black hole for Internet advertisements. https://pi-hole.net/, 2019. [Online; accessed 2019-05-11].

[13] A. Acar, H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi, and S. Uluagac. Peek-a-boo: I see your smart home activities, even encrypted! In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 207–218, New York, NY, USA, 2020. Association for Computing Machinery.

[14] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, and Z.-L. Zhang. A tale of three CDNs: An active measurement study of Hulu and its CDNs. In *2012 Proceedings IEEE INFOCOM Workshops*, pages 7–12. IEEE, 2012.

[15] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1362–1380, 2019.

[16] J. Althouse, J. Atkinson, and J. Atkins. JA3. `https://github.com/salesforce/ja3`.

[17] Amazon.com, Inc. Amazon DSP. `https://advertising.amazon.com/products/amazon-dsp`, 2019. [Online; accessed 2019-05-10].

[18] B. Anderson and D. McGrew. Machine Learning for Encrypted Malware Traffic Classification: Accounting for Noisy Labels and Non-Stationarity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1723–1732, New York, NY, USA, 2017. Association for Computing Machinery.

[19] B. Anderson and D. McGrew. Accurate TLS Fingerprinting using Destination Context and Knowledge Bases, 2020.

[20] Antidot. Content Delivery Platform. `https://www.antidot.net/content-delivery-platform/`, 2019. [Online; accessed 2019-12-05].

[21] Apple Inc. iTunes Preview. `https://apps.apple.com/us/genre/ios/id36`.

[22] Apple Inc. User Interface Testing. `https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html`, 2017.

[23] N. Apthorpe, D. Reisman, and N. Feamster. A Smart Home is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic, 2017.

[24] L. Bernaille and R. Teixeira. Early Recognition of Encrypted Applications. In S. Uhlig, K. Papagiannaki, and O. Bonaventure, editors, *Passive and Active Network Measurement*, pages 165–175, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[25] L. Bernaille, R. Teixeira, and K. Salamatian. Early Application Identification. In *Proceedings of the 2006 ACM CoNEXT Conference*, CoNEXT '06, New York, NY, USA, 2006. Association for Computing Machinery.

[26] S. Bhat, D. Lu, A. Kwon, and S. Devadas. Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning. *Proceedings on Privacy Enhancing Technologies*, 4:292–310, 2019.

[27] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In G. Danezis and D. Martin, editors, *Privacy Enhancing Technologies*, pages 1–11, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[28] T. Böttger, F. Cuadrado, G. Tyson, I. Castro, and S. Uhlig. Open Connect Everywhere: A Glimpse at the Internet Ecosystem through the Lens of the Netflix CDN. *ACM SIGCOMM Computer Communication Review*, 48(1):28–34, 2018.

[29] L. Brotherston. TLS Fingerprinting: Smarter Defending & Stealthier Attacking. `https://blog.squarelemon.com/tls-fingerprinting/`, 2015.

[30] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[31] C. Cawley. What's the difference between Google TV and Android TV? `https://www.androidpolice.com/whats-the-difference-between-google-tv-and-android-tv/`.

[32] B. Charyyev and M. H. Gunes. Voice command fingerprinting with locality sensitive hashes. In *Proceedings of the 2020 Joint Workshop on CPS&IoT Security and Privacy*, CPSIOTSEC'20, page 87–92, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Consumer Reports, Inc. Samsung and Roku Smart TVs Vulnerable to Hacking. `https://www.consumerreports.org/televisions/samsung-roku-smart-tvs-vulnerable-to-hacking-consumer-reports-finds/`, 2018. [Online; accessed 2019-04-22].

[34] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde. Analyzing Android Encrypted Network Traffic to Identify User Actions. *IEEE Transactions on Information Forensics and Security*, 11(1):114–125, 2016.

[35] B. Copos, K. Levitt, M. Bishop, and J. Rowe. Is Anybody Home? Inferring Activity From Smart Home Network Traffic. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 245–251, 2016.

[36] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *2012 IEEE Symposium on Security and Privacy*, pages 332–346, 2012.

[37] D. Eastlake. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, RFC Editor, January 2011. `http://www.rfc-editor.org/rfc/rfc6066.txt`.

[38] eMarketer. US Connected TV Users, by Brand, 2018 & 2022. `https://www.emarketer.com/Chart/US-Connected-TV-Users-by-Brand-2018-2022-of-connected-TV-users/220767`, 2018. [Online; accessed 2019-11-26].

[39] S. Englehardt, J. Han, and A. Narayanan. I never signed up for this! Privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies*, 2018(1):109–126, 2018.

[40] S. Englehardt and A. Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1388–1401, New York, NY, USA, 2016. ACM.

[41] J. Erman, A. Gerber, K. Ramadrishnan, S. Sen, and O. Spatscheck. Over The Top Video: The Gorilla in Cellular Networks. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, pages 127–136. ACM, 2011.

[42] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[43] Federal Trade Commission. VIZIO to Pay $2.2 Million to FTC, State of New Jersey to Settle Charges It Collected Viewing Histories on 11 Million Smart Televisions without Users' Consent. `https://www.ftc.gov/news-events/press-releases/2017/02/vizio-pay-22-million-ftc-state-new-jersey-settle-charges-it`, 2017. [Online; accessed 2019-04-22].

[44] X. Feng, Q. Li, H. Wang, and L. Sun. Acquisitional Rule-based Engine for Discovering Internet-of-Things Devices. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 327–341, Baltimore, MD, Aug. 2018. USENIX Association.

[45] FTC Staff. A Look at What ISPs Know About You: Examining the Privacy Practices of Six Major Internet Service Providers, Oct. 2021.

[46] Future Today Inc. `http://www.stuffwelike.com/`, 2019. [Online; accessed 2019-12-05].

[47] M. Ghiglieri. I Know What You Watched Last Sunday - A New Survey Of Privacy In HbbTV. In *Web 2.0 Security and Privacy Workshop (W2SP) 2014*, W2SP '14, 2014.

[48] M. Ghiglieri and E. Tews. A privacy protection system for HbbTV in Smart TVs. In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pages 357–362, 2014.

[49] M. Ghiglieri, M. Volkamer, and K. Renaud. Exploring Consumers' Attitudes of Smart TV Related Privacy Risks. In T. Tryfonas, editor, *Human Aspects of Information Security, Privacy and Trust*, pages 656–674, Cham, 2017. Springer International Publishing.

[50] M. Ghiglieri and M. Waidner. HbbTV Security and Privacy: Issues and Challenges. *IEEE Security & Privacy*, 14(3):61–67, 2016.

[51] P. Gill, V. Erramilli, A. Chaintreau, B. Krishnamurthy, K. Papagiannaki, and P. Rodriguez. Follow the Money: Understanding Economics of Online Aggregation and Advertising. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 141–148. ACM, 2013.

[52] Google LLC. Android Debug Bridge (adb). `https://developer.android.com/studio/command-line/adb`.

[53] Google LLC. apkanalyzer. `https://developer.android.com/studio/command-line/apkanalyzer`, 2019. [Online; accessed 2019-04-29].

[54] W. Gordon. How to Stop Your Smart TV From Tracking What You Watch. `https://www.nytimes.com/2018/07/23/smarter-living/how-to-stop-your-smart-tv-from-tracking-what-you-watch.html`, 2018. [Online; accessed 2019-05-10].

[55] Gray Television, Inc. `https://gray.tv/`, 2019. [Online; accessed 2019-12-06].

[56] H. Guo and J. Heidemann. IP-Based IoT Device Detection. In *Proceedings of the 2018 Workshop on IoT Security and Privacy*, IoT S&P '18, page 36–42, New York, NY, USA, 2018. Association for Computing Machinery.

[57] J. Hayes and G. Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1187–1203, Austin, TX, Aug. 2016. USENIX Association.

[58] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, page 31–42, New York, NY, USA, 2009. Association for Computing Machinery.

[59] A. Hintz. Fingerprinting websites using traffic analysis. In R. Dingledine and P. Syverson, editors, *Privacy Enhancing Technologies*, pages 171–178, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[60] N. P. Hoang, A. A. Niaki, P. Gill, and M. Polychronakis. Domain name encryption is not enough: privacy leakage via IP-based website fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2021(4):420–440, 2021.

[61] P. Hoffman and P. McManus. DNS Queries over HTTPS (DoH). RFC 8484, RFC Editor, October 2018.

[62] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, RFC Editor, May 2016.

[63] D. Y. Huang, N. Apthorpe, F. Li, G. Acar, and N. Feamster. IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(2), June 2020.

[64] Hub Research LLC. 2021 Connected Home. https://hubresearchllc.com/reports/?category=2021&title=2021-connected-home, 2021. [Online; accessed 2021-10-27].

[65] Hub Research LLC. 2021 Evolution of the TV Set. https://hubresearchllc.com/reports/?category=2021&title=2021-evolution-of-the-tv-set, 2021. [Online; accessed 2021-10-27].

[66] M. Husák, M. Čermák, T. Jirsík, and P. Čeleda. HTTPS Traffic Analysis and Client Identification Using Passive SSL/TLS Fingerprinting. *EURASIP J. Inf. Secur.*, 2016(1), Dec. 2016.

[67] J. Hyland, C. Schneggenburger, N. Lim, J. Ruud, N. Mathews, and M. Wright. What a SHAME: Smart Assistant Voice Command Fingerprinting Utilizing Deep Learning. In *Proceedings of the 20th Workshop on Privacy in the Electronic Society*, WPES '21, page 237–243, New York, NY, USA, 2021. Association for Computing Machinery.

[68] J. Irion. adb_shell. https://github.com/JeffLIrion/adb_shell.

[69] H. Jin, M. Liu, K. Dodhia, Y. Li, G. Srivastava, M. Fredrikson, Y. Agarwal, and J. I. Hong. Why Are They Collecting My Data? Inferring the Purposes of Network Traffic in Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(4), dec 2018.

[70] John Kurkowsi. tldextract. https://github.com/john-kurkowski/tldextract. [Online; accessed 2019-08-23].

[71] Jump PR. Beachfront Releases 2018 CTV Ad Data, Roku Still Leads, Amazon Growing Quickly. https://www.broadcastingcable.com/post-type-the-wire/2018-ctv-ad-data-realeased-by-beachfront, 2018. [Online; accessed 2019-05-10].

[72] G. Kelly, J. Graham, J. Bronfman, and S. Garton. Privacy of Streaming Apps and Devices: Watching TV that Watches Us. San Francisco, CA: Common Sense Media, 2021.

[73] S. Kennedy, H. Li, C. Wang, H. Liu, B. Wang, and W. Sun. I Can Hear Your Alexa: Voice Command Fingerprinting on Smart Home Speakers. In *2019 IEEE Conference on Communications and Network Security (CNS)*, pages 232–240, 2019.

[74] K. Kollnig, A. Shuba, R. Binns, M. V. Kleek, and N. Shadbolt. Are iPhones Really Better for Privacy? Comparative Study of iOS and Android Apps, 2021.

[75] S. Krishnan and F. Monrose. DNS Prefetching and Its Privacy Implications: When Good Things Go Bad. In *Proceedings of the 3rd USENIX Conference on Large-Scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'10, page 10, USA, 2010. USENIX Association.

[76] Kromtech Alliance Corp. Stopad for tv. https://stopad.io/tv, 2019.

[77] F. Le, J. Ortiz, D. Verma, and D. Kandlur. *Policy-Based Identification of IoT Devices' Vendor and Type by DNS Traffic Analysis*, pages 180–201. Springer International Publishing, Cham, 2019.

[78] H. Le, S. Elmalaki, A. Markopoulou, and Z. Shafiq. AutoFR: Automated Filter Rule Generation for Adblocking. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*, Anaheim, CA, Aug. 2023.

[79] Leichtman Research Group, Inc. 39% of Adults Watch Video via a Connected TV Device Daily. `https://www.leichtmanresearch.com/39-of-adults-watch-video-via-a-connected-tv-device-daily/`. [Online; accessed 2021-10-27].

[80] Y. Li, Z. Yang, Y. Guo, and X. Chen. DroidBot: a Lightweight UI-guided Test Input Generator for Android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[81] M. Liberatore and B. N. Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 255–263, New York, NY, USA, 2006. Association for Computing Machinery.

[82] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret. Network Traffic Classifier With Convolutional and Recurrent Neural Networks for Internet of Things. *IEEE Access*, 5:18042–18050, 2017.

[83] L. Lu, E.-C. Chang, and M. C. Chan. Website Fingerprinting and Identification Using Ordered Feature Sequences. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Computer Security – ESORICS 2010*, pages 199–214, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[84] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, page 313–326, New York, NY, USA, 2006. Association for Computing Machinery.

[85] X. Ma, M. Shi, B. An, J. Li, D. X. Luo, J. Zhang, and X. Guan. Context-aware Website Fingerprinting over Encrypted Proxies. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[86] S. Maheshwari. How Smart TVs in Millions of U.S. Homes Track More Than What's On Tonight. `https://www.nytimes.com/2018/07/05/business/media/tv-viewer-tracking.html`, 2018. [Online; accessed 2019-05-10].

[87] N. Malkin, J. Bernd, M. Johnson, and S. Egelman. "What Can't Data Be Used For?" Privacy Expectations about Smart TVs in the US. In *Proceedings of the 3rd European Workshop on Usable Security (EuroUSEC), London, UK*, 2018.

[88] E. C. Malthouse, E. Maslowska, and J. U. Franks. Understanding programmatic TV advertising. *International Journal of Advertising*, 37(5):769–784, 2018.

[89] Manta Media Inc. Htvma Solutions, Inc. `https://www.manta.com/c/mhqfv38/htvma-solutions-inc`, 2019. [Online; accessed 2019-12-05].

[90] J. R. Mayer and J. C. Mitchell. Third-Party Web Tracking: Policy and Technology. In *2012 IEEE Symposium on Security and Privacy*, pages 413–427, May 2012.

[91] McAfee, LLC. Customer URL Ticketing System. `https://www.trustedsource.org/`. [Online; accessed 2019-08-24].

[92] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, April 1976.

[93] D. McGrew, B. Enright, B. Anderson, L. Messenger, A. Weller, and S. Acharya. Mercury. `https://github.com/cisco/mercury`.

[94] L. McInnes, J. Healy, and S. Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11):205, 2017.

[95] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 319–333. IEEE, 2017.

[96] S. Miskovic, G. M. Lee, Y. Liao, and M. Baldi. AppPrint: Automatic Fingerprinting of Mobile Applications in Network Traffic. In J. Mirkovic and Y. Liu, editors, *Passive and Active Measurement*, pages 57–69, Cham, 2015. Springer International Publishing.

[97] H. Mohajeri Moghaddam, G. Acar, B. Burgess, A. Mathur, D. Y. Huang, N. Feamster, E. W. Felten, P. Mittal, and A. Narayanan. ott-tracking. `https://github.com/citp/ott-tracking`, 2019.

[98] H. Mohajeri Moghaddam, G. Acar, B. Burgess, A. Mathur, D. Y. Huang, N. Feamster, E. W. Felten, P. Mittal, and A. Narayanan. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 131–147, New York, NY, USA, 2019. Association for Computing Machinery.

[99] A. W. Moore and K. Papagiannaki. Toward the accurate identification of network applications. In C. Dovrolis, editor, *Passive and Active Network Measurement*, pages 41–54, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[100] Mozilla Foundation. Public Suffix List. `https://publicsuffix.org/`. [Online; accessed 2019-08-23].

[101] T. OConnor, R. Mohamed, M. Miettinen, W. Enck, B. Reaves, and A.-R. Sadeghi. HomeSnitch: Behavior Transparency and Control for Smart Home IoT Devices. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, page 128–138, New York, NY, USA, 2019. Association for Computing Machinery.

[102] S. E. Oh, N. Mathews, M. S. Rahman, M. Wright, and N. Hopper. GANDaLF: GAN for Data-Limited Fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2021(2):305–322, 2021.

[103] S. E. Oh, S. Sunkam, and N. Hopper. p1-FP: Extraction, classification, and prediction of website fingerprints with deep learning. *Proceedings on Privacy Enhancing Technologies*, 2019(3), 2019.

[104] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle. Website fingerprinting at internet scale. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2016.

[105] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, WPES '11, page 103–114, New York, NY, USA, 2011. Association for Computing Machinery.

[106] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[107] M. Perathoner. suffix-tree. `https://github.com/cceh/suffix-tree`.

[108] R. Perdisci, T. Papastergiou, O. Alrawi, and M. Antonakakis. IoTFinder: Efficient Large-Scale Identification of IoT Devices via Passive DNS Traffic Analysis. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 474–489, 2020.

[109] Pi-hole LLC. Blocking Mode. `https://docs.pi-hole.net/ftldns/blockingmode`, 2018.

[110] Qualys SSL Labs. HTTP Client Fingerprinting Using SSL Handshake Analysis. `https://www.ssllabs.com/projects/client-fingerprinting/`.

[111] Raspberry Pi Foundation. Setting up a Raspberry Pi as an access point in a standalone network (NAT). `https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md`, 2019. [Online; accessed 2019-03-04].

[112] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS Usage in Android Apps. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 350–362, New York, NY, USA, 2017. Association for Computing Machinery.

[113] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. *NDSS*, 2018.

[114] A. Reed and M. Kranch. Identifying HTTPS-Protected Netflix Videos in Real-Time. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 361–368, New York, NY, USA, 2017. Association for Computing Machinery.

[115] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *Proceedings of the Internet Measurement Conference*, IMC '19, page 267–279, New York, NY, USA, 2019. Association for Computing Machinery.

[116] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug Fixes, Improvements, ... and Privacy Leaks: A Longitudinal Study of PII Leaks Across Android App Versions. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2018.

[117] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 361–374, New York, NY, USA, 2016. Association for Computing Machinery.

[118] I. Reyes, P. Wijesekera, A. Razaghpanah, J. Reardon, N. Vallina-Rodriguez, S. Egelman, C. Kreibich, et al. "Is Our Children's Apps Learning?" Automatically Detecting COPPA Violations. In *Workshop on Technology and Consumer Protection (ConPro 2017), in conjunction with the 38th IEEE Symposium on Security and Privacy (IEEE S&P 2017)*, 2017.

[119] S. Rezaei, B. Kroencke, and X. Liu. Large-Scale Mobile App Identification Using Deep Learning. *IEEE Access*, 8:348–362, 2020.

[120] S. Rezaei and X. Liu. Deep Learning for Encrypted Traffic Classification: An Overview. *IEEE Communications Magazine*, 57(5):76–81, 2019.

[121] I. Ristić. HTTP client fingerprinting using SSL handshake analysis. `https://blog.ivanristic.com/2009/06/http-client-fingerprinting-using-ssl-handshake-analysis.html`.

[122] L. Rokach and O. Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.

[123] Roku, Inc. External Control Protocol (ECP). `https://developer.roku.com/docs/developer-program/debugging/external-control-api.md`.

[124] Roku, Inc. Roku Channel Store. `https://channelstore.roku.com/`.

[125] Roku, Inc. Roku Distribution Agreement. https://docs.roku.com/published/developerdistribution/en/us.

[126] Roku, Inc. Roku Developer Documentation: Development Environment Overview. https://sdkdocs.roku.com/display/sdkdoc/Development+Environment+Overview, 2019. [Online; accessed 2019-04-22].

[127] Roku, Inc. Roku Developer Documentation: Roku Advertising Framework. https://sdkdocs.roku.com/display/sdkdoc/Roku+Advertising+Framework, 2019. [Online; accessed 2019-04-22].

[128] Roku, Inc. Roku Developer Documentation: Security Overview. https://sdkdocs.roku.com/display/sdkdoc/Security+Overview, 2019. [Online; accessed 2019-04-22].

[129] Roku, Inc. The Roku Advantage. https://advertising.roku.com/advertising-solutions, 2019. [Online; accessed 2019-05-10].

[130] S. J. Saidi, A. M. Mandalari, R. Kolcun, H. Haddadi, D. J. Dubois, D. Choffnes, G. Smaragdakis, and A. Feldmann. A Haystack Full of Needles: Scalable Detection of IoT Devices in the Wild. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 87–100, New York, NY, USA, 2020. Association for Computing Machinery.

[131] B. Saltaformaggio, H. Choi, K. Johnson, Y. Kwon, Q. Zhang, X. Zhang, D. Xu, and J. Qian. Eavesdropping on Fine-Grained User Activities Within Smartphone Apps Over Encrypted Network Traffic. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, Aug. 2016. USENIX Association.

[132] Sapna Maheshwari. How Smart TVs in Millions of U.S. Homes Track More Than What's On Tonight. https://www.nytimes.com/2018/07/05/business/media/tv-viewer-tracking.html, 2018. [Online; accessed 2019-05-11].

[133] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1357–1374, Vancouver, BC, Aug. 2017. USENIX Association.

[134] A. Sherman. How Roku used the Netflix playbook to beat bigger players and rule streaming video. https://www.cnbc.com/2021/06/18/how-roku-dominated-streaming-anthony-woods-new-content-obsession.html, 2021.

[135] A. Shuba, A. Le, E. Alimpertis, M. Gjoka, and A. Markopoulou. AntMonitor: A System for On-Device Mobile Network Monitoring and its Applications. *arXiv preprint arXiv:1611.04268*, 2016.

[136] A. Shuba, A. Markopoulou, and Z. Shafiq. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. *Proceedings on Privacy Enhancing Technologies*, 2018(4):125–140, 2018.

[137] S. Siby, M. Juarez, C. Diaz, N. Vallina-Rodriguez, and C. Troncoso. Encrypted DNS ⇒ Privacy? A Traffic Analysis Perspective. In *Network & Distributed System Security Symposium (NDSS)*. Internet Society, 2020.

[138] P. Sirinam, M. Imani, M. Juarez, and M. Wright. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1928–1943, New York, NY, USA, 2018. Association for Computing Machinery.

[139] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright. Triplet Fingerprinting: More Practical and Portable Website Fingerprinting with N-Shot Learning. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1131–1148, New York, NY, USA, 2019. Association for Computing Machinery.

[140] G. Sloane. AMAZON IS NOW TAKING A 30 PERCENT CUT OF AD SALES FROM FIRE TV. `https://adage.com/article/design/amazon-taking-30-percent-ad-sales-fire-tv/315678`, 2018. [Online; accessed 2019-05-10].

[141] J.-P. Smith, P. Mittal, and A. Perrig. Website Fingerprinting in the Age of QUIC. *Proceedings on Privacy Enhancing Technologies*, 2021(2):48–69, 2021.

[142] Statista. Connected TV advertising spending in the United States from 2019 to 2026. `https://www.statista.com/statistics/1048897/connected-tv-ad-spend-usa/`.

[143] StuffWeLike. `http://www.stuffwelike.com/`, 2019. [Online; accessed 2019-12-05].

[144] Q. Sun, D. Simon, Y.-M. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted Web browsing traffic. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 19–30, 2002.

[145] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 439–454, 2016.

[146] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. Robust smartphone app identification via encrypted network traffic analysis. *IEEE Transactions on Information Forensics and Security*, 13(1):63–78, 2018.

[147] telekrmor. Round 3: What Really Happens On Your Network? `https://pi-hole.net/2017/07/06/round-3-what-really-happens-on-your-network/`, 2017. [Online; accessed 2019-05-11].

[148] The SciPy community. scipy.cluster.hierarchy.fcluster. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.fcluster.html`.

[149] The SciPy community. scipy.cluster.hierarchy.linkage. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html`.

[150] The SciPy community. scipy.spatial.distance.pdist. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html.

[151] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky. Packet-Level Signatures for Smart Home Devices. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, February 2020.

[152] UCI Networking Group. FingerprinTV. https://github.com/UCI-Networking-Group/fingerprintv.

[153] UCI Networking Group. The TV is Smart and Full of Trackers: Project Page. http://athinagroup.eng.uci.edu/projects/smarttv/, 2019.

[154] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, September 1995.

[155] T. van Ede, R. Bortolameotti, A. Continella, J. Ren, D. J. Dubois, M. Lindorfer, D. Choffnes, M. van Steen, and A. Peter. FlowPrint: Semi-supervised mobile-app fingerprinting on encrypted network traffic. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2020.

[156] J. Varmarken, J. A. Aaraj, R. Trimananda, and A. Markopoulou. FingerprinTV: Fingerprinting Smart TV Apps. *Proceedings on Privacy Enhancing Technologies*, 2022(3), 2022.

[157] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq. Firetastic. https://github.com/UCI-Networking-Group/firetastic, 2020.

[158] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq. Rokustic. https://github.com/UCI-Networking-Group/rokustic, 2020.

[159] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq. The TV is Smart and Full of Trackers: Measuring Smart TV Advertising and Tracking. *Proceedings on Privacy Enhancing Technologies*, 2020(2), 2020.

[160] WaLLy3K. The Big Blocklist Collection. https://firebog.net, 2019. [Online; accessed 2019-04-29].

[161] C. Wang, S. Kennedy, H. Li, K. Hudson, G. Atluri, X. Wei, W. Sun, and B. Wang. Fingerprinting Encrypted Voice Traffic on Smart Speakers with Deep Learning. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 254–265, New York, NY, USA, 2020. Association for Computing Machinery.

[162] K. Wang, J. Zhang, G. Bai, R. Ko, and J. S. Dong. It's Not Just the Site, It's the Contents: Intra-Domain Fingerprinting Social Media Websites Through CDN Bursts. In *Proceedings of the Web Conference 2021*, WWW '21, page 2142–2153, New York, NY, USA, 2021. Association for Computing Machinery.

[163] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 143–157, San Diego, CA, Aug. 2014. USENIX Association.

[164] S. Zhao, S. Li, J. Ramos, Z. Luo, Z. Jiang, A. K. Dey, and G. Pan. User profiling from their use of smartphone applications: A survey. *Pervasive and Mobile Computing*, 59:101052, 2019.

# Appendix A

# Appendix for Chapter 3

This appendix complements Chapter 3. Appendix A.1 provides further details on how domains are categorized as either first party, third party, or platform-specific party, w.r.t. the app that contacts it. Appendix A.2 offers more details on how Rokustic and Firetastic automate app exploration. Appendix A.3 discusses the limitations of Rokustic and Firetastic, including when the automation does not lead to video playback and how the automated exploration compares to manual testing of apps that require login. Ultimately, this discussion leads to directions for how to improve Rokustic's and Firetastic's automatic app exploration. Appendix A.4 evaluates the success rate of Firetastic's TLS interception. Appendix A.5 provides additional analysis of datasets that was omitted from the main text.

## A.1 Labeling Datasets Continued

To complement Section 3.2, this section provides the full details for how we label each endpoint as either first party, third party, or platform-specific party, w.r.t. the app that contacts it:

1. We first tokenize app identifiers and the eSLD of the contacted FQDN (we obtain the eSLD using Mozilla's Public Suffix List [100, 70]). For Fire TV, we tokenize the package names and developer names. For Roku, we rely on app and developer names since its apps do not have package names. For app/package tokens, we ignore common and platform-specific strings like "com", "firetv", "roku", etc., while retaining all tokens from the developer names. We then match the resulting identifiers with the tokenized eSLD.

2. If the tokens match, we label the destination as *first party*. We note that since we keep all developer tokens, we will map "roku" related eSLDs as first party if the app was developed officially by Roku.

3. Otherwise, we label a destination as *platform-specific party* if it originated from platform activity rather than app activity. For Fire TV, we rely on AntMonitor's [135] ability to label each connection with the responsible process. For Roku, we simply check if the eSLD contains "roku".

4. Otherwise, if the destination is contacted by at least two different apps from different developers, we label it as *third party*.

5. Finally, if the destination does not fall into any of the other categories, we resort to labeling it as *other*, which thus captures domains that are only contacted by a single app and are not identified as a first party nor platform-specific party.

We acknowledge that the variations in labeling methodology for platform-specific parties for Roku and Fire TV may impact comparability. However, we believe that our choice provides the more accurate platform-specific labeling for each testbed platform.

## A.2 App Exploration Implementation Details

In this section, we provide details on the implementation of our automatic app exploration tools, Rokustic for Roku (Appendix A.2.1) and Firetastic for Fire TV (Appendix A.2.2), in addition to what is described in Sections 3.4.1 and 3.4.2, respectively.

### A.2.1 Roku Automation

To scale testing of apps, we implement a software system, Rokustic, that automatically installs and exercises Roku apps on a Roku Express 3900X (Roku for short).

**Setup.** We run Rokustic on a Raspberry Pi 3 Model B+ set up to host a standalone network as per the instructions given in [111]. The Pi's wireless interface (`wlan0`) is configured as a wireless access point with DHCP server and NAT, and the Roku is connected to this local wireless network. The Pi's wired interface (`eth0`) connects the Pi and thus, in turn, the Roku to the WAN. This setup enables us to collect all traffic going in and out of the Roku by running `tcpdump` on the Raspberry Pi's wireless interface. We do not attempt to decrypt TLS traffic as we cannot install our own self-signed certificates on the Roku.

Rokustic utilizes the Roku External Control (ECP) API [123] to control the Roku. The ECP is a REST-like API exposed by the Roku to other devices on the local network. The ECP includes a set of REST-endpoints that provides the ability to press keys on the Roku remote, query the Roku for various information such as the set of installed apps, programmatically browse the Roku Channel Store (RCS), etc.

**App Installation.** Given a set of apps to exercise, Rokustic installs each app by invoking the ECP endpoint that opens up the on-device version of the RCS page for the app, and then sends a virtual key press to click the "Add Channel" button that starts download and

installation of the app. Rokustic then waits for five seconds, and then queries the Roku for the set of installed apps to check if the installation has completed, and if so continues to install the next app, otherwise the wait-and-check is repeated (until a fixed threshold).

**App Exploration.** From manual inspection of a few apps (e.g.,YouTube and Pluto TV), we find that playable content is often presented in a grid, where each cell is a different video or live TV channel. Generally, the user interface defaults to highlighting one of these cells (e.g., the first recommended video). Pressing "Select" on the Roku remote immediately after the app has launched will therefore result in playback of some content. From this insight, we devised a simple algorithm (see Listing A.1) that attempts to cause playback of three different videos for each installed Roku app.

```
for app in roku.installedApps:
        startPacketCapture(app.id);
        # Play default video
        launch(app); sleepSeconds(20); press("SELECT"); sleepMinutes(5);
        # Play some other video by selecting a different cell in the grid
        relaunch(app); press("DOWN"); press("DOWN"); press("RIGHT"); press("RIGHT");
        press("SELECT"); sleepMinutes(5);
        # Play a 3rd video
        relaunch(app); press("DOWN"); press("DOWN"); press("SELECT"); sleepMinutes(5);
        # Quit the Roku app
        press("HOME");
        stopPacketCapture(app.id);
```

Listing A.1: Algorithm for exercising Roku apps.

For each Roku app, the algorithm first starts a packet capture so as to produce a `.pcap` file for each Roku app, thereby essentially labeling traffic with the app that caused it: since

Roku apps cannot execute in the background (see Section 3.4.1), all traffic captured during execution of a single app will belong to that app and the Roku system. The target app is launched, and the algorithm pauses, waiting for the app to load. Next, a virtual "Select" key press is sent to attempt to start video playback, and the algorithm subsequently pauses to let the content play. The app is then relaunched by returning to the Roku's home screen and then launching the app again. The purpose of this is to safely return to the app's home screen such that different content can be selected for playback. This is repeated two times, with slight variations in the sequence of navigational key presses, such that each app will (presumably) end up playing three different videos, making the total exploration time approximately 16 minutes per app. We note that the `relaunch` procedure internally performs short sleeps to let the app quit and launch again, and that we have omitted a one second sleep after each navigational key press in the pseudo code in Listing A.1.

Although the Roku remote has a "Back" button, which behaves similarly to the back button in Android, we purposefully avoid using it as a means to return to the app's home screen: if the video selected for playback is shorter than the sleep duration, the app will return to its home screen automatically, and pressing "Back" will therefore quit the app and return to the Roku's home screen. This would pollute our data as the subsequent navigational key presses would cause a different app to be highlighted and then launched by the next "Select" key press.

### A.2.2 Fire TV Automation

Since Fire TV is based on Android, we can use existing Android tools to capture network traffic. Although there are various methods for capturing traffic on Android on the device itself (e.g., `androidtcpdump` [9]), most of them require a rooted device. While it is possible to root a Fire TV, it may make applications behave differently if they detect root. Thus, to

collect measurements that are representative of an average user, we use a VPN-based traffic interception method that does not require rooting the device [135, 7]. We discard incoming traffic because video content results in huge pcap files, which due to a technical limitation of ADB (very slow transfer speeds for large files) slows down the automated experiments significantly. The outgoing traffic is sufficient for our domain and PII analysis.

To automatically explore each Fire TV application, we utilize Droidbot [80], as it treats each app as a tree of possible paths to explore instead of randomly generating events, which results in higher test coverage of the application. Furthermore, we deduce that developers would minimize the necessary clicks in order to reach the core sections of their applications, especially for playing video content. Thus, we configure DroidBot to utilize its breadth first search algorithm to explore each application. The intuition is that this should cover more distinct UI paths of the app, thus increasing the chance of content playback (in contrast, the Depth First Search algorithm may cause the automation to deep end into a path that we do not care about, such as a settings menu). With some trial and error, we selected the input command interval as three seconds which leaves enough time for applications to handle the command and load the next view during app exploration.

We summarize Firetastic's automation algorithm in Listing A.2. For each app, Firetastic first starts the local VPN to capture (and decrypt) traffic. Next, it invokes DroidBot, which in turn launches the app and begins exploring it. When the 15-minute exploration completes, Firetastic stops the local VPN and extracts the `.pcapng` files that were generated during testing.

```
device = "10.0.1.xx:5555"; pcapng_dir = "/path/to/store/pcapng"; apk_dir = "/path/to/apk/batch"
for app in apk_dir:
        start_antmonitor(device) # Start AntMonitor on Fire TV
        ensure_antmonitor_connected(device) # Ensure VPN connection up
        # Run DroidBot command
        params = { duration: 15min, policy: "bfs_naive", install_timeout: 5min, interval: 3sec }
        run_droidbot(app, params, device)
        stop_antmonitor_vpn(device) # Stop AntMonitor on Fire TV
        extract_pcapng_files(app, pcapng_dir, device) # Extract the pcap files
        remove_pcapng_files(device) # Clean up before testing next app
```

Listing A.2: Algorithm for exercising Fire TV apps.

## A.3 Limitations of App Exploration

Our automated app exploration has, admittedly, limitations. For example, it can miss video playback for some apps, and cannot fully explore apps that require login. In this section, we evaluate our automated exploration vs. a more realistic manual exploration by a real user, for 20 apps. We report the cases where our automatic exploration succeeds and fails, and we provide insights into the reasons why, and ideas for future improvements.

First, we evaluate our tools' ability to perform playback, because it affects our ability to capture traffic related to ads delivered at the start of or during content (video/audio) playback. We evaluate Firetastic's and Rokustic's content playback rates in Appendix A.3.1 by performing a case study of 15 apps. We find that the two tools perform similarly to the the state-of-the-art [98], and we identify how we can further improve the tools to increase the chance of incurring content playback.

In Appendix A.3.2, we evaluate the automated approach's ability to discover an app's domain space by comparing the eSLDs and ATS domains discovered by the tools to the eSLDs and ATS domains discovered during manual interaction with the same 15 apps. We find that both tools generally do well at mapping a large fraction of the number of domains discovered during manual interaction.

Finally, a common limitation of app exploration in general (not just for smart TV apps) is that it is difficult to explore apps that require user login. In Appendix A.3.3 we compare the eSLDs and ATS domains discovered by Firetastic and Rokustic for 5 popular streaming apps (while logged out) to the eSLDs and ATS domains discovered during manual interaction (while logged in). As expected, we find that the automation misses parts of the domain space of apps that serve third-party ads as part of content that is only accessible after logging in. Interestingly, we also observe cases where the automation discovers more (ATS) domains than the manual experiments.

### A.3.1 Video Playback

**Setup.** To evaluate Firetastic's and Rokustic's ability to incur playback of an app's main content (video/audio) for the set of apps that do not require login to access (parts of) their content, we run the full (∼16 minutes) automation for 15 apps on each platform, while observing for content playback. We pick the 15 apps according to their popularity (as defined in Section 3.4): (1) the top-10 most popular apps to capture the most influential apps; and (2) 5 additional randomly sampled apps, spread evenly across the popularity spectrum, to represent the dataset as a whole.

**Results.** We present the content playback results in Table A.1. Firetastic manages to play content for 67% of the 15 apps (60% for the top-10 apps, and 80% for the random apps). A common characteristic of the 33% unsuccessful apps is that the majority of their content

is locked, and that free content is deferred to the least accessible sections of the UI (e.g., the bottom row of a grid layout). This decreases the chance that Firetastic will discover a playable video during the 16 min experiment, especially since attempts to play locked content often redirects to login/activation screens where additional time is lost. We can improve on this in future work by mixing BFS and DFS exploration: the automation can explore each level up to a certain threshold before drilling down into the next nested view.

Rokustic manages to play content for 53% of the 15 apps (50% for the top-10 apps, and 60% for the random apps). Unsuccessful attempts are primarily due to nested menus, where additional "Select" key press(es) are necessary to start playback. Through manual interaction with a few apps prior to executing our top-1000 apps measurement, we observe that if an app starts playback on the first "Select" key press, additional key presses may result in pausing and/or exiting the video, or skipping past an ad. Therefore, we opt for a conservative approach that, when successful, will collect as much ad/video traffic as possible. In future work, we can further improve content playback by repeatedly sending "Select" key presses (with short pauses in between) until the network throughput for the Roku stays above a certain threshold for a short duration (e.g., the bitrate of 720p video for 5 seconds). This dynamic strategy can handle more nested menus and thus be resilient to future changes to an app's UI.

**Takeaways.** In summary, both tools work well when an app adheres to good UI design principles, such as reducing the number of user actions required to reach the main content. The content playback success rates are on par with state-of-the-art concurrent work [98], with Firetastic slightly ahead, possibly due to its dynamic approach to UI exploration (as opposed to the static, heuristic-based approach used in [98]), and with Rokustic slightly behind. While Firetastic leverages existing advanced Android tools (Droidbot), similar tools do not currently exist for Roku, thus we had to build them from scratch, and it is therefore natural that Rokustic falls slightly behind its Fire TV counterpart.

| | | App Name | Play-back (auto) | Distinct eSLDs | | | Distinct ATS Domains | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Auto (A) | Manual (M) | $\frac{A}{M}$ | A | M | $\frac{A}{M}$ |
| **Roku** | Top-10 | YouTube | ✓ | 8 | 15 | 53% | 5 | 15 | 33% |
| | | Sling TV | ✗ | 10 | 21 | 48% | 6 | 21 | 29% |
| | | The Roku Channel | ✓ | 16 | 13 | 123% | 7 | 5 | 140% |
| | | Crackle | ✓ | 9 | 34 | 26% | 8 | 42 | 19% |
| | | JW Broadcasting | ✗ | 3 | 4 | 75% | 2 | 2 | 100% |
| | | PBS KIDS | ✓ | 4 | 7 | 57% | 4 | 6 | 67% |
| | | ESPN | ✗ | 6 | 16 | 38% | 4 | 16 | 25% |
| | | Tubi - Free Movies & TV | ✗ | 3 | 19 | 15.79% | 5 | 34 | 15% |
| | | DisneyNOW | ✗ | 6 | 9 | 67% | 4 | 5 | 80% |
| | | Pluto TV - It's Free TV | ✓ | 24 | 9 | 267% | 36 | 7 | 514% |
| | Random | Roku Newscaster | ✗ | 4 | 5 | 80% | 3 | 2 | 150% |
| | | ChuChu TV | ✓ | 10 | 10 | 100% | 9 | 17 | 53% |
| | | Elvis | ✓ | 20 | 38 | 53% | 14 | 54 | 26% |
| | | Mondo | ✗ | 4 | 5 | 80% | 2 | 2 | 100% |
| | | tik tok | ✓ | 1 | 3 | 33% | 2 | 3 | 67% |
| | **Total** | | 8/15 (53%) | 74 | 113 | 65% | 64 | 122 | 52% |
| **Fire TV** | Top-10 | Pluto TV - It's Free TV | ✓ | 15 | 30 | 50% | 13 | 35 | 37% |
| | | ABC | ✓ | 14 | 13 | 108% | 5 | 6 | 83% |
| | | Fox Now | ✓ | 22 | 24 | 92% | 18 | 18 | 100% |
| | | AMC | ✗ | 18 | 28 | 64% | 8 | 19 | 42% |
| | | Fox Sports GO | ✗ | 12 | 19 | 63.16% | 7 | 17 | 41% |
| | | Kids for Youtube | ✓ | 16 | 19 | 84% | 7 | 5 | 140% |
| | | PBS Kids | ✓ | 12 | 15 | 80% | 7 | 9 | 78% |
| | | CNN Go | ✓ | 31 | 34 | 91% | 41 | 34 | 121% |
| | | Sundance TV | ✗ | 13 | 23 | 57% | 5 | 11 | 45% |
| | | MTV | ✗ | 56 | 38 | 147% | 38 | 54 | 70% |
| | Random | Vimeo | ✓ | 12 | 11 | 109% | 5 | 3 | 167% |
| | | Dog TV Online | ✓ | 10 | 14 | 71% | 2 | 5 | 40% |
| | | WCSC Live 5 News | ✗ | 13 | 12 | 108% | 5 | 5 | 100% |
| | | WFXG FOX 54 | ✓ | 18 | 18 | 100% | 8 | 7 | 114% |
| | | 13abc WTVG Toledo, OH | ✓ | 12 | 11 | 109% | 5 | 2 | 250% |
| | **Total** | | 10/15 (67%) | 125 | 117 | 107% | 115 | 138 | 83% |

Table A.1: Content playback success for Rokustic and Firetastic, and a comparison of the number of domains discovered by Rokustic and Firetastic to the number of domains discovered during manual interaction with the same app, for 15 apps that do *not* require login. For each app, we perform approximately 16 minutes of automated and 16 minutes of manual interaction.

### A.3.2 Automation vs. Manual Testing

**Setup.** Next, we evaluate Firetastic's and Rokustic's ability to successfully map an app's domain space. We manually interact with the 15 apps from Appendix A.3.1, and compare the network behavior observed during automated testing to the network behavior observed during manual testing. For a fair comparison, we interact with each app for approximately the same duration as in the automated experiments (∼16 minutes). For consistency, we follow a protocol in which we attempt to play 7 different videos for approximately 2 minutes each, leaving a few minutes to navigate between videos. We compare the network behavior in terms of the number of eSLDs and ATS domains (as defined by the union of the blocklists from Section 3.5) contacted by each app. The results are presented in Table A.1.

**Results.** Firetastic is successful in mapping the domain space for 10 out of 15 apps (67%), uncovering 0.8 times (or more) the number of eSLDs, and 0.7 times (or more) the number of ATS domains discovered in the manual experiments. In fact, Firetastic even discovers *more* eSLDs and ATS domains than the manual experiment for 6 (40%) and 7 (47%), respectively, of the 15 apps. Rokustic is less successful, but still manages to uncover 0.67 times (or more) the number of eSLDs and ATS domains in the manual experiment for 7 (47%) and 8 (53%), respectively, of the 15 apps. Moreover, Rokustic even discovers 2.67 times as many eSLDs as the manual experiment for one of the apps (Pluto TV).

**Intuition.** The two tools have very different approaches to app exploration: Firetastic seeks to explore as much app functionality as possible, but is likely to exit content playback early, whereas Rokustic seeks to mimic a real user that sits through 3 videos of 5 minutes. Each approach has its own merit: Firetastic is good at discovering many ATS domains for apps that present ads *before* content playback begins, whereas Rokustic is the more successful tool when it comes to discovering ATS domains for apps that defer ad delivery to later in the video/audio stream, as was the case for Pluto TV. Finally, we note that even in the worst

case, i.e., when Firetastic and Rokustic do not manage to incur content playback, they still uncover several ATS domains.

**Takeaways.** This case study, and the fact that Firetastic and Rokustic uncovered approximately twice as many domains as the state-of-the-art [98] for the top-1000 apps measurement described in Section 3.4, show that our tools already provide sufficient means to automatically estimate a lower bound on the ATS domains of the two platforms. This lower bound should improve if the changes suggested earlier are implemented.

### A.3.3 Apps that Require Login

**Setup.** To understand how well the automation manages to map the domain spaces of apps that require login, we pick 5 of the top subscription-based streaming apps and run the automation *without* logging in, and also manually interact with the same apps *while logged in* (following the same protocol as in Appendix A.3.2). We compare the network behavior using the same metrics as in Appendix A.3.2. The results are presented in Table A.2.

**Results.** Firetastic actually discovers *more* eSLDs than the manual experiments for 3 out of the 5 apps. We also observe that the STARZ app contacts *more* ATS domains when automatically tested than in the manual experiment. These findings are interesting as they indicate that an app's domain space and ATS-related activity possibly changes after the user logs in and is not necessarily tied to video playback. An ideal experiment would thus need to thoroughly exercise the app in both states. The results for Rokustic are more in line with what is to be expected: Rokustic discovers fewer eSLDs and ATS domains than the manual experiments (55% and 53%, respectively, on average). Finally, for both platforms, we observe that the number of ATS domains contacted by Hulu increases significantly for the manual experiments. This is to be expected as Hulu is the only of the 5 apps that deliver third-party ads during content playback.

| | App Name | Distinct eSLDs | | | Distinct ATS Domains | | |
|---|---|---|---|---|---|---|---|
| | | Auto (A) | Manual (M) | $\frac{A}{M}$ | A | M | $\frac{A}{M}$ |
| **Roku** | HBO NOW | 3 | 7 | 43% | 2 | 5 | 40% |
| | Hulu | 6 | 18 | 33% | 3 | 21 | 14% |
| | Netflix | 3 | 4 | 75% | 3 | 3 | 100% |
| | SHOWTIME | 4 | 8 | 50% | 3 | 6 | 50% |
| | STARZ | 5 | 7 | 71% | 3 | 5 | 60% |
| | **Total** | 14 | 30 | 47% | 6 | 26 | 23% |
| **Fire TV** | HBO NOW | 7 | 9 | 78% | 2 | 2 | 100% |
| | Hulu | 9 | 19 | 47% | 4 | 17 | 24% |
| | Netflix | 11 | 9 | 122% | 4 | 3 | 133% |
| | SHOWTIME | 10 | 8 | 125% | 4 | 3 | 133% |
| | STARZ | 18 | 14 | 129% | 12 | 7 | 171% |
| | **Total** | 27 | 42 | 64% | 15 | 27 | 56% |

Table A.2: Comparison of the number of domains discovered by Rokustic and Firetastic to the number of domains discovered during manual interaction with the same five apps that *require* login. The automation was performed while logged out, and the manual interaction was performed while logged in. For each app, we perform approximately 16 minutes of automated and 16 minutes of manual interaction.

**Takeaways.** As expected, Rokustic can only map parts of the (ATS) domain spaces of apps that require login. For Firetastic, we observe that some apps contact *more* ATS domains while logged out, and an ideal experiment would thus need to explore the apps in both states.

## A.4 Fire TV TLS Interception

Firetastic attempts to decrypt TLS traffic to facilitate detection of PII exposures in encrypted traffic. However, the decryption may fail: (i) for apps that attempt to mitigate TLS interception, for example through use of certificate pinning, or (ii) if the cipher suites used in the TLS connection are not supported by the TLS decryption library used in AntMonitor. To approximate the impact that such decryption failures may have on the PII exposure results, we evaluate the failure rate of Firetastic's TLS interception across all apps in the

Fire TV testbed dataset from Section 3.4.

**Methodology.** For each app in the Fire TV testbed dataset, we first identify the set of TCP connections, $t$, initiated by this app and labeled as TLS by tshark. Next, we identify the subset $h$ of TCP connections in $t$ that also contained at least one packet labeled by tshark as HTTP: these are the connections that are successfully decrypted. Finally, we compute the decryption failure rate of each app as $\frac{|t|-|h|}{|t|}$. We note that our methodology conservatively computes an upper bound compared to the actual failure rate, as any non-HTTP over TLS (e.g., proprietary binary protocols) will be counted as decryption failures.

We note that in $t$, we only include TLS connections, where the TLS handshake concluded successfully. We assume that an app will retry the connection if it rejects AntMonitor's certificate. AntMonitor stops intercepting an app's connections if it detects that the app rejects its certificate, thus the second TLS handshake should complete successfully. This restriction on $t$ therefore also prevents double counting (i.e., the original failed connection does not contribute to the total number of TLS connections initiated by the app). Although we only recorded upstream data for Fire TV, we use the presence of an upstream TLS Application

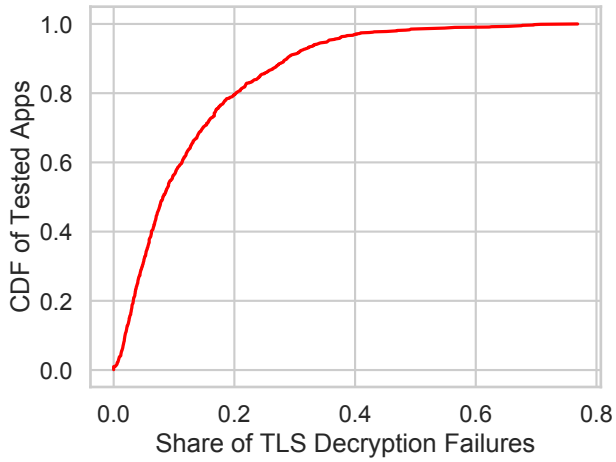



Figure A.1: Empirical CDF of TLS decryption failures per app. The decryption generally worked well: decryption fails for 1 out of 10 (or fewer) TLS connections for 55% of all apps; 1 out of 5 (or fewer) TLS connections for 80% of all apps.

Data packet as a proxy for inferring that the TLS handshake concluded successfully.

**Results.** In Figure A.1, we show the empirical CDF for the TLS decryption failure rates for all apps in the Fire TV testbed dataset. We note that the decryption generally works well. For example, decryption fails for 1 out of 10 (or fewer) TLS connections for 55% of all apps, and 1 out of 5 (or fewer) TLS connections for 80% of all apps. Since TLS decryption was generally successful, this validates the PII exposure results.

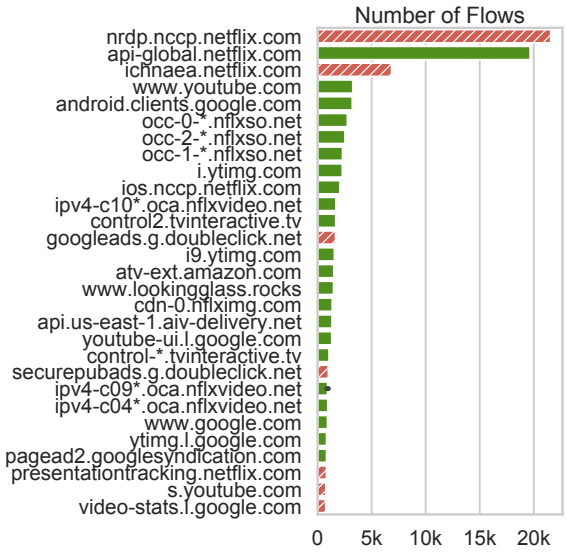
## A.5 Additional Analysis of Datasets

### A.5.1 In the Wild Dataset Continued

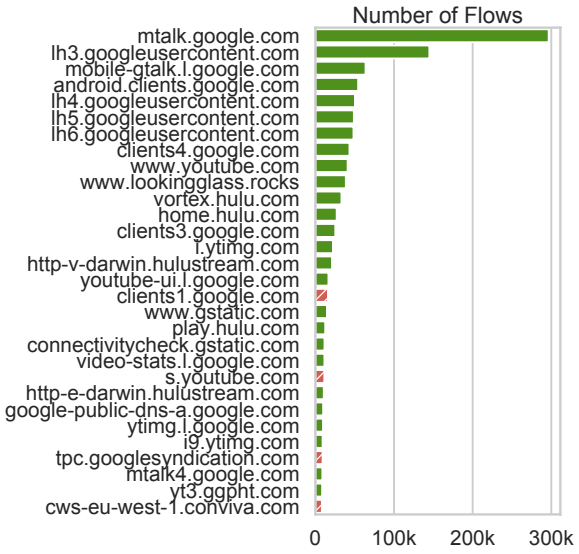
Figure A.2 provides data for the platforms that were omitted from Figure 3.1 in Section 3.3

### A.5.2 Key Players Continued

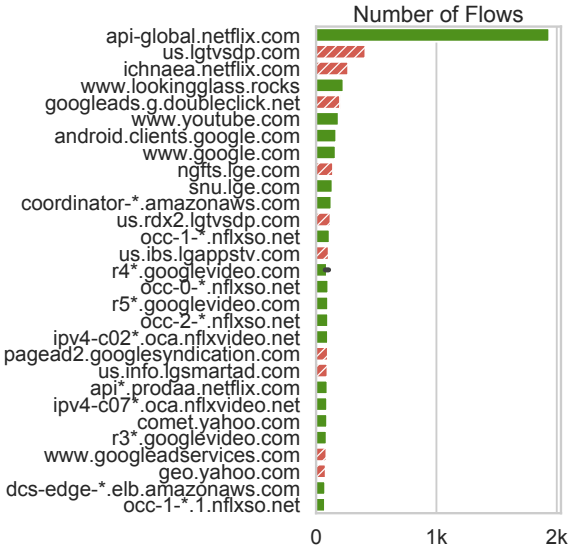
Figure A.3 provides further insight into the "Key Players" discussion in Section 3.4.3 by accumulating flows to subdomains of the top-30 eSLDs of each platform. Each eSLD is in turn mapped to its parent organization. Finally, all flows to domains under that parent organization are separated based on whether they are labeled as advertising and tracking or not. A large amount of the flows to the two platform operators are labeled as advertising and tracking.




(a) Vizio




(b) Chromecast




(c) LG

Figure A.2: Continuation of Figure 3.1 from Section 3.3: Top-30 fully qualified domain names in terms of number of flows per device for the remaining devices in the in the wild dataset. Domains identified as ATSes are highlighted with red, dashed bars.

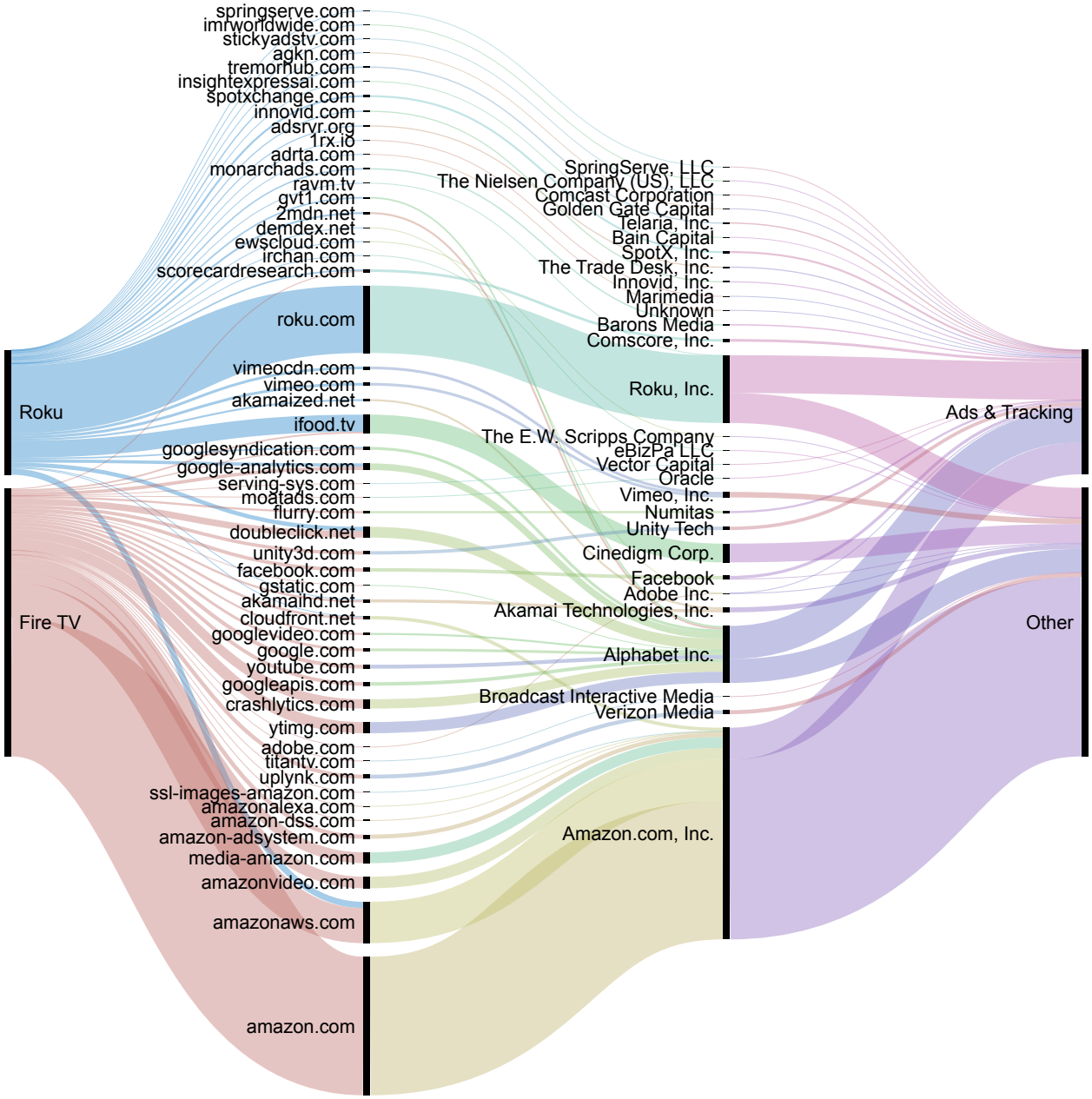



Figure A.3: Left to right: (1) mapping of flows to subdomains of the top-30 eSLDs (see Figures 3.3c and 3.3d) of each testbed platform; (2) mapping from eSLD to its parent organization; (3) separation of the flows to the organization by advertising and tracking flows and other flows. An edge's width represents the number of flows.

# Appendix B

# Appendix for Chapter 4

This appendix complements Chapter 4.

## B.1 PingPong Characterization

In Section 4.3.2, we discuss how we use PingPong to extract packet pairs, which are then used as input for our clustering algorithm that analyzes the PBFs of smart TV apps. Most notably, (1) we treat app launch samples as equivalent to smart home device events: we merge the per-app launch samples into one trace and analyze it using PingPong; (2) from the output of PingPong's pair clustering step that uses DBSCAN, we make the conservative choice to only consider DBSCAN clusters that consist of identical packet pairs, i.e., we allow no variability in packet sizes in the pairs of a PBF; and (3) we use PingPong's default configuration that sets DBSCAN's $\epsilon$ parameter, which informally specifies how far a packet pair can reside from an existing DBSCAN cluster of packet pairs to become part of that cluster, to 10 [151]. However, our methodology for clustering apps based on their fingerprints is flexible enough to accommodate other design choices for the underlying packet pair extraction performed by

PingPong. Next, we elaborate on the aforementioned choices (2) and (3).

**(2) Packet Size Variations.** In [151], PingPong is designed to allow the sizes of packets in a pair of a PLS to vary slightly. As evident from our results, in a lot of cases, smart TV apps exhibit deterministic packet pairs that always appear with identical packet sizes, e.g., C-882 S-219. However, in some cases, packet sizes can have slight variations, e.g., C-882 S-219, C-892 S-219, etc. In our work described in Chapter 4, we make the most conservative choice and only consider packet sizes with no variations as candidates for inclusion in a PBF, discarding the remaining packet pairs that have slight variations. This choice is consistent with DBFs: fully qualified domain names (FQDN) considered as candidates for inclusion in a DBF are unique.

By loosening this strict requirement, one can potentially increase the PBFs' sizes and/or PBF prevalence, but this may come at the cost of additional PBF collisions. Our approach may be generalized to accommodate packet size variations by fine-tuning $\epsilon$ and modifying the logic that decides what clusters (as output by PingPong) are considered for inclusion in a PBF. In fact, one could even consider distance of domains in DBFs by considering not only exactly matching FQDNs, but also common effective second-level domains (eSLD).

**(3) Interaction Between Pair Clustering by PingPong and PBF Clustering by FingerprinTV.** Throughout Chapter 4, we use PingPong with its default DBSCAN parameter $\epsilon = 10$ to cluster packet pairs. When we observe the output of PingPong, we sometimes find clusters that contain two different packet pairs that are within a distance of 10. For instance, we observe this phenomenon for Roku apps: the omnipresent Roku-specific packet pair described in Section 4.3.2 is sometimes clustered (by PingPong) together with other pairs that are within a distance of 10, e.g., C-882 S-219 and C-892 S-219. Since FingerprinTV discards clusters that are not exclusively comprised of identical packet pairs, it only includes the Roku-specific pair in the PBFs of 834 of the 1000 Roku apps when $\epsilon = 10$.

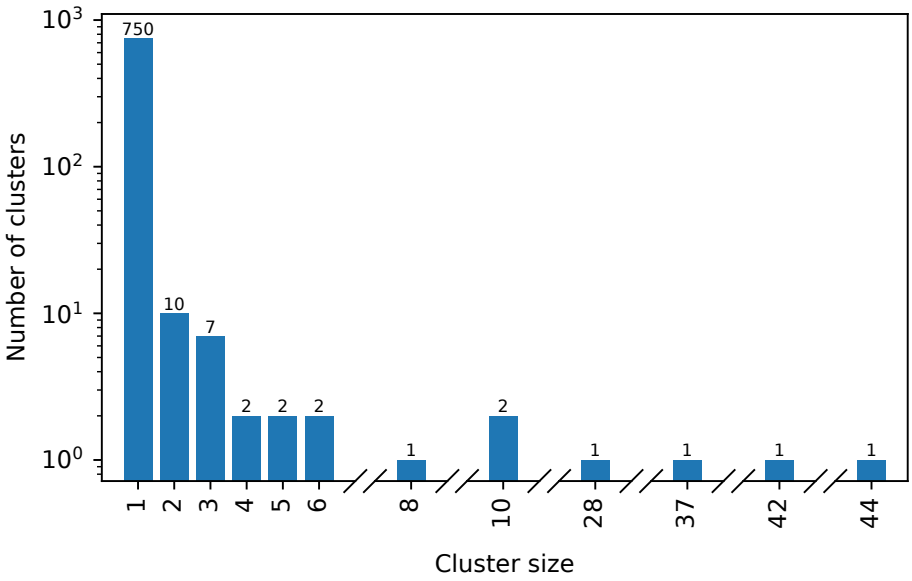



Figure B.1: Distribution of clusters by cluster size for PBFs for Roku when PingPong's $\epsilon$ parameter is set to 0.

To eliminate this behavior, one may set $\epsilon = 0$ so as to force PingPong to only output clusters consisting of identical packet pairs. To examine what effect this has on the inclusion of the omnipresent Roku-specific packet pair in the PBFs of Roku apps, we also run PingPong with $\epsilon = 0$ and then use FingerprinTV to perform PBF extraction and performance analysis on this PingPong output. With this configuration change, the Roku-specific packet pair becomes part of the PBF for 974 Roku apps. For the remaining 26 apps, the Roku-specific packet pair appears more than $L$ times across the $L$ launch samples of each app and is therefore discarded (see Definition 4.2 and recall that $U = L = 10$). Compared to Figure 4.5c that shows 717 apps with distinct PBFs, namely apps in clusters of size 1, when we run PingPong with $\epsilon = 10$, Figure B.1 shows that we have 750 apps that have distinct PBFs when we run PingPong with $\epsilon = 0$.

In summary, changing the value of $\epsilon$ in PingPong's DBSCAN clustering is a design choice. More generally, a careful co-design of DBSCAN clustering in the underlying PingPong with PBF clustering by FingerprinTV is required to achieve the desired trade-off between prevalence and distinctiveness of the extracted PBFs.