

Measurement Techniques to Understand How Diversity
in TLS Implementations & Deployments
Influences Protocol Security

by

Muhammad Talha Paracha

A dissertation submitted in satisfaction of the requirements
for the degree of Doctor of Philosophy in Computer Science
at Northeastern University in Boston, Massachusetts

thesis committee

David Choffnes, Northeastern University

Alan Mislove, Northeastern University

Christo Wilson, Northeastern University

Taejoong Chung, Virginia Tech

November 2023

**Northeastern University
Khoury College of
Computer Sciences**




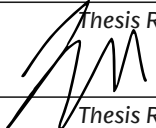
PhD Thesis Approval

Measurement Techniques to Understand How Diversity in TLS
Thesis Title: Implementations & Deployments Influences Protocol Security

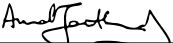
Author: Muhammad Talha Paracha

PhD Program: Computer Science Cybersecurity Personal Health Informatics


PhD Thesis Approval to complete all degree requirements for the above PhD program.

 _____ Thesis Advisor	<u>11/27/23</u> _____ Date
 _____ Thesis Reader	<u>11/27/23</u> _____ Date
 _____ Thesis Reader	<u>11/27/2023</u> _____ Date
 _____ Thesis Reader	<u>11/27/23</u> _____ Date
_____ Thesis Reader	_____ Date

KHOURY COLLEGE APPROVAL:

 _____ Associate Dean for Graduate Programs	<u>11/28/2023</u> _____ Date
--	------------------------------------

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

 _____ Recipient's Signature	<u>29 November 2023</u> _____ Date
---	--

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to the Northeastern University-UMI Website.

To my parents, Ammi and Abbu. May you two keep smiling forever.

Abstract

TLS is a fundamental and widely-used network security protocol. On one hand, the protocol has undergone rigorous development over the past 25 years and offers sophisticated theoretical guarantees. At the same time, its adoption has grown from traditional computers to handheld devices and IoT ones, with these settings presenting varying constraints and caveats. As a consequence, a large number of TLS implementations and deployments exist and cater to different needs. Unfortunately, this results in a gap between what the protocol offers in theory *vs* how it works in practice; the diversity in the ecosystem not only increases the probability of a mistake during protocol development and use, but also leads to customizations with unexpected side effects.

The thesis of this dissertation is that the rich diversity in TLS implementations & deployments introduces opportunities to harm protocol security, and that the harms can be identified (and mitigated) using rigorous measurement techniques.

My work sheds light on previously unexplored aspects of TLS deployment in three different settings; web, mobile and IoT devices. More specifically, I (a) study web content availability and consistency over HTTP/S to better understand the obstacles to a TLS-by-default web, (b) conduct longitudinal experiments on a large number of consumer IoT devices to evaluate TLS effectiveness in that setting, and (c) revisit certificate pinning policies in mobile applications to examine implementations with advanced network security techniques that go beyond what the protocol offers.

In addition to exploring diversity in deployments, my work leverages the diversity in TLS implementations alongside recent advances in generative language models to automate bug discovery. More specifically, I present a novel approach of generating synthetic TLS certificates using language models that reveal a wide range of previously unobserved and interesting implementation differences with security implications.

My work has led to vulnerability disclosures, a security feature at a major CDN provider, a presentation at an IRTF body to inform protocol engineering, and novel auditing techniques that enable greater transparency about real-world protocol effectiveness. I believe the insights from my work can assist in better modeling of software security beyond TLS, the techniques proposed push state-of-the-art for network measurement, and the use of language models to generate synthetic test cases can prove valuable in domains where software inputs can be expressed in natural language.

Contents

1	Introduction	1
2	Background and Related Work	5
2.1	Protocol Basics	5
2.2	TLS Security	6
2.3	Network Measurement	7
3	TLS Usage in Consumer IoT Devices	9
3.1	Goals	10
3.2	Methodology	11
3.2.1	Testbed	11
3.2.2	Instrumentation	13
3.3	Results	16
3.3.1	TLS Connection Security	16
3.3.2	Certificate Validation	23
3.3.3	Diversity of TLS Behavior	26
3.4	Discussion	28
3.5	Conclusion	30
4	Web Content Availability and Consistency over HTTP/S	32
4.1	Methodology	33
4.1.1	HTTP/S Inconsistencies	33
4.1.2	Crawling Overview	33
4.1.3	Identifying Inconsistencies	35
4.2	Results	38
4.2.1	Summary Results	38
4.2.2	Factors Influencing Inconsistencies	40
4.2.3	Comparing HTTPS Adoption Metrics	41
4.3	Discussion	42
4.4	Conclusion	42
5	TLS Certificate Pinning in Mobile Applications	44
5.1	Background and Motivation	45
5.2	Goals	46
5.3	Methodology	47

5.3.1	Datasets	47
5.3.2	Static Analysis	48
5.3.3	Dynamic Analysis	49
5.3.4	Circumventing Pinning	51
5.3.5	PII Analysis	52
5.3.6	iOS Background Traffic	52
5.4	Results	53
5.4.1	Pinning in Common Apps	55
5.4.2	Pinning in Popular vs Random Apps	57
5.4.3	Certificate Analysis	58
5.4.4	Connection Security	62
5.4.5	PII in Pinned vs Non-Pinned Traffic	62
5.4.6	Limitations	63
5.5	Discussion	64
5.6	Conclusion	65
6	Testing TLS Certificate Validation Using Generative Language Models	66
6.1	Background and Motivation	67
6.2	Goals	71
6.3	Methodology	71
6.3.1	Certificate Datasets	72
6.3.2	Language Models	73
6.3.3	Differential Testing Framework	74
6.4	Results	75
6.4.1	Discrepancies and Code Coverage	76
6.4.2	Certificate Diversity	78
6.4.3	Security Implications	80
6.5	Discussion	82
6.6	Conclusion	83
7	Concluding Remarks	84

Acknowledgements

First and foremost, I want to acknowledge my advisor David Choffnes for his absolute mentorship throughout my doctoral journey. I am deeply grateful for having witnessed and learned from Dave's vision for research, attention to detail in methodology, and aspiration to do impactful work. I know I will continue to cherish these learnings in my career going forward. Over the years, I have also realized the importance of kindness, understanding, and support I received from Dave that enabled me to succeed in my program. Work and health during COVID-19 were particularly challenging to handle, but Dave always helped brainstorm ideas and suggest resources to manage such situations. Thank you, Dave.

I want to thank members of my thesis committee Alan Mislove, Christo Wilson, and Taejoong (Tijay) Chung for their support and guidance. I have been fortunate to meet these people at the start of my program and I admire the positivity they brought into all of our encounters. I now understand well how the company of pleasant colleagues makes research all the more interesting. In the same vein, I want to thank Martina Lindorfer for providing me with support and guidance (and, a desk with her students at TU Wien) during the last few months of my program amid my visa issues. And I want to thank Daniel Dubois for being a mentor and friend as our most experienced and senior labmate.

I want to thank my friends and community at Northeastern University and in Boston for all the amazing memories. Thank you Prasanth for being a great friend, and for all the discussions on work (or politics) during our table tennis games. Thank you Ali, Amogh, Domien, Harshad, Shuwen, Ben, Clifton, Aziz, Johanna, Tianrui, Narmeen, Fan for all the fun hangouts and conversations. I am also thankful to my dear friends outside of work, Shah Rukh, Asad, Hassan, Tooba, Rida, Mahnoor, and Arsalan for their presence in my life. And I am very thankful to Geeticka for her support and belief in me to continue my program with a passion and excitement that had diminished somewhere along the way.

Last, I want to acknowledge my family: *Ammi* (Farhat), *Abbu* (Shaukat), *Aapi* (Sidrah), *Bhai* (Suleman), Tayyab, and Noor. Although my Mom and Dad cannot relate to the journey of a doctoral program, their deep encouragement towards me in pursuing my academic ambitions has always been there. And although my Mom and Dad cannot relate to the journey of living in a foreign country, their deep care towards me to make me feel belonged has always been there. I am thankful to all of my family for their continuous love and support that enables me to keep going forward in my life and career.

Chapter 1

Introduction

Transport Layer Security (“TLS”, the counterpart to web-specific HTTPS) is the de facto, IETF-standard, Internet security protocol that provides *confidentiality*, *integrity*, and *authenticity* to network communications. By November 2022, 99% of pages browsed through Google Chrome used TLS [1]. Both Android and iOS default to using TLS for network traffic from apps (unless explicitly opted-out) [2], [3]. Prior research indicates that IoT devices mainly rely on TLS as well for network security [4].

TLS is designed using cryptographic primitives and standardized through protocol specification (“spec”) documents in plain writing. The protocol functionality is then programmed into TLS libraries (“implementations”) in accordance with the spec. At the time of this writing, at least 18 TLS implementations exist and cater to different needs. For instance, some implementations aim to meet particular performance demands (MbedTLS has a smaller footprint for embedded devices), while others differ in their coding style and language (LibreSSL and BoringSSL are forks of OpenSSL managed by different teams). These implementations are then configured and deployed by administrators in different settings (“deployments”), either as-is, or as a part of a larger framework (e.g., NGINX web server uses OpenSSL).

The thesis of this dissertation is that the rich diversity in TLS implementations & deployments introduces opportunities to harm protocol security, and that the harms can be identified (and mitigated) using rigorous measurement techniques.

First, this diversity increases the probability of a mistake during protocol development and use. Programming bugs are regularly found in TLS implementations (e.g., *Heartbleed*), while misconfigurations are common in TLS deployments (e.g., using deprecated protocol features). Second, this diversity results in customizations with side effects that cannot be predicted from the protocol spec alone. For instance, TLS-enabled websites typically support the insecure HTTP protocol as well, due to browsers HTTP-by-default policy – this behavior unfortunately undermines protocol security and leaves site visitors vulnerable to a multitude of attacks (e.g., [5]–[7]).

With respect to TLS deployments, my work sheds light on previously unexplored aspects of protocol use in three different settings; web, mobile and IoT devices. For each setting, my research introduces novel network measurement techniques that highlight issues in protocol

security specific to that setting. With respect to the TLS implementations, my work proposes a novel technique that uses differential testing alongside generative language models with the aim to automate bug discovery. While it is substantially time-consuming to manually scrutinize a TLS implementation for programming bugs, prior works have used different TLS implementations as cross-referencing oracles to systematically find bugs. My research extends this line of work by learning X.509 TLS certificate representations using generative language models, and sampling from the learned representations to obtain synthetic certificates that reveal implementation differences with security implications.

In detail, my published works (Tables 1.1 and 1.2) explore TLS use in:

Consumer IoT Devices (IMC 2021) To evaluate TLS effectiveness in this setting, we gather more than two years of TLS network traffic from IoT devices, conduct active probing to test for vulnerabilities, and develop a novel blackbox technique for exploring the trusted root stores in IoT devices by exploiting a side-channel through TLS Alert Messages. We find a wide range of behaviors across devices, with some adopting best security practices but most being vulnerable in one or more of the following ways: use of old/insecure protocol versions and/or ciphersuites, lack of certificate validation, and poor maintenance of root stores. Specifically, we find that at least 8 IoT devices still include distrusted certificates in their root stores, 11/32 devices are vulnerable to TLS interception attacks, and that many devices fail to adopt modern protocol features over time.

Popular Websites (TMA 2020) To better understand the obstacles to a TLS-by-default web, we crawl beyond the landing page to understand HTTPS content unavailability and inconsistency issues that remain in today’s popular HTTPS-supporting websites. Our analysis shows that 1.5% of the HTTPS-supporting websites from the Alexa top 110k have at least one page available via HTTP but not HTTPS. Surprisingly, we also find 3.7% of websites with at least one URL where a server returns substantially different content over HTTP compared to HTTPS. We propose new heuristics for finding these unavailability and inconsistency issues, explore several root causes, and identify mitigation strategies. Taken together, our findings highlight that a low, but significant fraction of HTTPS-supporting websites would not function properly if browsers use TLS-by-default.

Mobile Applications (IMC 2022) To examine implementations with advanced network security techniques that go beyond what the TLS protocol offers, we thoroughly investigate the use of certificate pinning on Android and iOS. We collect 5,079 unique apps from the two official app stores: 575 common apps, 1,000 popular apps each, and 1,000 randomly selected apps each. We develop novel, cross-platform, static and dynamic analysis techniques to detect the usage of certificate pinning. We find certificate pinning as much as 4 times more widely adopted than reported in recent studies. More specifically, we find that 0.9% to 8% of Android apps and 2.5% to 11% of iOS apps use certificate pinning at run time (depending on the aforementioned sets of apps). We then investigate which categories of apps most frequently use pinning (e.g., apps in the “finance” category), which destinations are typically pinned (e.g., first-party destinations vs those used by third-party libraries), which certificates are pinned and how these are pinned (e.g., CA vs leaf certificates), and the connection security for pinned connections vs unpinned ones (e.g., the use of weak ciphers

Table 1.1: List of peer-reviewed publications part of this dissertation. (*) indicates equal contribution.

Conference	Publication
TMA 2020	A Deeper Look at Web Content Availability and Consistency over HTTP/S <i>Talha Paracha, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin</i>
IMC 2021	IoTLS: Understanding TLS Usage in Consumer IoT Devices <i>Talha Paracha, Daniel Dubois, Narseo Vallina, David Choffnes</i>
IMC 2022	A Comparative Analysis of Certificate Pinning in Android & iOS <i>Amogh Pradeep*, Talha Paracha*, Protick Bhowmick, Ali Davanian, Abbas Razaghpanah, Taejoong Chung, Martina Lindorfer, Narseo Vallina, Dave Levin, David Choffnes</i>

Table 1.2: List of related peer-reviewed publications.

Conference	Publication
PETS 2020	When Speakers Are All Ears: Characterizing Misactivations of IoT Smart Speakers <i>Daniel Dubois, Roman Kolcun, Anna Maria, Talha Paracha, David Choffnes, Hamed Haddadi</i>
PETS 2021	Blocking without Breaking: Identification and Mitigation of Non-Essential IoT Traffic <i>Anna Maria, Daniel Dubois, Roman Kolcun, Talha Paracha, Hamed Haddadi, David Choffnes</i>
IMC 2022	Respect the ORIGIN! A Best-case Evaluation of Connection Coalescing in The Wild <i>Sudheesh Singanamalla, Talha Paracha, Suleman Ahmad, Jonathan Hoyland, Luke Valenta, Yevgen Safronov, Peter Wu, Andrew Galloni, Kurtis Heimerl, Nick Sullivan, Christopher A. Wood, Marwan Fayed</i>
IMC 2023	Behind the Scenes: Uncovering TLS and Server Certificate Practice of IoT Device Vendors in the Wild <i>Hongying Dong, Hao Shu, Vijay Prakash, Yizhe Zhang, Talha Paracha, David Choffnes, Santiago Torres-Arias, Danny Huang, Yixin Sun</i>

or improper certificate validation).

Further, my work explores the diversity in TLS implementations using:

Differential Testing and Generative Language Models (in preparation) Certificate validation is a crucial step in TLS. Prior research has shown that differentially testing a corpus of synthetic TLS certificates can reveal critical security issues (e.g., accidentally accepting untrusted *v1* certificates). Such work relies on a variety of techniques (e.g., random mutations, program coverage guidance) to obtain synthetic certificates that execute diverse code paths and trigger new bugs, which are then caught by comparing outputs of multiple TLS libraries against each other. Inspired by this prior work, we identify a new opportunity for differential testing: generative language models based on neural networks (e.g., ChatGPT), which are popular today for applications such as generating content, writing code and conversing with users. The core insight in our work is that TLS certificates can be expressed in natural language, as they are defined in a standard (X.509) that supports human readability. In this work, we present a novel approach of generating synthetic TLS certificates using off-the-shelf generative language models. Our models are able to (i) learn representations for TLS certificates, and (ii) generate new certificates that (usually) conform to the protocol standard but produce diverse behavior during the certificate validation process for several TLS implementations. Our results show that these synthetic certificates produce an order of magnitude more “distinct discrepancies” than baseline during differential testing, and reveal a wide range of previously unobserved and interesting behavior with security implications.

Apart from these technical contributions, the work on IoT devices led to vulnerability disclosures to 8 manufacturers. It was also presented to a group at the Internet Research Task Force (IRTF) [8] with the aim to inform protocol engineering and practice. The work on web HTTP/S content consistency inspired the development of SSL/TLS Recommender at Cloudflare [9] that helps websites upgrade to TLS. All works introduce novel auditing techniques that can be further used by administrators and investigators to better understand real-world protocol effectiveness.

This dissertation is structured as follows. Chapter 2 provides background and related work. Chapters 3-5 detail peer-reviewed works that explore TLS deployments in distinct settings. Chapter 6 presents the use of generative language models for differentially testing TLS implementations. Chapter 7 provides concluding remarks with an overview of the lessons learned from work in this dissertation.

Chapter 2

Background and Related Work

Since Netscape started work on the TLS predecessor, Secure Sockets Layer (SSL), ≈ 25 years ago, TLS has undergone rigorous development featuring various standardization efforts and releases—SSL 2.0 (1995), SSL 3.0 (1996), TLS 1.0 (1999), TLS 1.1 (2006), TLS 1.2 (2008) and TLS 1.3 (2018). This chapter provides relevant background information about TLS alongside prominent implementation and deployment insights from the literature that relate to the work in this thesis.

2.1 Protocol Basics

Root stores TLS generally uses digital certificates that bind host identities with cryptographic material. These certificates are issued by Certificate Authorities (CAs) and can be “revoked” if they get compromised. Server authentication is the most common TLS usage where clients store relevant information about one or more trusted CAs root certificates and require that servers present valid certificates from one of them to authenticate themselves. These certificates form a trusted set of “root” store certificates deployed on end systems; if the private key for any of these trusted root certificates is compromised, the attacker can circumvent security guarantees of *all* TLS connections by a client.¹ Currently, web browsers and operating systems (OS) ship with dozens of root certificates in their root stores to enable TLS communication with a wide range of servers. Due to the crucial importance of root certificates, these platforms actively maintain their root stores to remove any certificates from CAs that violate CA guidelines or get compromised.

Secure connection establishment A TLS “handshake” is the set of messages that establishes a secure session between two end hosts. The process is initiated by a client to advertise its supported protocol versions, ciphersuites (i.e., cryptographic algorithms), and extensions (i.e., other advanced features). In response, a TLS server decides the protocol version and ciphersuite to use based on its compatibility. During the handshake, the client and server can authenticate each other and compute cryptographic keys to be used for confidentiality and integrity of the future communication. The authentication typically involves

¹Except for applications that use strategies such as certificate pinning.

validating the received certificate chain using a root store. Any data sent after a successful handshake is encrypted.²

2.2 TLS Security

Handshake vulnerabilities By 2020, major browsers had deprecated all TLS versions below 1.2 due to serious security flaws [10]. Yet, some TLS clients voluntarily downgrade connection security upon handshake failure to improve compatibility with older servers. The POODLE (2014) [11] attack exploited the behavior of major web browsers and other clients to fallback to SSL 3.0 (known to be insecure at the time) and highlighted the risks of any fallback behavior. In general, the TLS *interception* class of attacks typically refer to weaknesses in handshake validation that are exploited by on-path attackers. These attacks are particularly severe as they allow secret eavesdropping of all TLS communication sent between a client and server on a compromised connection.

Web vulnerabilities HTTP-by-default behavior in web browsers, and limited deployment of TLS results in severe security and privacy risks to users (e.g., [5]–[7]). Prior work argues for TLS to be supported on every page of a website, and for web browsers to use TLS-by-default, to mitigate these threats.

Mobile vulnerabilities Georgiev et al. [12] conducted one of the earliest studies about TLS usage in non-browser software on multiple platforms including Android and iOS. They found instances of insecure TLS implementations on both platforms. TLS usage on Android was also studied by Fahl et al. [13] in 2012. Using static and dynamic analysis techniques, they found 8% of apps studied to be vulnerable to interception attacks. On the other hand, Orikogbo et al. [14] studied the validity of TLS certificate validation logic in iOS apps. They found rare instances of implementations that would bypass all or some of the critical validation steps (e.g., certificate expiration check).

Root stores can include expired, unknown, or obscure CA certificates (e.g., [15]–[17]), which can expose clients to TLS interception attacks. An attacker with access to the private key for a CA certificate in the trust store can use it to sign arbitrary certificates (for arbitrary domains) and trick the client into accepting these malicious certificates as valid. *Certificate pinning* is an alternate to trusting OS root stores, where mobile applications include a custom certificate to be trusted (in source code). Pinning prevents attacks by limiting certificate trust to a pre-determined set of certificates instead of trusting a certificate issued by any CA certificate in the OS root store. Note that certificate pinning not only protects against malicious actors, but also against investigators and auditors seeking to analyze the data exchanged between devices and servers (e.g., to understand personal data exfiltration).

IoT vulnerabilities Alrawi et al.’s SoK [4] provides a broad security evaluation of IoT devices. Their work covers 45 devices from four different dimensions; devices themselves along with their cloud endpoints, communication channels and mobile apps. The authors find that these devices typically rely on TLS for network security, but often contain trivial

²Assuming the NULL ciphersuite is not selected.

flaws in the protocol implementation and deployment.

For a comprehensive and systemic review of TLS security issues and how the protocol has evolved over time to deal with those challenges, please refer to [18].

Differential testing In order to systematically find bugs in certificate validation logic of TLS implementations, a seminal differential testing approach was introduced by Brubaker et al. [19]. The authors (i) generated a corpus of synthetic test certificates by randomly combining and mutating parts of real certificates, and (ii) provided the corpus as input to multiple TLS libraries to use them as cross-referencing oracles to find differences in implementations (and bugs). Large body of prior research extends this line of work with the aim to improve the synthetic certificate generation process, including, but not limited to: *Mucerts* [20] that uses code coverage guidance, *Coveringcerts* [21] that uses combinatorial methods with theoretical guarantees, *SymCerts* [22] that adds symbolic execution, *RFCcerts* [23] that leverages certificate rules from protocol specification documents, *Transcerts* [24] that relies on coverage transfer graphs, *NEZHA* [25] that keeps track of behavioral asymmetries across multiple programs, and *DRLgencert* [26] that uses deep reinforcement learning to perform mutations on a certificate. Note that in contrast to these techniques that automatically generate synthetic certificates, Barengi et al. [27] work to first manually obtain a grammar for TLS certificates, and then build a parser to find legitimate issues in certificates that are missed by various implementations.

2.3 Network Measurement

TLS adoption There is a significant body of research on analyzing TLS usage from different vantage points i.e., passive monitoring of university networks [28]–[30], server-side connection logs [17], active Internet scans [31]–[33], browser telemetry data [34], and Android usage statistics [35]. Overall, the results from prior work indicate a steady trend towards ubiquitous use of TLS.

Root stores analysis Fadai et al. [36] investigated the historical data for Mozilla’s root certificates. They evaluated the trust implications of root certificates from several platforms in terms of the owner status (i.e., private entity or governmental organization) and country of origin. Other works proposed techniques to restrict the set of root CAs trusted by users based on the insights that (i) CAs commonly sign a handful of top-level domains [28], (ii) some CAs have not signed any certificates used by the HTTPS servers [37], and (iii) unique browsing history enables individualization of the trusted CAs set [38]. Some works have also focused on the user-trusted certificates present in the wild and that do not belong to audited root stores—Vallina-Rodriguez et al. [15] explored vendor and app-specific additions to the official Android root store, and Durumeric et al. [17] explored the additions due to middleboxes such as an antivirus software or a corporate proxy.

Pinning adoption Razaghpanah et al. [35] found 150 apps in their dataset that implemented some form of pinning (2% of apps analyzed). In 2015, Android 6.0 introduced Network Security Configurations (NSCs) [39] that enable apps to customize certain network

security settings. Possemato et al. [40] studied Network Security Policies (a single line configuration in Android Manifests, or NSC files) and found that 13.02% of 125k+ apps used these policies; with only 0.62% (of the 13.02%) using pinning. Oltrogge et al. [41] studied NSC adoption using static analysis and found that 7.43% of 1.3M apps used NSCs; with only 0.67% (of the 7.43%) using pinning.

TLS fingerprinting TLS fingerprinting has been used frequently in the past to infer client behaviors – from detecting malware [42] to the usage of censorship circumvention tools [43] and client identification (e.g., [17], [29], [30], [35]).

Chapter 3

TLS Usage in Consumer IoT Devices

Consumer Internet-of-Things (IoT) devices such as voice assistants, smart TVs and video doorbells are popular, with their prevalence projected to be 75 billion by 2025 [44]. Most IoT devices rely on TLS to provide confidentiality, integrity and authenticity of their network communications [4]. Numerous prior works have shown that TLS security properties can be compromised due to development errors (e.g., [19]), insecure configurations (e.g., [12]), and outdated clients (e.g., [45]). While TLS usage has been studied extensively in mobile applications and web browsers (e.g., [41], [14], [34]), there is little insight into its effectiveness in the IoT ecosystem (e.g., [4]).

More specifically, there exists a research gap in understanding whether TLS implementations in IoT devices: (i) establish connections using secure TLS versions and ciphersuites, (ii) correctly perform certificate validation while using a generally trusted set of root certificates, and (iii) adopt new features as the protocol evolves over time (e.g., modern ciphersuites). There are several challenges that prevent the use of existing methodologies to study these aspects of IoT devices. *First*, understanding TLS support on a significant number of IoT devices requires blackbox testing techniques; this is because source code is generally unavailable and firmware analysis is not scalable. *Second*, most IoT devices provide limited ways to trigger TLS traffic for measurement—the timing, destination, and contents of their communication are all dependent on device functionality and interactions. *Third*, existing vantage points offer limited opportunities to track device behavior over time (e.g., recent work considers only manufacturer-level device tracking using ISP/IXP data [46]).

In this chapter, we address these challenges to study a large number of TLS-enabled consumer IoT devices (with over 200 million units sold collectively). First, we shed light on the security of TLS implementations and configurations in these devices using existing and novel active measurement techniques that require only TLS traffic interception. Second, based on the insight that devices generate significant network traffic when powered on, we automate device reboots using smart plugs to trigger TLS activity for our experiments. And third, we analyze ≈ 2 years of network traffic from these devices via uncontrolled experiments to study how their TLS usage changes over time. Altogether, we conduct *active* experiments on 32 devices, and collect *passive* data from 40 devices, each generating TLS traffic for at least 6 months.

Our key objectives are to evaluate the security of TLS connections established by IoT devices, how this changes over time, and whether they correctly validate certificates. More specifically, we study how devices’ TLS implementations are configured with respect to TLS versions and ciphersuites supported, and provide the first longitudinal analysis of how these properties change over time, as the TLS protocol and attacks against it evolve. We further check whether the devices properly validate certificates to protect against the TLS interception attacks, extending prior work by including a more comprehensive set of invalid certificates in our tests. We develop a novel active testing strategy to reveal the trusted set of root certificates on a device. These certificates form the “*trusted root*” of all security guarantees provided by the TLS protocol and auditing them is particularly important given the recent rise in supply-chain attacks by powerful adversaries [15], [47]. While prior works have studied root stores in open platforms (e.g., operating systems and browsers), to the best of our knowledge we are the first to investigate the validity of root certificates used in IoT devices.

3.1 Goals

Our goals are to answer three research questions (RQs):

RQ1: Do devices *securely establish* TLS connections? Securely establishing TLS connections means that devices use secure TLS versions and ciphersuites. In this work, we consider whether devices are resilient to in-network adversaries (e.g., a network provider) that have the ability to capture and manipulate TLS traffic between an IoT device and the destinations it contacts. We focus on device support for the latest, secure protocol versions, modern ciphersuites, and negotiated TLS configurations between IoT clients and their destinations. We use ≈ 2 years of passively collected longitudinal IoT traffic to determine whether devices adopt new features and abandon deprecated ones.

RQ2: Do devices *properly validate* TLS certificates? In this work, we focus on evaluating whether devices accept connections with invalid certificates, and understanding whether their root stores contain deprecated and/or distrusted root certificates. Specifically, we focus on validation of the server certificate chain, hostname and various X.509 extensions specified in RFC 2818 [48] and RFC 5280 [49]. In addition, we evaluate the configured set of trusted root certificates to determine whether devices protect against distrusted and/or stale root certificates.

RQ3: What is the *diversity of behaviors* within and across devices? The effectiveness of a single attack vector is limited to the set of devices that share the same vulnerability. To understand the breadth of the impact of potential attacks, we investigate how many devices exhibit the same TLS behavior and, potentially, the same security issues. We further investigate whether individual devices exhibit different TLS behavior for different connections—indicative of multiple TLS instances on the same device.

To answer these questions, instead of modeling IoT devices as monolithic implementations, we treat them as complex devices that can integrate third-party components and even allow users to install third-party software as in the case of Smart TV platforms. These cases



Figure 3.1: Mon(IoT)r lab at Northeastern University. The lab is designed to resemble a smart home with consumer IoT devices.

can lead to additional risk of vulnerabilities due to the need to maintain the security of these multiple TLS deployments and development errors, both at the OS level and across third-party developers.

Assumptions For this study, we assume that when a device uses TLS, the corresponding traffic must be safeguarded to provide authenticity, confidentiality and integrity. Note that we cannot in general know whether the content of any specific TLS connection is sensitive (e.g., contains personal data). When such connections are used to transmit sensitive content such as users’ personal data or a manufacturer’s confidential machine-learning models, there is a clear need to protect it against attackers. While some TLS endpoints may be public services that exchange non-sensitive data, we cannot *a priori* distinguish such entities, and thus treat all endpoints that use TLS as *potentially sensitive*.

3.2 Methodology

This section provides an overview of our methodology, analyses, and how they relate to the research questions.

3.2.1 Testbed

IoT Devices We study 40 TLS-supporting IoT devices across 6 categories; *Cameras*, *Smart Hubs*, *Home Automation*, *TV*, *Audio* and *Other Appliances* (Table 3.1). The testbed is configured to represent a smart home with a wide range of consumer IoT devices connected to the Internet (Figure 3.1). All devices are located in an isolated space designed to resemble

Table 3.1: List of the 40 TLS-supporting devices in our study. (*) denotes devices used only in *passive* experiments.

Cameras (n = 7)	Smart Hubs (n = 7)	Home Automation (n = 7)	TV (n = 5)	Audio (n = 7)	Appliances (n = 7)
Blink Camera*	Blink Hub	Smartlife Bulb	Fire TV	Google Home	GE Microwave
Amazon Cloud-cam*	Smartthings Hub	Smartlife Remote	Samsung TV*	Mini	Samsung Washer*
Zmodo Doorbell	Philips Hub	Meross	LG TV	Amazon Echo	Samsung Dryer
Yi Camera	Wink Hub 2	Dooropener	Roku TV	Plus	Samsung Fridge
D-Link Camera	Sengled Hub*	TP-Link Bulb	Apple TV	Amazon Echo	Smarter iKettle
Amcrest Camera	Switchbot Hub	Nest Thermostat		Dot 3	Behmor Brewer
Ring Doorbell*	Insteon Hub*	TP-Link Plug		Amazon Echo	LG Dishwasher*
		Wemo Plug		Spot	
				Harman Invoke	
				Apple HomePod	

a studio apartment. To interact with devices that support companion apps, we installed and used these apps on smartphones connected to the same network. Network traffic collection is performed at a gateway that provides network access only to our IoT testbed.

We use a software/firmware update discipline that we assume to be typical of an IoT home scenario. Specifically, devices that receive automatic updates are updated at whatever cadence the manufacturer specifies. For devices that require manual intervention for updates, we accepted the updates when explicitly asked by the companion apps of devices. Note that we accept these updated in an ad-hoc manner, and as such, these devices are not regularly updated. We decided to use this approach because (a) we expect many users to also use these devices in a similar way, and (b) getting *all* devices to update at a regular interval could not be automated.

Experimental Setup and Dataset Our study uses a combination of *passive* and *active* experiments. The key difference between the two experiment types is that *active* experiments involve the usage of *mitmproxy* [50] to intercept traffic while *passive* experiments do not. Both experiments need some form of interaction with the IoT devices for generating network activity.

In *passive* experiments we simply record the network traffic generated by devices. This includes data while devices are not in use, and also from interactions with ≈ 40 consenting study participants enrolled in our IRB-approved study. These participants are members of our academic institution, and are directed simply to use these devices as they please. The passive dataset covers ≈ 2 years of traffic from January 2018 to March 2020. Among the 40 devices in *passive* experiments, every device generated traffic for at least 6 months, while 32 devices did so for more than 12 months. Passive data allows us to observe the real-world behavior of the devices (a) when they are connected to the network without user interactions, and (b) when users interact with them.

In *active* experiments, we intercept the traffic from our devices by impersonating the server-side of TLS connections. To induce the devices to generate TLS traffic for interception, we leverage the observation that IoT devices generate significant traffic when powered on [51]. Thus we programmatically use TP-Link power plugs to turn devices off and back on again,

Table 3.2: Overview of the TLS interception attacks.

Attack	Description
NoValidation	Use a self-signed certificate to check whether a device performs any certificate validation.
WrongHostname	Use an unexpired legitimate certificate for a domain under our control to check whether a device performs hostname validation. We send the full chain linking to a trusted root authority during handshake.
InvalidBasicConstraints	Use certificate from the previous attack as a root CA to check whether a device validates <i>BasicConstraints</i> extension. We send the full chain linking to a trusted root authority during handshake.

causing them to boot and potentially establish TLS connections. All 32 devices in *active* experiments generated at least one TLS connection. The bulk of our experiments were performed in March 2021.

Some devices broke, lost manufacturer support or would lose WiFi connectivity until reconfigured again. As a result, such devices did not generate traffic continuously throughout the entirety of our *passive* experimentation period, and were omitted from the *active* experiments (resulting in the discrepancy between number of devices in *active* vs *passive* experiments).

In total, we gathered ≈ 17 M TLS connections (per device average: ≈ 422 K, median: ≈ 138 K connections). Note that our active experiments comprise controlled, repeatable experiments that are conducted without study participants present and represent a snapshot in time (at least 3 minutes after a device reboot). Passive experiments are uncontrolled and they may include participant interactions, thus they enable us to study longitudinal insights across a variety of connections.

3.2.2 Instrumentation

We use the following instrumentation to gather data for analysis.

TLS handshake analysis (RQ1 and RQ3) To determine whether devices establish secure TLS connections, we extract information about TLS versions and ciphers advertised by clients and selected by servers. We further parse the *ClientHellos* to extract client fingerprints, and use this information to explore the diversity of TLS implementations in the IoT ecosystem.

Certificate Validation Analysis (RQ2) We investigate whether devices are susceptible to several TLS interception attacks that an adversary can use to compromise TLS connections (Table 3.2). We picked these attacks because they do not require significant resources (e.g., compromising a root CA, or breaking a cipher using cryptanalysis). They are related to proper certificate chain validation and have previously been found effective against a wide variety of non-browser TLS clients [12], so we extend these to IoT devices. We use *mitmproxy* [50] for performing these attacks.

Note that a potential limitation of our study is that attempts to test vulnerabilities (e.g.,

Table 3.3: Sources for obtaining historical data for CA root certificates trusted by various platforms.

Platform	Total versions	Earliest version year	Comments
Ubuntu	9	2012	We install the <i>ca-certificates</i> package and fetch the <i>/etc/ssl/certs/ca-certificates.crt</i> file from official Docker images.
Android	10	2010	We use version-tagged commits for either <i>/platform /system/ca-certificates</i> or <i>luni/src /main/files/cacerts</i> [52], [53].
Mozilla	47	2013	We extract different file versions from commit history for NSS’s <i>security/nss/lib/ckfw/builtins/certdata.txt</i> [54].
Microsoft	15	2017	We use the historical information published by Microsoft about its trusted root store certificates [55].

using self-signed certificates) will lead to connection errors, and those in turn may cause a device (or some of its functionality) to cease to work, thus suppressing further network connections. To test the potential impact of this issue, we restart devices and repeat all the above attacks with *TrafficPassthrough* where we do not intercept any connections that previously failed when under attack [56]. Encouragingly, we find that *TrafficPassthrough* experiments did not lead to finding *any* new certificate validation failures, even though they produced $\approx 20.4\%$ more connections (average, in terms of new DNS or TLS hostnames) from these devices. We speculate that these additional connections might be based on success responses from some earlier connections (e.g., a login request) and, as such, only appear in *TrafficPassthrough* tests.

Root stores analysis (RQ2) We present a novel technique to detect if a Certificate Authority (CA) root certificate is in the trusted root store of an IoT device. Our key insight is that the TLS protocol specifies different steps for clients when validating a certificate with an *unknown issuer* compared to a certificate with *known issuer but invalid signature*—opening a side channel to infer the presence of trusted root certificates in a client’s root store. In this work, we exploit this side channel using TLS *Alert Messages*.

We first use a self-signed root certificate with arbitrary *Subject Name* to intercept a TLS connection originating from the device. The device should fail to establish the connection if it is doing proper certificate validation because our CA certificate is not in its root store. We then intercept the same TLS connection using a *spoofed CA certificate*, i.e., a self-signed root certificate with its *Subject Name*, *Issuer Name* and *Serial Number* matching that of a legitimate root certificate being tested. The client should reject this certificate due to a signature validation error: while the subject name, issuer name, and serial number all match a trusted root certificate, we do not have the root CA’s private key to generate a valid signature for the leaf certificate in chain. Thus our interception attempt fails in both cases, but the failure could either be due to the client not recognizing the arbitrary *Subject Name* in its root store, or because it does recognize a *Subject Name* that is in its root store but the leaf certificate has an invalid signature. If we are able to observe this difference in device behavior, we can infer whether a given CA certificate is trusted by the device or not.

Table 3.4: Testing our technique for exploring root stores in various TLS libraries. Only two were found to be amenable (shown in italics).

Library	Response for known CA with invalid signature	Response for unknown CA
<i>MbedTLS (v2.21.0)</i>	<i>Bad Certificate</i>	<i>Unknown CA</i>
<i>OpenSSL (v1.1.1i)</i>	<i>Decrypt Error</i>	<i>Unknown CA</i>
Oracle Java (v1.8.0)	Certificate Unknown	Certificate Unknown
WolfSSL (v4.1.0)	Bad Certificate	Bad Certificate
GNU TLS (v3.6.15)	No Alert	No Alert
Secure Transport (macOS v11.3)	No Alert	No Alert

We found that the TLS specification provides a mechanism to observe this difference in behavior: per RFC 5246 (TLS 1.2) or RFC 8446 (TLS 1.3), a TLS client may choose to send a TLS *Alert Message* during a connection failure. More specifically, clients can choose to send `unknown_ca` alert to indicate that a trusted CA root certificate could not be found when forming the chain and `decrypt_error` alert to indicate for a signature check failure. For this work, we consider a device amenable to our technique of root store exploration if it sends different alerts based on the type of experiment run.

To realize this experiment, we use the approach from TLS interception attacks to boot devices, intercept their TLS connections, and respond with self-signed certificates as described previously. We then record any TLS *Alert Messages* that appear. It is crucial that a connection from the same TLS instance is triggered from a device every time a root CA is investigated. Otherwise, we cannot know if our exploration is targeted towards one root store or multiple root stores on the same device. For our experiments, our expectation is that devices will follow the same procedure every time they are rebooted.

To obtain a set of CA certificates to spoof, we gathered historical data for CA certificates trusted by various platforms through the sources described in Table 3.3. We use this data to make two distinct set of certificates:

1. *Common CA certificates*: we use the latest version of the root store for each platform and extract currently unexpired certificates common to all of them.
2. *Deprecated CA certificates*: we start with the earliest version of the root store for each platform, and extract all certificates removed from the successor version(s) of the store, but that are currently unexpired. We exclude any certificate if it was once removed but is still present in the latest version of the root store.

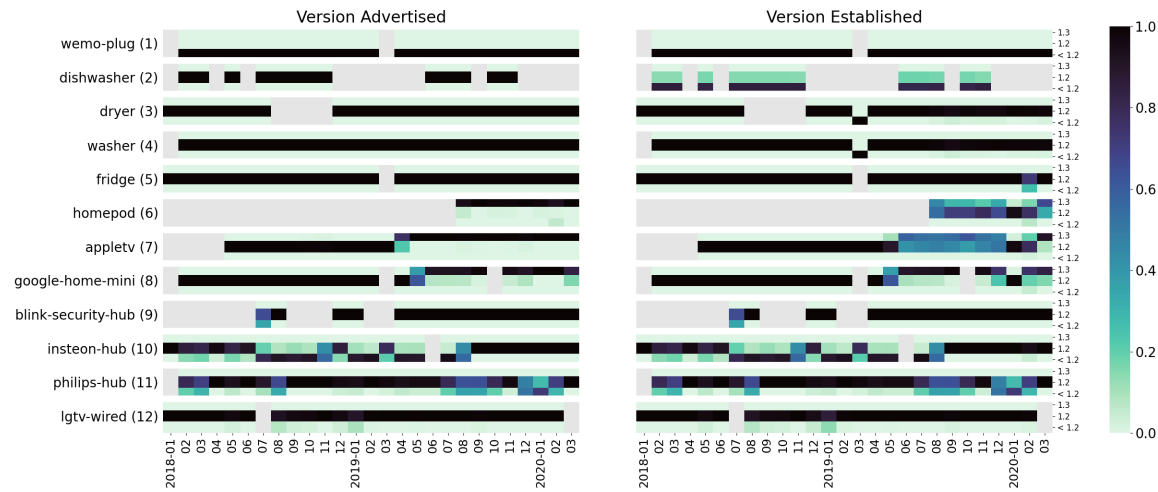


Figure 3.2: TLS version support for IoT devices. Devices often use multiple versions (rows 2-12), can encounter lack of server support (rows 2-8) and rarely adopt better TLS versions over time (rows 7-10). 28 devices use TLS 1.2 for the vast majority of their advertised and established connections, and are not shown in this figure.

Common CA certificates represent the ones trusted by all major (non-IoT) platforms, and thus can be considered likely trustworthy. *Deprecated CA certificates* represent cases where root certificates are retired before expiration, or in some cases explicitly distrusted (e.g., due to noncompliance with CA guidelines), and thus their trustworthiness is (more) questionable. Note that our approach cannot in general reveal all certificates in the root store; rather, it can reveal only those included in our testing set. As such, our analysis may omit non-public root, such as those in private PKIs.

We validated the efficacy of our approach in popular TLS libraries and present results in Table 3.4. Among the 2/6 libraries that are amenable to this analysis, *MbedTLS* is generally known to be deployed in IoT ecosystems [22]. As we show later in our results, we empirically found that *OpenSSL* is used by multiple devices that are also amenable to this measurement strategy.

3.3 Results

We now present the results to answer our research questions.

3.3.1 TLS Connection Security

In this section, we rely on more than two years of *passively* collected data to study TLS protocol version and ciphersuites, and whether their support improves over time.

Protocol Version As explained earlier, the TLS version used in a connection is determined during the handshake and is based on the highest version supported by both client

Table 3.5: IoT devices that *downgrade* security upon connection failures (✓ indicates downgrade).

Device	Failed Handshake	Incomplete Handshake	Behavior	Downgraded / Total Destinations
Amazon Echo Dot	✗	✓	Falls back to using SSL 3.0	7 / 9
Amazon Echo Plus	✗	✓	Falls back to using SSL 3.0	6 / 7
Amazon Echo Spot	✗	✓	Falls back to using SSL 3.0	11 / 15
Amazon Fire TV	✗	✓	Falls back to using SSL 3.0	13 / 21
Apple Homepod	✗	✓	Falls back to using TLS 1.0	7 / 9
Google Home Mini	✗	✓	Falls back to supporting a weaker ciphersuite and signature algorithm (TLS_RSA_WITH_3DES_EDE_CBC_SHA and RSA_PKCS1_SHA1)	5 / 5
Roku TV	✓	✓	Falls back from offering 73 ciphersuites to just 1 (TLS_RSA_WITH_RC4_128_SHA)	8 / 15

and server. Since versions prior to 1.2 are deprecated due to security concerns, we focus on the prevalence of such connections in our dataset, and whether their use is due to lack of client and/or server support for newer versions.

Our first observation is good news. A large majority of the devices (28/40) use TLS 1.2 exclusively and are thus not using deprecated versions. However, for other devices, we find a mix of traffic that includes the use of deprecated TLS versions over time.

To visualize this phenomenon and understand how it impacts the security of established connections, in Fig. 3.2 we visualize a heatmap of the fraction of connections for which each TLS version is *advertised* via *Client Hellos* (left), and *established* via *Server Hellos* (right) over a 2-year period. For each device (y-axis), we use three rows to represent the TLS connections observed over 1.3 (top), 1.2 (middle) or older versions (bottom). Each cell represents the fraction of TLS connections over each TLS version during a particular month of our study (x-axis). Gray cells indicate months where a device did not generate any TLS traffic. Note that Fig. 3.2 omits the 28 devices that established connections using only TLS 1.2. We make the following observations:

The vast majority of connections happen over TLS 1.2. Only the *Wemo Plug* advertises an insecure TLS version throughout the entire measurement period for all its connections.

Devices tend to support newer protocol versions than the servers they connect to. We find that 32 devices *advertised* support for TLS 1.2 in more than 95% of their connections every month; however, only 24 *established* connections consistently with TLS 1.2. For example, the *LG Dishwasher*, *Samsung Dryer*, *Samsung Washer*, *Samsung Fridge* devices advertise TLS 1.2, and the *Apple Home Pod* and *Apple TV* devices advertise TLS 1.3, but all of them establish connections using older protocol versions. The finding highlights that the security of TLS connections from IoT devices in many cases is limited by servers rather than the devices themselves.

Table 3.6: IoT devices that support older TLS versions.

Device	TLS 1.0 Available?	TLS 1.1 Available?
Zmodo Doorbell	✓	✓
Wink Hub 2	✓	✓
Yi Camera	✓	✓
Philips Hub	✓	✓
Smarter Brewer	✓	✓
TP-Link Bulb	✓	✓
Roku TV	✓	✓
Meross Dooropener	✓	✓
LG TV	✓	✓
Google Home Mini	✓	✓
Amazon Fire TV	✓	✓
Amazon Echo Spot	✓	✓
Amazon Echo Plus	✓	✓
Amazon Echo Dot	✓	✓
Amcrest Camera	✓	✓
Samsung Fridge	✗	✓
Samsung Dryer	✗	✓
Wemo Plug	✓	✗

Devices rarely upgrade to newer protocol versions. The vast majority of devices supported the same TLS versions during the two-year study. The exceptions are *Apple TV* and *Google Home Mini*, which transitioned to using TLS 1.3 (5/2019), and the *Blink Security Hub*, which transitioned to TLS 1.2 (7/2018), for the majority of its advertised connections. (TLS 1.3 was finalized by IETF in 8/2018.)

We do not have ground truth to indicate whether changes in advertised TLS versions are due to TLS software upgrades on the device or due to connections established using a different existing TLS instance on the same device. For the three cases above, we believe they are likely software upgrades because the new protocol versions are used exclusively after the transition. In contrast, the *Insteon Hub* appeared to downgrade its advertised and established connections to older TLS versions for a brief period of time (7/2018–8/2019). We manually inspected these cases and found that changes in fractions of connections using older TLS versions were explained by a single set of destinations that were contacted more or less frequently from one month to the next. As such, we do not believe these were due to any TLS software changes. Note, however, that the transition to TLS 1.2 (9/2019) is more likely due to an upgrade in protocol support because older TLS versions are not seen at all after this date.

Devices that advertise multiple maximum TLS versions. We find that 20 devices advertise support for more than one TLS version, with 15 of those advertising multiple maximum versions for the same destinations. This was surprising, since a device with a more secure configuration would advertise only the most recent TLS version as its maximum. There are several potential explanations for this behavior. One explanation could be that different

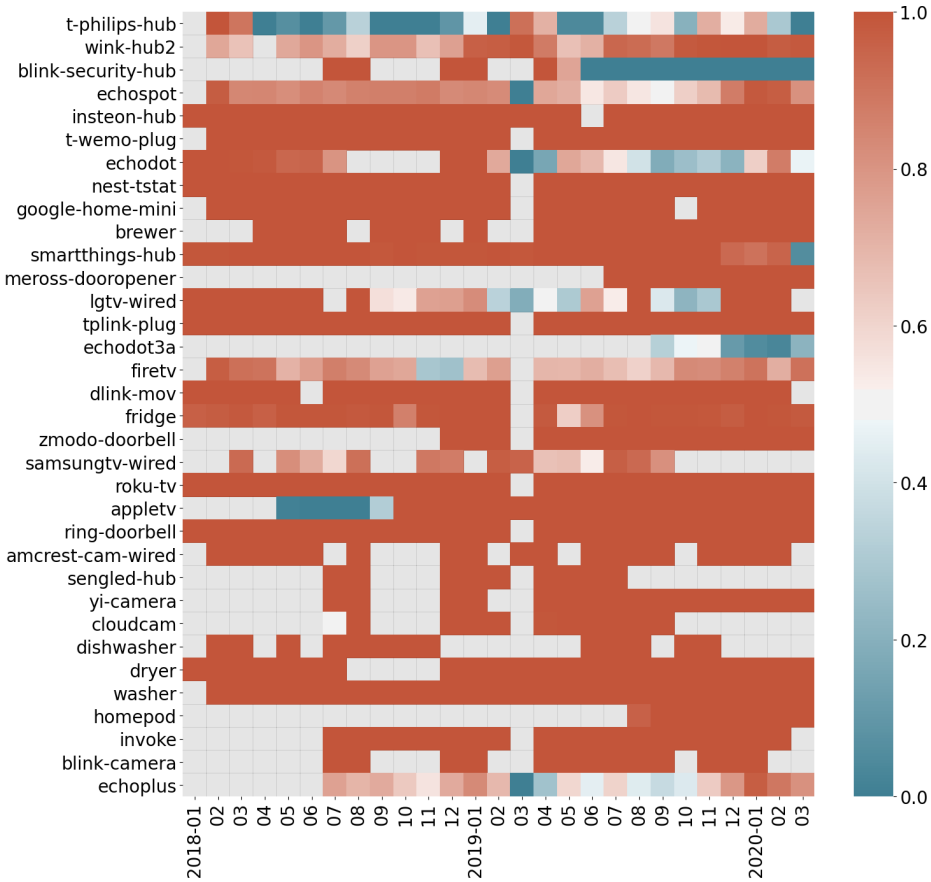


Figure 3.3: IoT devices that *advertise* handshakes with insecure ciphersuites (lower is better). Most devices do not deprecate these ciphersuites over time. 6 devices rarely advertise such ciphersuites, and are not shown in this figure.

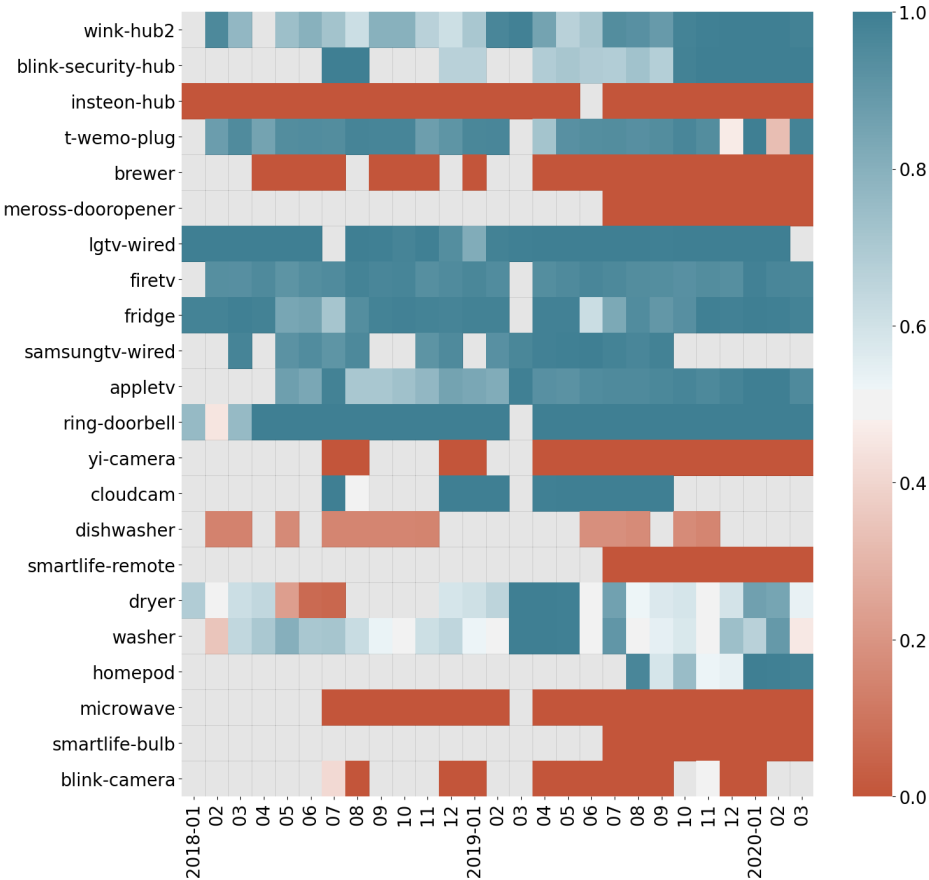


Figure 3.4: IoT devices that *establish* connections with strong ciphersuites (higher is better). Most devices do not adopt these ciphersuites over time. 18 devices use such ciphersuites for the vast majority of their established connections, and are not shown in this figure.

IoT device functionality (e.g., third-party software) uses the same TLS implementation but different configurations. In this case, we hypothesize that connections to different parties would consistently use different TLS configurations. To test this, we labeled each TLS connection as first or third-party using an approach inspired by Ren et al. [51]. We found no patterns that indicate bias toward one TLS version depending on the destination type contacted, and thus we found no evidence to support this hypothesis. Another explanation is that each device contains multiple TLS instances and different software components of a device use them independently. While we do not have any ground truth to confirm it, the observed behavior is consistent with this explanation. We explore this behavior and its implications further in §3.3.3.

Connection security under attacks The results above focus on connection security observed passively, and only shed light on the maximum advertised protocol version from devices. To better understand the susceptibility of these devices to even weaker security in the face of an active on-path attacker, we conducted active experiments that attempt to force devices to downgrade connection security through connection failures, or negotiate connections with older TLS versions by using them in *ServerHellos*.

We ran experiments using two types of TLS connection failures; *IncompleteHandshake* where we do not reply to a *ClientHello* with *ServerHello*, and *FailedHandshake* where we use a self-signed certificate to cause an unsuccessful handshake. Table 3.5 lists the 7 devices that downgrade security upon connection failures, the types of handshake errors that lead to downgrades, how security was downgraded, and how many destinations were susceptible. The most likely reason for such behavior is that clients intentionally want to maximize compatibility with old servers. Interestingly, the majority of—but not all—destinations (i.e., unique domains identified via SNI or DNS) for a device are affected by downgrades.

The exception is the *Google Home Mini*, which is susceptible to downgrades on all its connections. The most significant downgrade that we observed was the fallback to SSL 3.0 (which is vulnerable to the POODLE attack) in 4 devices, all from the *Amazon* family.

Next, we investigate which devices support TLS versions older than 1.2 and will establish connections using those older versions, if triggered to do so. Table 3.6 lists the 19 devices that support TLS versions older than 1.2. We note that despite the large number of these devices, TLS 1.2 was the most common protocol seen in *established connections* from passive data. As such, the finding highlights that completely protecting against active attackers requires devices to not only advertise TLS 1.2, but also completely disable support for older TLS versions.

Ciphersuites Similar to the protocol version, the selection of a connection’s ciphersuite also happens during a connection handshake and depends on client and server compatibility. For a connection to follow best security practices, *strong* ciphersuites that offer forward-secrecy (DHE, ECDHE) should be chosen, while those that are either *insecure* (RC4, DES, 3DES, EXPORT) or do not offer encryption or authentication (ANON, NULL) must be avoided.

To study the prevalence and client/server support for these ciphersuites, we plot heatmaps for the advertised and established ciphersuites over time. Each row represents a device,

where each cell is the fraction of connections that are insecure (Fig. 3.3) or strong (Fig. 3.4) for a given month of the study. As before, gray cells indicate months where there was no TLS traffic from the device. We make the following observations:

Devices never support (ANON, NULL) ciphersuites. We did not observe any TLS connection advertised or established using these.

Devices support weaker ciphersuites than the servers they talk to. 34 devices advertised insecure ciphersuites (Fig. 3.3) but only 2 ever established connections using those (*Wink Hub 2* and *LG TV*). In contrast to support for TLS versions, the devices in our study generally offered to use weaker security than what servers chose to establish.

Devices tend to have better support for perfect forward secrecy than the servers they connect to. 33 devices advertise support for forward secrecy, but a large majority of devices (22) establish most of their connections without it (Fig. 3.4).

Devices rarely improve usage of ciphersuites over time. Only 2 devices (*Blink Security Hub - 5/2019*, *SmartThings Hub - 3/2020*) stopped advertising/using weak ciphers during our two-year study (Fig. 3.3), while 5 (*Apple HomePod - 1/2020*, *Ring Doorbell - 4/2018*, *Apple TV - 3/2019*, *Wink Hub & Blink Security Hub - 10/2019*) adopted perfect forward secrecy (Fig. 3.4). Surprisingly, Apple TV (10/2018) appeared to increase support for weak ciphers over time.

Devices show varying support for ciphersuites during multiple months. Many devices support insecure ciphersuites in a fraction of their connections as opposed to all or none. Similar to the case with protocol version, the varying support suggests the presence of multiple TLS instances in a device.

Comparison with prior work We now compare TLS versions seen from the IoT devices in our testbeds with those observed in prior work. Note that prior work [29], [30] looked at all traffic from a network provider, not only IoT devices. Specifically, when looking at North American vantage points in November, 2019, a recent study [29] found that $\approx 60\%$ of client connections support TLS 1.3, while our study found only $\approx 17\%$ of IoT device connections support TLS 1.3. In April, 2018, Kotzias et al. [30] found that $\approx 10\%$ connections advertise RC4 ciphersuite support while we find $\approx 60\%$ of connections do. Relative to other sources of Internet traffic such as browsers, IoT devices and their online infrastructure are slow to adopt modern protocol features and to deprecate insecure ones.

Takeaways Our longitudinal study revealed good and bad news about TLS usage in IoT devices. On the positive side, the IoT devices in our study often rely on TLS1.2 or above, do not support (NULL, ANON) ciphersuites and often support better protocol versions than the servers they connect to. On the negative side, many of the devices in our study do not use the latest protocol version, still support some weak ciphersuites, and tend to not upgrade to modern protocol features over time. Our findings suggest that although most IoT devices establish reasonably secure TLS connections, device manufacturers can improve when it comes to maintaining updated TLS libraries and configurations over time. This will help to reduce their exposure to attacks over time.

Table 3.7: IoT devices vulnerable to TLS *interception* attacks. (✓ indicates vulnerability).

Device	No-Validation	InvalidBasic-Constraints	Wrong-Hostname	Vulnerable/Total Destinations
Zmodo Doorbell	✓	✓	✓	6 / 6
Amcrest Camera	✓	✓	✓	2 / 2
Smarter Brewer	✓	✓	✓	1 / 1
Yi Camera	✓	✓	✓	1 / 1
Wink Hub 2	✓	✓	✓	1 / 2
LG TV	✓	✓	✓	1 / 2
Smartthings Hub	✓	✓	✓	1 / 3
Amazon Echo Plus	✗	✗	✓	1 / 8
Amazon Echo Dot	✗	✗	✓	1 / 9
Amazon Echo Spot	✗	✗	✓	1 / 17
Amazon Fire TV	✗	✗	✓	1 / 21

3.3.2 Certificate Validation

In this section, we use active experiments to evaluate how well IoT devices validate TLS certificates for the connections they establish. It is important to note that failure to properly validate certificates makes devices susceptible to interception attacks, where the attacker can recover the plaintext content of encrypted connections. To understand the correctness of certificate validation, we test three aspects. First, we identify whether devices are susceptible to interception attacks via the techniques presented in Table 3.2. Second, we determine whether devices conduct certificate revocation checking. Last, we evaluate our novel probing strategy to reveal the set of trusted root CAs and determine whether devices continue to trust unexpired root certificates that have been deprecated, particularly focusing on *distrusted* certificates.

Invalid certificates We begin by understanding whether devices perform validation correctly when presented with invalid certificates (Table 3.7). In summary, *seven devices do not perform any certificate validation* and are thus vulnerable to traffic interception. Four other devices (all from the *Amazon* family) do not check for correct *Common Name* in certificates and we were thus able to decrypt their TLS traffic using a free certificate obtained from *ZeroSSL* for a domain under our control. Interestingly, the *Yi Camera* disables certification validation completely upon 3 consecutive failed connections.

Through manual inspection of successfully intercepted TLS connections, we found that 7/11 devices transmitted potentially sensitive data to first-party destinations (e.g., “encrypt_key” for *Zmodo Doorbell*, “command server” for *Amcrest Camera*, “deviceSecret” for *LG TV* and “bearer” authentication tokens for *Amazon* devices). This provides strong evidence that lack of certificate validation can have implications for user and device security/privacy.

Interestingly, we also found that 7/11 vulnerable devices (Table 3.7, column 5) initiated TLS connections to other first or third-party destinations that were *not* vulnerable (likely

Table 3.8: Exploring the root stores of 8 IoT devices. Each cell denotes the number of root certificates present in an IoT device over the number of certificates whose inclusion could be successfully checked.

Device	Commonly-trusted certs (total = 122)	Deprecated certs (total = 87)
Google Home Mini	100%	6%
Amazon Echo Plus	98%	18%
Amazon Echo Dot	98%	19%
Amazon Echo Dot 3	90%	27%
Wink Hub 2	92%	38%
Roku TV	91%	41%
LG TV	93%	59%
Harman Invoke	82%	59%

due to the presence of multiple TLS instances—we explore this behavior and its implications further in §3.3.3).

Revocation Checking An important aspect of establishing secure connections is for clients to determine whether the server certificate for a connection has been revoked. To test whether devices perform such checks, we use passive data to look for communication with standard revocation endpoints (CRLs, OSCP servers), requests for OCSP staples in *ClientHellos* and presence of *Must Staple* extension in certificates. We find that a large majority of devices (28) do not ever conduct certificate revocation checks, and thus only 12 devices ever attempt to check for revocation for any of the certificates received throughout the measurement period. Of those devices, 11 support OCSP Stapling but never encounter a certificate with a *Must Staple* extension. We conclude that the IoT ecosystem provides only limited support for revocation checking, similar to what has been observed by prior work in desktop and mobile browsers [57].

Root Stores When devices continue to trust deprecated or distrusted (and unexpired) CA certificates, they can become susceptible to interception attacks against all destinations if an attacker obtains the corresponding secret key. We now investigate the extent to which IoT devices are vulnerable to this issue.

We use the methodology introduced in §3.2.1 to detect the inclusion of deprecated-yet-unexpired root store CAs in IoT devices. We excluded appliances not suitable for repeated reboots (i.e., *Washer*, *Dryer*, *Thermostat*, *Fridge*) and the devices that did not validate certificates in any of their TLS connections. For 8/24 remaining devices in the testbed, our methodology successfully triggered different *Alert Messages* to enable root stores exploration.

A summary of the results is provided in Table 3.8. In some cases, our experiments were inconclusive in determining the inclusion of a particular certificate (e.g., if the device did not generate any traffic). We exclude such cases and present the total number of certificate inclusions divided by the total number of successful experiments for each device in the table. We find the majority of unexpired certificates common to all platforms to be present in

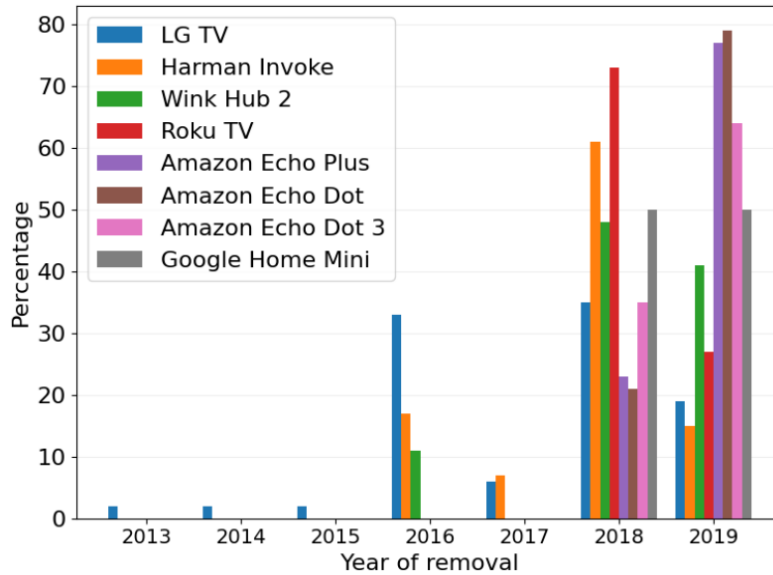


Figure 3.5: For deprecated CA root certificates still present in IoT devices, we track their year of removal from major platforms.

all devices probed (second column in the table). This is good news, as it suggests that IoT devices, web browsers, and OSEs trust a similar set of (presumably trustworthy) CA certificates.

Interestingly, however, all devices also contain at least one deprecated-yet-unexpired root certificate, i.e., that has already been removed from one or more major platforms. With the exception of the Google Home Mini, the IoT devices we tested contain significant fractions (if not a majority) of root certificates that were deprecated from other platforms.

To understand how long such deprecated-yet-unexpired root certificates remain in device root stores, we plot the *staleness* of each root certificate in terms of the year it was removed from one of the four reference platforms in Figure 3.5. (If a certificate was removed from multiple stores, we use the latest year of removal.) Devices with large numbers of certificates that were removed years ago are either not updating their root stores or not interested in deprecating certificates.

We find that the majority of observed stale root certificates were deprecated in the years 2018 and 2019, likely biased by the fact that the devices were manufactured at or shortly before those years. Surprisingly, we find that one device (*LG TV*) contains unexpired root CAs that were deprecated as early as 2013. We note that the devices in our testbed were able to receive regular updates during our study. More specifically, *LG TV* was last updated in July 2019 and *Roku TV* in September 2020, while the bulk of our experiments were performed in 2021. Other devices such as voice assistants from Google and Amazon receive updates automatically as long as they are connected to the Internet. This suggests that some manufacturers are not updating root stores at the same cadence (if at all) as other software updates.

A root certificate that is deprecated is not necessarily untrusted, as some may be removed for “administrative” reasons such as regular key rotations (e.g., [58]). However, the *TurkTrust* (2013) and *Certinomis* (2019) CAs were explicitly distrusted by Mozilla while *CNNIC* (2015) and *WoSign* (2016) are in the Google blacklist due to a failure to comply with CA guidelines (e.g., *TurkTrust* was responsible for an unauthorized certificate for `google.com`) [59]–[61]. Arguably these root certificates should not be trusted by any devices. Surprisingly, we found that one or more of these CAs explicitly distrusted by various platforms were *still trusted by all devices*. The fact that these root certificates remain trusted by devices can open them to arbitrary interception attacks if the private key for those certificates were shared with adversaries (*WoSign* incident [62]).

We note that these IoT devices tend to contact a small set of destinations, but nonetheless contain root stores used by web browsers/OSes that are expected to contact arbitrary destinations. An important question is whether these devices all need to use such large root stores, or instead some of the devices can reduce their trusted set of certificates to cover only the destinations that are required for the device.

Takeaways 28 IoT devices show some form of certificate validation limitations. Some devices skip certificate validation altogether and most do not bother to check for revoked certificates. All of the affected TLS connections were contacting first-party destinations. We conclude that even popular IoT devices from major manufacturers exhibit poor TLS validation for at least some of their connections. Further, all of the devices that we could successfully probe for root certificates contained at least one that was deprecated and distrusted, despite the fact that the devices themselves install regular updates. These deprecated CAs root certificates—particularly ones that are distrusted—can be perceived as the weakest link in TLS security for IoT devices.

3.3.3 Diversity of TLS Behavior

In this section, we explore the diversity of TLS behaviors observed for individual devices and across devices. The goal of this analysis is to shed light on how IoT devices use shared or different TLS implementations and configurations, and the potential ramifications on security.

Our primary investigative tool is *TLS fingerprinting*; namely, we generate TLS fingerprints for 32 devices and compare them to a publicly available database of 1,684 fingerprints that covers a wide variety of sources such as different browsers, multiple versions of TLS libraries, and malware samples [30]. Each fingerprint is labeled with the *application* that generated it (e.g., *OpenSSL*, *curl*, *android-sdk*). We generate fingerprints for the TLS connections from our devices in the same way as done during the database compilation. Since devices can update their libraries and that may affect the corresponding fingerprints, here we only study TLS traffic from active experiments that represent a snapshot in time.

Devices with more than one TLS fingerprint. For 18/32 IoT devices in our experiments, we found only a single TLS fingerprint per device, likely due to the use of a single TLS instance. This can simplify TLS security management by having only one instance to maintain. However, we found that 14/32 devices had connections with more than one

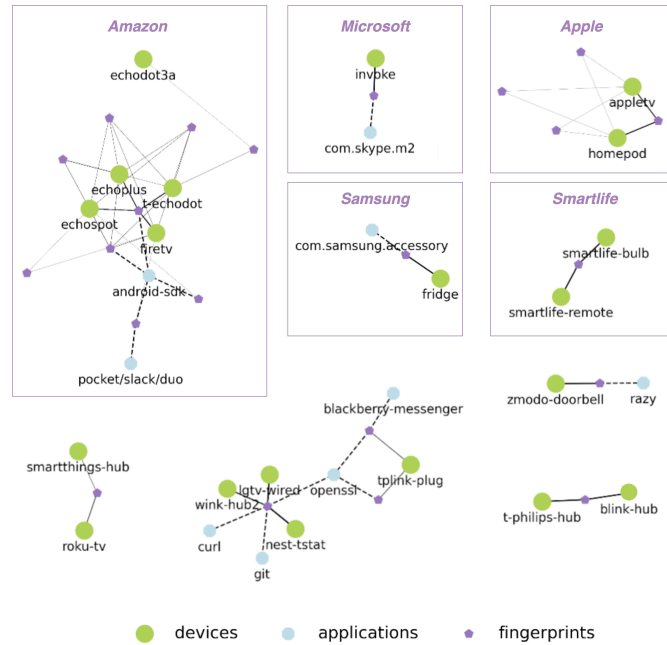


Figure 3.6: IoT devices that likely share TLS libraries with other devices and applications.

fingerprint, indicating the presence of multiple instances. This can help explain the mixed support for TLS connection security and the presence of TLS interception vulnerabilities in some (but not all) connections from a device.

While we do not know why there are multiple TLS instances on a single device (because we lack access to firmware for these devices), one conjecture is that these devices may contain different first- and third-party components, each using different TLS instances. These components can come from a variety of sources such as user-installed software (e.g., app stores) or the usage of multiple frameworks during software development (e.g., Golang, Java, and Python come pre-bundled with different TLS instances). If true, such behavior can make it harder to maintain TLS security over time, as both device manufacturers and other developers need to secure and maintain all of these instances.

TLS fingerprints shared across devices. We find that 19 devices share at least one TLS fingerprint with *other* devices and/or applications (e.g., *OpenSSL*). This is likely because multiple devices share the same (and in many cases, open source) TLS library.

To better understand the nature of shared TLS instances, we produced a graph of devices and applications with the same fingerprints. There are three types of nodes in the graph: devices (from our study) and applications (from Kotzias et al.[30]) that generate TLS fingerprints, and the set of unique fingerprints that are shared among them. Edges between a device/application and fingerprint indicate that we observed a device or application using that fingerprint. Figure 3.6 visualizes this graph. In the figure, the thicker edges correspond to the most-used fingerprint (and likely, the most-used TLS instance) for each device. Note that the graph includes an edge only if the TLS fingerprint it connects to is shared with at least one other node, i.e., all non-shared fingerprints and edges are removed from the figure to

improve readability. Dashed edges represent a fingerprint shared with a labeled application from Kotzias et al.[30], and thus they do not represent observed traffic in our study.

Our first observation is that devices and applications from the same manufacturer share fingerprints—this can be observed with labeled clusters (e.g., *Amazon*, *Microsoft*, and *Apple*). It is not surprising that these devices are likely using the same TLS instances, but it nonetheless could be good news for maintaining security because it indicates that the manufacturer likely needs to maintain one set of TLS instances across devices. These shared instances also suggest that many of our findings apply to other devices belonging to the same manufacturers that are not in our testbed.

Our next observation is about devices that share fingerprints with applications in the fingerprint database. For example, the dominant fingerprint from Amazon Fire TV is the same as one from *android-sdk*, and we verified that the device runs a fork of Android OS [63]. Similarly, six devices exhibit the same TLS fingerprints as the *OpenSSL* library, likely indicating that *OpenSSL* is used on those devices. This helps to explain why our technique for root stores exploration worked for *Invoke*, *LG TV*, and *Wink Hub 2*: despite being produced by different manufacturers, they all share fingerprints with *OpenSSL*—one of the two libraries we found amenable to the root stores exploration technique.

While we pointed out above that shared TLS instances can be good in the sense that they are easier to maintain, sharing can also be a double-edged sword. Specifically, a security vulnerability in one TLS instances can immediately impact large numbers of devices. For example, in the TLS certificate validation analysis, we found that *Amazon* devices fall back to TLS 1.0 during a downgrade attack. The TLS fingerprinting analysis shows that this is likely because they share the same vulnerable implementation. (Interestingly, the *Echo Dot 3* is the only *Amazon* device in our testbed not susceptible to the downgrade attack, and its fingerprints have smaller overlap with those from other *Amazon* devices.) Importantly, our observations hint at a way for an attacker to scale attacks by identifying and exploiting vulnerable TLS implementations that are shared among multiple devices.

Takeaways IoT devices show similarity of TLS fingerprints with (i) other devices from the same manufacturer (e.g., all *Amazon* devices), and (ii) various TLS clients (e.g., *LG TV* and *Wink Hub 2* with *OpenSSL*)—suggesting that our findings apply to many more devices not tested in our experiments, and that security vulnerabilities found in one instance can affect large numbers of devices. We also found that multiple TLS instances are deployed in the same device in many cases, potentially making it difficult to maintain TLS security over time.

3.4 Discussion

Recommendations Client support for TLS security has been an underexplored area in recent research. Our findings, however, paint a complex picture of connection security and certificate validation in connections from IoT devices. For instance, some devices support the latest secure TLS features but still negotiate weak connections due to lack of server support. Similarly, some devices fail to validate certificates, but only for some connections. Device

root stores are infrequently updated (if at all), and several devices likely include multiple TLS instances.

The user risks due to insecure/incorrect TLS implementations in their IoT devices are similar to the risks for any other systems using TLS, such as web browsers and other apps. For example, MITM attacks may be carried out *not only* by any on-path attackers (e.g., a malicious router), but by other devices on the same user network as well, such as a malicious IoT device using ARP spoofing. If the attack is successful, it can expose potentially sensitive user data, such as microphone data from a smart speaker or login credentials.

To mitigate this, our key recommendation to consumer IoT device manufacturers is to audit, upgrade and maintain their devices' TLS instances in a consistent and uniform way that safeguards all of their network traffic. One way to do this is to provide TLS as an operating system service (i.e., POSIX socket call) as proposed by O'Neill et al. [64]. Multiple components within a device, and multiple devices in the IoT ecosystem can then use the service to enable TLS in a consistent way. In a similar vein, we encourage industry groups like the IoxT alliance [65] to incorporate TLS security standards into their guidelines for manufacturers to follow, as well as verification tests. In fact, the IoxT alliance can also join the CA/Browser Forum consortium [66] to adopt the same standards as web browsers when it comes to trust in root certificates.

IoT devices can also rely on certificate pinning, a technique to mandate the use of particular certificates in the chain sent by a server, to mitigate some of the vulnerabilities found in our study. More specifically, the interception attacks we presented (Table 3.7) could have been prevented with the proper use of certificate pinning. But it is important to highlight that certificate pinning is not a panacea—pinning can help only in cases of compromised root stores if the leaf certificate is pinned (rather than the root). Further, certificate validation checks are necessary even if pinning is implemented. Otherwise, devices might appear secure but will remain susceptible to sophisticated MITM attacks (e.g., [67]).

An internal or third-party auditing service can also help IoT vendors keep their TLS instances up-to-date with the evolving security recommendations. IoT devices can be configured to create TLS connections to the auditing service at regular intervals (e.g., once every reboot). The service can then audit the security of the connections (e.g., ciphersuites offered by the device during handshake). As new attacks are discovered, the service can contact manufacturers to alert them about new vulnerabilities and mitigations.

Another possible mitigation strategy that IoT users can use is to interpose a trusted network component between their IoT devices and the Internet, similar to the one proposed by Hesselman et al. [67], to verify that TLS connections are being securely established. If such verification fails, the component pauses the connection and reports the issue to the user, which is left with the choice whether to allow the insecure TLS connection or not, as it happens for web browsers.

Limitations Our study had several limitations. First, we chose a limited number of devices to make the scope of our experiments practical. As such, our results are biased by the selection of (a) popular consumer devices, and (b) multiple devices from the same manufacturer. Second, our choice of TLS interception attacks reflected the ones that are easily

exploitable by an in-network adversary. Other sophisticated attacks that use cryptanalysis on a sufficiently large amount of network traffic (e.g., POODLE, SWEET32) are difficult to mount (e.g., need JavaScript injection to repeatedly trigger requests) but could nonetheless compromise TLS security in some IoT devices. Third, the coverage of our analyses could be improved by (i) relying on techniques from other works to automate device interactions (e.g., using smartphones [68], reverse-engineering exposed APIs [69]), and (ii) inspecting source-code when possible (e.g., firmware extraction from memory, rooting Android-based devices, crawling third-party marketplaces).

Unfortunately, all these techniques require device-specific efforts and do not generally scale well to other devices. Finally, our technique to explore root stores does not generalize for all devices. One reason is that some implementations choose to not send any TLS alerts over connection failures. Moreover, unlike TLS 1.2, which mandated the usage of “appropriate” alerts on encountering fatal errors, TLS 1.3 made it optional. This motivates the need to search for better techniques to exploit the side-channel and explore root stores in more IoT devices.

Responsible disclosure We contacted manufacturers of the 11 IoT devices to responsibly disclose our successful *interception* attacks (Table 3.7). Unlike other devices that showed weaknesses due to stale root stores or compatibility with older protocol versions and weaker ciphersuites, these devices had vulnerabilities severe enough that we were able to actively exploit them and extract decrypted TLS communications from their first-party connections.

Ethical considerations This study involved human subjects that participated after completing informed consent materials that are part of our IRB-approved study. No personal or sensitive data about individuals is collected as part of this study. The active experiments exploited vulnerabilities only for the devices in our lab, and we did not use any information gleaned from these experiments to attack other devices or cloud services.

3.5 Conclusion

Our work fills an important knowledge gap in our understanding of TLS behavior from consumer IoT devices using more than two years passive measurements along with active experiments to reveal TLS vulnerabilities. We find a wide range of security-related TLS behaviors ranging from good (a large majority of tested devices use TLS 1.2 or higher), to bad (more than half of the devices advertise deprecated TLS versions or insecure ciphersuites in a significant fraction of their connections), and critically flawed (11 devices are vulnerable to TLS interception attacks because they do not properly validate server certificates). Further, we find that devices are slow to adopt new TLS versions and to secure the set of supported ciphersuites, and they also rarely remove deprecated and distrusted CA certificates from their root stores.

Finally, we used TLS fingerprinting to identify cases where individual devices use multiple distinct TLS instances, and those where different devices use the same TLS instances—each with implications for security, e.g., shared vulnerabilities that can facilitate attack scaling. We conclude that TLS clients in IoT devices have much room for improvement, and we

recommend that manufacturers adopt uniformly secure TLS instances and industry standards [65], and conduct regular auditing and updating to ensure their devices' connections remain secure.

To ensure reproducibility and enable new research, we have made all of our longitudinal TLS handshake data, controlled experimentation data and analysis software publicly available at: <https://github.com/NEU-SNS/IoTLS>.

Chapter 4

Web Content Availability and Consistency over HTTP/S

The importance and ease of deploying HTTPS websites has increased dramatically over recent years. Recent studies of HTTPS adoption have found rapidly increasing adoption, leading some to speculate that most major websites will soon be able to redirect all HTTP requests to HTTPS [70]. Indeed, some client-side tools even go so far as to force *all* web requests to be made via HTTPS [71].

However, to our knowledge, all previous work measuring the deployment of HTTPS [70], [72]–[75] makes two basic but fundamental assumptions: (1) They assess server-side HTTPS support by looking at a single page: the domain’s *landing page* (also known as its root document, “/”), and (2) They assume that any resource that is available over both HTTP and HTTPS has the same content (in the absence of an attack, of course)—that is, that the only difference between `http://URI` and `https://URI` is that the latter is served over TLS.

Unfortunately, neither of these assumptions has been empirically evaluated. This is important because, if they do not hold, they threaten our understanding of how HTTPS is truly deployed. For example, the fact that a landing page is (or is not) secure might not necessarily indicate the security of the site writ large. Moreover, if there are content differences between HTTP and HTTPS, then merely defaulting to the more secure variant—as many papers and tools have suggested—risks unexpected side-effects in usability.

In this chapter, we empirically show that this conventional wisdom does not universally hold (Fig. 4.1). We identify inconsistencies between HTTP and HTTPS requests to the same URI, in terms of content unavailability over one protocol and content differences between them. Rather than restrict our study to landing pages, we conduct a deep crawl (up to 250 pages) of each of the Alexa top 100k sites, and for a 10k sample of the remaining 900k sites on the Alexa top 1M list [76]. We then analyze the retrieved pages to identify substantial content inconsistencies between HTTP and HTTPS versions of the same page, using a novel combination of state-of-the-art heuristics.

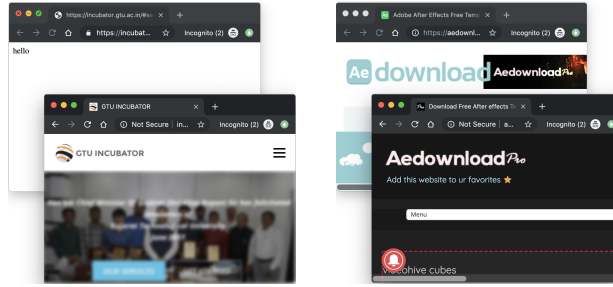


Figure 4.1: Examples of webpages with different content using HTTP vs HTTPS.

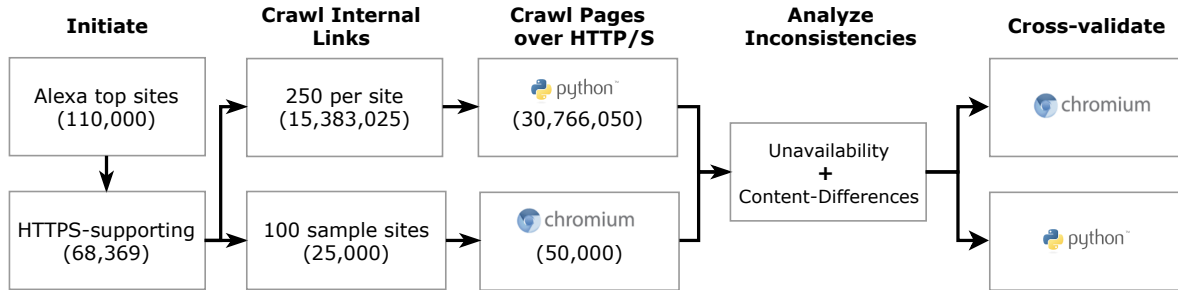


Figure 4.2: Summary of our pipeline. We use a custom crawler for the full crawl, but rely on a real browser to detect any under/overestimation of inconsistencies.

4.1 Methodology

This section describes our measurement methods for crawling websites and identifying inconsistencies.

4.1.1 HTTP/S Inconsistencies

We focus our study on websites that support both HTTP and HTTPS on the landing page. For this set of sites, we measure whether there are different results when accessing the same resource over HTTP and HTTPS. We categorize such cases of *inconsistencies* between protocols as follows:

Content unavailability This occurs when a resource is successfully retrieved via HTTP, but not HTTPS (i.e., HTTP status code ≥ 400 ,¹ or an error²).

Content difference This occurs when a resource is available over both HTTP and HTTPS, but the content retrieved differs significantly (as defined in §4.1.3).

4.1.2 Crawling Overview

We limit our analysis to differences in HTML content served by a website, which we refer to as *pages*, and do not consider embedded resources (e.g., CSS and Javascript files). When crawling a website, we conduct a depth-first search of all links to other webpages with the

¹In Section 4.2, we show how some websites rely only on splash pages to report a failure, instead of also returning an error status codes.

²Except for timeout or connection-reset errors.

same second-level (TLD+1) domain—which we call *internal links*)—starting at the landing page for the site.

The websites we crawled are a subset of the Alexa global top 1 M domains. We chose this list because it represents sites visited by users via web browsers [77]. Our analysis covers all of the top 100K most popular domains. We supplement this set with a randomly selected 10K sample of the 900K least popular domains, since rankings beyond 100K are not statistically meaningful³.

For each domain, we crawled at most 250 internal links to limit the load on websites induced by our crawls while still covering significant fractions of site content. We could not identify an efficient way to inform this limit empirically, and picked this number to keep the crawl duration at an acceptable length. Note that we consider only sites where the landing page is accessible using HTTPS (and apply the same restriction to subdomains of a site).

To conduct the crawl, we developed a (i) Python-based crawler using the Requests [79], BeautifulSoup [80] and HTML5Lib [81] libraries, and a (ii) Chromium-based crawler using the Chrome DevTools Protocol. The former implementation does not attempt to render a page by executing JavaScript and/or fetching third-party resources, and is thus considerably faster, enabling us to crawl more pages in a limited set of time; we use it for finding the inconsistencies across all websites and rely on a real browser only for cross-validating our results. The crawler visited each page using both HTTP and HTTPS on their default ports (unless otherwise specified in an internal link, e.g., via a port number in the URL). A summary of the pipeline is presented in Fig. 4.2.

Identifying internal links To identify internal links for a site, we first access the landing page of each website⁴ at the URL `http://www.<website-domain-name>`; for websites that include a subdomain, we do not add the “www.” prefix. We parse the fetched webpage and retrieve all internal links (i.e., URLs from anchor tags in the page). We prune this set of internal links to include only URLs that respect the `robots.txt` file (if one exists). We also filter out URLs from subdomains according to the subdomain’s `robots.txt` file. Of the 110K sites in our original list, 4,448 were filtered out due to `robots.txt` entries.

If the number of internal links retrieved from the landing page is less than the maximum number of pages per site in our crawl (250), we recursively crawl the website by following the internal links to find more URLs. We repeat this process until we have 250 URLs or we fail to find new links. Fig. 4.3a shows the number of pages crawled per site, with the vast majority of sites (86%) hitting the limit of 250 pages and 92% of sites yielding 50 or more pages.

Ethical considerations Our crawler followed the “Good Internet Citizenship” guidelines proposed by Durumeric et al. [72]. We used a custom user-agent string with a URL pointing to our project webpage with details on the research effort and our contact information. We honored all `robots.txt` directives, carefully spread the crawl load over time, and minimized the number of crawls performed. To date, we have received only one *opt-out* request from a

³Alexa states that they “do not receive enough data from [their] sources to make rankings beyond 100,000 statistically meaningful” [78].

⁴The Alexa list only provides the domain name for each website, and not the complete URL.

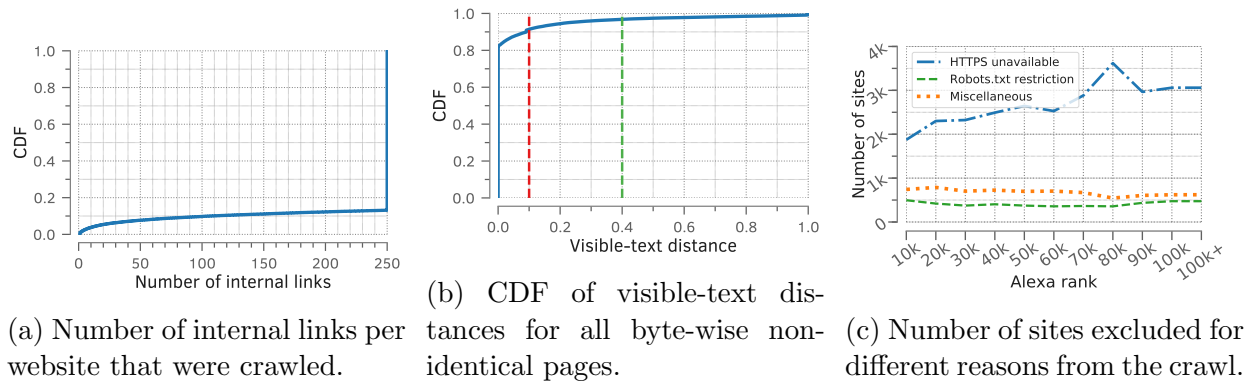


Figure 4.3: Overview of crawling data.

website administrator, which we promptly honored.

4.1.3 Identifying Inconsistencies

We analyze the crawled pages for inconsistencies as follows.

Identifying unavailability Despite being conservative in the rate at which we crawl web pages, it is still possible that some servers may block our requests. For example, they may blacklist the IP address of the machine running the crawler, and *not* indicate this using the standard “429 – Too Many Requests” response status code. Additionally, we may encounter transient 5xx error codes. Under such circumstances, our analysis might misinterpret the temporary blocking as content unavailability (i.e., a false positive).

To mitigate such false positives, one day after the initial crawl we conduct a follow-up crawl of all detected cases of content unavailability. The initial crawl visited each consecutive page after the previous one finished loading, but for this smaller crawl, we use a large (20 seconds) delay between visiting consecutive pages. This follow-up crawl is designed not only to avoid rate-limiting, but also changes the order of fetches from HTTP-first to HTTPS-first to detect differences in website behavior that are dependent on order of HTTP/S access.

Our approach is susceptible to false negatives. Namely, if a server permanently blocks our crawler IP address based on just the initial crawl, the follow-up crawl would miss any potential inconsistencies for that site. This assumes that blocking is consistent for both HTTP and HTTPS requests.

Identifying significant content differences While it is straightforward to detect pages with non-identical content using byte-wise comparisons, using this approach would flag content differences between HTTP and HTTPS for pages that are essentially identical (e.g., due to a difference in timestamp at the bottom of a page) or have dynamic nature (e.g., a product catalog page rotating featured items on each access). Thus, to better understand meaningful content differences, we need heuristics to help filter out such cases.

In particular, we use heuristics inspired by prior research on near-duplicate detection for webpages [82], [83]. This prior work did not provide open-source implementations or sufficient justification for parameter settings, so we base our own heuristics on three steps common to

all of the prior work:

- Parse HTML content and remove nonvisible parts (e.g., markup [82] and headers [83]).
- Retrieve a set of word-based n-grams⁵ from the remaining content, ignoring all whitespace ($n=10$ [82] or $n=3$ [83]).
- For each pair of processed pages, compare the similarity of their n-grams.

A limitation of prior work is that it is tuned to find near-identical pages, while our goal is to find cases that are *not*. Thus, our approach is inspired by the following insights. First, we can filter out dynamic webpages by loading the same page multiple times over the same protocol—if the page content changes, then any such changes should not be counted when comparing HTTP to HTTPS. Second, we observe that dynamic pages often use the same page structure (e.g., use the same HTML template) and the changes occur only in a small set of regions in the document.

Based on these insights, we develop an algorithm for calculating “distance” between two pages, which we then use as a metric to quantify their differences on a scale of 0 to 1. First, we parse the HTML document, filtering either all text visible to a user,⁶ or a list of HTML tag names (to capture the page structure). Next, we transform the result into a set of 5-grams. After getting such 5-grams for two pages, we compute the Jaccard distance (i.e., the size of the intersection of the two sets divided by the size of their union).

We then determine that there is a significant content difference between HTTP/S versions of a page if the following properties hold with parameters $\alpha, \beta, \gamma \in [0, 1]$:

1. The page-structure distance between HTTP and HTTPS, is greater than γ .
2. The visible-text distance between HTTP and HTTPS, *d-across-protocols*, is greater than α . This filters differences appearing due to minor changes such as timestamps in visible-text, and/or cookie-identifiers in the source.
3. The visible-text distance between the same page fetched twice over HTTP + β , is less than *d-across-protocols*. This ensures that if a page is dynamic over HTTP, then the difference it presents over HTTP/S must be greater than the baseline difference due to dynamicity, by an amount controlled by β , in order for the page to be counted in our analysis.

We obtain and use the data for computing the distances as follows. Our initial crawl loads each page over both HTTP/S to find the ones with non-identical bodies. A day later, we run a slow follow-up crawl only on the pages with non-identical bodies over HTTP/S, to identify any false positives from the initial crawl. For cases where non-identical bodies persist, we then identify pages with significant content differences satisfying the above properties. For assessing properties 1 and 2, we compare the HTTP/S versions of the page from the slower crawl. For property 3, we compare the HTTP version of the page from the initial crawl with the HTTP version of the same page from the follow-up crawl.

This method provides us with an objective way of measuring visual differences across pages

⁵An n-gram is a contiguous sequence of n items from a given list. For example, word-based 2-grams generated from “to be or not to be” include “to be”, “be or”, “or not” and so on.

⁶Using the code found here: <https://stackoverflow.com/a/1983219>.

Type	Description	Example
Misconfigured redirections (82.6%)	Choosing a default protocol for all visitors, but accidentally setting up redirections which do not preserve resource paths.	<i>developers.foxitsoftware.cn/pdf-sdk/free-trial</i> gets redirected to the website homepage if the request was over HTTP (instead of being redirected to the requested page at HTTPS).
Unintentional support	Accepting HTTPS connections without serving meaningful content.	<i>www.historyforkids.net</i> presents default server page over HTTPS accesses but provides site-specific content otherwise.
Misconfigured headers	Incorrectly using the response status codes.	<i>www.onlinerecordbook.org/register/award-unit</i> returns 200 HTTP OK status for both HTTP/S accesses, but the actual content at the latter says “Page not found! The page you’re looking for doesn’t exist”.
Different versions	Providing a potentially upgraded version at HTTPS, which might have different content for the same resource request.	<i>video.rollingout.com</i> provides different content at the index page of the website based on the protocol used during request.

Table 4.1: Characterization of content differences. The fraction of all pages with inconsistencies that fall into a category is reported when available.

served over HTTP/S, while taking into account their inherent dynamic nature. We note that whether a user actually finds a set of pages different is subjective to some extent. For the purposes of this study, we assume that the greater the visual differences across pages, the greater the probability of a user also finding them different.

Fig. 4.3b presents a CDF of the visible-text distances for all byte-wise non-identical sets of HTTP/S pages crawled. The majority of pages (82.4%) have a visible-text distance of 0, and are thus essentially identical. To validate this metric and determine thresholds to use for significant differences, we manually inspected more than a dozen pages at random from the set clustered around 0, and indeed find them all to have minor differences that we would *not* consider meaningful.⁷

However, there is a long tail of remaining pages with potentially significant visible-text differences—and it is possible for two pages with few visual differences to be semantically very different. The curve in Fig. 4.3b does not reveal any obvious thresholds for detecting significant page differences, so we provide results using a low threshold ($\alpha = 0.1$, $\beta = 0.1$, $\gamma = 0.4$) that finds more inconsistencies, and a stricter, high threshold ($\alpha = 0.4$, $\beta = 0.2$, $\gamma = 0.6$) that finds fewer. These thresholds for α are marked on the figure using vertical lines.

Although the selection of parameters entailed some manual tuning, our root cause analysis indicates that this approach worked well for identifying inconsistencies. Namely, in Section

⁷For the sample size we used, the estimated fraction of pages with minor differences in the cluster is 0.90 \pm 0.10, with a 95% confidence interval.

Type	Description	Example
Misconfigured redirections (19.7%)	Fixing broken links through redirections, but only over one protocol.	<i>www.sngpl.com.pk/web/tel:1199</i> redirects to the website homepage when accessed with HTTP, but presents a 404 Not Found otherwise.
Fixed Ports (8.7%)	Embedding or enforcing port numbers in the URLs.	<i>facade.com/content</i> redirects to <i>facade.com:80/content/</i> , resulting in a connection error on accesses over HTTPS.
Partial support	Not supporting HTTPS at a portion of site content.	<i>www.indiana.edu/~iubpc/</i> , and all resources under the directory.
Different versions	Providing a potentially upgraded version at HTTPS, which might not require hosting resources from the old version.	<i>aedownloadpro.com/category/product-promo/</i> is only available at the HTTP version of the site.

Table 4.2: Characterization of content unavailability issues. The fraction of all pages with inconsistencies that fall into a category is reported when available.

4.2.1, we attribute the vast majority of the identified content differences (at least 82.6% of pages) to server misconfigurations.

4.2 Results

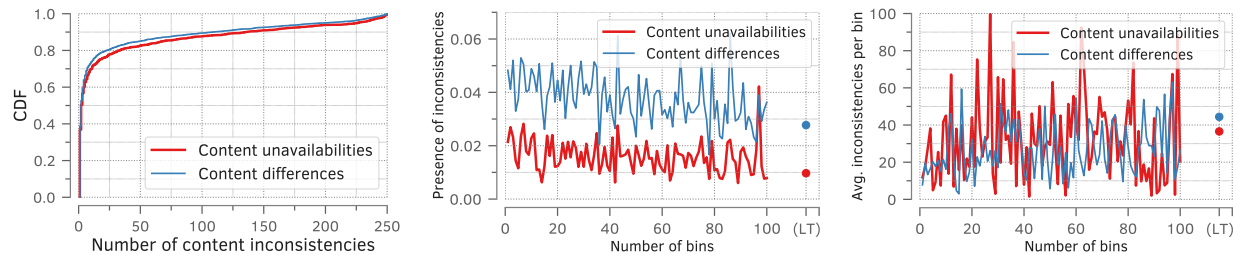
We performed the measurements in October 2019 from a university network in Boston, Massachusetts (USA). From the 110K sites (the *Alexa* list in §4.1.2) investigated, our crawler found at least one internal link for 68,369 websites available over both HTTP and HTTPS. We plot the number of sites excluded from the crawl due to various reasons, using bins of 10K websites, in Fig. 4.3c. The *miscellaneous* category includes cases where our crawler encountered parsing errors, pages relying on JavaScript, or a domain hosting only one page with no internal links.

For the rest, we show the number of links crawled per site as a CDF in Fig. 4.3a. The average number of internal links captured and crawled was 225. The cluster at $x = 250$ is due to the threshold we had set (§4.1.2), our crawler stops searching for new links after the limit has reached.

4.2.1 Summary Results

We observe 1.5% of websites (1036 of 68,369) have content unavailabilities, and 3.7% (2509 of 68,369) have content differences (3.1% with stricter significance thresholds). For websites with at least one inconsistency, the average number of pages with inconsistencies is 31.9 and 27.2, respectively. Fig. 4.4a plots CDFs of the number of inconsistencies per site, showing that while most sites have few inconsistencies, there is still a significant percentage of websites (15%) where we find inconsistencies for at least 50 internal links.

We identify several root causes for these inconsistencies by manually analyzing 50 random



(a) Number of inconsistencies (b) Alexa rank vs. inconsistency (c) Alexa rank vs. #inconsis-
per website. presence. tencies.

Figure 4.4: Inconsistencies are prevalent on a small but significant number of websites, and are not correlated with site popularity. (LT) refers to Long Tail results.

instances for both unavailability and content differences. Tables 4.1 and 4.2 provide a description and one example for each (the types represent commonly observed behaviors, and do not represent a complete taxonomy). We then label pages as (i) “Misconfigured redirections” if their final URIs are different over HTTP and HTTPS (i.e., after performing any redirections), and (ii) “Fixed ports” if their URIs include port numbers 80 or 8080. We note that it is not trivial to estimate the fractions for all categories (e.g., assessing whether a webpage presents meaningful content vs. a custom error page), and thus present the fractions only for the two above types. Most of the cases are server misconfigurations, and as such the corresponding inconsistencies are likely easy to fix (e.g., by providing suitable redirections). We note that two error types dominate content unavailability inconsistencies; *404 Client Error* (82.6%) and *SSLError* (9.1%).⁸

Cross-validation with real browser Our results could be biased due to our reliance on a Python-based crawler that does not execute JS instead of using a real browser. This could lead to false negatives if HTTP/S differences are visible only to a JS-supporting crawler, or false positives if differences are only visible to our custom crawler.

We cross-validate our Python crawler’s results by comparing them with the Chromium-based crawler, and found their results to be highly consistent. Specifically, both pipelines observed 95.8% of domains with at least one unavailability inconsistency and 86.0% of domains with at least one content difference (85.1% with stricter distance thresholds).

Upon manual inspection, we observed that domains with inconsistencies visible only via the Python-based crawler reflect (i) TLS implementation differences (e.g., minimum TLS version allowed, certificate validation logic), (ii) presence of `<meta http-equiv="refresh">`⁹ redirections in page content and/or (iii) content differences that are less significant when JS is enabled. While there are differences in inconsistency results, as expected, our findings suggest that a large fraction of inconsistencies observed by the custom crawler are ones that also would affect users visiting sites via web browsers.

It is also possible that the Python crawler missed inconsistencies that manifest only through a browser. To estimate whether this is the case, we used 100 sample sites in the following

⁸The remaining errors are various 4xx and 5xx errors.

⁹An HTML-based instruction for web browsers to redirect to another page after a specified time.

way. First, Alexa Top 100k list was divided into bins of size 1,000 each. From each bin, we found a sample site that had at least 250 internal pages and zero inconsistencies according to the Python-based pipeline. We fed these 100 sample sites to the Chromium-based pipeline to check for any inconsistencies that manifest only through a browser. In summary, we found no such inconsistencies.

More specifically, the browser did not flag any of these websites with unavailability issues (its usage should not have affected the availability of a page anyway), but it did find 5 with content differences (and only 1 site when using stricter significance thresholds). Upon manual analysis, we found all of these cases were due to normal web page dynamics that surpassed our content-difference significance thresholds (which were tuned for content based on a crawler that did not support JS execution).

4.2.2 Factors Influencing Inconsistencies

Website popularity We test whether inconsistency rates correlate with site popularity, and find that there is no strong relationship between the two. Specifically, we distribute entries from Alexa Top 100K list into bins of size 1,000 each, while preserving the popularity rank. Note that we use a separate bin for the 10K sample of the 900K least popular sites. In Fig. 4.4b, we plot bins on the x-axis and on the y-axis the fraction of websites with at least one inconsistency (the denominator is the number of websites with at least one link crawled). In Fig. 4.4c, we use the same x-axis, but the y-axis is the average number of inconsistencies, for all websites with at least one inconsistency. Content differences seem slightly more likely on popular pages than unpopular ones, which is somewhat counterintuitive since one might expect more popular sites to be managed in a way that reduces content differences. But one can visually see high variance in the relationship between inconsistencies and popularity; further, we compute the Pearson’s correlation coefficients and find only weak correlations—not strong enough to infer that the rate of inconsistencies depends on site popularity.

Self vs. third-party hosting Cangialosi et al.[84] studied the prevalence of outsourcing TLS management to hosting providers, as it pertains to certificate issuance and revocation. They find it to be common, and that such outsourced providers manage certificates better than self-hosted sites. Based on this observation, we investigate whether outsourced site management also reduces inconsistencies between HTTP and HTTPS.

For the case of outsourced site management, we focus on one ground-truth example: Cloudflare. We choose them because they manage certificates for the vast majority of their hosted websites.¹⁰ For other hosting providers, it is not clear what percent of domains are being self-managed vs. service-managed. We begin by mapping a server’s IP address to AS number, then use CAIDA’s AS-to-Organization dataset to retrieve the organization name. We then focus on the 20,131 websites whose server organization is “Cloudflare, Inc.” We find content unavailability inconsistencies in only 0.6% of these sites (a decrease of 60.0% as compared to the average across all sites), and content-differences in 2.0% (a decrease of 45.9%). Thus for this one example, Cloudflare management seems to reduce inconsistencies ($\chi^2 = 200$ with

¹⁰According to a recent investor report [85], $\approx 97\%$ of Cloudflare customers use the free-tier product that provides only Cloudflare-managed HTTPS certificates (consistent with estimates from prior work [84]).

Type	Sources	Total	Inconsistency issues	
			Unavailability	Content-diff.
HTTPS Available	[86]–[88]	74,778	741 (1.0%)	1933 (2.6%)
	[89]	66,206	607 (0.9%)	1691 (2.6%)
Default HTTPS	[86], [89]	67,813	591 (0.9%)	1697 (2.5%)
	[87]	16,221	99 (0.6%)	368 (2.2%)
HSTS Available	[86]	17,557	108 (0.6%)	410 (2.3%)

Table 4.3: Comparing HTTPS adoption metrics by calculating number of websites with issues over number of websites fulfilling the metric criteria.

p -value < 0.00001 for Pearson’s test of independence).

We compare these error rates with the set of 3,977 identified self-hosted websites. We define them as the ones whose organizations host only one domain from the Alexa 110k sites in our analysis, and thus are either self-hosted or hosted through a small provider.¹¹ For these, we find content unavailability inconsistencies in 4.5% of such websites (an increase of 200% as compared to the average across all sites), and content differences in 8.5% (an increase of 129.7%). Thus, self-hosted sites seem to be much more likely to have inconsistencies between HTTP and HTTPS ($\chi^2 = 226$ with p -value < 0.00001).

We posit the following reason that might explain why third-party certificate management can help reduce inconsistencies. Prior work [84] suggests third-party services perform better certificate management. They likely (i) can also notice server misconfigurations comparatively earlier due to their large number of customers and dedicated support staff, and (ii) have default TLS-related settings in place to reduce the chance of accidental mistakes when a site is migrated to HTTPS.

Certificate issuing authority We now investigate whether the rate of inconsistencies is related to certificate issuing authority (CA). We found that across all domains crawled in the study, the most commonly used CAs are Let’s Encrypt (LE; 21.6%), DigiCert (19.7%), Comodo (18.2%) and Cloudflare (8.2%). But for domains with inconsistencies, the shares change to 10.0%(↓), 31.6%(↑), 15.1%(↓) and 2.5%(↓) respectively. As such, we did not see any clear trend indicating how a CA can affect inconsistency rates.

4.2.3 Comparing HTTPS Adoption Metrics

Prior work identifies several metrics for characterizing the extent to which a website supports HTTPS [70]. In Table 4.3, we compare our inconsistency findings with those metrics, using the Alexa list (§4.1.2). In some cases, our crawls identified inconsistencies on subdomains, but we exclude these from the analysis to ensure a fair comparison. We find that there exists a small but nontrivial number of sites where such metrics indicate support for HTTPS/HSTS, but we identify inconsistency issues. The results for HSTS are particularly surprising, as users

¹¹We manually analyzed a small sample of the 3,977 sites, and estimate the fraction of them that are self-hosted (versus hosted via a small provider) to be 0.79 ± 0.15 , with a 95% confidence interval.

are guaranteed to be affected by inconsistencies since the browser must fetch all content over HTTPS. A key takeaway is that, for a more accurate view of website HTTPS support, future scans should take into account inconsistencies and scan beyond landing pages.

4.3 Discussion

While our findings provide a dose of good news about the quality of HTTPS adoption on popular pages (the rate of inconsistencies is low), the sobering fact is that there are still a substantial number of inconsistencies—even on some of the most popular websites. We now discuss the implications of our findings, and why even a small number of page inconsistencies is a finding that has broad impact.

Security Prior work argues for HTTPS support on every page of a website. Thus even a single page with an unavailability issue can undermine security for all others in a site, as a single access to an insecure page (due to unavailability over HTTPS) can be a vector to enable site-wide downgrade attacks.

Usability From a usability perspective, content differences mean that a user on a HTTPS-by-default browser may view content that the website owner did not intend to be shown. This could lead to confusion, loss of revenue (e.g., for retail sites with missing product pages or one that have incorrect details), and user abandonment. A caveat for our study is that we do not have any data regarding page popularity within a site, so we cannot tell how many users are affected by pages by content differences.

Persistency & Prevalence We found that the identified many inconsistencies are persistent over time, and the total number of issues is not getting better over time. To test this, we ran a second crawl over the Top 100K websites four months after the initial study and observed a similar-scale set of consistency issues: 1.4% of websites with unavailability issues and 3.6% of websites with content differences. Further, the *union* of all websites among the two crawls was 2% for unavailability issues and 4.8% for content-differences (with the intersection being 0.9% and 2.4% respectively). This suggests that although the issues affect a small portion of the web at *any given point of time*, the problem is much more widespread when considering the number of websites affected by it over time.

4.4 Conclusion

This work explored whether websites counterintuitively provide different results when the same URI is accessed over HTTP and HTTPS. We found a small but significant fraction of sites have inconsistency in terms of content availability and differences, and this occurs across all levels of popularity. We find that their root causes are often simple server misconfigurations, and thus recommend automated processes to identify and remediate these issues. Our findings also highlight that moving web browsers to HTTPS-by-default would still incur substantial problems for users and site accessibility, motivating the need for more study on the impact of such approaches.

We argue that website administrators need tools like our crawler to help them identify and mitigate inconsistency issues to facilitate the transition to HTTPS-by-default. To encourage this, our efficient web crawler code, dataset, and analysis code can be publicly accessed at <https://github.com/NEU-SNS/content-differences>.

Chapter 5

TLS Certificate Pinning in Mobile Applications

Mobile applications (apps) are extremely popular – 230 billion apps were installed on devices in 2021 [90] alone – and often transmit sensitive data over the Internet to deliver their service (e.g., credentials, financial and health information). Thus, network connection security is critically important in this context. While the standard TLS PKI provides sufficient security for most apps, several classes of attacks have revealed gaps in its protection: tampered, misconfigured, or poorly maintained certificate authority (CA) root stores [15] can enable highly targeted or large-scale monkey-in-the-middle (MITM) attacks [91], [92]. To address this issue, app developers and third-party libraries can use *certificate pinning*, which establishes a developer-specified relationship between a hostname and its cryptographic identity (certificate or hash of the public key)—one that is typically hard-coded (hence “pinned”) and that adds another layer of security compared to certificate validation that uses only the trusted system CA root store.

Although beneficial from a security standpoint, pinning is known to introduce maintenance overheads, misconfiguration errors, and other problems which could expose users to more attacks. Unfortunately, there is no clear community consensus on whether the benefits of pinning outweigh potential risks of misconfiguration and developer errors. On the web, it was first introduced in 2011 [93], but has since been deprecated by all major desktop and mobile browsers [94]. Android officially supports pinning since version 4.2 (released in 2012) [95], but has since moved to not recommending pinning due to the risk of app breakage when server configurations change [96], [97]. Apple does not provide clear recommendations for iOS, but notes that pinning might be necessary to meet regulatory requirements [98], and recommends long-term strategies to handle certificate changes. It is, therefore, vital to know whether app developers implement certificate pinning; and to identify common deployment errors that could compromise apps’ security.

In this chapter, we provide the first multi-perspective look at certificate pinning, by developing more complete methodologies for detecting it that leverage the complementary strengths of static and dynamic analysis, characterizing its prevalence across iOS and Android, and investigating the implications of observed implementations.

We develop novel static and dynamic techniques to detect and measure the adoption of pinning. Specifically, our methodology includes more complete rules for searching app binaries for evidence of certificates or pinning APIs, an analysis of which code is responsible for pinning, as well as run-time analysis that reliably distinguishes pinned connections from other confounding types of TLS connection behavior. Our work builds upon and extends prior work in this space [13], [40], [41], [99].

5.1 Background and Motivation

During a TLS handshake, clients obtain a *certificate chain* (ordered list of certificates) from servers, where each certificate is signed by the previous one. Clients trust the chain if they trust the *root* certificate, and the signatures from the *root* (first) to the *leaf* (last) are all valid. A *root store* or *CA (certificate authority) store* is a collection of such trusted *root* certificates, which is included in OSes including Android and iOS [15], [100].

Certificate pinning is an alternate to trusting OS root certificates, where apps include a custom certificate to be trusted (in their source code or metadata in the app package), instead of the set of certificates present on the OS. We define pinned certificates as such custom certificates that *must* be present in the certificate chain to successfully establish a TLS connection. These pinned certificates could be any certificate in the chain, i.e., leaf, intermediate, or root certificates. They could also be pinned in any form, i.e., storing the entire certificate, a hash of the certificate, or some other identifier.

Pinning for Protection: Mobile root stores are known to include expired, unknown, or obscure CA certificates [15], which can expose clients to TLS interception attacks. An attacker with access to the private key for a CA certificate in the system trust store can use it to sign arbitrary certificates (for arbitrary domains) and trick the client into accepting these malicious certificates as valid. Using certificate pinning prevents such attacks by limiting certificate trust to a pre-determined set of certificates instead of trusting a certificate issued by any CA certificate in the system trust store. Note that certificate pinning not only protects against malicious actors, but also against investigators and auditors seeking to analyze the data exchanged between devices and servers (e.g., to understand personal data exfiltration, cross-border data transfers, *etc.*).

Pinning for Customization: Certificate pinning enables developers to define a specific certificate to trust. This allows developers to issue and sign their own trusted certificates instead of obtaining one from a trusted third-party CA, thus regaining more control over their internal certificates at the cost of limited utility since custom CAs will not be trusted by browsers or other software that does not trust the custom CA. Note, however, that verifying if a pinned certificate is present in a chain is not sufficient to ensure that the chain is correct; rather, the TLS library must still validate all other properties of certificates (i.e., *Common Name* matching, revocation checking, *etc.*) to protect against various other attacks.

Pinning and HPKP: Certificate pinning methods found in mobile apps differ greatly from *HTTP Public Key Pinning* (HPKP). HPKP is an obsolete technique for *web browsers* that allowed website owners to specify pinned certificates for their domain. One key reason HPKP was proposed is that website owners in general cannot directly control the trust store for a

Rank	Android			iOS		
	Random	Popular	Common	Common	Popular	Random
1	Education 12%	Games 36%	Games 18%	Games 18%	Games 21%	Games 15%
2	Games 12%	Weather 2%	Productivity 12%	Productivity 14%	Photography 11%	Business 11%
3	Tools 6%	Finance 2%	Business 7%	Business 8%	Social 6%	Education 11%
4	Music 6%	Shopping 2%	Communication 6%	Social 7%	Education 6%	Food 7%
5	Books 6%	Entertainment 2%	Finance 6%	Education 6%	Finance 6%	Lifestyle 7%
6	Business 5%	Food 2%	Education 5%	Finance 6%	Lifestyle 5%	Utilities 6%
7	Lifestyle 5%	Social 2%	Social 5%	Utilities 5%	Entertainment 4%	Entertainment 4%
8	Entertainment 4%	Productivity 2%	Health 4%	Photography 4%	Utilities 4%	Health 4%
9	Travel 4%	Photography 2%	Travel 3%	Health 3%	Productivity 4%	Travel 4%
10	Personalization* 4%	Music 2%	Lifestyle 3%	Lifestyle 3%	Weather 4%	Shopping 3%

Table 5.1: An overview of our app datasets. We present the top 10 app categories from each dataset, along with their percentages over the total number of apps in that dataset.

browser, and HPKP gave them a way to specify custom certificates for pinning on a domain. In contrast, mobile services that use pinning can control both the client software (the app) and the web servers they communicate with. As such, there is no need for any additional protocol like HPKP to specify how pinning should occur—mobile apps simply include the pinned certificate material in the app code and/or metadata.

We also note that the threat models and stakeholders in the two techniques are different. For HPKP, the website owner does not trust the OS or browser root store, but assumes that browser will enforce a specified pinned certificate. Further, HPKP trusts the first seen certificate (and thus does not solve the problem for adversaries that can intercept the first TLS connection) and also does not support changing the pinned certificate. In contrast, mobile services that use certificate pinning do not trust the OS root store, but trust that the OS will faithfully execute its specified certificate validation and pinning code. In addition, mobile services can change the pinned certificate in numerous ways, e.g., by pinning a CA certificates that can issue additional trusted leaf certificates, or releasing a new version of the app with a new pinning specification.

5.2 Goals

Our study is organized around the following key research questions:

RQ1: How can we reliably detect pinning and its prevalence in mobile apps, in a platform-agnostic way?

RQ2: What are the characteristics of apps that deploy pinning (popular vs unpopular apps, app categories, pinned destinations) and what are their implications?

RQ3: How consistently do developers use pinning across the Android and iOS versions of the same apps?

RQ4: How is pinning implemented (e.g., nature of certificate chains, code that contributes to pinning)?

RQ5: How secure is pinning in mobile apps? And what kind of data is protected by



Figure 5.1: Our methodology to detect certificate pinning. We (1) crawl Android and iOS apps, (2) search app contents for certificate files or hashes, (3) retrieve certificates corresponding to the hashes using publicly available Certificate Transparency logs, (4) launch every app on a real device and collect network traffic in two distinct settings: (5) when app traffic is not intercepted, and (6) when app traffic is intercepted through monkey-in-the-middle (MITM) technique. We identify and mark TLS connections that transmit data in the former setting but not the latter as pinned.

pinning?

5.3 Methodology

In this section, we detail our datasets as well as the novel static and dynamic approaches we use to detect certificate pinning (RQ1) and shed light on the implementation aspects related to it. Figure 5.1 presents an overview of our methodology.

5.3.1 Datasets

To understand the prevalence of pinning in different parts of the Android and iOS ecosystems, we collect a wide and diverse range of apps on both platforms. We group the apps in three different datasets: popular apps, random apps, and “common” apps. The “common” apps dataset contains the same app on Android and iOS, thus enabling us to perform head-to-head comparisons of the two platforms. We collect these apps at various points in time in 2021.

Collecting Android apps from the Google Play Store is simpler than collecting iOS apps from the Apple App Store. For Android, we use GPlayCLI [101] to download apps directly from the Play Store. For iOS, we automate GUI interactions with the deprecated iTunes 12.6 application to download apps, based on previous work [14]. We obtain the category of each app (e.g., gaming or finance) directly from the metadata set by the developers and available in the respective stores.

Common Apps (n = 575): Linking apps present on one market with those present on another is non-trivial. We create the set of common apps using AlternativeTo [102]. This website crowdsources information, recommendations, and reviews for software. Apps listed on this website can have links to the Google Play Store and Apple App Store if they are present on both platforms. We retrieve $\approx 1,000$ app pages sorted by popularity on this website and look for apps listed on both stores. Using this technique, we obtain 575 apps; we manually verify (on a small random sample of 30 apps) that these apps are in fact the

same. To respect community norms related to crawling, we add our contact information in the User-Agent field, and limit our crawler to request 1 page per second.

Popular Apps (n = 1,000): For popular apps on Android, we use the *google-play-scraper* [103] to crawl “Top Free” lists for each category on the Google Play Store. We pick at random 1,000 apps from these lists ($\approx 12k$ in total). For iOS, we use the iTunes Search API to fetch top apps using 19 generic category names as search terms (e.g., productivity, finance, music). The API returns at most 100 results per call. We repeat the process for each category and collect unpaid apps that are compatible with our test device, compiling a set of 1,000 apps. Both sets contain apps that capture the notion of popularity for each store; they do not necessarily represent the top 1,000 apps for either platform. We note that we used the US version of the app stores while compiling these listings.

Random Apps (n = 1,000): To compile a list of random apps, we start out with fetching details about as many apps as possible. Unfortunately, the list of all apps on either platform is not public. For Android, we use a list of 1.35M app IDs compiled by prior work [41]. For iOS, we crawl 1.25M app IDs from the official store listings [104]. From each of these lists, we randomly select 1,000 apps and download them from the respective stores. We believe that the large size of our lists provides a sufficient degree of randomness.

We perform all crawls from North America; We collected the Common and Popular sets from February to May 2021, and the Random sets in October 2021. Due to our app collection technique, we see app collisions between the three sets; in such cases the same app is used in the sets they appear in. Accounting for collisions, we collect 2,564 unique apps for Android (11 collisions for Common and Popular sets). For iOS we collect 2,515 unique apps (60 collisions for Common and Popular sets). We see no collisions between the Random app set and other sets on either platform. Thus in total, we collect 5,079 unique apps, counting Android and iOS apps as different apps for the Common set.

5.3.2 Static Analysis

Static analysis involves studying apps without actually executing them. In this section, we discuss the parts of apps we study and the exact techniques used to infer whether certificate pinning is being implemented across apps.

Configuration Files In Android, Network Security Configuration (NSC) files are used to customize network security settings without having to modify app code [39]. This technique allows apps to define general security settings or per-domain settings, with the option to specify certificates to trust, and to pin certificate hashes. We use static analysis to extract the *Android Manifest* file, which we parse to check if an app is using an NSC. If it is found, we extract the pertinent configuration file and parse that to obtain certificates and hashes that the app uses, extracting files as needed.

In iOS, App Transport Security Settings provide a similar feature of specifying pinned hashes in an app’s configuration files [105]. We note that it is a recent feature introduced in iOS 14 in September 2020, and is unavailable in the version of iOS used in our study. Because this feature was released close to our data crawls, we do not check for its prevalence in our datasets.

Embedded Certificates Pinning implementations typically specify which certificates to pin in an app code. Therefore, we search for these certificates in app code by looking for any files ending with *.der*, *.pem*, *.crt*, *.cert*, and *.cer* extensions, or by extracting strings with delimiters such as “-----BEGIN CERTIFICATE-----”. In addition, we also search for SHA-1/256 hashes of the *SubjectPublicKeyInfo* (SPKI) field of certificates that is traditionally used in various protocols (e.g., HTTP Public Key Pinning [106] and DANE [107]), but also seen in some pinning implementations (e.g., Chrome [93] and the Android OkHttp library [108]).

We decompile the Android apps using *Apktool* [109]. As iOS apps are encrypted, we use Flexdecrypt [110] or Frida-iOS-Dump [111] to extract decrypted payloads. Flexdecrypt is faster than Frida-iOS-Dump because it does not require opening an app for decryption. We then employ a fast recursive grep tool, *ripgrep* [112], to search for the regex patterns of interest, i.e., hashes or certificates. For hashes we use the regular expression $sha(1-256)/[a-zA-Z0-9+/=]{28,64}$. The length allows us to search for hashes that are either base64- or hex-encoded. In addition, we use *libradare2* [113] to analyze strings from native libraries and/or executables present in the apps. We note that we do not attempt to de-obfuscate any decompiled files, nor can we handle any code that an app dynamically downloads during run time, which is a limitation common for any static analysis approach.

Associated Certificates We search for certificates associated with *SubjectPublicKeyInfo* hashes that our analysis found in the apps using the *crt.sh* certificate search [114] that indexes data from Certificate Transparency (CT) logs.

Third-party Pinning Code Because we have information about the path in the app code where each pin or certificate is found, we can use this information to shed light on the source of pinning code. To check for third-party pinning, we manually review all the certificate paths that appear in more than 5 apps, and infer whether the source is indeed a third party using publicly available knowledge (e.g., code in the *sensibill* folder reflects the billing API of the Sensibill SDK in Android apps).

5.3.3 Dynamic Analysis

There are several reasons why support for pinning found statically in apps might not lead to pinning being used at run time. For example, we may detect code that is unused (e.g., due to a library that is never loaded, a library that provides optional support for pinning, or an outdated app version dynamically disabling pinning at run time). To address this, we use dynamic analysis; namely, we install and run apps on devices while collecting device logs and network traffic to determine if apps pin certificates at run time. Thus, we consider results from dynamic analysis to be the ground truth about whether apps actually use pinning or not. In this section, we describe the components used in our dynamic analysis in detail and how they tie together to detect pinning.

Dynamic Pipeline We execute apps on real devices and collect network traffic. Our dynamic pipeline relies on automation frameworks for both platforms that control the devices

via USB connections, and can install/run/uninstall apps on them. Our devices connect to a WiFi hotspot under our control. We use *mitmproxy* [115] to proxy all the traffic from devices to servers, and to have the ability to MITM the traffic using an arbitrary CA certificate.

For the Android tests, we use a Pixel 3 device running a factory system image of Android 11 (released in September 2020; modified to include the *mitmproxy* certificate in the certificate store). For the iOS tests, we use an iPhone X running iOS 13.6 (released in July 2020; with trust for the *mitmproxy* certificate enabled). Our iPhone is jailbroken using Checkra1n [116] to facilitate various aspects of the study (e.g., app decryption for static analysis, pinning circumvention). Our choice to use the particular iPhone model and OS is based on the fact that there were no jailbreaks available for the latest combinations at the time of our experiments.

During dynamic testing, the automation framework installs one app at a time on the test device to ensure traffic isolation, waits 30 seconds to collect app traffic, and uninstalls the app before moving on to the next. For each dataset of apps, we run two experiments. Our first baseline experiment (“non-MITM”) records TLS traffic triggered by apps without any interference. The second experiment (“MITM”) runs with *mitmproxy* enabled, which tries to MITM any TLS connection. Based on the difference in app behavior in these two experiments, we extract information about pinned connections, as described in detail in the next section.

App Interaction We experimented with various techniques to automate interacting with apps using UI Automator [117] for Android and a similar tool for iOS. While automation is itself feasible, the key issue is that apps on iOS and Android often present different UIs and we could not identify a general way to exercise the same functionality across platforms and thus could not conduct an apples-to-apples comparison. Given this, we also explored the potential for using *random* interactions that are commonly used in prior work (e.g., [118], [119]). While the interactions would not be identical across platforms, they also should not have any particular bias overall. We conducted a small set of experiments where we used automated, random UI interactions, and we found no significant change in the number of domains contacted when compared to tests without any UI interactions. Thus, we do not perform any automated or manual interactions with apps in our study.

We varied the amount of sleep time (15s, 30s, and 60s) from installing an app to uninstalling it, and heuristically found 30 seconds to be the best value for our study. More specifically, we found the average number of TLS handshakes performed by a small random sample of apps in the three settings to be 20.78, 23.5, and 24.62 respectively. As the vast majority of TLS connections are established within 30 seconds, we believe the diminishing returns beyond this point are not worth the additional wait.

Detecting Pinned Connections A key challenge for detecting pinned connections is that it can be difficult to distinguish between a connection that fails due to TLS interception on a pinned connection, as opposed to a connection that fails for other reasons (e.g., server-side issues). At a high level, our approach is to use a differential analysis, where we detect differences in connection behavior with and without TLS interception. Specifically, if a connection to a destination carries traffic beyond the handshake *without* TLS interception,

and never carries traffic when *with* TLS interception, we mark the destination as pinned.

More specifically, we observed that a pinned TLS connection exhibits two key properties, and neither is unique to TLS interception. First, pinned TLS connections typically send failure signals via a TLS alert or TCP connection reset if an attempt is made to MITM them. However, these signals may also appear in an app traffic for reasons other than pinning (e.g., TLS alert due to an unsupported protocol version). Second, pinning may result in a connection being successfully established, but it will never be used for transmitting useful application data if there is an attempt to MITM the connection. However, even without TLS interception, apps will create redundant connections and never use some of them to transmit application data. Thus, we need a way to account for such confounding factors. By comparing connection behavior in the two settings, we can attribute any observed connection failures to the presence of pinning. We further rely on the following definitions to evaluate a connection status:

Used Connection To determine whether a TLS connection sends application data, we rely on the following tests. For TLS 1.2 or below, the presence of any “Encrypted Application Data” packets is sufficient to infer that the corresponding TLS connection is being used by a client. This inference does not work for TLS 1.3, where all encrypted records (data, alerts, or handshake messages) are disguised as TLS 1.2 “Encrypted Application Data” to reduce issues with middleboxes. Thus for TLS 1.3, we rely on the following two heuristics to identify connections that send application data: 1) clients either send more than two “Encrypted Application Data” packets, or 2) the second packet is not the same length as an encrypted TLS alert. The reasoning behind this is that the first encrypted client packet must always be “Client Handshake Finished” for successful connections according to the protocol specification, the second packet may or may not be an alert to indicate connection closure and third (if present) can only mean that application data has already been transmitted.

Failed Connection We define a failed connection as any TLS connection that goes unused, *and* where the clients abort connections with TCP RST or TCP FIN flags. This helps avoid false positives for cases where a connection simply remained unused in our experiments due to the limited recording time.

After collecting that status of connections, we evaluate which *destinations* are used at least once. Such information is readily available: 99% of the TLS traffic in our experiments have a non-empty SNI field, indicating the destination hostname for the connection. If a destination has any TLS connection that is used in the non-MITM setting, but TLS connections that always failed in the MITM setting, we mark it as pinned. We note that the heuristic to mark used connections does not need to be perfect, as we ultimately rely on whether an app behaves differently in the two experiment settings to determine pinning status.

5.3.4 Circumventing Pinning

Using the aforementioned methodology, we detect apps that implement pinning. In order to understand why apps implement pinning (RQ5), we attempt to look at the traffic sent in those pinned connections. To this end, we use Frida [120] to hook into various popular TLS libraries and disable certificate validation checks. When successful, this allows us to continue

using our dynamic pipeline to MITM connections and obtain data that apps send to servers in pinned connections. We note that pinning circumvention is not guaranteed to succeed, as developers can always use custom TLS implementations rather than relying on popular ones. Using this approach, we were able to successfully circumvent pinning for $\approx 51.51\%$ unique destinations on Android, and $\approx 66.15\%$ unique destinations on iOS.

5.3.5 PII Analysis

Pinning can either be used to protect sensitive user data, or hide data collection from auditors. As we do not interact with the apps, we cannot check if pinned connections are being used to protect user data (e.g., banking credentials). However, we can still check for the presence of other sensitive information that apps are known to collect from prior studies [118], [121]–[123].

More specifically, if we are successful in circumventing pinning, we inspect the decrypted application data to check for the presence of sensitive personally identifiable information (PII) that can harm user privacy. We also check whether PII presence differs significantly in the pinned vs non-pinned traffic. The PII we search for includes the following information for both platforms: *IMEI*, *advertisement ID*, *WiFi mac address*, *user email*, *state*, *city* and *latitude/longitude*. Although this list is not exhaustive by any means, it is sufficient for the purposes of this study as we are mainly interested in comparing PII prevalence across pinned vs non-pinned traffic, rather than finding out whether apps transmit any PII.

5.3.6 iOS Background Traffic

For Android, our manual analysis did not detect any background traffic that could interfere with our experiments. However, the situation for iOS turned out to be difficult to handle. First, we noticed TLS traffic to various Apple-controlled domains (*icloud.com*, *apple.com* and *mzstatic.com*) that spanned the whole duration of dynamic testing (mainly due to connection retries in MITM experiments). We simply excluded these destinations from our analysis.

Second, and more importantly, we also needed to ignore traffic to many first-party destinations for apps, because it might have been triggered by the OS, rather than the app. This is due to a feature in iOS that contacts all destinations that are marked as “associated” in an app’s entitlements. When an app is installed, iOS triggers TLS communication with these destinations to verify that they are indeed controlled by the app’s developer (by going to a specific pre-defined path). The purpose of this feature is to facilitate connections between the app and its website(s) (e.g., to share credentials, to navigate from the browser to the app when the user visits one of its websites). In our testing, we noticed that all this traffic appears as pinned, likely because the underlying iOS service does not trust our MITM certificate. Unfortunately, the traffic from OS exhibits a similar TLS fingerprint as regular app traffic. As such, we could not find a way to distinguish traffic to these destinations triggered by the app vs the iOS background service. To avoid falsely attributing pinning to apps, we ignored all associated destinations from an app’s entitlements during our analysis. More specifically, 66% of apps did not specify any associated domain, so no traffic was excluded for these. For the rest, there were on average 4.8 unique associated domains present in the configuration.

Study	Year	Prevalence	Analysis	Dataset size	Dataset source
Fahl et al. [13]	2012	10%	Dynamic	20	High-profile Android apps
Oltrogge et al. [124]	2015	0.07%	Static	639,283	Apps from the Google Play store
Razaghpanah et al. [99]	2017	2%	Dynamic	7,258	Android apps found in the wild
Stone et al. [125]	2017	28%	Dynamic	135	Security sensitive Android apps
Possemato et al. [40]	2020	0.62%	Static	16,332	Android apps using NSCs
Oltrogge et al. [41]	2021	0.67%	Static	99,212	Android apps using NSCs

Table 5.2: Certificate pinning prevalence mentioned in prior work. Note that the variety of analysis techniques, datasets, and passage of time make direct comparisons difficult.

Dataset type		Dynamic analysis		Static analysis	
				Embedded Certificates	Configuration Files*
Common (n = 575)	Android	8.17% (47)		26.96% (155)	2.78% (16)
	iOS	8.52% (49)		22.96% (132)	-
Popular (n = 1,000)	Android	6.7% (67)		19.7% (197)	1.8% (18)
	iOS	11.4% (114)		33.4% (334)	-
Random (n = 1,000)	Android	0.9% (9)		9.9% (99)	0.6% (6)
	iOS	2.5% (25)		9.5% (95)	-

Table 5.3: Certificate pinning prevalence found using various methods across different datasets. Each cell denotes the number of apps with one instance of pinning, over the total number of apps in that dataset. (*) denotes the method used by prior work.

Note that this generic approach of excluding traffic can only cause false negatives (i.e., filter out domains that actually pin), not false positives.

Since our goal is to conduct a head-to-head comparison of pinning prevalence in Android vs iOS, we re-ran our dynamic pinning detection pipeline for apps in the Common dataset that were found to be pinning in either Android or iOS through the prior methodology (72 apps in total). We modified our setup for the re-run in the following way in order to avoid the issue with associated destinations: after installing an app, we waited 2 minutes to let the OS finish communicating with these first-party destinations. We launched the app afterwards, and then collected data for 30 seconds as we had done before. We use results from this re-run whenever we mention iOS Common dataset in the rest of this work. On a positive note, the limited re-run did not reveal any false negatives in the initial run. Thus, we do not believe our methodology of handling iOS background traffic affects the results significantly.

5.4 Results

We apply the techniques presented in Section 5.3 on each dataset we have collected in order to understand the prevalence of certificate pinning. We present the certificate pinning we find, per dataset and platform, for our static and dynamic analyses in Table 5.3. To help us compare our findings with prior studies, Table 5.2 summarizes pinning prevalence indicated in prior work. It is clear, however, that these prior studies entail a wide range of techniques for detecting pinning, use different app datasets, and were conducted over a wide time range. Thus, it is difficult to conduct a meaningful apples-to-apples comparison. Instead, to enable comparison with prior work, we focus on the NSC-based static analysis technique used by

Category (Rank)	Pinning %	No. of Apps
Finance (9)	22.99 %	20
Social (14)	17.81 %	13
Events (28)	15.0 %	3
Dating (33)	14.29 %	2
Food & Drink (15)	13.64 %	9
Shopping (18)	12.96 %	7
Comics (32)	12.5 %	2
Automobile (25)	8.33 %	2
Travel (12)	6.49 %	5
Weather (24)	5.88 %	2

Table 5.4: Top 10 categories of apps that pin in Android across all datasets and pinning prevalence per category. Ranks indicate the popularity of a category in our dataset.

multiple prior studies, using the datasets we collect.

Pinning by Technique: Prior research mainly relies on Network Security Configurations (NSCs) to detect pinning in Android [13], [40]. For our Android datasets, using the same approach, we find relatively few apps to pin (from 0.6% to 2.78% depending on the dataset). In contrast, our dynamic analysis technique finds up to 4 times more pinning (i.e., 1.8% to 6.7% in popular apps). Our findings suggest that apps likely have many options other than NSCs to deploy pinning.

We further find that pinning prevalence varies substantially for the novel static and dynamic approaches we develop. While static approaches provide us with potentially pinning apps, dynamic analysis gives us stronger evidence of pinning as we observe pinned behavior through network connections. Due to this reason, for the remainder of this work, we call an app to be pinning if we find at least one instance of a pinned connection from the app in our dynamic analysis results.

Pinning by Platform: We find more pinning apps in iOS as compared to Android across all datasets. While we present a head-to-head comparison of apps in the next section, we find it interesting that even random iOS apps pin substantially more (2.5%) than the set of random Android apps (0.9%). Upon closer inspection, we notice that two destinations, `www.paypalobjects.com` and `firestore.googleapis.com`, get pinned in 10 and 5 apps respectively in the iOS random dataset. In comparison, for the Android random dataset, we do not find any common pinned destination. As such, increased pinning in iOS might be due to these and other third-party libraries that are pervasive in the iOS ecosystem, and choose to pin, as compared to the ones for Android.

Pinning by Category: To understand the characteristics of apps that use pinning (RQ2), we check whether apps belonging to certain categories pin more frequently than others. For each category from the two platforms, we normalize the number of apps that pin by the number of apps we have in that category. We present the top 10 app categories that pin in Android (Table 5.4) and iOS (Table 5.5).

We find that the top category in both platforms is “Finance,” suggesting the use of pinning

Category (Rank)	Pinning %	No. of Apps
Finance (9)	20.63 %	26
Shopping (13)	16.48 %	15
Travel (14)	13.48 %	12
Social Networking (8)	11.02 %	14
Photo & Video (6)	10.67 %	16
Lifestyle (5)	8.7 %	14
Food & Drink (11)	8.49 %	9
Sports (16)	8.16 %	4
Navigation (22)	8.0 %	2
Books (19)	7.69 %	3

Table 5.5: Top 10 categories of apps that pin in iOS across all datasets and pinning prevalence per category. Ranks indicate the popularity of a category in our dataset.

is to protect sensitive user data in these apps. The next two categories are “Social” and “Shopping,” likely again due to the sensitivity of data shared on apps in these categories. In terms of categories within a platform, we find it interesting that none of the top 3 app categories for either platform appears in the respective top 10 pinned categories list. In fact, “Games” is the most prevalent category across all our datasets, but does not appear in the top 10 pinned categories for either platforms.

5.4.1 Pinning in Common Apps

To understand whether pinning apps pin the same domains on both Android and iOS (RQ3), we consider apps from our Common dataset. From this dataset, we find 69 apps that pin on at least one platform. Of these, 27 apps pin on both Android and iOS, 20 apps pin solely on Android, and 22 apps pin solely on iOS. For each app, we compare the set of pinned and unpinned domains across platforms.

By definition, a single entity controls both Android and iOS versions of the same app in the common dataset, and one might hypothesize that they pin domains in the same way. However, we find in practice that they do not always do so, and thus define **inconsistent** and **consistent** pinning for this common dataset as follows. An app has *inconsistent pinning* if a domain pinned on one platform is not pinned on the other. An app has *consistent pinning* if it pins at least one common domain on both platforms and has no inconsistent pinning. Based on these definitions, we present Figure 5.2. For a set of apps, we have inconclusive results, as domains pinned on one platform do not appear on the other at all. For these, we can not determine if the domains would be pinned or not, as we have not observed them.

Apps Pinning on Both Platforms: Of the 27 apps that pin on both platforms, we find that 15 apps have consistent pinning. For these apps, we aim to understand the number of common domains pinned on both platforms. To this end, we compare pinned domains on Android to pinned domains on iOS for each app. We find that 13 apps have the same set of domains pinned on both platforms. For the remaining two apps, we see that one domain is pinned on both platforms (Android pins one other and iOS pins two others).

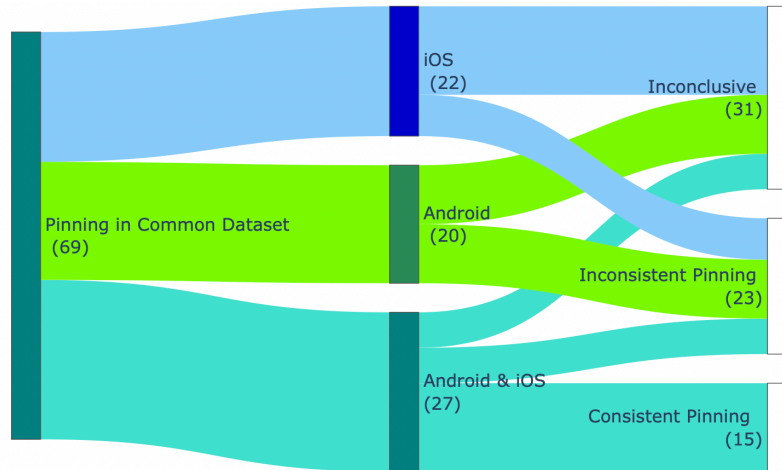


Figure 5.2: Pinning found in the Common dataset split by platforms. We classify pinning found in this dataset into: Inconsistent, Consistent, and Inconclusive as defined in Section 5.4.1.

To understand the inconsistent pinning apps, we compare pinned domain sets to not pinned domain sets. To compare similarities in pinned domains for two pinning sets, we use Jaccard indices. To compare a pinning set to a non-pinning set, we look at the percentage of pinning domains present in the non-pinning set. We use this instead of Jaccard indices here as we care about domains that are pinned in one set and not pinned in the other, as opposed to similarities between the two sets. We present a heatmap of these calculations in Figure 5.3. Each inconsistent app is represented as a row with the first column giving us the overlap of pinned domains on both platforms. The second gives us the percentage of pinned domains on Android that appear as unpinned on iOS; the third gives the percentage of pinned domains on iOS that appear as unpinned on Android. Of the 6 apps, we see that 2 have overlaps of pinned domains; 3 have pinned domains on android that they do not pin on iOS and 3 pin domains on iOS that they do not pin on Android. For the remaining 6 apps, all values on such a heatmap would be 0. On analyzing these, we see that they share no common domains on the two platforms; thus all overlaps would be 0 and hence inconclusive.

Apps Pinning on One Platform: To understand how domains pinned on one platform are handled on another platform, we look at apps that pin exclusively on one platform. For this set, pinning consistencies are nonexistent as the other platform does not pin. To understand these pinning inconsistencies, we compare pinned domains on one platform that appear as not pinned on the other. Apps that have pinned domains on one platform that do not appear as unpinned on the other are marked inconclusive. For 20 apps pinning exclusively on Android, we have 10 inconsistent and 10 inconclusive apps. For 22 apps pinning exclusively on iOS, we have 7 inconsistent and 15 inconclusive apps.

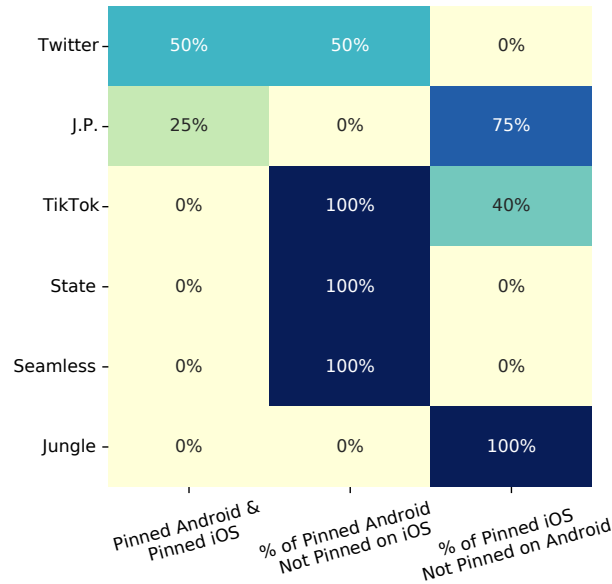


Figure 5.3: Inconsistent pinning in apps that pin on both platforms. We see that the first two apps have overlapping pinned domains but are inconsistent as they pin domains on one platform and not on the other.

We calculate the percentage of pinned domains that appear as not pinned on the other platform and plot a heatmap of these in Figure 5.4. We see that 7 Android apps have all traffic pinned on Android appearing as not pinned on iOS. All apps on iOS marked as inconsistent (7) have all pinned domains appearing as not pinned on Android. Thus, for both these sets of apps, we see that developers talk to common domains and pin them on one platform while not pinning them on the other. This indicates that the pinning policies of these apps is inconsistent and vary greatly based on the platform.

5.4.2 Pinning in Popular vs Random Apps

To understand the prevalence of pinning in popular and arbitrary apps (RQ2), we apply our detection methodology on the Popular 1,000 and Random 1,000 datasets. For Android, we find that 67 apps from Popular and 9 apps from Random use pinning. For iOS, we find 114 apps from Popular and 25 apps from Random use pinning. Thus, we see that pinning is more prevalent on iOS as compared to Android.

To further understand the nature of pinning, specifically the parties involved in pinning in an app, we dig deeper into the number of pinned connections and present the results in Figure 5.5. Each bar on the x-axis represents an app that pins at least one domain, split by dataset and platform. The y-axis shows the percentage of pinned and not pinned domains that each app contacts in our tests. Blue represents pinned domains and green represents not pinned domains. We divide domains contacted by an app into first and third party, attributing each domain for an app using various points of information (whois data, certificate subject names, *etc.*). We annotate each bar with first and third party data marking first parties with dark and third parties with light colors.

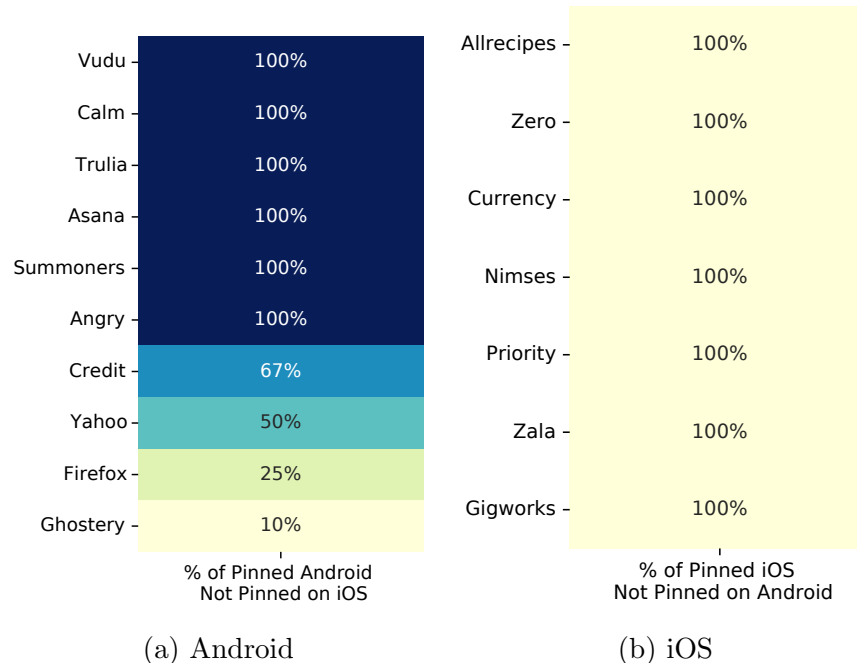


Figure 5.4: Inconsistent pinning in apps that pin exclusively on one platform. Each cell represents the percentage of pinned domains on a platform found as not pinned on the other. For 6 Android and 7 iOS apps, all pinned domains appear as not pinned on iOS and Android respectively.

We see that almost all Android apps that contact first party domains also pin those domains (28); with a single exception, Trulia. On the other hand, Android apps that pin third parties (51) rarely pin all third parties (4); many apps pin some third parties and do not pin others (47).

In contrast, for iOS we observe many cases where first parties are not pinned (18), often when other first parties are pinned (6, dark blue and dark green on the same bar). Similar to Android, many iOS apps pin all first party domains they contact (39). All iOS apps in our dataset that pin third party connections also have other unpinned flows (99).

Based on the results in Figure 5.5, we observe that apps on both platforms almost always have inconsistent pinning practices; domains (regardless of first or third party) are selectively pinned, disregarding other traffic. Only 5 apps on Android (AskURA, Auto Kiosk, Edmtrain, FFBaD, and Private Fostering Awareness) and 4 apps on iOS (Bank of America, CandyCrush, Facebook, and Surge proxy) pin all domains they contact.

5.4.3 Certificate Analysis

In this section, we explore how certificate pinning is implemented in apps from the two platforms (RQ4). To do this analysis, we gather certificate chains that are served at destinations found to be pinning via dynamic analysis, as well as the certificates found in apps using static analysis. Our static analysis techniques (1) search for raw certificates present

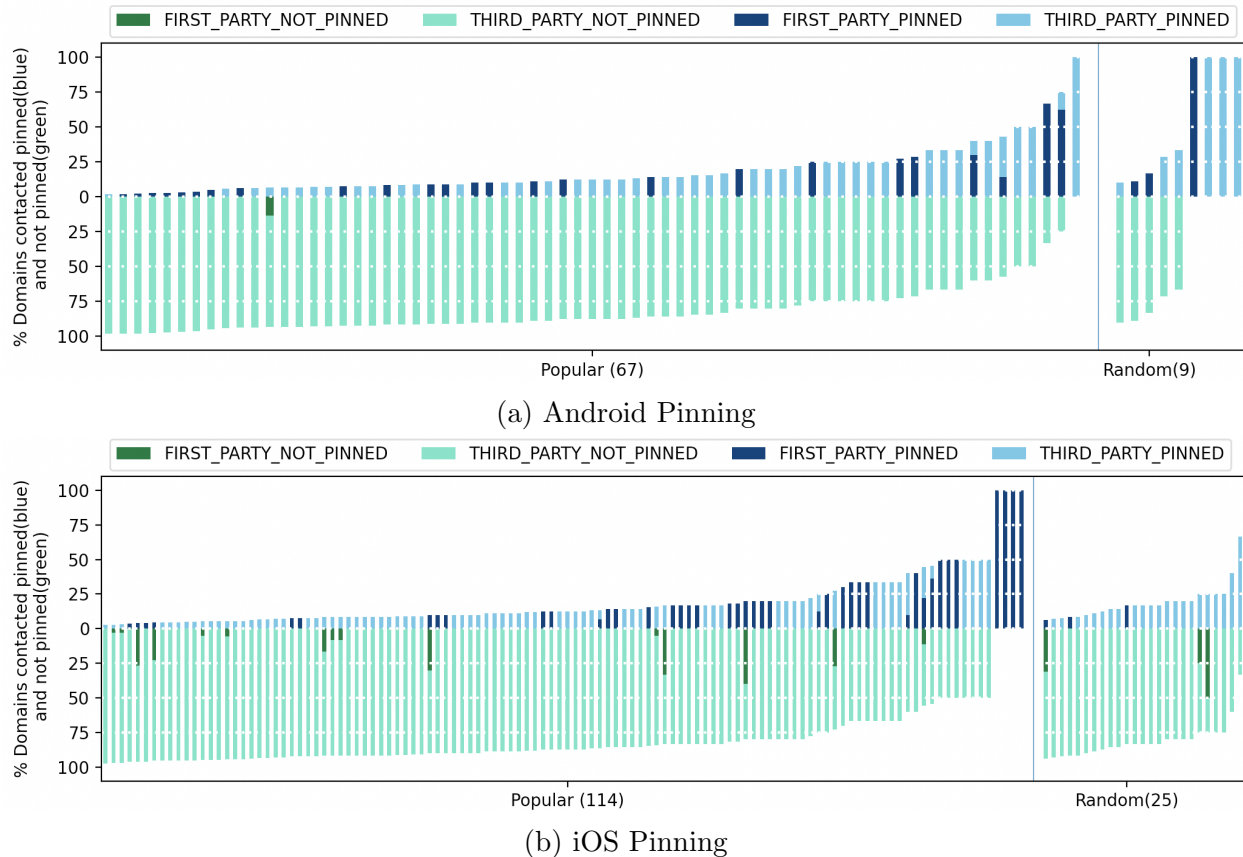


Figure 5.5: Percentage of domains contacted that are pinned vs not pinned. Dark colors represent first parties and light colors represent third parties. Each bar represents an app, from the Popular and Random datasets for Android and iOS respectively.

in an app, and (2) attempt to fetch certificates associated with any SPKI hashes present in the app. More specifically, our static analysis techniques discovered 966 unique certificates across all apps present in raw format, as well as 170 unique certificates associated with 50% of the unique pins from all apps. Using this static and dynamic certificate data, we shed light on the following aspects of pinning implementations:

The Public Key Infrastructure (PKI) Used: Both Android and iOS come pre-bundled with a default set of root CA certificates (the “default PKI”). We note that in the case of Android, Original Equipment Manufacturers (OEMs) may add additional root certificates in addition to those included in Android’s Open Source Project (AOSP) [15], [126]. Apps

Platform	Default PKI	Custom PKI	Data Unavailable
Android	163	4	11
iOS	238	1	14

Table 5.6: Type of Public Key Infrastructure (PKI) used by pinned destinations. The majority of pinning happens with certificates that tie to the default PKI.

that wish to implement pinning can either pin certificates that still tie to the default PKI, or use a “custom PKI” altogether by trusting their own root CA. To understand which of these two mechanisms are prevalent in apps, we validate certificate chains served at all pinned destinations using OpenSSL, configured with the latest version of Mozilla CA certificate store [127]. Further, we manually review the ones OpenSSL could not validate to confirm that they are indeed certificates tied to custom PKIs. Our results, summarized in Table 5.6, reveal that the vast majority of pinning happens with default PKI in use.

Interestingly, we also find two cases, one per platform, where the destination presents a self-signed certificate, rather than a chain. Although these destinations are reaping the benefits of certificate pinning, they are likely missing the flexibility provided by a PKI. To illustrate this, we note that the expiry dates for these certificates are 27 and 10 years. Due to this long validity period, and because these certificates cannot be revoked (revocation only applies to leaf certificates), any key compromise will mandate app updates to protect connection security.

Pinning Root vs Leaf Certificate: Apps can choose to either pin the root or a leaf certificate from the certificate chain, with the former offering more flexibility while the latter offering more security. More specifically, pinning a leaf certificate protects the TLS connection from all CAs, including the issuer. But on the other hand, leaf certificates have shorter expiry periods, and their keys are more likely to be rotated for security reasons. As such, pinning leaf certificates demands more management, and can even render apps unusable if they are not updated to reflect the latest leaf served from a destination.

To understand which type of certificates apps choose to pin, we cannot only rely on dynamic data alone as this data reveals certificate chains presented at pinned destinations. Rather, we need to investigate which certificate in the chain is likely pinned in an app’s code using static analysis methods (see Section 5.3.2). In our analysis, we find $\approx 31\%$ of pinning apps across the two platforms, for which there is at least one certificate that appears in both static and dynamic data (certificate matching is done in terms of the Common Name). We find the majority of these certificates to be CAs (80/110), and the remaining (30/110) to be leaf certificates.

Pinning Entire Certificate vs Its Key: As mentioned earlier, pinning leaf certificates can lead to unavailability issues if the certificates are updated at the server but an outdated app version is used, or if the developers forget to update pinned data on the app. There is one exception since pinning can be done via a certificate’s Subject Public Key Information (SPKI): app developers can update certificates on the servers as long as the certificate key remains unchanged. Our data indicates that this is indeed how app developers implement pinning. More specifically, out of the 30 leaf certificates that we found to be pinned in the previous section, 24 of them were pinned via SPKI hashes. The remaining 6 leaf certificates are present in their raw format in the apps, thus the developer could either pin the whole certificate or just the public key. In 5 of these 6 cases, we notice that destinations serve new leaf certificates during dynamic testing, which still result in pinned connections. This suggests that app developers likely pinned public keys for these certificates. Although this is good news for app usability, it also implies that certificate keys are reused which, in-turn, defeats the purpose of certificate renewals.

Platform	Framework	# apps
Android	Twitter	29
	Braintree	27
	Paypal	25
	Perimeterx	9
	MParticle	9
iOS	Amplitude	45
	Stripe	34
	Weibo	24
	FraudForce	16
	Adobe Creative Cloud	13

Table 5.7: Top 5 third-party frameworks that include certificate in Android and iOS. We combine paths where certificates are found across apps and provide occurrences here.

Subverting Proper Certificate Validation: Because certificate pinning only protects TLS connections against particular attacks, any pinning implementation still needs to conduct other certificate validation checks as defined in the TLS protocol (e.g., certificate subject name match, date validation) to protect against other attacks [125]. To see if any apps that use pinning bypass other standard certificate validation checks, we check for expiry dates of certificates served at pinned destinations. We do not find any certificates that are expired but were considered valid by apps during dynamic analysis. As such, we do not find any evidence of apps subverting normal certificate validation to only rely on pinning as the protection mechanism.

Third-party Frameworks That Introduce Certificates: We finally look at the package code paths in apps where our static analysis detects certificates and/or pins to attribute the behavior to first-party or third-party code. We observe that many of these paths appear in multiple apps. Upon manually investigating for the top common paths with certificates, and removing generic ones (such as `config.json`), we present the list of various third-party frameworks that we identify to likely be introducing certificate pinning logic to the apps in Table 5.7. For some of these, we are able to trace the pinning code in their open-source repositories (e.g., Twitter SDK, MParticle SDK). We note that some of these frameworks are also associated with popular pinned domains from our dynamic analysis (e.g., `config2.mparticle.com`, `*.perimeterx.net`). Last, we believe that the end-points that did not appear during our dynamic analysis are likely the ones for which we were unable to automatically trigger the associated code paths. We particularly believe this to be the case with Paypal that appears as a popular pinned domain in iOS, but never appears in Android (except for the Paypal app). Overall, our analysis reveals social networks, payment processing systems, and app analytics frameworks are the common sources of third-party code that introduces certificate pinning in apps.

Dataset	Bad Ciphers		
	Overall	Pinning apps	
Common	<i>Android</i>	8.35%	23.4%
	iOS	93.39%	55.77%
Popular	Android	18.3%	1.49%
	iOS	95.2%	46.09%
Random	Android	3.1%	0.0%
	iOS	82.6%	52.94%

Table 5.8: Weak ciphers found in pinned vs all connections across all datasets for Android and iOS. In general, we see pinning apps increase connection security in pinned connections as they disable weak ciphers more often than other apps in the dataset. The Common Android dataset (*italics*) is an exception to this trend supporting weak ciphers more often than the rest of the dataset.

5.4.4 Connection Security

In this section, we explore whether apps that use pinning also adopt other security practices in their pinned TLS connections (RQ5). More specifically, we check whether these TLS connections advertise support for bad ciphersuites (e.g., DES, 3DES, RC4 or EXPORT) that are susceptible to attacks. We compare their prevalence with connections from all apps to contrast security practices of apps that implement pinning. Table 5.8 shows our results. “Overall” shows the percent of all apps in a dataset that have at least one TLS connection with bad ciphers, while “Pinning apps” shows the percent of apps with certificate pinning that have at least one *pinned* TLS connection with bad ciphers.

Across all three iOS datasets, we see an increase in connection security of pinning connections when compared to the overall connections in every set. Weak ciphers drop from 93.39% to 55.77% for Common iOS, 95.2% to 46.09% for Popular iOS, and 82.6% to 52.94% for Random iOS datasets. However, trends in Android are more nuanced. For the Common Android dataset, we see that pinning apps reduce connection security as the percentage of bad ciphers in pinning apps is higher (23.4%) than that of the overall dataset (8.35%). But for the Popular and Random Android datasets, we see an increase in connection security of pinning connections as compared to other apps in those sets. Weak ciphers drop from 18.3% to 1.49% for the Popular set and from 3.1% to 0.0% for the Random set. Thus, with the exception of the Common Android dataset, our data suggests that pinning apps likely have better connection security for their pinned connections when compared to non-pinning apps on both Android and iOS.

5.4.5 PII in Pinned vs Non-Pinned Traffic

Since pinned TLS connections are harder to inspect by device users and auditors, in this section we try to understand whether app developers implement pinning in order to hide sensitive PII data collection, rather than to improve user security (RQ5). To do so, we inspect PII prevalence in decrypted TLS connections for all apps that implement pinning

Platform	PII	Pinned	Non-Pinned
iOS	Ad. ID*	25.85 %	18.06 %
	City	0 %	0.94 %
	State	0 %	0.31 %
	Lat./Lon.	0 %	0.04 %
Android	Ad. ID	25.74 %	19.96 %
	Email	0.99 %	0.52 %
	State	0.99 %	1.12 %
	City	0 %	0.45 %

Table 5.9: PII found in pinned connections, and how the prevalence differs from non-pinned TLS connections. (*) marks results that are statistically significant.

using the methodology described in 5.3.5.

Our results are presented in Table 5.9 and reveal what PII is found in pinned traffic, and how does the prevalence differs for non-pinned traffic. Since the number of non-pinned destinations is orders of magnitude more than pinned ones on both platforms, we cannot simply compare the PII prevalence across the two categories. As such, we highlight the results where differences in PII prevalence are statistically significant (found using Chi-square test of independence with a p-value ≤ 0.05). We find that advertisement ID is the key identifier that appears substantially in both pinned and non-pinned traffic. Although it appears more in pinned traffic, the differences we see are statistically significant in only one platform. We do not find substantial presence of other identifiers that we checked for. As such, our results suggest that app developers likely do not use pinning as a method to hide PII data collection.

5.4.6 Limitations

We discuss limitations of our methodology here. We also claim to find a lower bound of certificate pinning, which remains unaffected by these limitations.

Embedded Certificates We search for certificates embedded in apps, but could miss certificate for various reasons, such as apps using obfuscated code, reconstructing certificates at run time, storing certificates in non-standard formats, *etc.*

Partial Observation Our dynamic testing is limited; we do not explore all code paths of an app. Thus, we miss certificate pinning that is not triggered during our testing. Similarly, we do not have ground-truth about all of the PII that apps collect. Our analysis instead is limited a subset of PII that we could infer automatically in network traffic.

iOS Background Traffic As discussed in Section 5.3.6, we exclude iOS “associated” domains from our pinning calculations to avoid introducing noise due to OS-initiated background traffic. This may lead to an underestimation of pinning on iOS.

Limited App Interaction Though we explored automated interactions with apps, we found they had a limited impact on results. We did not log into or interact with apps after doing so. Thus, we potentially miss pinned connections in such scenarios.

Dataset Given that the dataset is collected from official stores, we do not capture the prevalence of pinning outside of official channels. Similarly, we do not cover prevalence for paid apps. Lastly, we tested a relatively modest number of apps ($\approx 3,000$), largely due to scalability constraints for dynamic testing. While we partially mitigate this by selecting different collections of apps (Popular, Random), our study represents only a sample of all available apps.

5.5 Discussion

Pinning Inconsistencies: We introduce the concept of *inconsistent* and *consistent* pinning for the Common dataset of apps. Apps from the Common dataset are developed and maintained by the same entity (developer, company, *etc.*). Thus, we expect the pinning policies to be consistent across these two mobile platforms. We find that this is rarely the case, with less than half the apps having completely consistent pinning. This indicates that the security practices of the same entity are different on Android and iOS, and is an interesting finding as it is unexpected.

We argue that pinning consistently, across platforms, is good practice. Although codebases for various platforms might vary, the reasoning behind pinning should be the same. We can only speculate about the reasons for such differences, e.g., they could be due to different pinning APIs across OSes causing confusion/inconsistency (e.g., as found by Oltrogge et al. [124]), or due to developers using different threat models for iOS compared to Android.

Developer Survey: Our work revealed practices that cannot be explained by our dataset alone. By surveying developers who use pinning, we can better understand the reasoning behind pinning, including why there are inconsistencies between Android and iOS. Such a survey can also help the community to better understand deployment/maintenance requirements, and compile a better set of guidelines for developers that wish to use certificate pinning.

App Exploration: An orthogonal problem we encountered during our study was app exploration. We tested random automated interactions with apps but found no significant change in traffic generated by the apps with or without these interactions. Developing a tool that automatically interacts with apps (signing up, logging in *etc.*) would be useful for various future studies.

Pinning Circumvention: In this work we used existing techniques to circumvent pinning to study data that is protected behind pinned connections. The number of connections we circumvented was limited ($\approx 50\%$ destinations). We leave it to future work to develop techniques that can circumvent a larger number of pinned connections, enabling studies of data protected by pinning.

Ethical Considerations: Our work does not entail human subject research. All tests were conducted using accounts set up for the sole purpose of our testing. Our methodology requires crawling app stores, and we used low crawling rates with accounts that are easily identified as being used for research purposes (in case the platforms took notice of our crawls and needed to contact us). Similarly, while crawling the AlternativeTo website, we limited

our crawler to request 1 page/second and included our contact details in the User-Agent field. We received no complaints about our crawls, nor were any of our accounts disabled or rate limited in any way.

5.6 Conclusion

In this work, we conducted the first large-scale study of certificate pinning across both Android and iOS apps. We found significantly higher prevalence of pinning than in prior studies, with at least 11% of popular iOS apps and 6.7% of popular Android apps doing so. Interestingly, we found that pinning behavior varies significantly across platforms, even for the same app. Based on our analysis, pinning is commonly added by third-party libraries and is likely deployed for the protection of financial data, with little evidence that pinning is used primarily to protect (non-credential) personal data. In future work, we will explore how results change with more app interactions, both automated and manual.

To support reproducibility and facilitate further research in the area, we make our dataset and code publicly available at: <https://github.com/NEU-SNS/app-tls-pinning>.

Chapter 6

Testing TLS Certificate Validation Using Generative Language Models

The sheer complexity of TLS spec makes it impractical for application developers to implement the protocol from scratch. Rather, a large number of open-source TLS implementations (e.g., OpenSSL, MatrixSSL, MbedTLS) exist and can be integrated directly in applications. As is the case with any computer software, ensuring that a TLS implementation follows the spec correctly and remains bug-free is hard to achieve. Unfortunately, programming bugs in implementations are regularly found, leading to various security issues such as denial-of-service vulnerabilities or potential leaks of user data from TLS-supporting servers.

Prior research (e.g., [19]–[21]) has leveraged an interesting insight to systematically find bugs in the certificate validation logic of TLS implementations, an essential step in the protocol to ensure server authenticity. Since multiple implementations exist and follow the same protocol spec, prior works use them as cross-referencing oracles to form a differential testing framework: given an input certificate, these works compare the validation outcomes of multiple libraries to find discrepancies (where one implementation accepts a certificate while another rejects). As these implementation differences should generally not exist, prior works find many of these discrepancies to reflect bugs and vulnerabilities in implementations.

Several different techniques exist to generate the corpus of test certificates to use in differential testing. For instance, randomly combining and mutating parts of real certificates [19], using code coverage statistics to guide generation [20], and modelling certificate parameters to use combinatorial methods [21]. Inspired by the prior work, we identify a new opportunity for differential testing: generative language models based on neural networks (e.g., ChatGPT), which are popular today for applications such as generating content, writing code and conversing with users.

In this chapter, we present a novel approach to generate synthetic TLS certificates for differential testing using generative language models. We demonstrate how these models are able to (i) learn X.509 representations for TLS certificates, and (ii) generate new certificates that (usually) conform to the protocol standard but produce diverse behavior during the certificate validation process for several TLS implementations. Our results show that these synthetic certificates produce an order of magnitude more “distinct discrepancies” than baseline during

differential testing, and reveal a wide range of previously unobserved and interesting behavior with security implications.

6.1 Background and Motivation

In this section, we provide the necessary background for TLS and differential testing, how it relates to language models, and how we use these models as blackbox tools in this work.

TLS Certificates A crucial part of TLS connection establishment is the validation of one or more “certificates” that contain cryptographic information for a host (e.g., the public key for *google.com*). Clients must validate these certificates according to the protocol specification (spec) that, among numerous other things, checks if a certificate is authentic based on its signature from a trusted Certificate Authority (CA), unexpired at the time of checking, and is indeed issued for the domain client wants to connect to.

The TLS spec is complex and defined semi-formally in various RFCs released over the past two decades [128], [129]. The latest protocol version is TLS 1.3, and any version below TLS 1.2 is considered unsafe today. TLS certificates are defined in a standard known as “X.509”, which itself is based on an interface description language “ASN.1”. Standards defined in ASN.1 can be efficiently serialized/deserialized in a cross-platform way. TLS certificates are usually encoded in a binary “DER” format (and stored/transferred in base64-encoded “PEM” strings with ASCII characters). An example TLS certificate in this encoding is present in Listing 6.1. The X.509 standard has also undergone various updates, with version 3 being the most commonly used today, and is defined in RFC 5280 [23].

A large number of TLS implementations conform to the TLS spec and are integrated into other software. Unfortunately, due to the complexity of TLS protocol and correspondingly high chance of bugs during software development, vulnerabilities in TLS implementations are common (e.g., [130], [131]). As a result, TLS has been the target of many automated efforts to discover and address such flaws in implementations.

Differential Testing While there are numerous works that assess TLS implementations, here we focus on the work by Brubaker et al. [19] that introduced the idea of differential testing in the context of certificate validation (an extended discussion of related works is in Chapter 2). The authors identified two fundamental requirements for systematically testing certificate validation logic in implementations: (i) generating a variety of valid test certificates to find bugs in rarely triggered code paths, and (ii) determining if a certificate validation result from an implementation is correct. The authors proposed a novel solution to meet these requirements by (i) randomly combining and mutating parts of real certificates to automatically generate a large synthetic corpus of certificates (*Frankencerts*) that are syntactically valid but may violate some of the semantic constraints, and (ii) using multiple TLS libraries as cross-referencing oracles to find bugs. For the latter, the key idea is that since TLS libraries attempt to conform to the same protocol spec, a “discrepancy” (where one library rejects a certificate but another does not) can indicate a potential issue in either of the libraries. The authors indeed observed that the majority of discrepancies found in their work reflected bugs and security issues in TLS implementations. While the technique

helped significantly improve TLS implementations, the randomness in the technique means it can take an arbitrarily long time to capture all bugs in a program.

Software Security and Language Models The core insight in our work is that since TLS certificates are defined in X.509 format, we can use a suitable compiler to deserialize the certificates from a memory-efficient DER/PEM format into a verbose textual format (Listing 6.2). An interesting side-effect of such conversion is that the textual format is suitable for language models (detailed in next section). These models can learn a representation of the training data (e.g., a corpus of TLS certificates), and one can generate new examples by sampling from the representation (e.g., to generate a synthetic certificate). Interestingly, these models may learn representations that are not perfect, resulting in sampled outputs that may differ from expectations. In the context of differential testing and fuzzing, such imperfect representations can actually help find bugs. That is because synthetic certificates that deviate slightly from the TLS spec, for example, can trigger rarely tested code paths, increasing the likelihood of finding bugs.

To the best of our knowledge, language models have not been used in the context of generating synthetic TLS certificates before, and have had limited use in software security so far. We believe Godefroid et al. [132] are the first to use modern language models for the purposes of fuzzing. The authors trained a model for fuzzing the complex PDF input format. While they were able to successfully learn a usable representation that can generate novel PDF objects, they could only find one stack-overflow bug. Notably, the authors did not rely on differential testing but used *AppVerifier*, a low-cost runtime monitoring tool, to catch memory bugs. Regardless, the authors found the technique promising in general and provided an early discussion on how the model training process influences quality of fuzzing (more details in Section 6.3). Our work can be seen as an extension to theirs—we complement the use of language models with differential testing, and do so in the domain of TLS certificate validation. Note that work by Sablotny et al. [133] is another such extension to use language models to generate synthetic HTML tags and fuzz web browsers, but they also did not rely on differential testing and did not find any bugs.

Listing 6.1: A TLS certificate shown in PEM format

```
-----BEGIN CERTIFICATE-----
MIIEHTCCAwwGAWIBAgIQToEtioJl4AsC7j41Akb...
...NBIVBAMTHkNPTU9ETyBDZXJOawZpY2F0aw9uIEF1
-----END CERTIFICATE-----
```

Listing 6.2: A TLS certificate shown in ASN.1 textual format

```
{
  toBeSigned {
    version 2 -- v3 --,
    serialNumber 104350513648249232941998508985834464573,
    signature {
      algorithm {1 2 840 113549 1 1 5}
    },
    issuer rdnSequence : {
      {
```

```

    {
      type {2 5 4 6} -- id-at-countryName --,
      value '4742'H
    }
  },
  ...
  validity {
    notBefore utcTime : "061201000000Z" -- Fri Dec 1 00:00:00 2006 --,
    notAfter utcTime : "291231235959Z" -- Mon Dec 31 23:59:59 2029 --
  },
  ...
  extensions {
    {
      extnId {2 5 29 14} -- id-ce-subjectKeyIdentifier --,
      extnValue '04140B58E58BC64C1537A440A930A921BE47365A56FF'H
    }
    ...
  }
}
}

```

Generative Language Models We now discuss language models in more detail and how to use them as blackbox tools. Generative language models based on neural networks have received widespread attention in the last few years. The human-like text generator ChatGPT [134], released in November 2022, accumulated 100 million monthly active users within just two months [135]. That is because of the remarkable performance of these purely language models on a variety of tasks such as writing code, solving mathematical questions and answering user questions.

In simple terms, generative language models learn to predict a sequence of tokens when given an input sequence. A token can represent one single character, but even multiple characters or a unit in some domain other than natural language (e.g., a musical note). This generalizability is what makes them extremely powerful, and enables us to use off-the-shelf modern language models as blackbox tools in this work. It is notable that by simply optimizing performance on a token prediction task, a (well-trained) model can learn a useful representation of what the training dataset expresses (e.g., how a joke is written in English). The representation can then be sampled from for different tasks (e.g., generating a new joke using "knock knock" as an input sequence).

We use two types of generative language models in this work; Generative pre-trained transformers (GPT), and, Recurrent neural networks (RNNs). We rely on these two because GPTs are the current state-of-the-art, and RNNs were the dominant sequence prediction models in the last two decades with widespread impact [136]. RNNs were also used in the prior study closely related to our work (mentioned in 6.1). For additional context, RNNs originated in 1986, while the more commonly used "LSTM" variant appeared in 1997. GPTs are comparatively recent, and rely on a "Transformers" architecture that was introduced in 2017. GPTs are part of what is colloquially known as "large language models" (LLMs) due to their massive scale of trainable model parameters, datasets, and compute requirements.

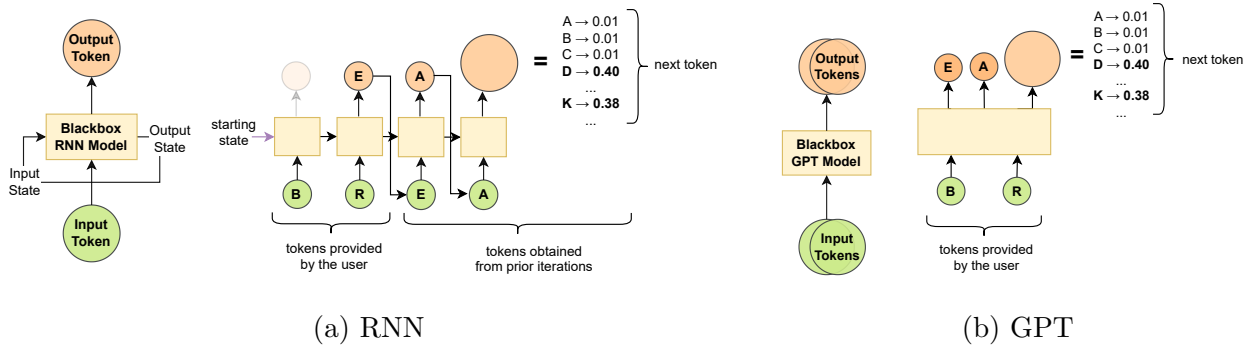


Figure 6.1: An abstraction for the language model architectures used in this work. A pipeline on how to use each model for generating text is on the right.

We present an abstraction of these models in Figure 6.1 that is suitable for understanding them as blackbox tools in this work. As security researchers, our approach is to intentionally rely on off-the-shelf language models with our basic machine learning skills, rather than to create custom model architectures specialized for the purposes of software security.

We now describe how these architectures can be used. As a blackbox, RNNs expect a single input token and an input “state”, and produce an output token and state. They are used in a sequential manner to predict token sequences of arbitrary length, with the key insight of using the output token and state at one step, as input to the subsequent step for predicting the next token. As such, the state can be used by the RNN to keep track of historical information and make token predictions at each step based on the entire sequence from prior steps. A pipeline of such use is present in Figure 6.1a. On the other hand, GPTs work on limited-length sequences and can be seeded with an input sequence of tokens to get an output sequence. A pipeline of such use is present in Figure 6.1b. The lack of state maintenance in GPT makes them much easier to train in parallel, as opposed to RNNs. It also means that GPTs are able to explicitly consider all prior input tokens. These two features, in practice, make GPTs much efficient than RNNs. Note that for both an RNN and GPT, the output tokens are actually sampled from a probability distribution (i.e., the models output a score for each token that can be used as a probability distribution, rather than a single token). The sampling strategy determines how creative or constrained outputs from a generative model get to be. It is also common to select the token with highest probability as the output token.

These language models are trained on a corpus of textual data. It is particularly common to use a GPT pre-trained on a very large corpus of generic data with extensive compute, and to only “fine-tune” it with a small set of data for the task at hand. The process of fine-tuning means model parameters can be updated for the new training data, but do not need to be bootstrapped from scratch. Intuitively, fine-tuning helps because a pre-trained model can quickly learn a new task if it has already acquired extensive knowledge for some overarching problem (e.g., sequence prediction).

Table 6.1: Overview of certificate datasets used for training language models.

Dataset	Total certificates	Versions			Extensions			Validity	
		v1	v3	others	Per certificate (median)	Names (unique)	Values (unique)	notBefore year (median)	notAfter year (median)
IPv4	100000	1790	98207	3	8	28	92630	2019	2025
2022	100000	11893	88102	5	9	31	110916	2021	2022
Balanced	100000	50036	49944	20	0	29	85480	2020	2023

6.2 Goals

While the ultimate goal of our work is to find new security issues in TLS implementations, we measure progress to our goal by monitoring the following three statistics, in increasing order of priority; (i) code coverage, as it generally reflects the ability to find bugs in different sections of a program, (i) number of unique discrepancies, as they should generally not exist in spec-conforming implementations, and (iii) software bugs, as they can often be exploited to cause security issues. With this work we aim to answer the following research questions (RQs):

RQ1: How can we use language models to learn a representation of X.509 TLS certificates?

RQ2: How can we generate diverse certificates from a learned representation to discover large numbers of discrepancies?

RQ3: Do these synthetic certificates help uncover new bugs and security issues through differential testing?

RQ4: How does certificate testing using language models compare with prior work?

6.3 Methodology

In this section, we describe the datasets, language models and the differential testing pipeline we use in our work. We start with a brief overview of our methodological approach.

Parameter selection Training modern machine learning models is a particularly parameterized process. The final system performance not only depends on the choice of training dataset and model architecture, but is also influenced by a wide range of other “hyperparameters” used (e.g., learning rate, number of layers, size of a training batch). Typically when training these models, the learning goal is sufficiently expressed through the training “loss function” used in the optimization process. Thus, the parameters are empirically selected to get a trained model with the smallest loss value, while the search space of parameters is only restricted on practical concerns such as time.

Interestingly, for the purposes of fuzzing, we do not know *a priori* how optimizing the loss function relates to the generation of synthetic certificates that trigger more bugs or discover new security issues. In general, a loss function specific to finding bugs cannot be designed either, since the set of *all* security issues in a program is unknown.

In more detail, Godefroid et al. [132] are the first to highlight an inherent tension between

learning and fuzzing – the former optimizes for generative outputs that conform to some inherent structure in the training data, while the latter exploits inputs that break the structure to reach bug-inducing code paths. Note that differential testing is able to find bugs even through well-formed software inputs, so it is not an inherent conflict in our case. But we still cannot assume whether a state-of-the-art model architecture trained extensively, or a simpler model trained briefly would be more suitable for the purposes of differential testing.

Given the open question of how to train and tune generative models and the large parameter space to explore, we rely on the following heuristic approaches. First, we train a variety of models (12 in total, based on 3 training datasets and 4 models detailed in the next section) using default and/or common-sense parameters (e.g., a learning rate not too high or too low). Second, we evaluate how these different models help us achieve goals in this work and, for scalability reasons, use only the best among them for in-depth experiments. As such, we do not claim that any particular model is best in general, but only how it compares to the different techniques we tried.

6.3.1 Certificate Datasets

Our approach relies on an input set of TLS certificates to train the language models. The models, in turn, are designed to learn the representation of a certificate based on the features present in the training set. For instance, if we train the models using only *v1* certificates, the models will naturally be incapable of generating certificates with TLS extensions (only available in *v3* certificates). To generate synthetic certificates with a diverse range of features, we use three distinct datasets for training:

IPv4 (n = 100,000): Certificates crawled from websites by randomly searching the IPv4 space for TLS-enabled hosts (until the desired number of certificates were found). The crawl was performed in February 2022. This dataset represents certificates typically found online. A limitation is that we could not specify the hostnames during this crawl and, as such, may obtain certificates that are less commonly seen by end users (i.e., “default certificates” instead of domain-specific ones).

2022 (n = 100,000): Certificates deployed in the year 2022 and obtained from the Rapid7[137] scans. This dataset represents certificates that are recently issued, and are thus expected to contain the latest TLS features with more prevalence than the previous dataset (as certificates can be long lived, and websites do not necessarily need to update them).

Balanced (n = 100,000): A mix of *v3* and *v1* certificates sampled from the Rapid7 scans. The sampling was done in a way that *v3* and *v1* certs are equally prevalent in the dataset.

An overview of the features contained in these datasets is present in Table 6.1. The diversity in these datasets can be observed by looking at the distribution of different features (e.g., expiry dates).

We cannot use raw TLS certificates for training purposes because they are encoded in the compact PEM data format (Listing 6.1). Instead, we rely on the *PyCrate*[138] framework to decode them into “ASN.1 textual format” that is suitable for language models (Listing

6.2) due to the verbosity afforded by that format. To facilitate model training, one change we make in the certificate training data is that we remove the public key and signature fields from the certificates. We do so as these are special fields that need to be calculated with cryptographic functions, and the language models cannot learn these functions without the appropriate information (e.g., corresponding private key). Note that synthetic TLS certificates with invalid signatures will likely provide limited value in finding bugs in a TLS implementation.

6.3.2 Language Models

As discussed earlier, we use the following two types of language models in this work:

RNNs We use the open-source *Char-RNN-PyTorch* implementation[139] with an LSTM variant, as done in prior work [132]. We train two 3-layer models from scratch, one with 256 hidden neurons per layer (henceforth called “RNN-Small”, ≈ 1 million trainable parameters in total) and the other with 1024 (“RNN-Medium”, ≈ 10 million trainable parameters). Note that the implementation treats each character as a token.

GPTs We use the open-source *gpt-neo-125m*[140] implementation, which is inspired by the GPT-3 architecture but with ≈ 125 million trainable parameters (as opposed to ≈ 1 billion for the base GPT-3 model) for faster training. Our results did not indicate a need to train larger models as we will discuss in the results section. We fine-tune one pre-trained model (“GPT-Finetuned”) and train another from scratch (“GPT”) using our datasets. For both models, the maximum sequence length is set to be 2048 tokens, the default for GPT-Neo.

In summary, we have 4 models with sizes that can differ by two orders of magnitude. Note that larger models typically lead to better performance. Using the previously described 3 datasets, we train 12 models in total. For each training certificate instance decoded into an ASN.1 text, we add a prefix string at its start and a suffix string at its end (e.g, “BEGINCERTIFICATE” and “ENDCERTIFICATE”). We train the models end-to-end on each certificate instance.

Generating certificates We simply use the prefix as input to a trained model and generate an output string that ends with the specified suffix (we also set a max output length limit in case the suffix is never produced). Because the models are trained on ASN.1 inputs, the outputs are expected to represent the ASN.1 language. As such, we encode each model output into the PEM format expected by TLS libraries. Note that the certificate outputs may not be well-formed ASN.1 or ASN.1 compatible with the X.509 certificate format, so the conversion is not guaranteed to succeed. We call each well-formatted PEM instance a “synthetic certificate”.

Once we obtain a certificate, we rely on the following three approaches for using it during differential testing:

Leaf Using the synthetic certificate directly with a dummy public key and dummy signature.

Leaf+CA Using a custom (manually-trusted) v3 root CA as the issuer CA, and attach

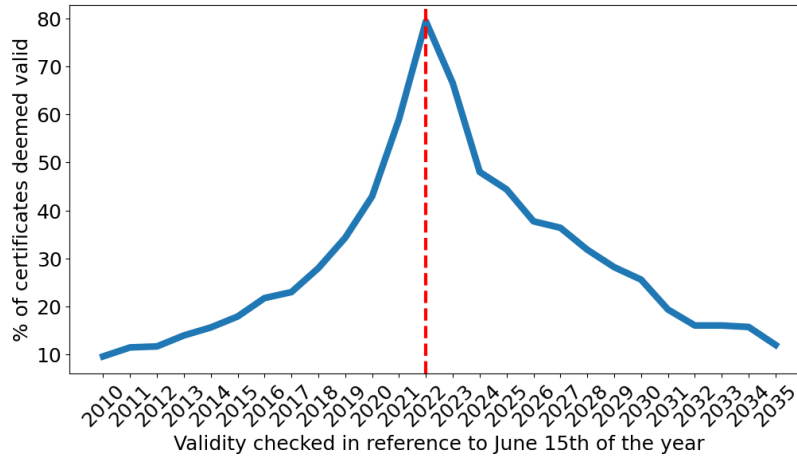


Figure 6.2: How certificate validity is influenced by the date of testing. Date used in our methodology is highlighted in red (*June 15, 2022*).

Library	Version
OpenSSL	1.1.1t
LibreSSL	3.6.2
GnuTLS	3.6.16
MbedTLS	3.3.0
MatrixSSL	4.6.0

Table 6.2: TLS libraries in our differential testing framework. All versions reflect the latest available on February 15th, 2023.

its valid signature.

Leaf+CA+Intermediate Chaining multiple synthetic certificates together to produce a valid chain with one root (manually trusted), one leaf, and $X \geq 1$ intermediate certificates.

6.3.3 Differential Testing Framework

After generating synthetic certificates through language models, we use them to test TLS libraries through our differential testing framework. In a nutshell, the framework takes a certificate as input, executes it against the validation code of multiple TLS libraries, and compares their output against each other to find discrepancy-producing certificates.

In more detail, the certificate validation output from each library is categorized as either success or failure (based on the program exit status code). A *discrepancy* is when at least one library differs in validation outcome from others. We test 5 popular open-source TLS libraries (Table 6.2) (our framework can easily be extended to support more.) With 5 libraries, the synthetic certificates can generate a maximum possible of 30 “distinct (sets of) discrepancies” (2^5 possible outcomes, minus 2 for all success/all failure outcomes). In addition, we collect any outputs to *stdout* and *stderr* streams alongside code coverage data

Table 6.3: Overview of PEM certificates obtained from the language models. Each model was used to output 100,000 strings.

Training Data	Model	Valid certificates	Versions			Extensions			Validity	
			v1	v3	others	Per certificate (median)	Names (unique)	Values (unique)	notBefore year (median)	notAfter year (median)
IPv4	RNN-Small	35310	678	34631	1	7	24	43397	2020	2024
	RNN-Medium	97151	1897	95252	3	7	35	107105	2020	2025
	GPT-Finetuned	82081	1258	80818	5	7	37	33010	2016	2027
	GPT	74431	369	74062	0	7	21	25719	2015	2027
2022	RNN-Small	69953	20650	49301	2	3	18	106509	2021	2022
	RNN-Medium	34886	7304	27580	2	3	19	46710	2021	2024
	GPT-Finetuned	74215	23356	50847	12	1	42	71693	2021	2024
	GPT	44468	12567	31897	4	3	19	64566	2021	2024
Balanced	RNN-Small	938	711	227	0	0	12	675	2020	2023
	RNN-Medium	95965	65833	30130	2	0	27	60235	2020	2023
	GPT-Finetuned	81295	61581	19710	4	0	39	28557	2020	2023
	GPT	64254	47050	17204	0	0	17	35101	2020	2023

using GCOV.¹ We execute certificate validation logic in these libraries without domain name validation, as our certificate generation process is not constrained to any particular domain.

Certificate validity periods (e.g., whether a certificate is expired or valid at the time of testing) can significantly affect the flow of code executed during differential testing. For consistency, we need to validate all certificates using the same date and time. To achieve this, we rely on *libfaketime*[141] to patch low-level system libraries and simulate an artificial time for our experiments.

There is a trade-off to be made when deciding the particular time to use. If we choose a date on which a certificate is invalid (expired or yet-to-be-valid), it should result in less code execution in a library as the certificate might be rejected without validation of any other details (assuming time validity checks are performed earlier by a library than other checks). On the other hand, a certificate that is valid for the current time may result in more code execution, but not necessarily with the code paths that are rarely triggered and that may expose subtle bugs. To understand the trade-off better, we take a representative sample of certificates generated from our language models and show how the date chosen affects certificate validity (Figure 6.2). Given that there is no *a priori* way to determine the optimal date to use for differential testing, we use a heuristic based on the intuition that exploring significantly more code paths in general, but not completely ignoring rarely triggered paths, should help us uncover substantial numbers of discrepancies. We thus use *June 15, 2022* in this work as this timestamp lets a significant fraction, but not all, of the synthetic certificates to remain valid during testing.

6.4 Results

We present our results in three phases of evaluation. In Section 6.4.1, we evaluate the performance of the 12 trained language models in terms of code coverage and unique discrepancies. In Section 6.4.2, we conduct additional experiments to understand how certificate diversity influence the discovery of discrepancies. Based on what we learn from these two sections, we

¹Code coverage data is collected as aggregate from all certificate validations for a particular dataset.

choose a final model to generate synthetic certificates, then we use these certificates to test against multiple TLS libraries and we discuss discrepancies in terms of their behavior and implications on software security in Section 6.4.3.

We begin by evaluating if our trained language models generate valid X.509 TLS certificates (RQ1). Using the prefix string as input, we obtain 100,000 output strings from each model that end with the desired suffix (we find the suffix 99% of the times within the max output length). We then encode these model outputs into PEM-formatted certificates with varying success.

Table 6.3 provides an overview of the synthetic certificates obtained from the models. A key observation we make is that the models, in general, generate valid PEM certificates. The only exception is the RNN-Small model trained on the Balanced dataset, which produces valid X.509 outputs only 0.938% of the time. On the other hand, the RNN-Medium model trained on the IPv4 dataset produces the largest number of valid outputs among all models (97.151%).

Further, these models learn distinctive X.509 representations, which we summarize in the table in terms of the diversity in version and extension information present in their certificates. While the models trained on IPv4 and 2022 datasets produce significantly more v3 certificates, that is not the case with models trained on the Balanced dataset. In fact, while the Balanced training dataset had an equal proportion of v1 and v3 certificates, the models trained on this dataset produce more v1 certificates. Interestingly, these Balanced models have learned that v1 certificates do not have extensions, alluded by the fact that the median number of extensions is zero for these cases.

Finally, the models learn not only constraints about the X.509 format, but also other constraints from real-world certificate training data. For instance, the median `notAfter` field (representing expiry date) is ahead of the `notBefore` field (typically representing the certificate issuance date) for certificates from all models.

6.4.1 Discrepancies and Code Coverage

We now use synthetic certificates in the differential testing framework. We find that no discrepancies are produced when using the certificates directly with TLS libraries (*Leaf*). On the other hand, when we provide a valid chain (*Leaf+CA* or *Leaf+Intermediate+CA*), we trigger a large number of discrepancies. This means all libraries mark a synthetic certificate as invalid when not signed by a valid CA. When focusing on the other two methods, they each find a similar set of discrepancies. For the sake of simplicity, we present results in this section using only *Leaf+CA*.

Figure 6.3 shows the performance of these models on two axes – code covered and unique discrepancies triggered. In addition to the 12 models, we include two benchmarks for reference: certificates from one of the training dataset sources (IPv4) and certificates generated using the methodology of prior work (Frankencerts).

The first key observation is that the all language models trigger significantly more discrepancies (*median* 17.5) than the two benchmarks (4 and 5 discrepancies respectively), while the training dataset explores more code. We speculate the reason is that real-world certifi-

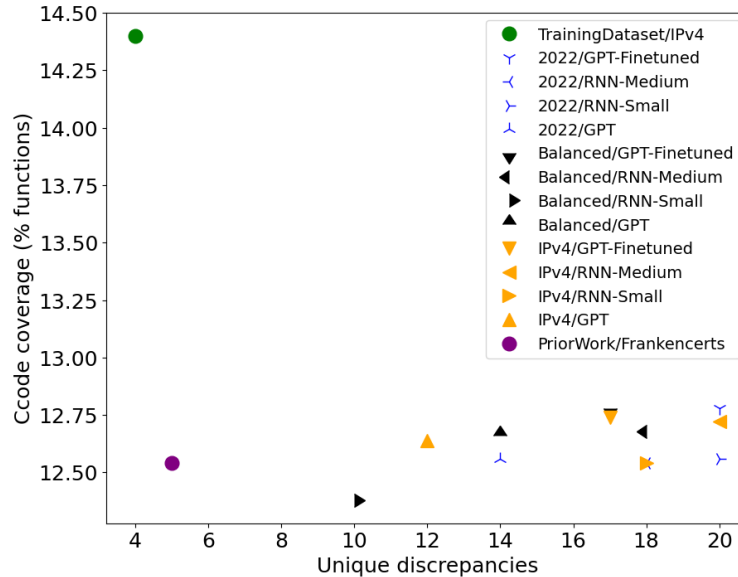


Figure 6.3: Performance of language models in triggering discrepancies and exploring certificate validation code. The models trigger significantly more discrepancies than benchmarks, despite covering less code than real-world certificates.

cates exercise more functionality in TLS libraries as they presumably conform more closely to the TLS spec. Note that exercising more code paths is not the same as exercising new code paths (e.g., code paths that may trigger bugs). While the code coverage may seem low across all datasets, it is because only a small portion of code in a TLS library deals with certificate validation.

Second, we note that none of the model architectures or training datasets optimize both code coverage and unique discrepancies. The various models that trigger large numbers of discrepancies do not necessarily use the same architecture, nor are they trained on the same dataset. In addition, the three models that produce the most discrepancies (20) may actually reflect similar type of discrepancies. On the other hand, models with fewer discrepancies might work as an ensemble to achieve an overall larger number of unique discrepancies.

To understand this further, Figure 6.4 uses a heat map to visualize the number of certificates that trigger each type of discrepancy, for each model. The number of unique discrepancies discovered across all synthetic and benchmark certificates is 23 (out of a maximum possible of 30). This means none of the individual models find all types of observed discrepancies. Three models find one discrepancy each that are not found elsewhere. Interestingly, a small number of certificates are responsible for these rare discrepancies. On the other hand, discrepancies found by a large number of models tend to be triggered by a larger number of certificates as well.

Our key takeaway is that all trained models help us find discrepancies significantly more than the benchmarks, but we do not have evidence to call any particular model architecture best among all.

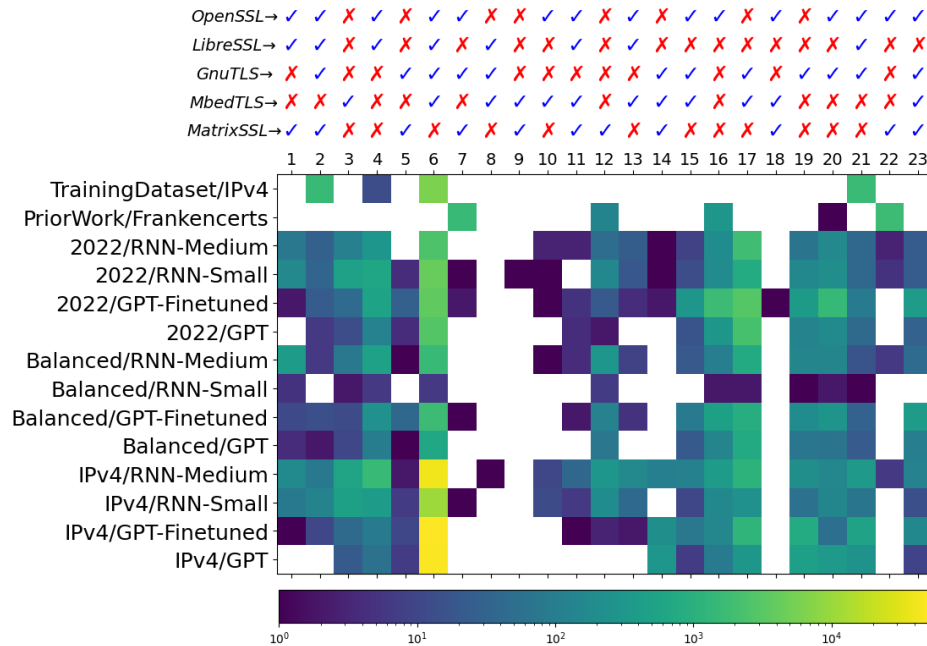


Figure 6.4: Exploring the discrepancy producing certificates. Each cell represents the number of certificates from a dataset (row) that trigger discrepancy of a particular type (column). Rows 1 and 2 represent benchmarks, and 3 onwards represent certificates generated by language models. The synthetic certificates trigger significantly more discrepancies than benchmarks, and mostly overlap on the type of discrepancies.

6.4.2 Certificate Diversity

We now evaluate the extent to which we can control the diversity in our synthetic certificates, and how this in turn leads to better code coverage and/or unique discrepancies observed (RQ2). Our intuition is that diverse TLS certificates (e.g., via an unexpected combination of extensions) may help trigger new discrepancies by executing rarely-tested code paths. Two simple ways to make a set of synthetic certificates more diverse is by (a) changing the sampling strategy used when generating certificates, and, (b) increasing the number of certificates obtained from a model. For practical purposes, we study these techniques on a single model type and training dataset. While there is no single “best” model as discussed in the previous section, we use *IPv4/RNN-Medium* because this model produced the most unique discrepancies. In addition, it has a moderate size in terms of trainable parameters as compared to the other two models with similar numbers of discrepancies. This makes the model more powerful than *2022/RNN-Small* yet significantly faster to train than *2022/GPT-Finetuned*.

Sampling strategy During certificate generation, we sample from a probability distribution to get output tokens. For instance, if the model outputs a distribution (0.5, 0.4, 0.1) for the three tokens **a**, **b** and **c**, the token “a” is selected as output with a probability of 50%. Interestingly, we can use a simple technique to sample in a way that makes outputs more diverse from, or constrained to, the representation learned by a model. Language models can

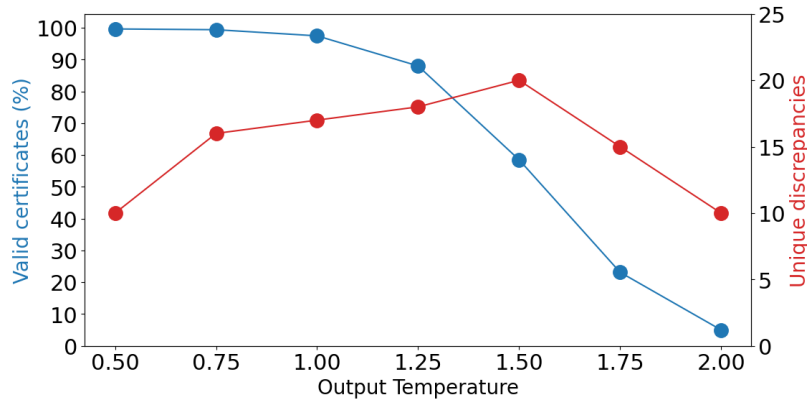


Figure 6.5: Trade-off between sampling strategy and discrepancies found. Sampling in a conservative way (left) makes more synthetic certificates valid, but they find less discrepancies.

be tuned with a hyperparameter called “Temperature” that controls how uniform the token probability distribution is. For instance, with $Temperature = 2.5$, the earlier distribution is scaled to (0.36, 0.34, 0.30) and with $Temperature = 0.1$ it is scaled to (0.72, 0.27, 0.01).

A more uniform sampling strategy will introduce diversity to synthetic certificates (by sampling different tokens), at the cost of making some certificates invalid (if some tokens violate ASN.1 syntax). To understand this trade-off, we test with multiple $Temperature$ values and plot our results in Figure 6.5. For each test, we generate 10,000 output strings, which are then encoded to PEM certificates. Note that by default, $Temperature = 1$ is used (i.e., results in the previous section). As expected, a low value leads to a large number of model outputs being encoded as valid TLS certificates. On the other hand, increasing the $Temperature$ helps discover more discrepancies but only up to a certain point, after which most certificates become invalid. Based on these empirical results, the best value to find most discrepancies is $Temperature = 1.5$.

Number of certificates Because we use each model to generate a maximum possible of 100,000 certificates, an interesting question is whether these models can find more unique discrepancies when given more time. To understand this potential, Figure 6.6 shows the number of certificates from a model (on average) that trigger a certain number of discrepancies. The grey dashed region represents certificates that did not help in finding any new discrepancies.

We observe that two-thirds of the discrepancies are produced by just 20,000 synthetic certificates. In addition, the arc of the curve indicates diminishing returns in terms of new discrepancies when using more certificates. Nonetheless, there is a nonzero likelihood of finding at least some new discrepancies with additional certificates beyond 20,000, so we use a larger sample in our experiments in the next section.

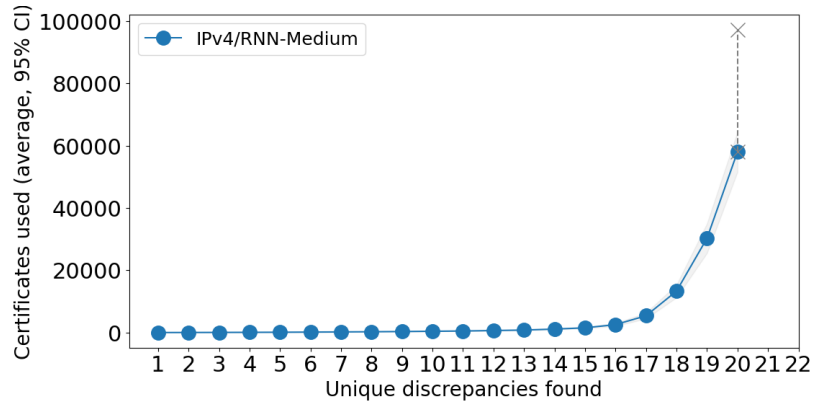


Figure 6.6: Number of certificates needed to observe new discrepancies. Majority of the unique discrepancies are observed with just 20,000 synthetic certificates.

6.4.3 Security Implications

Based on what we learned from the previous section, we use our trained *IPv4/RNN-Medium* model with $Temperature=1.5$ to generate 1M certificates. We use results from this experiment to discuss the behavioral and security implications of our findings. Interestingly, this certificate set triggers all 23 unique discrepancies reflected in Figure 6.4, but no additional new discrepancies.

Our focus on this section is understanding the root causes for the discrepancies we observe. While a large number of synthetic certificates may trigger a particular discrepancy, the number of root causes behind that discrepancy may only be a handful. These root causes will reflect differences in TLS implementations. The differences can either be benign (i.e., where the TLS spec allows for the libraries to differ) or indicate programming bugs, with or without security implications.

We analyze discrepancies in two ways to understand the root causes: (i) by parsing security-relevant information from output logs of libraries, and, (ii) by manually debugging a small set of discrepancies with output logs that do not contain any meaningful information.

Parsing Output Logs Output logs for a discrepancy-producing certificate can point us to a potential vulnerability if one library in the differential testing framework rejects the certificate explicitly for a security reason (e.g., the error message contains “certificate has expired”). Recall that a discrepancy means at least one other library accepts this certificate. This difference in behavior may reflect a mistake by the rejecting library, or a security bug in the accepting one(s).

We systematically investigate such cases by first making a list of all security-relevant strings from output logs of *all* results, and then analyzing discrepancies with output logs containing any such strings. We categorize the results based on the reason for failure and observe the following:

- 1. Certificate expired or not yet valid** All discrepancy-producing certificates rejected due to certificate lifetime validity periods produce just one type of discrepancy – discrepancy

#12 in Figure 6.4. Upon closer inspection, we identified a single root cause for all of them: MatrixSSL allows for a grace period of 24 hours² when matching certificate’s *notBefore* and *notExpiry* time values with the local clock. This means that MatrixSSL still finds certificates with *June 16, 2022* and *June 14, 2022* dates valid while others do not.

2. Invalid time format We find some discrepancy-producing certificates with peculiar time values which cannot be parsed by every TLS library. More specifically, we find that the invalid dates such as *February 31st* are accepted by GnuTLS, and that the timestamp containing leap seconds (the value *60* in the seconds field) is accepted by MatrixSSL and GnuTLS. On manual analysis, we observe these libraries still reject certificates if their validity is reasonably far behind (or far ahead) of the local clock. Given this, we do not classify these as vulnerabilities.

3. Unsupported critical extension One discrepancy-producing certificate has a “critical” Certificate Policies extension with arbitrary data. We find that OpenSSL, LibreSSL and GnuTLS consider this certificate valid while others do not. We reported this behavior to OpenSSL³ since the protocol spec says “*if this extension is critical, the path validation software MUST be able to interpret this extension (including the optional qualifier), or MUST reject the certificate*”. We find that the behavior was indeed closely linked to a security bug of moderate severity, CVE-2023-0465, published a few months before our report. In more detail, there was an issue with OpenSSL where it would silently skip *all* certificate policy checks upon encountering invalid policies. That said, our synthetic certificates could not directly find this bug because OpenSSL disables policy checking by default (even for critical policies).

4. Mismatch between certificate SKID and IKID We notice that the TLS spec does not require that the Subject Key ID (SKID) and Issuer Key ID (IKID) in a leaf and issuer certificate match, and thus consider related discrepancy-producing certificates to be benign. They merely reflect differences in library choices when handling such cases.

Manual analysis A large number of discrepancies in our results do not contain any meaningful information in their output logs. In order to understand their behavior, we manually debug a small set of these discrepancies and observe the following:

1. Invalid email in certificate subject We find a handful of certificates, accepted only by OpenSSL, that contain badly formatted emails in their subject names (e.g., email with multiple ‘@’ characters) . While emails are typically not a part of a certificate’s subject name, the TLS spec allows for the possibility due to legacy implementations. Interestingly, LibreSSL outputs the error number 1 for these certificates (“unspecified error; should not happen”). We notice that LibreSSL parses email fields from a subject (and points out this issue) so it can eventually perform name constraints checking, if required. As the issuer certificate in our experiments did not specify any name constraints, we are unable to say, at this point, if the OpenSSL behavior can be exploited.

²Reference code can be found at [/crypto/keyformat/x509.h#L54-L59](https://crypto.keyformat/x509.h#L54-L59) and [/crypto/keyformat/x509.c#L5119-L5128](https://crypto.keyformat/x509.c#L5119-L5128).

³Github issue link: <https://github.com/openssl/openssl/issues/21359>

2. V1 certificate with extensions We find V1 certificates only accepted by OpenSSL and LibreSSL that contain TLS extensions (undefined for a V1 certificate). Note that this behavior was also reported by [142] in their analysis of the discrepancies.

3. Parsing error We find a synthetic certificate with an extra byte in the value of its *KeyUsage* extension. The extra byte breaks parsing for TLS implementations other than OpenSSL and LibreSSL. During our manual analysis, we also find a violation of the TLS spec by all implementations. As per the spec, “*when the keyUsage extension appears in a certificate, at least one of the bits MUST be set to 1.*”. We are able to craft a TLS certificate that violates that but is validated fine by all implementations. Interestingly, we find that this property was present in one of the synthetic certificates from our work, but it was not flagged as it does not generate a discrepancy.

6.5 Discussion

False negatives We note that differential testing can only find software bugs when one of the TLS implementations mistakenly accepts or rejects a certificate that another does not. This implies that a software bug affecting *all* TLS implementations will not be captured by the framework. In other words, the technique suffers from false negatives. In practice, increasing the number of reference implementations to use in the differential testing framework can help reduce the number of false negatives (due to the higher probability that one of the implementations will not contain the same bug as others). In this work, we rely on 5 TLS implementations, but our framework can easily be extended to support more.

Comparison with other works While in this work we compare our technique with *Frankencerts*, we must acknowledge a fundamental limitation that accompanies the comparison. Unfortunately in the area of software testing and fuzzing, there is not a universal consensus on the metric to use for benchmarking different techniques. Heuristics such as percent of code covered or number of discrepancies found help but do not enable a completely fair comparison. For instance, while we use the same number of synthetic certificates during comparison, our technique relies on training language models on GPUs with extensive compute. In comparison, *Frankencerts* does not require any such bootstrapping. Further, differential testing in our context is semi-automated, as there is still a need to manually analyze discrepancies to find root causes and previously unknown software bugs. Calculating the time taken for such analysis further complicates a comparison. A technique that helps find less discrepancies that are easy to debug and denote security vulnerabilities may be more useful than another that finds more discrepancies that are of low value.

As such, we do not conclude that synthetic certificate generation using language models is superior to other techniques, but only that our results indicate the potential of language models to generate synthetic certificates that trigger diverse behavior and find novel discrepancies. Ultimately, we believe an ensemble of different certificate generation techniques can provide the most value in terms of improving software security.

Prompt engineering In this work we trained and/or finetuned off-the-shelf language models using custom training datasets and built a pipeline for sampling synthetic certificates

from the trained representations. Yet another potential way to generate synthetic certificates is to interact directly with a language model such as ChatGPT and give a prompt relevant to the task of differential testing (e.g., “Generate a TLS certificate that contains an unexpected combination of extensions”). While ChatGPT at this moment cannot generate a certificate with such a simple prompt, it is an interesting direction to further continue interaction with ChatGPT, guiding it for what a TLS certificate represents, what fuzzing aims to achieve, and then asking to generate novel synthetic certificates. Such prompt engineering methods may represent another useful way to leverage language models for differential testing. Note that a recent work from Meng et al. [143] shows promising results for the use of systematic interaction with a pre-trained language model to guide a protocol fuzzer.

Using language models for other input formats Our work uses language models for learning representations for X.509 TLS certificates. Given the success of our method in learning representations useful for the purposes of software testing, we believe there is potential in using this technique in other domains where software inputs can also be expressed in natural language. Some examples include ASN.1 grammars for structures other than TLS certificates, Javascript-based attack vectors such as invalid Content Security Policies, or verbose REST APIs used in various online services.

6.6 Conclusion

In this work, we evaluated the usefulness of generative language models in learning X.509 TLS certificate representations for use in differential testing. Our results indicate the potential of these models to produce synthetic certificates that trigger diverse behavior and can help find previously unknown bugs. We present a pipeline to train language models and generate synthetic certificates, evaluate ways to increase the diversity in sampled outputs, and assess the corpus performance on different metrics. Our work motivates the use of language models for differential testing in other domains, and sparks interest in the use of artificial intelligence to not only learn input grammars, but to also generate test cases that deviate from the learned grammar in ways that enable software testing for bugs and vulnerabilities.

Chapter 7

Concluding Remarks

The thesis of my work is that *the rich diversity in TLS implementations & deployments introduces opportunities to harm protocol security, and that the harms can be identified (and mitigated) using rigorous measurement techniques*. My work in this dissertation defends this thesis using two distinct approaches. First, by studying different TLS deployments in isolation and developing customized measurement techniques, we reveal security issues that exist only due to the nature of each deployment. Second, by setting multiple TLS implementations to consistently validate a novel corpus of synthetic certificates, we find a wide range of implementation differences that reveal protocol violations with implications on security. Our results demonstrate how even if the TLS spec is assumed to be correct, protocol security will still remain malleable due to practical factors that come into play during protocol development or deployment.

In more detail, in Chapter 3, we evaluate TLS effectiveness in consumer IoT devices. Our results reveal how effectively deploying TLS not only requires the choice of a secure protocol configuration, but also requires a mechanism to update the deployed stack to keep up with latest protocol features and avoid deprecated ones. In addition, we are the first to show how the lack of interfaces to explore file system inside an IoT device opens up a new attack vector due to the difficulty in inspecting device root stores. In Chapter 4, we evaluate issues with TLS support on the web that exist due to the insecure HTTP-by-default behavior in web browsers. Unlike prior works that show a rapid support towards TLS-everywhere, our results show how such an approach will cause usability issues in a small set of websites, hindering the progress to a TLS-by-default web. We also notice issues with deploying TLS consistently on *all* pages of a website, often as a result of server misconfigurations. In Chapter 5, we shed light on certificate pinning mechanisms that are not a part of the TLS spec, but are often found in mobile apps with hardened security measures. We evaluate whether mobile apps consistently pin across different platform versions of an app as well as whether pinning mechanisms can actually harm protocol security or user privacy by making it harder to audit network connections. While our results do not suggest any evidence of reduced protocol security or privacy, they nonetheless indicate yet another customization in a TLS deployment that influences protocol security.

Finally, in Chapter 6, we explore diversity in TLS implementations in tandem. We use gen-

erative language models to generate synthetic TLS certificates that execute diverse behavior during differential testing and reveal differences in implementations with security implications. Our work shows the potential of language models to generate synthetic test cases to help improve software security. Unlike diversity in deployments where we find previously unexplored issues related to protocol security, we show how diversity in implementations can also be leveraged to cross-check them for correctness.

The overarching takeaway from my work in this dissertation is that a rigorous and correct protocol design is *not* sufficient to ensure effective TLS use. In addition, it requires examination of the context in which the protocol gets implemented and deployed, the interplay between an end-user, developer and various protocol features, and, any emergent properties that may influence protocol security in ways that do not appear in the design alone.

Bibliography

- [1] *Https encryption on the web – google transparency report*, <https://transparencyreport.google.com/https/overview?hl=en>, (Accessed on 11/28/2022).
- [2] *Apple will require https connections for ios apps by the end of 2016 — techcrunch*, <https://techcrunch.com/2016/06/14/apple-will-require-https-connections-for-ios-apps-by-the-end-of-2016/>, (Accessed on 11/28/2022).
- [3] *Https will now be the default for all android apps*, <https://www.thesslstore.com/blog/https-will-now-be-the-default-for-all-android-p-apps/>, (Accessed on 11/28/2022).
- [4] 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)*, 2019, pp. 1362–1380. DOI: 10.1109/SP.2019.00013.
- [5] M. Marlinspike, “More tricks for defeating ssl in practice,” *Black Hat USA*, 2009.
- [6] S. Sivakorn, I. Polakis, and A. D. Keromytis, “The cracked cookie jar: Http cookie hijacking and the exposure of private information,” in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 724–742.
- [7] L. Chang, H.-C. Hsiao, W. Jeng, T. H.-J. Kim, and W.-H. Lin, “Security implications of redirection trail in popular websites worldwide,” in *Proceedings of the 26th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2017, pp. 1491–1500.
- [8] *Https://datatracker.ietf.org/meeting/112/materials/agenda-112-maprg-04.html*, <https://datatracker.ietf.org/meeting/112/materials/agenda-112-maprg-04.html>, (Accessed on 11/29/2022).
- [9] *Introducing ssl/tls recommender*, <https://blog.cloudflare.com/ssl-tls-recommender/>, (Accessed on 11/29/2022).
- [10] *Apple, google, microsoft, and mozilla come together to end tls 1.0 — ars technica*, <https://arstechnica.com/gadgets/2018/10/browser-vendors-unite-to-end-support-for-20-year-old-tls-1-0/>, (Accessed on 05/14/2021).
- [11] B. Möller, T. Duong, and K. Kotowicz, “This poodle bites: Exploiting the ssl 3.0 fallback,” *Security Advisory*, 2014.

- [12] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, “The most dangerous code in the world: Validating ssl certificates in non-browser software,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 38–49.
- [13] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, “Why eve and mallory love android: An analysis of android ssl (in)security,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 50–61, ISBN: 9781450316514. DOI: 10.1145/2382196.2382205. [Online]. Available: <https://doi.org/10.1145/2382196.2382205>.
- [14] D. Orikogbo, M. Büchler, and M. Egele, “Crios: Toward large-scale ios application analysis,” in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2016, pp. 33–42.
- [15] N. Vallina-Rodriguez, J. Amann, C. Kreibich, N. Weaver, and V. Paxson, “A tangled mass: The android root certificate stores,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 141–148.
- [16] M. T. Paracha, D. J. Dubois, N. Vallina-Rodriguez, and D. Choffnes, “Iotls: Understanding tls usage in consumer iot devices,” in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021, pp. 165–178.
- [17] Z. Durumeric, Z. Ma, D. Springall, *et al.*, “The security impact of https interception,” in *NDSS*, 2017.
- [18] J. Clark *et al.*, *Revisiting past challenges and evaluating certificate trust model enhancements, may 19-22, 2013*.
- [19] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, “Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations,” in *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 114–129.
- [20] Y. Chen and Z. Su, “Guided differential testing of certificate validation in ssl/tls implementations,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 793–804.
- [21] K. Kleine and D. E. Simos, “Coveringcerts: Combinatorial methods for x. 509 certificate testing,” in *2017 IEEE International conference on software testing, verification and validation (ICST)*, IEEE, 2017, pp. 69–79.
- [22] S. Y. Chau, O. Chowdhury, E. Hoque, *et al.*, “Symcerts: Practical symbolic execution for exposing noncompliance in x. 509 certificate validation implementations,” in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 503–520.
- [23] *Rfc 5280 - internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile*, <https://datatracker.ietf.org/doc/html/rfc5280>, (Accessed on 12/04/2023).

- [24] J. Zhu, C. Wan, P. Nie, Y. Chen, and Z. Su, “Guided, deep testing of x. 509 certificate validation via coverage transfer graphs,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 243–254.
- [25] J. Wei, X. Ren, X. Li, *et al.*, “Nezha: Neural contextualized representation for chinese language understanding,” *arXiv preprint arXiv:1909.00204*, 2019.
- [26] C. Chen, W. Diao, Y. Zeng, S. Guo, and C. Hu, “Drlgencert: Deep learning-based automated testing of certificate verification in ssl/tls implementations,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 48–58.
- [27] A. Barengi, N. Mainardi, and G. Pelosi, “Systematic parsing of x. 509: Eradicating security issues with a parse tree,” *Journal of Computer Security*, vol. 26, no. 6, pp. 817–849, 2018.
- [28] J. Kasten, E. Wustrow, and J. A. Halderman, “Cage: Taming certificate authorities by inferring restricted scopes,” in *International Conference on Financial Cryptography and Data Security*, Springer, 2013, pp. 329–337.
- [29] R. Holz, J. Hiller, J. Amann, *et al.*, “Tracking the deployment of tls 1.3 on the web: A story of experimentation and centralization,” *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 3, pp. 3–15, 2020.
- [30] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero, “Coming of age: A longitudinal study of tls deployment,” in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC ’18, Boston, MA, USA: Association for Computing Machinery, 2018, pp. 415–428, ISBN: 9781450356190. DOI: 10.1145/3278532.3278568. [Online]. Available: <https://doi.org/10.1145/3278532.3278568>.
- [31] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz, “Mission accomplished? https security after diginotar,” in *Proceedings of the 2017 Internet Measurement Conference*, 2017, pp. 325–340.
- [32] Z. Durumeric, F. Li, J. Kasten, *et al.*, “The matter of heartbleed,” in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC ’14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488, ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. [Online]. Available: <https://doi.org/10.1145/2663716.2663755>.
- [33] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, “Analysis of the https certificate ecosystem,” in *Proceedings of the 2013 conference on Internet measurement conference*, 2013, pp. 291–304.
- [34] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, “Measuring https adoption on the web,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1323–1338.

- [35] 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*, 2017, pp. 350–362.
- [36] T. Fadai, S. Schrittwieser, P. Kieseberg, and M. Mulazzani, “Trust me, i’m a root ca! analyzing ssl root cas in modern browsers and operating systems,” in *2015 10th International Conference on Availability, Reliability and Security*, IEEE, 2015, pp. 174–179.
- [37] H. Perl, S. Fahl, and M. Smith, “You won’t be needing these any more: On removing unused certificates from trust stores,” in *International Conference on Financial Cryptography and Data Security*, Springer, 2014, pp. 307–315.
- [38] J. Braun and G. Rynkowski, “The potential of an individualized set of trusted cas: Defending against ca failures in the web pki,” in *2013 International Conference on Social Computing*, 2013, pp. 600–605. DOI: 10.1109/SocialCom.2013.90.
- [39] G. Developer, *Network security configuration*, <https://developer.android.com/training/articles/security-config>, (Accessed on 09/08/2021).
- [40] A. Possemato and Y. Fratantonio, “Towards HTTPS everywhere on android: We are not there yet,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 343–360.
- [41] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes, and S. Fahl, “Why eve and mallory still love android: Revisiting tls (in) security in android applications,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [42] *Tls fingerprinting in the real world - cisco blogs*, <https://blogs.cisco.com/security/tls-fingerprinting-in-the-real-world>, (Accessed on 11/25/2020).
- [43] S. Frolov and E. Wustrow, “The use of tls in censorship circumvention,” in *NDSS*, 2019.
- [44] *Number of iot devices 2015-2025 — statista*, <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>, (Accessed on 12/02/2020).
- [45] *This poodle bites: Exploiting the ssl 3.0 fallback*, <https://www.openssl.org/~bodo/ssl-poodle.pdf>, (Accessed on 05/03/2021).
- [46] S. J. Saidi, A. M. Mandalari, R. Kolcun, *et al.*, “A haystack full of needles: Scalable detection of iot devices in the wild,” in *Proceedings of the ACM Internet Measurement Conference*, 2020, pp. 87–100.
- [47] *What you need to know about the solarwinds supply-chain attack — sans institute*, <https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/>, (Accessed on 04/04/2021).
- [48] *Rfc2818*, <https://datatracker.ietf.org/doc/html/rfc2818>, (Accessed on 05/22/2021).
- [49] *Rfc5280*, <https://datatracker.ietf.org/doc/html/rfc5280>, (Accessed on 05/22/2021).

- [50] *Mitmproxy - an interactive https proxy*, <https://mitmproxy.org/>, (Accessed on 05/26/2021).
- [51] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi, “Information Exposure for Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach,” in *Proc. of the Internet Measurement Conference (IMC)*, 2019.
- [52] *Refs - platform/system/ca-certificates - git at google*, <https://android.googlesource.com/platform/system/ca-certificates/+refs>, (Accessed on 05/19/2021).
- [53] *Platform/libcore - git at google*, <https://android.googlesource.com/platform/libcore/>, (Accessed on 05/19/2021).
- [54] *Mozilla-central: Certdata.txt*, <https://hg.mozilla.org/mozilla-central/file/tip/security/nss/lib/ckfw/builtins/certdata.txt>, (Accessed on 05/19/2021).
- [55] *Microsoft trusted root certificate program: Participants - technet articles - united states (english) - technet wiki*, <https://social.technet.microsoft.com/wiki/contents/articles/31634-microsoft-trusted-root-certificate-program-participants.aspx>, (Accessed on 05/19/2021).
- [56] *Mitmproxy/tls_passthrough.py at main · mitmproxy/mitmproxy*, https://github.com/mitmproxy/mitmproxy/blob/main/examples/contrib/tls_passthrough.py, (Accessed on 05/26/2021).
- [57] Y. Liu, W. Tome, L. Zhang, *et al.*, “An end-to-end measurement of certificate revocation in the web’s pki,” in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC ’15, Tokyo, Japan: Association for Computing Machinery, 2015, pp. 183–196, ISBN: 9781450338486. DOI: 10.1145/2815675.2815685. [Online]. Available: <https://doi.org/10.1145/2815675.2815685>.
- [58] *1493822 - removal of "visa ecommerce root" ca from mozilla root program*, https://bugzilla.mozilla.org/show_bug.cgi?id=1493822, (Accessed on 05/16/2021).
- [59] *1552374 - remove certinomis - root ca*, https://bugzilla.mozilla.org/show_bug.cgi?id=1552374, (Accessed on 05/16/2021).
- [60] *Revoking trust in two turktrust certificates - mozilla security blog*, <https://blog.mozilla.org/security/2013/01/03/revoking-trust-in-two-turktrust-certificates/>, (Accessed on 05/16/2021).
- [61] *Net/data/ssl/blocklist - chromium/src - git at google*, <https://chromium.googlesource.com/chromium/src/+refs/heads/main/net/data/ssl/blocklist/>, (Accessed on 05/26/2021).
- [62] *Google online security blog: Distrusting wosign and startcom certificates*, <https://security.googleblog.com/2016/10/distrusting-wosign-and-startcom.html>, (Accessed on 05/26/2021).
- [63] *Fire os overview — amazon fire tv*, <https://developer.amazon.com/docs/fire-tv/fire-os-overview.html>, (Accessed on 11/21/2020).

- [64] M. O’Neill, S. Heidbrink, J. Whitehead, *et al.*, “The secure socket api: Tls as an operating system service,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 799–816.
- [65] *Ioxt - the global standard for iot security*, <https://www.ioxtalliance.org/>, (Accessed on 05/26/2021).
- [66] *Cab forum — certification authorities, web browsers, and interested parties working to secure the web*, <https://cabforum.org/>, (Accessed on 09/27/2021).
- [67] C. Hesselman, J. Jansen, M. Davids, and R. d. O. Schmidt, “Spin: A user-centric security extension for in-home networks,” SIDN Labs Technical report SIDN-TR-2017-002, Tech. Rep., 2017.
- [68] A. M. Mandalari, D. J. Dubois, R. Kolcun, M. T. Paracha, H. Haddadi, and D. Choffnes, “Blocking without Breaking: Identification and Mitigation of Non-Essential IoT Traffic,” in *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*, 2021.
- [69] 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*, vol. 2020, no. 2, 2020.
- [70] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, “Measuring HTTPS Adoption on the Web,” in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, 2017, pp. 1323–1338.
- [71] Electronic Frontier Foundation, *HTTPS Everywhere*, <https://www.eff.org/https-everywhere>, [Last accessed: June 17, 2019], 2019.
- [72] Z. Durumeric, E. Wustrow, and J. A. Halderman, “ZMap: Fast Internet-wide Scanning and Its Security Applications,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C.: USENIX, 2013, pp. 605–620.
- [73] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz, “Mission Accomplished?: HTTPS Security After Diginotar,” in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC ’17, New York, NY, USA: ACM, 2017, pp. 325–340.
- [74] Google, *HTTPS encryption on the web – Google Transparency Report*, <https://transparencyreport.google.com/https/overview>, [Last accessed: June 16, 2019], 2019.
- [75] Electronic Frontier Foundation, *The EFF SSL Observatory*, <https://www.eff.org/observatory>, [Last accessed: June 16, 2019], 2010.
- [76] Alexa Support, *Does Alexa have a list of its top-ranked websites?* <https://support.alexa.com/hc/en-us/articles/200449834-Does-Alexa-have-a-list-of-its-top-ranked-websites->, [Last accessed: June 22, 2019], 2019.
- [77] Q. Scheitle, O. Hohlfeld, J. Gamba, *et al.*, “A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists,” in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC ’18, New York, NY, USA: ACM, 2018, pp. 478–493.

- [78] Alexa Support, *How are Alexa's traffic rankings determined?* <https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined->, [Last accessed: June 22, 2019], 2019.
- [79] K. Reiz, *Requests III: HTTP for Humans and Machines, alike*. <https://3.python-requests.org>, [Last accessed: June 17, 2019], 2018.
- [80] L. Richardson, *PyPI: beautifulsoup4*, <https://pypi.org/project/beautifulsoup4/>, [Last accessed: June 17, 2019], 2019.
- [81] J. Graham, *PyPI: html5lib*, <https://pypi.org/project/html5lib/>, [Last accessed: June 17, 2019], 2017.
- [82] M. Henzinger, "Finding near-duplicate web pages: A large-scale evaluation of algorithms," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2006, pp. 284–291.
- [83] Moz Developer Blog, *Near-duplicate Detection at Moz*, <https://moz.com/devblog/near-duplicate-detection/>, [Last accessed: June 17, 2019], 2015.
- [84] F. Cangialosi, T. Chung, D. Choffnes, *et al.*, "Measurement and analysis of private key sharing in the https ecosystem," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 628–640.
- [85] Cloudflare, *Investor Presentation Q4 2019*, https://cloudflare.net/files/doc_downloads/Presentations/2020/NET-Q4-2019-Investor-Presentation_FINAL.pdf, [Last accessed: May 10, 2020], 2020.
- [86] Mozilla, *Mozilla Observatory*, <https://observatory.mozilla.org/>, [Last accessed: June 26, 2019], 2017.
- [87] HTTPSWatch, *About*, <https://httpswatch.com/about>, [Last accessed: June 26, 2019], 2017.
- [88] Censys, *Censys*, <https://censys.io/>, [Last accessed: June 26, 2019], 2017.
- [89] Google, *HTTPS FAQs*, <https://support.google.com/transparencyreport/answer/7381231/>, [Last accessed: June 26, 2019], 2017.
- [90] Statista, *Number of mobile app downloads worldwide from 2016 to 2021*, <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/>, (Accessed on 05/18/2022), 2022.
- [91] *Rogue web certificate could have been used to attack iran dissidents — google — the guardian*, <https://www.theguardian.com/technology/2011/aug/30/faked-web-certificate-iran-dissidents>, (Accessed on 05/17/2022).
- [92] *Dell does a lenovo: Ships laptops with rogue root ca - ghacks tech news*, <https://www.ghacks.net/2015/11/23/dell-does-a-lenovo-ships-laptops-with-rogue-root-ca/>, (Accessed on 05/17/2022).
- [93] *Imperialviolet - public key pinning*, <https://www.imperialviolet.org/2011/05/04/pinning.html>, (Accessed on 01/17/2021).
- [94] Can I Use, *HTTP Public Key Pinning*, <https://caniuse.com/?search=hpkp>, (Accessed on 05/18/2022), 2022.

- [95] Google Developer, *Jelly Bean*, <https://developer.android.com/about/versions/jelly-bean.html#android-4.2>, (Accessed on 05/18/2022), 2012.
- [96] Android Developer, *Security with HTTPS and SSL (version updated 2021-01-26)*, <https://web.archive.org/web/20210301223141/https://developer.android.com/training/articles/security-ssl>, (Accessed on 05/18/2022), Jan. 2021.
- [97] Google Developer, *Security with HTTPS and SSL*, <https://developer.android.com/training/articles/security-ssl>, (Accessed on 05/18/2022), Oct. 2021.
- [98] Apple Developer, *Identity Pinning: How to configure server certificates for your app*, <https://developer.apple.com/news/?id=g9ejcf8y>, (Accessed on 05/18/2022), Jan. 2021.
- [99] 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*, ser. CoNEXT '17, Incheon, Republic of Korea: Association for Computing Machinery, 2017, pp. 350–362, ISBN: 9781450354226. DOI: 10.1145/3143361.3143400. [Online]. Available: <https://doi.org/10.1145/3143361.3143400>.
- [100] Z. Ma, J. Austgen, J. Mason, Z. Durumeric, and M. Bailey, "Tracing your roots: Exploring the tls trust anchor ecosystem," in *Proc. of the ACM Internet Measurement Conference (IMC)*, 2021.
- [101] *Matlink/gplaycli: Google play downloader via command line*, <https://github.com/matlink/gplaycli>, (Accessed on 05/18/2022).
- [102] *Alternativeto*, <https://alternativeto.net/>, (Accessed on 05/17/2022).
- [103] *Google-play-scraper*, <https://pypi.org/project/google-play-scraper/>, (Accessed on 05/18/2022).
- [104] Apple, *App store downloads on itunes*, <https://apps.apple.com/us/genre/ios/id36>, (Accessed on 05/11/2022).
- [105] Apple Developer Documentation, *Nspinneddomains*, https://developer.apple.com/documentation/bundleresources/information_property_list/nsapptransportsecuritynspinneddomains, (Accessed on 05/19/2022), 2022.
- [106] *Rfc 7469 - public key pinning extension for http*, <https://tools.ietf.org/html/rfc7469>, (Accessed on 01/17/2021).
- [107] *Rfc 6698 - the dns-based authentication of named entities (dane) transport layer security (tls) protocol: Tlsa*, <https://tools.ietf.org/html/rfc6698>, (Accessed on 01/17/2021).
- [108] *Certificatepinner (okhttp 3.14.0 api)*, <https://square.github.io/okhttp/3.x/okhttp/okhttp3/CertificatePinner.html>, (Accessed on 01/17/2021).
- [109] *Apktool*, <https://ibotpeaches.github.io/Apktool/>.
- [110] *Flexdecrypt*, <https://github.com/JohnCoates/flexdecrypt>.
- [111] *Frida-ios-dump*, <https://github.com/AloneMonkey/frida-ios-dump>.

- [112] *ripgrep (rg)*, <https://github.com/BurntSushi/ripgrep>.
- [113] *Radare2*, <https://rada.re/>.
- [114] *Crt.sh — certificate search*, <https://crt.sh/>, (Accessed on 05/17/2022).
- [115] *mitmproxy*, <https://mitmproxy.org/>.
- [116] *Checkra1n*, <https://checkra.in>.
- [117] Android Developer, *Write automated tests with UI Automator*, <https://web.archive.org/web/20220907074832/https://developer.android.com/training/testing/other-components/ui-automator>, (Accessed on 09/07/2022), Mar. 2022.
- [118] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, “ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic,” in *Proc. of the International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2016.
- [119] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, “50 ways to leak your data: An exploration of apps’ circumvention of the android permissions system,” in *Proc. of the USENIX Security Symposium*, 2019.
- [120] *Frida*, <https://frida.re/>, (Accessed on 11/15/2021).
- [121] K. Kollnig, A. Shuba, R. Binns, M. V. Kleek, and N. Shadbolt, “Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps,” in *Proc. of the Privacy Enhancing Technologies Symposium (PETS)*, 2022.
- [122] J. Ren, M. Lindorfer, D. 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 *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.
- [123] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, *et al.*, “Apps, Trackers, Privacy, and Regulators A Global Study of the Mobile Tracking Ecosystem,” in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.
- [124] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl, “To pin or not to {pin—helping} app developers bullet proof their {tls} connections,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 239–254.
- [125] C. M. Stone, T. Chothia, and F. D. Garcia, “Spinner: Semi-automatic detection of pinning without hostname verification,” in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2017.
- [126] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, “An analysis of pre-installed android software,” in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [127] *Curl ca extract: Extract ca certs from mozilla*, <https://curl.se/docs/caextract.html>, (Accessed on 05/19/2022).
- [128] *Rfc 8446 - the transport layer security (tls) protocol version 1.3*, <https://datatracker.ietf.org/doc/html/rfc8446>, (Accessed on 12/04/2023).

- [129] *Rfc 5246 - the transport layer security (tls) protocol version 1.2*, <https://datatracker.ietf.org/doc/html/rfc5246>, (Accessed on 12/04/2023).
- [130] *Imperialviolet - apple's ssl/tls bug*, <https://www.imperialviolet.org/2014/02/22/applebug.html>, (Accessed on 12/04/2023).
- [131] *Openssl 'heartbleed' vulnerability (cve-2014-0160) — cisa*, <https://www.cisa.gov/news-events/alerts/2014/04/08/openssl-heartbleed-vulnerability-cve-2014-0160>, (Accessed on 12/04/2023).
- [132] P. Godefroid, H. Peleg, and R. Singh, “Learn&fuzz: Machine learning for input fuzzing,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2017, pp. 50–59.
- [133] M. Sablotny, B. S. Jensen, and C. W. Johnson, “Recurrent neural networks for fuzz testing web browsers,” in *Information Security and Cryptology—ICISC 2018: 21st International Conference, Seoul, South Korea, November 28–30, 2018, Revised Selected Papers 21*, Springer, 2019, pp. 354–370.
- [134] *Chatgpt*, <https://chat.openai.com/auth/login>, (Accessed on 12/04/2023).
- [135] *Chatgpt sets record for fastest-growing user base - analyst note — reuters*, <https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/>, (Accessed on 12/04/2023).
- [136] *The 2010s: Our decade of deep learning / outlook on the 2020s*, <https://people.idsia.ch/~juergen/2010s-our-decade-of-deep-learning.html>, (Accessed on 12/04/2023).
- [137] *Rapid7 - practitioner-first cybersecurity solutions*, <https://www.rapid7.com/>, (Accessed on 12/04/2023).
- [138] *Github - p1sec/pycrate: A python library to ease the development of encoders and decoders for various protocols and file formats; contains asn.1 and csn.1 compilers*, <https://github.com/P1sec/pycrate>, (Accessed on 12/04/2023).
- [139] *Github - nikhilbarhate99/char-rnn-pytorch: Minimal implementation of multi-layer recurrent neural networks (lstm) for character-level language modelling in pytorch*, <https://github.com/nikhilbarhate99/Char-RNN-PyTorch>, (Accessed on 12/04/2023).
- [140] *Eleutherai/gpt-neo-125m · hugging face*, <https://huggingface.co/EleutherAI/gpt-neo-125m>, (Accessed on 12/04/2023).
- [141] *Github - wolfcw/libfaketime: Libfaketime modifies the system time for a single application*, <https://github.com/wolfcw/libfaketime>, (Accessed on 12/04/2023).
- [142] C. Tian, C. Chen, Z. Duan, and L. Zhao, “Differential testing of certificate validation in ssl/tls implementations: An rfc-guided approach,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–37, 2019.
- [143] R. Meng, M. Mirchev, M. Bohme, and A. Roychoudhury, “Large language model guided protocol fuzzing,” in *NDSS*, 2024.