ABSTRACT

Dissertation Title: MEASURING AND MITIGATING POTENTIAL RISKS OF THIRD-PARTY RESOURCE INCLUSIONS

Soumya Indela
Doctor of Philosophy, 2021
Electrical and Computer Engineering

Advised by: Professor Dave Levin
Computer Science

In today's computer services, developers commonly use third-party resources like libraries, hosting infrastructure and advertisements. Using third-party components improves the efficiency and enhances the quality of developing custom applications. However, while using third-party resources adopts their benefits, it adopts their vulnerabilities, as well. Unfortunately, developers are uninformed about the risks, as a result of which, the services are susceptible to various attacks. There has been a lot of work on how to develop first-hand secure services. The key focus in my thesis is quantifying the risks in the inclusion of third-party resources and looking into possible ways of mitigating them. Based on the fundamental ways that risks arise, we broadly classify them into Direct and Indirect Risks. Direct risk is the risk that comes with invoking the third-party resource incorrectly—even if the third party is otherwise trustworthy whereas indirect risk is the risk that comes with the third-party resource potentially acting in an untrustworthy manner—even if it were invoked correctly.

To understand the security related direct risks in third-party inclusions, we study cryptographic frameworks. Developers often use these frameworks incorrectly and in-

troduce security vulnerabilities. This is because current cryptographic frameworks erode abstraction boundaries, as they do not encapsulate all the framework-specific knowledge and expect developers to understand security attacks and defenses. Starting from the documented misuse cases of cryptographic APIs, we infer five developer needs and we show that a good API design would address these needs only partially. Building on this observation, we propose APIs that are semantically meaningful for developers. We show how these interfaces can be implemented consistently on top of existing frameworks using novel and known design patterns, and we propose build management hooks for isolating security workarounds needed during the development and test phases.

To understand the performance related direct risks in third-party inclusions, we study resource hints in webpage HTML. Today's websites involve loading a large number of resources, resulting in a considerable amount of time issuing DNS requests, requesting resources, and waiting for responses. As an optimization for these time sinks, websites may load resource hints, such as DNS prefetch, preconnect, preload, pre-render, and prefetch tags in their HTML files to cause clients to initiate DNS queries and resource fetches early in their web-page downloads before encountering the precise resource to download. We explore whether websites are making effective use of resource hints using techniques based on the tool we developed to obtain a complete snapshot of a webpage at a given point in time. We find that many popular websites are highly ineffective in their use of resource hints, causing clients to query and connect to extraneous domains, download unnecessary data, and may even use resource hints to bypass ad blockers.

To evaluate the indirect risks, we study the web topology. Users who visit benign, popular websites are unfortunately bombarded with malicious popups, malware- loading

sites, and phishing sites. The questions we want to address here are: Which domains are responsible for such malicious activity? At what point in the process of loading a popular, trusted website does the trust break down to loading dangerous content? To answer these questions, we first understand what third-party resources websites load (both directly and indirectly). I present a tool that constructs the most complete map of a website's resource-level topology to date. This is surprisingly nontrivial; most prior work used only a single run of a single tool (e.g., Puppeteer or Selenium), but I show that this misses a significant fraction of resources. I then apply my tool to collect the resource topology graphs of 20,000 websites from the Alexa ranking, and analyze them to understand which third-party resource inclusions lead to malicious resources. I believe that these third-party inclusions are not always constant or blocked by existing Ad-blockers. We argue that greater accountability of these third parties can lead to a safer web.

ANALYZING AND MITIGATING POTENTIAL RISKS OF THIRD-PARTY RESOURCE INCLUSIONS

by

Soumya Indela

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Advisory Committee:
Professor Dave Levin, Chair/Advisor
Professor Tudor Dumitras
Professor Charalampos Papamanthou
Professor Ashok Agrawala
Professor Lawrence Washington

Acknowledgments

I would like to start by praying to God who continually showers His blessings.

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I would like to thank my advisor, Professor Dave Levin for giving me an invaluable opportunity to work on extremely interesting projects over the past four years. He has always made himself available for advice and help both in academics and personal growth. It has been a pleasure to work with and learn from such an extraordinary individual. He will be an ideal that I aspire to for the rest of my career and I hope I can live up to the high standards that he has instilled in me.

I am very grateful to Professor Tudor Dumitras whose guidance in the initial stages of my PhD helped me develop interest in cyber security research and also inculcate the quality to collaborate and initiate technical discussions. I am immensely grateful that working under the guidance of such a proficient person paved way to my thesis.

I would like to acknowledge my defense committee members, Professor Charalampos Papamanthou, Professor Ashok Agrawala and Professor Lawrence Washinton for their valuable time and feedback. I would also like to thank my undergraduate advisor Professor Sanjay Bose and the professors at University of Maryland, who instilled the thrill of researching in me and trained me with the skill set to pursue research. I

would also like to appreciate the help and technical support from some the University of Maryland staff members.

I owe my deepest thanks to my co-authors, Mukul Kulkarni, Kartik Nayak, colleagues Matthew Lentz, Ivan Petrov, Zhihao Li, Stephen Herwig, Preston Tong, Melissa Hoff, for their aid in improving my skillset and research expertise in cyber security. All our discussions have always kept me motivated, and helped me to keep my graduate student life exciting. My dear friends Sriram Vasudevan, Srikanth Govindarajan, Akshaya Sharma, Nikhil Valluru, Vidya Raju, Arun Shankar, Dwith CYN, Amit Kumar have enriched my graduate life in many ways and deserve a special mention. Our interactions always made me think of the practical considerations in my research and expanded my world-view.

I feel greatly obliged to my parents (Raghuramulu, Jyothi), in-laws (Vidyasagar, Durga), siblings (Sravya, Paavan, Lalitsagar), grandparents and extended family members for their endless support and encouragement in all my endeavours. They have played a significant role in shaping me into what I am today. The journey would not have been possible without my family and their belief in me. It is only because of their teachings and guidance that I could achieve what I have.

Words cannot express my emotions and it is almost impossible for me to verbalize my friendship with Raghuvaran Yaramasu, Pranali Shetty, Vidya Vemparala, Harika Matta, Sharadha Kalyanam, Naresh Maruthi, Rengarajan Sankaranarayanan, Ratan Vishwanath, Saif Mohammed, Kamala Raghavan, Shalvi Raj, Aakanksha Mishra, Sai Sumana, Vandana Banapuram, Sravya Amudapuram, Alok Kumar and Ranjitha Devikere. Without their company, my journey would not have been so much gratifying and I would have

found it extremely difficult to wade through all the highs and lows.

Last but not the least I am extremely indebted to my husband, Amarsagar Reddy Ramapuram Matavalam, who has always stood by me and guided me through the final phases of my PhD. He gave me the strength to persevere and courage to pull through impossible odds at times. I would also like to express my gratitude for the substantial support during the Covid pandemic.

Finally, I have to particularly recognize the financial support received from National Science Foundation, Department of Defense and Maryland Procurement Office.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AES	Advanced Encryption Standard
API	Application Program Interface
CA	Certificate Authority
CDF	Cumulative Distribution Function
CDN	Content Delivery Network
CRL	Certificate Revocation List
CWE	Common Weakness Enumeration
DES	Data Encryption Standard
DNS	Domain Name System
DOM	Document Object Model
DoS	Denial-of-Service
E2LD	Effective Second Level Domain
EC2	Elastic Compute Cloud
ECB	Electronic CodeBook
FQDN	Fully Qualified Domain Name
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
JSSE	Java Secure Socket Extension
MD5	Message Digest 5
MITM	Man-in-the-Middle
NaCl	Networking and Cryptography library
NIST	National Institute of Standards and Technology
OCSP	Online Certificate Status Protocol
OpenCCE	Open CROSSING Crypto Expert
OS	Operating System
RC4	Rivest Cipher 4
SDK	Software Development Kit
SHA-1	Secure Hash Algorithm 1
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URL	Uniform Resource Locator
VT	Virus Total

Chapter 1: Introduction

Computer services such as software and web development are large and sophisticated in design and usage. These services involve many complex components, including hosting infrastructure (CDNs, cloud) and content (libraries, advertising, tracking scripts, web fonts). Libraries are commonly used in advertising, analytics, cloud and social media (Facebook) [12]. In most cases, developers use third-party resources. For example, 66% PyPI packages are used by developers [23] and Content Delivery Networks like Akamai [85] deliver more than 20% of web traffic.

Component-oriented development improves the efficiency and the quality of developing custom applications, thus enhancing the use of third-party resources. Additionally, re-creating these resources incurs a waste of time and redundant memory allocation. Sometimes, it is safer to use trusted third-party code—especially in cryptography, where it is suggested 'not' to write your own code as it is challenging to implement an algorithm that is secure [93, 96]. Overall, the benefits of using third-party content include improved operational speed, delivery time, and scalability.

Incorporating third-party resources involves a high level of trust as it imports not only the features, but also the vulnerabilities associated with it. Thus, if the third-party resource gets compromised, the application is also compromised. Besides vulnerabilities

in the resources, there is a possibility of incorporating the resources incorrectly in the application. Thus, the interface via which the third-party resource is incorporated may also be at risk. Additionally, in the case of websites, most of the resources included are scripts, which have complete access to the Document Object Model (DOM), thus providing access to personal data like credentials and session cookies, in turn compromising the website. Downloading malware without users' knowledge is another common risk associated with various applications.

Unfortunately, developers are oblivious of the vulnerabilities when using third-party resources, and thus seem to be taking on more risk than they might realize. The developers need to be provided with good documentation to ensure security, but this is likely insufficient [5]. Sometimes, the developer has no control; for instance in online advertising, the website developer is unaware of the actual advertisements that the ad networks on the website embed. Developers do not understand the various security threats and make critical mistakes like disabling security checks during the testing and development phase and these checks remain disabled in production [43,93]. Security critical errors can creep into applications by large organizations with well funded security teams as well. For example, 88% of Android applications using cryptographic APIs make at least one mistake [40].

As a result of the mistakes, there are severe real-world consequences. Recently, the crypt32.dll windows module [86] and ads in Microsoft's ad-supported apps [112] have led to installing malware on the users' devices. Another example is the Heartbleed [39] bug in the OpenSSL library, which had long-term consequences. Due to the mistake of improper input validation, the input value for the request messages missed a check on the bounds

causing a buffer overflow. This compromised upto 64KB of sensitive data, including keys for certificates, resulting in the revocation of over 73% of vulnerable certificates [124].

The entire ecosystem could be improved by having tools that better inform developers about the risks they are taking on. In my thesis, I aim to study the hypothesis that "The potential risks in the use of third-party inclusions can be measured and mitigated."

To this end, I observe that risks arise either (a) at the interface via which the third-party resource is included or (b) via inclusions of yet another third-party resource because the developer no longer has control. Thus, based on the fundamental ways that risks arise, we broadly classify them into Direct and Indirect Risks. Direct risk originates from invoking the third-party resource incorrectly—even if the third party is otherwise trustworthy whereas indirect risk originates from the third-party resource potentially acting in an untrustworthy manner—even if it were invoked correctly.

An example of direct and indirect risk arises in the context of online advertising. Including a DNS prefetch link is an example of a direct risk that may occur due to incorrect invocation. An ad network may resell the space to other ad networks, which in turn allow third-party advertisers to load Javascript into that iframe. In this case, malicious behaviour by either the ad networks or the advertisers can lead to indirect risks. Other examples of direct risks include critical mistakes in cryptography [40, 102, 118]. Pop-up ads [81], [77] and malicious website redirections are well known examples of indirect risks.

Direct and indirect risks by our definition are orthogonal—it is perfectly possible that a third-party resource may be both malicious and invoked incorrectly. Since the risks with inclusion of third-party resources are inherently different, I evaluate my thesis by

exploring them separately.

My goal in studying direct risks is to understand the lapses in developing a secure interface and ultimately defining some semantic rules to design a secure API. To address this, I consider cryptographic APIs and identify some of the common mistakes including exchanging keys without authenticating the endpoint [93], storing sensitive information in cleartext [93] or with weak protection [40], using parameters known to be insecure (e.g. the Electronic Codebook mode or non-random initialization vectors) for block ciphers [40], using encryption keys that are constant [40] or are generated from insufficient randomness [40,93], or performing improper TLS certificate validation [43,47,93]. More specific examples of library misuses include using defaults which are insecure (`ssl.VERIFY_CRL_CHECK_CHAIN` should be used instead of `ssl.VERIFY_default` to check for certificate revocation) and using incorrect parameters due to differences across libraries (for hostname verification and certificate validation `ssl.CERT_REQUIRED` requires integer 2 in cURL whereas it is boolean value TRUE, which corresponds to integer 1 in JSSE) [60].

Besides security related issues of direct risks as in the case of cryptographic APIs, another concern is performance. We study this in the context of resource hints that can be included in the HTML of a webpage to instruct browsers to speculatively perform some part of fetching [50] (e.g., to issue a DNS request for a domain that is likely to be loaded later in the HTML page). HTML resource hints are easy to include, easy to reason about, and, *if used correctly*, have the potential to significantly decrease webpage load times. Our analysis of resource hints especially on websites including Cloudflare show that majority of the websites do not make effective use of resource hints. For our comprehensive

study of resource hints on Alexa top-100,000 websites, we use the techniques based on the sound methodology that we developed for downloading resource-level topology of a website.

My goal in studying indirect risks is to understand the amount of trust developers have in third-party resources and inform the users and/or developers where there are risks related to trust breakdown. Towards this, I perform a measurement study to expose the trust relationships involved in incorporating third-party resources at various levels. For this measurement study, I consider the web topology and study the multiple redirections in the Web. For example, a webpage includes an ad network, which in turn loads various ads. Specifically, yahoo.com (Alexa rank 11), which loads the malicious site https://dt.adsafeprotected.com both directly and through https://s.yimg.com. In the case of mangapanda.com, a click at any point on the webpage, would load an ad in a new tab. This occurs due to a redirect to the malicious cobalten.com through the script srv.aftv-serving.bid and go.pub2srv.com. From these examples, it is clear that an important first step in the measurement study is to learn what the third-party resources are. To obtain a good snapshot of the topology of a webpage, I developed a sound methodology to construct the most comprehensive map of a website's resource-level topology. Once the web topology is obtained, I analyze the data as a large graph with trust relationships using different metrics and provide information to users and developers on the third-party resources that can be delegated to websites.

My dissertation ultimately seeks to understand some of the potential risks involved with using third-party resources, and to develop techniques to empirically measure and reason about these risks. That said, it is important to note that this is not a comprehen-

sive study of all forms of risk across all possible applications. The two broad domains I consider in this dissertation—crypto APIs and the web—are, I believe, extremely important and highly representative, and the kinds of risks that I consider—weaker security, bad performance, or inclusion of malicious content—are also common in many settings. My hope is that my findings and methodologies can, to some extent, generalize to other domains, as well; I speak briefly to this in the conclusion of this dissertation.

Chapter 2: Related Work

In this chapter, I will describe related work as it pertains to measuring and mitigating misuse of cryptographic APIs in the context of direct risks. In the context of indirect risks, I will describe related work on downloading websites and measuring third-party resource inclusions in the web.

2.1 Measuring and mitigating misuse of cryptographic APIs

The related work includes empirical observations of cryptography misuse and attempts to prevent such misuse by formulating security design patterns and simplifying cryptographic APIs.

2.1.1 Misuse of Cryptography.

Egele et al. [40] performed an empirical study of cryptographic misuse in Android applications. Reaves et al. [93] studied 46 Android applications that perform financial transactions and reported instances of incorrect certificate validation, storing login credentials in clear text and using poor authentication practices such as do-it-yourself cryptography. Georgiev et al. [47] found that many widely used applications (Amazon's EC2 Java library, PayPal's merchant SDK), shopping carts (osCOmmerce, Ubercart) and mo-

bile applications (Chase mobile banking) all perform broken certificate validation. Fahl et al. [43] showed that root causes in SSL development are not simply careless developers, but also limitations and issues of the current SSL development paradigm. Acar et al. [5] perform a user study to show that participants who use Stack Overflow produced significantly less secure code than those using official documentation or published books. We add to this body of work by identifying systematic behavior differences among popular cryptographic frameworks when implementing the same functionality. However, our main focus is in presenting a solution to these problems.

2.1.2 Design patterns for application security and privacy.

Prior work has introduced several design patterns for application security and privacy [33, 53, 95, 98, 105, 107]. The early work of Yoder et al. [120] introduced several architectural patterns for enabling application security and presented a framework using the patterns to build secure applications. Sommerlad [105] introduced reverse proxy patterns to protect servers at the application layer at the network perimeter. Schumacher [99] proposed patterns for protection against cookies and pseudonymous email in the seminal paper on privacy patterns, and Hafiz [53] extended this work by suggesting patterns for the design of anonymity systems. An authentication enforcer pattern [108] is used to ensure that authentication happens at all relevant parts of the code. Kern et al. [64] designed a database query API that avoids SQL injection vulnerabilities. The effectiveness of these patterns has been questioned in recent studies. Heyman et al. [56] identify 220 security patterns, introduced over a ten year period, and Hafiz et al. [54] report duplicates

among these patterns, as different authors describe similar concepts but give them different names. Yskout et al. [121] quantify the benefits of security patterns to developer productivity and conclude that design patterns do not reduce development time. However, they do not assess whether the use of these patterns leads to more secure code. Rather than identifying design patterns that capture common implementation techniques, we characterize and bridge the semantic gap between the needs of developers and the cryptographic APIs. Starting from common misuse patterns of existing APIs and from incorrect recommendations found in widely used programming resources, we design semantic APIs that can be used correctly by developers who lack an in-depth knowledge of cryptography.

2.1.3 Simplified usage of cryptographic APIs.

The NaCl cryptographic library [20] introduced simplified APIs, aiming to avoid some misuse patterns observed with existing cryptographic libraries. Fahl et al. [43] propose modifying the Android OS to provide the main SSL usage patterns as a service that can be added to apps via configuration, to prevent developers from implementing their own SSL code. OpenCCE [11] is a tool for managing software product lines that builds on the observation that many cryptographic solutions represent combinations of common cryptographic algorithms, parameterized at compile time. OpenCCE guides developers through the selection of the appropriate algorithms, and synthesizes both Java code and a usage protocol. This approach requires monitoring the code changes over time, through static analysis, to ensure that the usage protocol is not violated. Additionally, the code synthesized is tied to the library used and does not account for the need to

incorporate external information at runtime. Rather than creating a new library or a new service, we introduce a semantic layer on top of existing libraries, which allows us to provide unified APIs across different libraries, programming languages and platforms. Additionally, we design portable semantic APIs for bridging the gap between developer needs and cryptographic expertise, instead of static analysis tools.

2.1.4 Static analysis and type systems.

Recent work on type systems and static analysis aims to remove the burden of implementing security checks from developers and to generate proofs that an application satisfies certain security properties. For example, the analysis of information flow allows determining if a program satisfies certain confidentiality policies [35, 58]. Van Delft et al. [114] extended this approach to information-flow properties which change during program execution. FlowTracker [94] focuses on discovering time-based side-channels in cryptographic libraries. Bodei et al. [22] focus on finding weaknesses in cryptographic protocols. We focus on misuses of the APIs exposed by popular frameworks, rather than on attacks against cryptographic protocols. Moreover, we observe that, for some security checks, developers need the flexibility to chose the most appropriate implementation. For example, there is currently no agreement about the best method for checking the revocation status of TLS certificates, and the choice is likely to be platform dependent [72]. Type systems and static analyses are complementary to a better API design, and they can improve the performance of well established security protocols by moving the checks to compile-time.

2.2 Downloading Websites

The related work includes the tools used to download websites, obtain the Web topology, and compare the various crawls to identify the appropriate parameters to download a website completely. We hope that our results will lead to more papers exploring these various parameters and reporting on them so that others may evaluate and reproduce more accurately.

2.2.1 Web Crawling Tools

Before the widespread use of dynamic content in webpages, it would suffice to use `curl` or `wget` to download web content. But, neither of these tools include a JavaScript engine, and thus miss a large portion of the web's content. However, even as early as 2012, Nikiforakis et al. [83] showed that more than 93% of the most popular websites include JavaScript from external sources. Many research projects seek to measure as many third-party resource inclusions as possible [9, 15–17, 59, 62, 67, 76, 78], making more sophisticated, headless browsers a necessity. To address this need, many researchers use Puppeteer [90], a Node.js library that gives programmatic control and data collection over a headless Chromium browser. In our research, we consider two tools that take complementary approaches in obtaining the Resource Tree: **ZBrowse** and **Crawlium**. ZBrowse [122] uses Node.js's built-in `getResourceTree` method for obtaining the DOM tree after the webpage has been loaded. It also augments this tree by collecting data from two network event triggers: `requestWillBeSent` and `responseReceived`. Rather than using Node.js's built-in resource tree construction, Crawlium [32] builds its own tree from

the collection of the various network events. Crawlium triggers on the same network events as ZBrowse, plus others to capture data sent and received via web sockets, frame navigation, new execution contexts, the parsing of scripts, and when the console API is called.

2.2.2 Web topology

In our study, we will demonstrate several measurement parameters that can have a significant effect on the proportion of the inclusion graph a tool is able to obtain. As their motivation for creating Crawlium, Arshad et al. [10] observe that merely obtaining the DOM tree can miss critical resource inclusions. ZBrowse uses Node.js's DOM tree, but overcomes this limitation at least in part by augmenting the tree with inclusions learned from network events [67, 122]. Although much prior research related to downloading content on webpages focuses solely on the resources, there is also work by Gibson et al. [48] where the links between the resources are analyzed. Barabasi et al. [13] utilize the incoming and outgoing link distribution of the Web to understand the network topology in the context of self-organization and scaling in random networks, whereas Castillo et al. [26] study both the link-based and content-based features, and use the topology of the Web graph to examine that linked hosts belong to the same class—either both are spam or both are non-spam. We study the web topology to identify the links, and ultimately the domains that lead to trust breakdown and load malicious content.

2.2.3 Comparison across crawls

The papers introducing Crawlium [10] and ZBrowse [67] specify their tools' network events, but do not investigate multiple page loads. Other studies have investigated the variation of page content from one page-load to another. Zeber et al. [123] and Englehardt and Narayanan [42]—as part of broader studies—both used OpenWPM, a Selenium-based web privacy measurement tool, to compare resources obtained between a pair of simultaneous or back-to-back crawls. Their results broadly agree, and indicate that the same third-party URLs are loaded 28% of the time and the typical overlap is about 90%. We study a slightly different question: how many page loads would we need in order to *exhaustively* obtain the inclusion graph? Our study also extends upon these prior efforts by comparing multiple tools and presenting an adaptive page-loading technique for obtaining more complete inclusion graphs.

2.3 Measuring third-party resource inclusion on the web

The related work includes advertising on desktop and mobile devices, the trust in third-party inclusions and the mechanisms by which the resources are incorporated into the wesite.

2.3.1 Online advertsing

Online advertising has become a cross-browser monetization platform introducing significant malicious activity. Prior work has shown that ad abuse leads to losses in the order of millions of dollars for advertisers. Blocking ads as well as forced redirects cost

publishers [104]. Using monetization tactics similar to DNSChanger, several large botnets (i.e., ZeroAccess and TDSS/TDL4) abuse the ad ecosystem at scale. Third party services also make money through ad abuse. Chen et al. [28] and Thomas et al. [111] analyze the depth of financial ad abuse. Thomas et al. [111] find that ad injection introduces malware and small number of software developers support a large number of ad injectors. Li et al. [70] study web traces and obtain ad redirection chains related to advertising networks, and identify that the top ranked Alexa websites and leading advertising networks are injected with malware.

Targeted advertising uses specialised user data and has become more common due to the prevalent use of social media. Plane et al. [87] conduct a pilot study and a multiple-step survey in which users are provided with various advertising scenarios to understand targeted advertising. While Andreou et al. [7] perform a case study on Facebook to identify that ads are displayed for different users based on the user interests as a result of advertising networks obtaining user data that sometimes contain sensitive information, Cabanas et al. [25] study how a large portion of Facebook users in the European Union are linked with potentially sensitive interests which lead to leakage of confidential personal data and that malicious third-party services reveal the identity of Facebook users. Recently, pop-up ads and forced redirections have become more prevalent where malware can be spread without the users knowledge. Identifying the websites (third party services, pop-ups, automatic redirect webpages, etc.) that lead to malware is important for safe web browsing. Many popular websites when accessed lead to fake advertisements [49] and pop-up ads [81], [77] which sometimes bypass ad blockers causing inconvenience [73].

There are many ad-blockers that prevent the loading of malicious domains that are

indirectly loaded by a website. But, some malicious third-parties evade these ad blocking mechanisms as well. The use of adblocking tools like Adblock Plus [1] and ublock origin affects the advertising revenue streams. Nithyanand et al. [84] analyze the arms race of ad blocking and anti-adblocking utilizing third-party services that are shared across multiple web pages. Thus, we need to analyze the detailed web topology to identify the source of malicious activity and propose to analyze the effectiveness of the adblockers. Guha et al. [51] study the challenges in measurement methodologies used for advertising networks and propose new metrics robust to noise present in ad distribution networks. They also identify measurement pitfalls and artifacts, and provide mitigation strategies. The paper studies ad network distribution as a whole, whereas we focus on malicious activity not only due to advertising, but also other ways like javascript obfuscation. Seifert et al. [100] utilize static attributes on an HTML page to detect and classify malicious web pages, whereas Poornachandran et al. [89] perform static analysis followed by a behavioral analysis to identify malicious advertisements. We move a step ahead and use the topology instead of the static HTML page to study malicious activity.

2.3.2 Mobile vs Desktop

While analyzing the web topology, we observed significant differences in the mobile version of the website compared to the desktop version. While Botha et al. [24] study the difference in security, Johnson and Seeling [63] study the webpage object requests in mobile and desktop browsers. Botha et al. [24] explore the availability of security mechanisms in a mobile context similar to the desktop environment and conclude that the same

protection level in the desktop environment cannot be achieved due to usability issues. Johnson and Seeling [63] identify that the number of webpage object requests increases steadily and that the growth is slightly higher in desktop versions of the webpage. Although there are mobile applications corresponding to web applications, an online survey and a user study conducted by Maurer et al. [74] identified that more people prefer using original content on mobile web browsers instead of the mobile application, especially for new generation mobile devices. Thus, we analyze the Web on both desktop and mobile devices and compare the topologies, both in terms of the number of malicious domains and in terms of the number of insecure redirection links to test our hypothesis that although desktop versions have a larger number of malicious domains, the number of links to these domains is less.

2.3.3 Mobile Advertising

With the rapid deployment of new mobile devices, there is a need to understand the security of mobile versions of websites. Mobile advertising has become an easy way to steal information on user's devices using advertising products [27]. Similar to web advertising, ad networks behind in-app advertising employ personalization to improve the effectiveness/profitability of their ad-placement. A lot of in-app advertisements work at the mobile app-web interface where users tap on an advertisement and are led to a web page which may further redirect, sometimes automatically until the user reaches the final destination. Mobile advertising is more prone to malicious activity, one of the reasons being less to no use of Adblockers on mobile devices.

Dong et al. [37] explore various new ad frauds in mobile applications that include both static placement and dynamic interaction fraud whereas Rastogi et al. [92] explore the interface between mobile applications and the web links and identify that destination webpages may result in scams. Due to these vulnerabilities in the applications and the interface, significant user data is leaked from mobile devices. Meng et al. [75] study the amount of sensitive user information that mobile in-app advertising networks learn and that personalized ads can be used to reconstruct the data obtained by the ad networks. Also, there are various ways in which the data can be accessed. While Son et al. [106] show that few applications require access to external storage to cache videos and images whereas Demetriou et al. [34] analyze the data that can be obtained by advertising networks from installed apps, the libraries, other files and user inputs; all of which results in revealing sensitive data. We aim to identify the web applications that are more prone to malicious activity and possibly detect where and how such behaviour arises. We also analyze whether the basic Adblockers currently available on mobile devices can protect the device from malicious domains.

2.3.4 Trust in Third-party Resource Inclusions

Ikram et al. [59] study dependency chains in the Web ecosystem focusing on suspicious or malicious third-party content that is indirectly loaded by first-party websites via dependency chains. While analyzing the malicious activity on the Web, we begin with what the paper does and obtain the dependency chains. We then utilize it to identify the links between first- and third-party domains and ultimately identify not just the domains

that are outright bad (by VirusTotal), but also the domains that indirectly load these domains when a user accesses multiple first-party websites (Alexa top ranked websites). The major reason behind malicious activity in the Web is the trust first-party websites place on third-party resources.

Chen et al. [29] study how network reputation and malicious activity is exploited in ad-bidding between ad exchanges and advertisers. For new or unknown domains, the reputation score is determined based on known legitimate and malicious domains. Antonakakis et al. [8] propose a dynamic reputation system by building a domain model using passive DNS query data and network related data. Sometimes, adversaries exploit domain ownership changes and use legitimate domains, which are no longer in use to introduce malicious activity. Lever et al. study that ownership changes are exploited by adversaries resulting in residual trust abuse reducing the safety of the domain and the users accessing such domains [69]. They additionally measure the extent of residual domain trust abuse and its growth in recent times [68]. Instead of using a known reputation, we first use VirusTotal to identify bad domains and propose to use the web topology to effectively define a better trust metric for the domains by back-tracing from the bad domains.

2.4 Resource Hints

There has been some prior work measuring the percent of websites that implement resource hints [55]. However, to the best of our knowledge, none investigate how optimally websites are in using resource hints. We intend to explore the actual fraction of resource hint links that websites use and whether they are properly enabling DNS

prefetching for HTTPS. In order to enable DNS prefetching, there are two main ways to do so. According to the popular web browsers Firefox and Chrome, websites can contain the following meta tag: ¡meta http-equiv=”x-dns-prefetch-control” content=”on”¿ in order to enable DNS prefetching [36] [119]. An alternative is to enable it within HTTP headers with the following syntax: ”X-DNS-Prefetch-Control: on” [119]. Chrome denotes this requirement as a response to prevent eavesdroppers from ”inferring the host names of hyperlinks” [36].

A wide variety of previous work has established that DNS resolution latency is one of the prime causes of slowdowns in page load times on the internet [52, 103, 109, 117]. Habib and Abrams were able to demonstrate the damage caused by DNS latency in slowing down web browsing, and suggest initial mitigations, such as increasing the size of the DNS cache [52]. In their analysis of how to improve internet speed, Singla et al. assessed the latency bottleneck of DNS resolution as causing a 7.4x inflation over c-latency (that is, “the time it would take for light to travel round-trip along the shortest path between the same end-points”) [103]. Sundaresan et al. empirically showed that DNS caching could reduce maximum page lookup times by between 15 and 50 milliseconds [109]. And Wang et al. used profiling to show that DNS lookup is the cause of almost 13% of page delay time during the loading process [117].

As a result of the significant contribution of DNS latency to slower internet speeds, a number of techniques have been proposed to mitigate this latency. These techniques include ideas varying from querying multiple DNS servers simultaneously to eliminating DNS resolvers. However, of all of these ideas, only one has caught on in practice [97,116]. DNS prefetching has been demonstrated to provide significant latency improvements in

theory and, as a result, has been implemented in practice by almost all major browsers [30, 82]. However, not every researcher is equally pleased with the idea of resource hints as a potential solution. There is a chance that resource hints can introduce new privacy concerns while browsing the web. In particular, as Krishnan and Monrose observed, because DNS prefetching causes all of the DNS requests for a web page to be sent at the same time, these requests are likely to be clustered together in DNS logs. If these requests are sufficiently unique to allow identification of individual pages of websites, this could potentially leak information about the actual *content* users are viewing, rather than just what IPs they are visiting [65, 66].

In addition, resource hints could be used for various vectors of attacks, such as framing attacks, targeted DoS attacks, cross-site forgery attacks, and data-analytic pollution attacks [115]. However, these attacks would require a malicious host for webpages. Although resource hints opens the door to new attacks, a malicious host already has the capability to implement various additional attacks against their users and others. Perhaps as a result of these security concerns, some major browsers have chosen to disable DNS prefetching by default under HTTPS, although it is still enabled by default under HTTP [36, 119]. Major browsers do not act on DNS prefetch hints within HTTPS files unless the page first explicitly turns DNS prefetching on, by including the `X-DNS-Prefetch-Control` HTTP header, or equivalently by including the HTML:

```
<meta http-equiv="x-dns-prefetch-control" content="on">
```

If at any point in the HTML DNS prefetching is turned off (`content="off"`), Chrome does not allow it to be turned back on.

Chapter 3: Toward Semantic Interfaces for Cryptographic Frameworks

This chapter focuses on direct risk—the risk that arises with invoking a third-party resource incorrectly, even if the third party is otherwise trustworthy. We study direct risks within the context of cryptographic frameworks. Several mature cryptographic frameworks are available, and they have been utilized for building complex applications. However, developers often use these frameworks incorrectly and introduce security vulnerabilities. This is because current cryptographic frameworks erode abstraction boundaries, as they do not encapsulate all the framework-specific knowledge and expect developers to understand security attacks and defenses. In our paper [60], we characterize this semantic gap between the needs of developers and the cryptographic APIs, and we present techniques for bridging this gap.

3.1 Overview

Cryptographic algorithms are often a necessary building block for complex applications and libraries, for instance to implement secure client-server communications, to store data securely or to process payments. Several mature cryptographic frameworks are currently available for this task, including Oracle JSSE, IBM JSSE, BouncyCastle, and OpenSSL. These frameworks are implemented by cryptography experts, include state-of-

the-art algorithms, and their code has been audited and analyzed with formal verification tools. They have also used to build real world software that provides strong security.

Unfortunately, software developers who lack cryptography expertise often make critical mistakes when using these frameworks, including exchanging keys without authenticating the endpoint [93], storing sensitive information in cleartext [93] or with weak protection [40], using parameters known to be insecure (e.g. the Electronic Codebook mode or non-random initialization vectors) for block ciphers [40] using encryption keys that are constant [40] or are generated from insufficient randomness [40, 93], or performing improper TLS certificate validation [43, 47, 93]. The Common Weaknesses Enumeration dictionary [31], which provides a comprehensive taxonomy of frequent programming mistakes, includes 14 common implementation errors related to the use of cryptography. These errors allow attackers to impersonate legitimate users [43, 47, 93], to harvest sensitive personal information [43, 47, 93] and even to steal money [93].

The solutions that have been proposed for this problem include simplified cryptographic APIs [3, 20, 43, 110], secure default values for the parameters of cryptographic algorithms [47] and using static analysis tools to discover bugs related to cryptography misuse [11, 40, 64]. These solutions do not address a more fundamental problem: the fact that current cryptographic frameworks *erode abstraction boundaries*, as they do not encapsulate all the framework-specific knowledge and expect developers to understand security attacks and defenses.

In this chapter, we characterize the semantic gap between the needs of developers and the cryptographic APIs, and we present techniques for bridging this gap. For example, simplified cryptographic APIs do not provide a true separation of concerns as they do not

provide the flexibility that developers need to implement the complex business logic required (e.g. authentication and authorization services, e-commerce SDKs and integrated shopping carts [47] or mobile payment systems [93]). This problem could be addressed in part by following best practices in API design [21], but some challenges are specific to the cryptography domain. In particular, the security of applications using cryptography often depends on *information external to the system*. For example, the SHA-1 cryptographic hash function, introduced two decades ago, is no longer considered secure given the performance of modern hardware, yet many web sites still advertise TLS certificates that use SHA-1 for generating digital signatures. The checks for implementation choices that may lead to insecurity must be done are runtime, as in some cases the information changes frequently. For example, when TLS certificates are compromised, they must be revoked and reissued immediately to prevent man in the middle attacks,[1] and client-side code must check for the revocation status of these certificates. Moreover, developers need the flexibility to select the most appropriate mechanism for incorporating this information. For example, an application can check the revocation status of TLS certificates by downloading certificate revocation lists (CRLs), by using the Online Certificate Status Protocol (OCSP) or by implementing OCSP stapling. There is currently *no agreement about what method is best*, and the choice is likely to be platform dependent [72]. In consequence, it is difficult to define simplified APIs or statically verifiable security protocols that cover all the ways cryptography is used in the real world.

Another domain specific challenge is that developers often need to *disable security*

[1] Exploits for the Heartbleed vulnerability, which enabled breaking TLS certificates at scale, were observed in the wild less than 24h after the vulnerability was disclosed [38].

checks in the development environment in order to run and test their application (e.g. by disabling server authentication with self-signed TLS certificates [43]), and sometimes these checks remain disabled in production because the developers do not understand the security threats associated with these workarounds [43, 93]. This suggests that designing good cryptography APIs is not sufficient for addressing the problem, and the solution must extend to the build management system.

To address these challenges, we propose *semantic APIs* for cryptographic libraries, which expose the security decisions without requiring in depth knowledge of attacks and defenses. We describe several *design patterns for implementing these APIs*, including three ways of incorporating external information, and we demonstrate how our APIs can be implemented on top of the existing cryptographic frameworks. Our APIs represent a first step toward striking the right balance between restricting the security decisions that developers make and giving them the flexibility needed for complex applications that use cryptography. In addition to these semantic APIs, we propose *compile-time checks* to separate the development environment from the production environment. This allows for a clean definition of workarounds during development that should not be used in production.

In summary, we make three contributions:

1. We identify new problems with the existing cryptographic APIs, and we classify the root causes of the new and the known programming mistakes related to using these APIs.

2. We present a solution to these problems by introducing semantic APIs for crypto-

graphic operations. We also discuss design patterns for implementing these interfaces on top of existing cryptographic frameworks.

3. We propose build management hooks for isolating the workarounds used during development and testing.

The rest of the chapter is organized as follows. In Section 3.2 we outline our goals and non-goals. In Section 3.3 we review problems with existing cryptographic APIs, not described in the prior work, and we categorize the needs of developers who use these APIs. In Section 3.4 we describe our solutions to these problems. In Section 3.5 we validate our solutions through several case studies. In Section 3.6 we discuss the remaining challenges.

3.2 Problem Statement

```
1 import socket, ssl
2 context = ssl.create_default_context()
3 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4 ssl_sock = context.wrap_socket(s, server_hostname='www.google.com')
5 ssl_sock.connect(('www.google.com', 443))
```

Figure 3.1: HTTPS request in Python.

Consider a developer Alice who wants to develop an application in Python, and she wants to use the HTTPS protocol in her software to communicate securely with a web service called Binary Object Broker (BOB). The HTTPS protocol allows clients to connect to servers they have not encountered before, and with which they have no shared secrets, by utilizing the TLS protocol to exchange keys during the initial handshake. A

common misuse of cryptographic APIs (CWE-322) is to perform a key exchange without first authenticating the server [31, 43, 93], which results in the establishment of a secure channel without first ensuring that the client has connected to the correct server. This programming mistake allows an adversary to intercept the communication through a man-in-the-middle attack [57]. To prevent this attack, an HTTPS server must present a digital certificate, signed by a Certification Authority that the client trusts, which authenticates the server to the client. While Alice is not a cryptography expert, she tries to avoid such common mistakes by looking up the best practices for using Python libraries to establish a secure connection with the web server. This results in the code shown in Figure 3.1.[2] As the figure shows, Alice creates a default context, which authenticates the BOB service by requesting a certificate and verifies that the web server's certificate hostname matches the one from the certificate. Although Alice believes that her implementation is secure, her code never checks if the certificate has been revoked, leaving the application exposed to a man-in-the-middle attack.[3] This is because the default option (`ssl.VERIFY_DEFAULT`) in Python does not check for revoked certificates. Alice must explicitly specify `context.verify_flags = ssl.VERIFY_CRL_CHECK_CHAIN` in Figure 3.1 to ensure this. The fundamental cause of this error is that the library expects developers to have a good understanding of security attacks and defenses and to know details about the library's implementation and configuration.

This is just one example of security vulnerabilities that can be introduced by misusing cryptography; recent studies [40, 43, 47, 93] have reported that such mistakes are

[2] From `https://docs.python.org/2/library/ssl.html#ssl-security`;

[3] Because TLS certificates are sometimes compromised (e.g. in the wake of the Heartbleed vulnerability [124]), client-side TLS code must check the revocation status of certificates presented by servers during the TLS handshake to prevent an adversary from eavesdropping on the connection.

common. The root causes of these mistakes fall into four categories:

1. **No separation of concerns.** Existing cryptographic frameworks are implemented by experts and can be used correctly. However, the APIs they expose do not encapsulate all the cryptographic knowledge and expect users such as Alice to understand security attacks and defenses and the subtle impact of various parameters used in the framework implementation. In consequence, when using these frameworks developers cannot focus only on the application logic but must also learn about cryptography.

2. **Diverse needs of developers.** Multiple types of developers may need to use cryptographic frameworks. For example, *functionality engineers* (such as Alice) often have simple requirements, such as communicating securely over the Internet, while *security engineers* must implement more complex services that rely on cryptography. Prior efforts to simplify cryptographic APIs, to make them more suitable for developers who lack cryptographic expertise [20, 43], do not take into account the diversity of these developers' needs.

3. **Need to incorporate external information.** One of the reasons why existing cryptographic frameworks expose a dizzying array of options and parameters is to allow developers to react to the community's evolving understanding of cryptographic attacks and defenses. Some implementation choices are found to jeopardize security, and the rate at which this information is updated ranges from years (in the case of cryptographic algorithms, e.g. SHA-1) to days (in the case of certificate revocations). Developers must find a way to incorporate such external information into

their systems.

4. **Reliance on secure default values.** The prior efforts to simplify cryptographic APIs remove choices available to developers, which leads to additional mistakes when programmers develop workarounds. For example, while the default settings for the Android SSL library ensure correct certificate validation, developers often need to disable these validation checks during development and testing, by using self-signed certificates [93], and do not understand that using this workaround in production code fails to authenticate the server [43]. In addition to cryptographic APIs that ensure a separation of concerns and that provide secure defaults, developers need appropriate build management tools to define workarounds that should only be used in the development environment.

Goals and non-goals. We present ideas that would help developers to use cryptography correctly, by addressing the four challenges identified above. However, we do not aim to enforce a secure usage of cryptographic APIs. We do not think this is currently feasible as developers could always resort to do-it-yourself crypto to bypass our enforcement points. In other words, we assume that, like Alice in our example, the developers are trying to write secure software and we aim to make it easier for them to achieve this goal. Furthermore, we do not propose new cryptographic techniques or algorithms, and we do not describe any cryptographic weaknesses in the existing algorithms. Instead, we focus on preventing common mistakes in the way these algorithms are used in applications. Finally, we do not consider the mistakes made by developers who implement their own custom ways of encrypting data, authenticating users, etc., thereby bypassing

cryptographic frameworks entirely.

3.3 Needs of Developers

From the context of programming mistakes reported in prior work [40, 43, 47, 93], we identify five general needs of developers who must use cryptographic APIs. By further investigating these needs, we also identify several challenges, not documented before, for using existing APIs correctly.

Need 1: *Establish secure connections.* One of the most common reasons to use cryptographic frameworks is to implement client-server applications that communicate over secure channels, often using the HTTPS protocol. However, the counter-intuitive interfaces and parameters exposed by these frameworks can lead to programming mistakes. For example, JSSE performs hostname verification only if an algorithm field is correctly set to "HTTPS" [47], but developers who are not familiar with the steps from the TLS handshake are likely to establish an insecure connection. Similarly, in Python, hostname verification is skipped if the developer forgets to specify `ssl.CERT_REQUIRED` (Figure 3.1). In the `cURL` library, the parameter for requesting hostname verification and certificate validation is 2 (an integer), while in JSSE it is TRUE (a boolean), which confuses developers who sometimes invoke the `cURL` API with 1 (the integer that corresponds to TRUE) [47].

We also identified cases where *different cryptographic frameworks behave differently when providing the same functionality*. This can lead to mistakes when developers move from one framework to another. For example, if a trusted certificate has expired, in an IBM JSSE implementation, the handshake will fail, even though the expired certifi-

cate is trusted. An Oracle JSSE implementation will flag such a connection as secure and expect the developer to check for expired certificates. Developers who are not security experts need a cryptographic API that abstracts away these details and that minimizes astonishment [21].

Need 2: *Store data securely.* Another frequent requirement is to store sensitive data, e.g. personally identifiable information, keys, credit card information, account balances and other application related information. However, developers sometimes store such information on the device in plaintext, use hard-coded keys or insufficiently random values for encryption, or allow the sensitive information to leak through log files [93]. Developers usually know which information handled by their applications is sensitive, but they make mistakes because they must invoke correct encryption operations each time the data is written to disk, in some form. Instead, a cryptographic API should decouple the task of specifying that certain data structures contain sensitive information and the secure storage primitives. For data marked as sensitive, these primitives should automatically encrypt the data in a way that ensures confidentiality (an unauthorized party cannot decrypt it) and integrity (the data cannot be forged or tampered with).

Need 3: *Incorporate security-critical external information.* Systems that were designed and implemented correctly a decade ago may be vulnerable today because of changes in the security landscape. For example, the DES encryption algorithm or the MD5 hashing algorithm were considered secure in the past, but today they are known to be insecure. Nevertheless, there are still applications that use DES and MD5. Similarly, as comput-

ing power increases, increasingly longer keys are necessary for providing security against brute force attacks. The cryptographic framework should check and enforce these recommendations transparently. This can be achieved by requesting the information, in a machine readable format, from a trusted third party—perhaps the National Institute of Standards and Technology (NIST), which periodically publishes recommendations for the use of cryptographic algorithms and key lengths [14]. Web browsers use a similar pattern for determining the revocation status of TLS certificates, which is another type of external information that is critical for security.

Need 4: *Use default parameters securely.* Developers who lack a background in cryptography will often end up using the default values of parameters required by various algorithms. Sometimes, the default values compromise security; for example, when the AES block cipher is used, the insecure ECB mode is the default in Python's PyCrypto [71], Java JSSE libraries (as well as resulting Android libraries) [40]. Cryptographic frameworks should provide default values that ensure security.

Need 5: *Disable security checks during development and testing.* Because security implies that certain operations will be disallowed, developers often need a way to bypass security checks in the development environment in order to test all the code paths. For example, when a developer starts building an SSL application, the code throws exceptions either due to the absence of a certificate or to the use of a self signed certificate. Many such developers then bypass SSL certificate validation, to be able to continue writing and testing their code. While these workarounds have a legitimate purpose in the develop-

Table 3.1: Mapping developer needs, mistakes and solution - CWE provides a unified, measurable set of software weaknesses [31]. For example from the CWE 297 in the list corresponds to the mistake "Improper Validation of Certificate with Host Mismatch" `https://cwe.mitre.org/data/definitions/297.html`.

Developer Needs	Example Mistake	Solution	CWE	Section
1. Establish secure connection	Allow expired certificates, Skip Hostname Verification	Semantic APIs, Integrating external information	295, 297, 599, 319, 321, 322, 324, 327	3.4.1, 3.4.2
2. Store data securely	Secret keys stored unencrypted	Semantic APIs	311, 312, 532	3.4.1
3. Incorporate security critical information	Using SHA1 digest	Integrating external information	299, 327 370, 676	3.4.2
4. Use default parameters securely	Using AES in ECB mode	Semantic APIs, Setup build configuration	276, 453	3.4.1, 3.4.3
5. Disable security checks during development	Using self-signed certificates in production environment	Setup build configuration	296	3.4.3

ment environment, they are sometimes deployed in a production environment, making the application vulnerable [43]. Cryptographic frameworks should provide a way for developers to specify that certain workarounds should be executed only in the development environment.

3.4 Unified Framework for Secure Application Development

The needs and few mistakes identified in the previous section have been summarized in Table 3.1. To address these needs, we propose a unified framework with the following components:

1. Semantic APIs, which present the high level functionality and security guarantees to the developers without exposing low level implementation specifics. We show that these APIs can be implemented consistently on different platforms, without modifying the underlying cryptographic framework (Section 3.4.1).

2. Design patterns for integrating information from external trusted sources, transpar-

Table 3.2: Proposed Semantic API with functions for both application and library developers [46].

API	Parameters	Semantics	Design Pattern
Functionality Engineers			
`Communicate Interface`			
`send(sock, msg)`	`addr`: address of the sender/receiver `msg`: data to be sent/received `sock`: socket for communication	Sends a message to the receiver	Regulator, Proxy, Template
`secureSend(sock, msg)`		Sends authenticated encrypted message	
`receive(sock)`		Receive a message from `addr`	
`secureReceive(sock)`		Receives authenticated encrypted message	
`connect(addr)`		Returns an established connection	
`secureConnect(addr)`		Returns a secure SSL connection (authenticity, confidentiality, integrity)	
`disconnect(sock)`		Disconnects connection	
`Storage Interface`			
`write(key, val)`	`key`: search key (or filename) `val`: data to be stored	Writes data in plaintext	Proxy, Template
`secureWrite(key, val)`		Writes encrypted data to file	
`read(key)`		Reads data from file	
`secureRead(key)`		Reads encrypted data from file	
Security Engineers			
`dispatch(sock, msg)`	`sock`: Receiver end-point	Sends the message through socket	
`receive(sock)`	`sock`: Sender end-point	Receives message through socket	
`isConfidential(msg)`	`msg`: data to be checked for	Returns whether data is confidential	

ently and at runtime (Section 3.4.2).

3. Ensuring correct compile time procedures to separate development environment from production environment. This can be done using code annotations in Java (for instance, using profiles in Spring framework) or using preprocessor macros in languages like C (Section 3.4.3).

3.4.1 Semantic APIs

In this section we introduce our semantic APIs, along with the use cases that motivated their design. Table 3.2 lists these APIs.

Our design goal is to allow only a few developers, which we call the Security Engineers, to be involved in making security decisions and provide them with the tools they

```
1  send(sock, msg) {
2    if(!isConfidential(msg)) {
3        dispatch(msg, sock);
4    } else
5        throw Exception("msg is confidential. Use secureSend()!");
6  }
7
8  secureSend(sock, msg) {
9    assert(sock instanceof SslSocket);
10   dispatch(sslSocket, msg);
11 }
12
13 connect(addr) {
14   /* create an HttpUrlConnection object and return socket */
15 }
16
17 secureConnect(addr) {
18   /* create HttpsUrlConnection object, which performs some validation */
19   /* Perform required SSL checks that are not performed by HttpsUrlConnection */
20   /* Perform appropriate check for certificate revocation */
21   /* return sslSocket */
22 }
```

Figure 3.2: Communicate interface - example `send` and `secureSend` functions to perform a connection using HTTP and HTTPS protocols.

need to implement custom protocols that meet certain security requirements, whereas the other developers, called Functionality Engineers should focus on functionality specific needs and their design choices should not affect security. We achieve this goal by ensuring that the functions written by the Functionality Engineers do not involve any security decisions whereas the functions written by the Security Engineers perform all the security tasks and these functions can be used by the Functionality Engineers directly without the complete knowledge of the underlying function and cannot be modified by the Functionality Engineers.

3.4.1.1 Functionality Engineers

Communicate Interface. The functionality required by developer need 1, described in Section 3.3, can be implemented with `connect`, `secureConnect`, `send`, `secureSend`,

`receive` and `secureReceive` functions. `connect`, `send` and `receive` allow communication without authentication or encryption, whereas `secureConnect`, `secureSend` and `secureReceive` ensure both properties.

From a functionality engineers' perspective, the secure functions would be used in the same manner as their insecure counterparts. Such developers would use `secureConnect`/ `secureSend` for sending sensitive data such as login credentials, SSN, credit card numbers, etc. We must therefore ensure that these functions perform all security checks that are required for what developers expect to be a secure channel. A detailed description of a TLS connection establishment for HTTPS and the necessary check to be performed is as follows. An HTTPS connection used for connecting to web servers is HTTP protocol using an SSL/TLS connection. HTTPS is primarily used for two purposes: 1. Authenticating the server 2. Ensuring confidentiality of the message sent to the server. An SSL connection is established by the following steps:

1. First, a *hello* message is exchanged between the client and the server. The client sends all the cryptographic information such as cipher suites that it supports, the SSL/TLS protocol version it supports, etc.

2. Based on the client's information, the server responds with the cipher suite and the version of protocol that will be used.

3. The server then presents a SSL certificates to the client to prove its identity. Each certificate contains details of the server such as name, location, etc., the time for which it is valid, a public key associated with a certificate and a digital signature from root certificate authority (or an intermediate certificate authority). Root cer-

tificate authority's certificates are self signed.

4. For every certificate, the client checks whether it is correctly signed by another certificate authority or whether it implicitly trusts the certificate authority (if the certificate is self signed). In addition, the client validates each certificate's expiry date and verifies the hostnames of the certificates.

5. In addition, the client needs to verify that the certificate has not been revoked in the recent past. This is performed by one of three ways: 1. Checking with a list of revoked certificates that was updated recently (CRLs) 2. Obtaining the revocation status for every request using online certificate status protocol (OCSP) 3. The server sends a time-stamped OCSP response in addition to the certificate (OCSP stapling).

6. Once the client validates the certificate and the server is authenticated, the client and server perform a key exchange protocol to setup a symmetric secret key that would be used to subsequently encrypt communication.

`send` can be used for all other communications and for channels that cannot be secured (e.g. text messages sent by a smartphone).

When sending a message on a secure channel, the functionality engineer specifies only the address `addr` and the data `msg`. The API implementation should ensure that confidential message is always sent using `secureSend` (as shown in Figure 3.2). This can be achieved by requiring security engineers to explicitly specify an `isConfidential` function, which takes a `msg` as a parameter and returns a boolean value, indicating whether the data is confidential.

The `send` function invokes `isConfidential` and returns an error rather than sending confidential data over an insecure channel. The method returns `true` by default, explicitly whitelisting data that is not sensitive. The security engineers define `msg` to be an instance of a specific superclass, which tags all data as sensitive or non-sensitive.

This decouples the task of specifying which data is confidential from the task of implementing network communications, and ensures that the network programmer cannot compromise security by mistakingly using `send` in cases where `secureSend` should be used.

Storage Interface. The functionality required by developer need 2 can be implemented with the `write`, `secureWrite`, `read` and `secureRead` functions. A `write` allows writing a value (or a file) to the storage system in plaintext whereas `secureWrite` ensures the integrity and confidentiality of the file. Whether the data stored is sensitive or not, a functionality engineer only needs to specify a value that is to be stored and a key that can be used to read the value. As in the Communicate interface, an `isConfidential` function must be specified that checks whether some data written in plaintext is indeed non-sensitive, thus decoupling the data sensitivity checks from the application logic.

The APIs presented to the functionality engineer when using the `write` and `secureWrite` (or `send` and `secureSend`) methods are very similar and do not require any parameters that control how security is achieved. The Communicate and Storage interfaces provide a clean separation of concerns by hiding all the cryptography details from the functionality engineer and by decoupling the security specification from the implementation of the input-output functionality.

3.4.1.2 Security Engineers

Unlike functionality engineers, security engineers implement more complex interaction protocols (e.g. authentication and authorization, online shopping, payment processing), and the resulting libraries may be included in third party applications. In consequence, security engineers need a better understanding of security principles, and are responsible for exposing intuitive APIs to the functionality engineers. However, security engineers can also benefit from semantic cryptographic APIs. For example, a security engineer might implement the `send` and `secureSend` functions discussed above. Figure 3.2 outlines an implementation, using the proxy design pattern [46] which adds a wrapper and delegation to protect the real component from undue complexity.

connect/send. The `connect` function creates a socket for communication whereas `send` transmits the message to the destined address. In this scenario, the data may be sent in plaintext. As mentioned earlier, this function invokes the `isConfidential` function to ensure that the data sent over the network is not sensitive.

secureConnect/secureSend. The `secureConnect` function creates a secure communication channel, e.g. by using `HttpsUrlConnection` for HTTPS in Java. The function performs all SSL checks that are not performed by the underlying libraries. For example, if the security engineer uses a JSSE library by Oracle, then this function checks for the expiry of certificates. The function then performs necessary checks for certificate revocation before transmitting the message to the destination. This requires incorporating information from external sources, and can be achieved using the Regulator pattern.

Semantic API

To address the semantic gap between the developers' needs and the cryptographic APIs provided, we allow security decisions by Security Engineers alone while Functionality Engineers focus on application specific design choices.

3.4.2 Integrating External Information

In this section we describe the Regulator pattern for incorporating external information, such as parameters known to be insecure or certificates that have been revoked. We will start with a motivating example for the proposed design pattern and then will explain the pattern in standard format.

- ***Regulator:*** This pattern can address one of the most common mistakes when using cryptographic APIs: not checking if TLS certificates have been revoked and thus accepting potentially compromised certificates for authentication. Certificate revocation check can be implemented by downloading certificate revocation lists (CRLs), through the Online Certificate Status Protocol (OCSP) or OCSP stapling [72]. Each of these methods require a different mechanism to integrate the information provided by a trusted external source and hence we propose separate (but closely related) design patterns for each method.

  X.509 certificates are issued to a web server by certificate authorities (CAs). These certificates serve as a *proof* to the client on the server that they are connecting to. The certificates are valid for a certain duration of time after which they need to be reissued. However, due to potential key compromises and other such reasons,

the CA may revoke the certificate. Thus, despite proving to the client about the existence of a certificate, the client needs to additionally verify if the certificate has been revoked. We present some of the methods used to check for revocation in the following.

1. **Certificate Revocation Lists (CRLs):** CRL is a list of revoked certificates maintained by the CAs. To use CRLs, the client intermittently retrieves a fresh CRL from the CA and verifies a certificate against this CRL. Whether a client has the most recent list depends on the frequency of requests made to the CA and this impacts the number of queries that the CA needs to support. An intermediary can reduce load on the CA by retrieving these lists more frequently and pushing them to the clients.

2. **Online Certificate Status Protocol (OCSP):** OCSP is an alternative to CRLs where, for every certificate, the client verifies for the certificate's revocation status with the CA (OCSP server).

3. **OCSP with Stapling:** In OCSP stapling the server provides the certificate and client validates the certificate. If the certificate is revoked, the client then checks the time elapsed since the revocation. If the certificate is presented in some pre-defined time-frame since the revocation, the client accepts the certificate. Otherwise the client forwards the certificate to CA (OCSP server) for validation.

Pattern Name and Classification: Regulator is a behavioral design pattern. Like the traditional Observer pattern [46], Regulator defines a one-to-many relationships

between objects and allows the dependent objects to update themselves automatically whenever the subject changes its state. However, there are two differences between the Observer and Regulator patterns:

1. The *Regulator* pattern does not expect the subject to be aware of all the dependent regulators, and hence the subject does not notify the regulators whenever the subject changes its state. On the contrary, it is regulators' responsibility to check if the subject has changed the state.
2. The *Regulator* pattern *does not* allow the regulators to modify the state of the subject. This is crucial since the subject is expected to be a standard or benchmark for all the regulators.

Intent: The intent of this design pattern is to facilitate mechanism for integration of information from external sources. For example it allows one to update various critical information based on the current standards. This ensures that the security checks are performed according to the current standards instead of outdated ones, thus avoiding any weak implementations. It is recommended that the regulator pattern should be used for every interface which requires to choose parameters from multiple available options by comparing them with some benchmarks.

Motivation: The prime reason for using this pattern is the unfortunate observation that many of the critical errors in the cryptographic implementations are caused by implementations of weak cryptographic algorithms like `RC4` or `MD5` hash function. Another example is using obsolete certificate revocation lists (CRLs) to validate revoked certificates. We therefore propose that the implementation of such interfaces

should incorporate the regulator pattern which will update the lists periodically on its own and synchronize the lists with the current approved standards (like those published by the NIST [14]). This pattern could be used for periodic updates of CRLs and list of secure algorithms used for SSL handshake negotiation. We wish to highlight that attacks on implementations using specific parameters are much more frequent than attacks on the algorithms themselves. It is of utmost importance, therefore to keep the parameter selection up-to-date with standards at run-time instead of compile-time.

Applicability: The regulator design pattern should be used in the design whenever security depends on performing checks with reference to content published by a trusted authority.

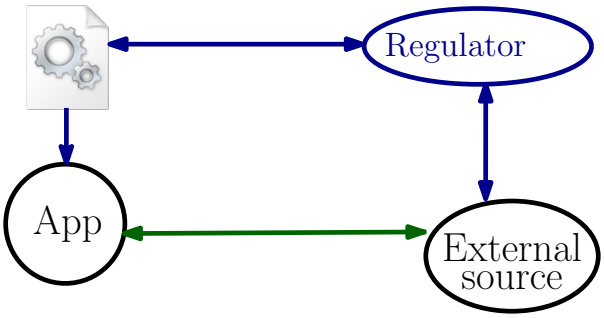


Figure 3.3: General structure of Regulator pattern - The darkblue arrows are used to indicate updating of parameters by a regulator, which retrieves this data from an external source, intermittently. The dark green arrows indicate update where an application directly contacts the subject.

Structure: Figure 3.3 shows the high level idea of how the regulator design pattern can be used for updating list of secure algorithms or list of secure parameters used for encryption. We later present detailed examples of regulator pattern that can be used in different models of updating the standard information.

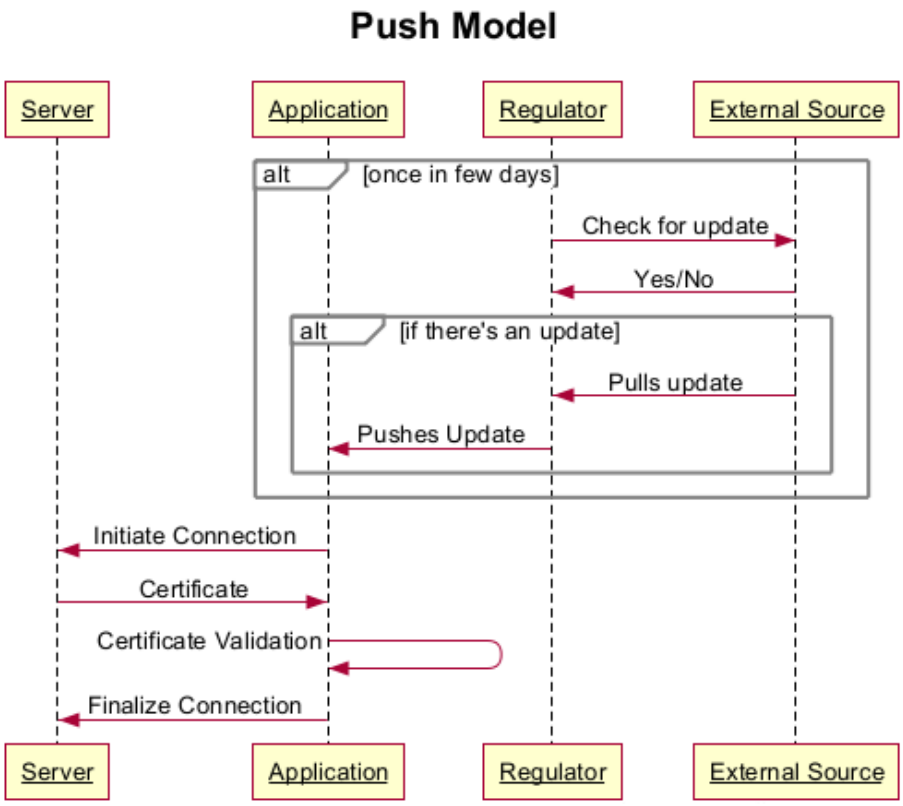


Figure 3.4: Sequence Diagram showing the Push Model

We now propose 3 ways of implementing the *regulator* pattern. These 3 implementation choices differ in the situations where they can be applied. We explain these differences below and give an example for each of the implementations.

– *Push*: This implementation is illustrated in Figure 3.4, which corresponds to certificate revocation checks using CRLs. Here, the application stores a local copy of CRL and uses it to verify whether the certificate provided by the server is valid or not. The regulator also maintains a copy of CRL and updates its own copy periodically by downloading the CRL updates from the certification authority (CA). The regulator then pushes the updated CRL to application and replaces the local CRL of the application with the updated

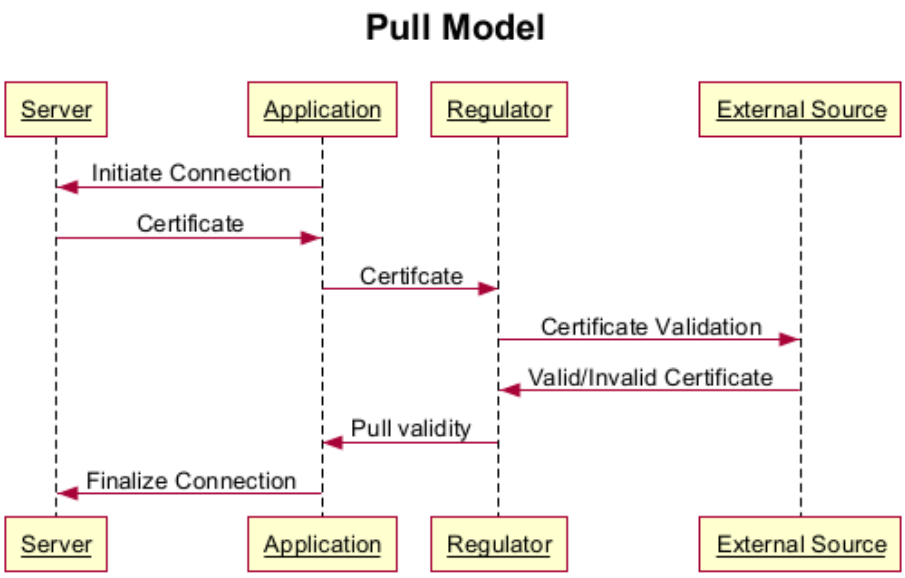


Figure 3.5: Sequence Diagram showing the Pull Model

one. The main advantage of this method is the application (or client) can check for the revocations locally. However, the disadvantage of this method is the application is required to store the (potentially large) list locally.[4]

This method can also be used in scenarios where the regulator must update the lists from multiple external sources.

– *Pull*: This implementation is illustrated in Figure 3.5, which corresponds to certificate revocation checks using the OCSP protocol. Here, the application checks for the validity of the certificate by sending the certificate to an OCSP server (or to another trusted authority, e.g. a CA). The certificate is validated based on the response from the OCSP server (or CA). Here, the regulator relays the responses between OCSP server (or CA) and the application.

We do not recommend this method in practice, because it adds overhead to the

[4]In the wake of events that triggered mass revocations (e.g. the Heartbleed vulnerability [124]), CRLs grew by up to two orders of magnitude.

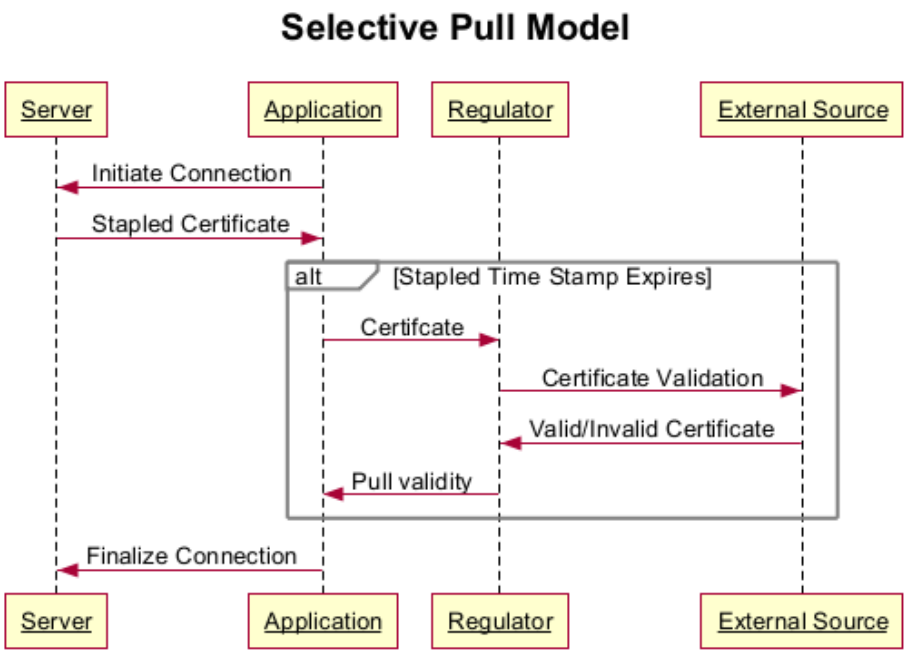


Figure 3.6: Sequence Diagram showing the Selective Pull Model

secure channel establishment and because of the potential privacy concerns related to disclosing to a third party every HTTPS site visited. As a result of these drawbacks, OCSP is used only in rare cases [72].

– *Selective Pull*: This implementation is similar to the *Pull* model, but in this case the information is downloaded only in certain conditions. This is illustrated in Figure 3.6, which corresponds to certificate revocation checks using OCSP Stapling. Here, the application requests the regulator to validate the certificate. The certificate is accepted as valid based on the time-stamp when the certificate is provided. If the regulator is not able to verify the certificate's validity locally then it forwards the certificate to OCSP server (or CA). The decision regarding accepting the certificate is then taken based on the response from the OCSP server (or CA) and relayed back to the application.

This method may be used instead of the *Pull* method, as it reduces the latency by allowing certificates to be caches for a certain period of time.

Beside certificate revocation, these patterns can also be used in other scenarios where information from some external source needs to be integrated transparently, without requiring the application developers to retrieve and interpret the information. For example, the Regulator pattern could be used in updating the list of secure algorithms; a trusted third party (e.g., the NIST, who already publishes standards for secure algorithms and key lengths [14]) could provide this information in a machine readable format, suitable for ingestion by our framework. As mentioned in Section 3.3, NIST periodically publishes recommendations about secure cryptographic algorithms to use. Presently, adhering to these recommendations is up to application developers, leading to wide use of old broken algorithms such as MD5 hashing. Instead, we propose that a trusted authority (such as NIST) maintains an service with a list of all secure algorithms at any point in time. Every client can periodically (say once every three months) update their local list using an automated procedure. The application would hence use only those algorithms that are recently listed as secure by NIST.

Consequences: The Regulator pattern allows application developers to use only algorithms and parameters that are considered secure, and it allows library developers to control the mechanism for retrieving external information. This is important, as there is currently no agreement on the best way to check the revocation status of TLS certificates [72], and the standards and protocols will likely continue to evolve. The main advantage of this pattern is that the subject does not need to be aware of all the regulators.

Further benefits and liabilities of the regulator pattern are:

1. It decouples the regulator from the subject, allowing the subject to be external and independent from the regulator.

2. The regulator can never alter the state of the subject, which ensures that all the regulators have the exact same view of the subject's state when they update themselves.

3. One of the liability is that the regulator adds an extra round of communication when used in the pull model.

The Regulator Pattern is related to the observer, mediator and iterator patterns. Similar to the observer pattern, it defines one-to-many relationships between objects, but in the regulator pattern the subject need not be aware of it's objects and does not notify any change in it's state. Similar to the mediator pattern, it encapsulates how objects interact and additionally, the regulator pattern captures changes in external information and pushes the changes to the objects. Similar to iterator pattern, it defines how objects access external information iteratively without loss of information. The regulator pattern is different from the observer pattern as it checks for updates and pushes the updates to it's objects besides just defining how the information is accessed.

External Information

Regulator pattern can be used to integrate security critical information at runtime from trusted external sources like NIST to avoid using revoked certificates and/or broken algorithms and cipher suites.

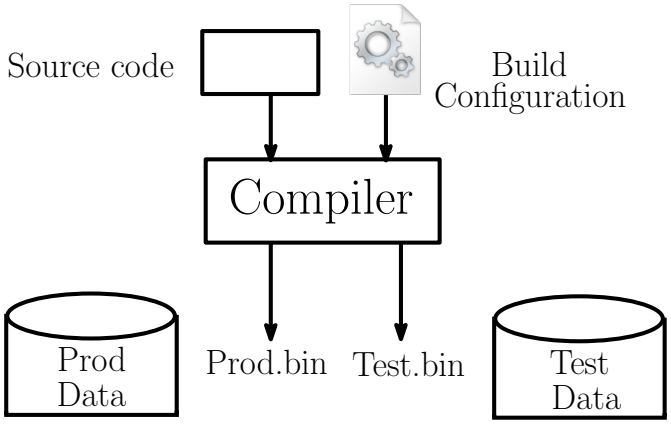


Figure 3.7: Producing separate binaries for test and production environments using a build configuration.

3.4.3 Managing Security Checks during Development and Testing

When developers disable security checks during development and testing, e.g. by utilizing self-signed certificates to bypass certificate validation checks [43], they need a way to specify that these workarounds should not be included in production releases. This separation should be ensured at compile time, which allows production releases to avoid the overhead of these workaround checks. As software projects typically maintain separate build environments for development and production, a natural way to fulfill developer need 5 is to ensure that workarounds are confined to the development build environment.

For example, when using self-signed certificates during development, developers could maintain separate validation rules for certificates. The validation rules for the test environment would check *TestKeystore*, which may contain self-signed certificates. The validation rules for the production environment would check *ProdKeyStore*, which only accepts valid certificates. The build configuration defines the properties of the two build environments, as illustrated in Figure 3.7, and the resulting binaries use the appropriate

keystores. This provides a clean separation between environments, which reduces the risk that security bypasses will propagate to production releases. This can be implemented with Maven build manager by specifying the tasks in the build file to select the correct keystore (*TestKeystore* or *ProdKeystore*) for the different build profiles corresponding to the different environments. The source code would check the certificates with the keystore that was selected during compilation.

Figure 3.8 shows a segment of the `pom.xml` build file that defines two build profiles, `test` and `production`. The tasks for each of the profiles define which of the keystores (*TestKeystore* or *ProdKeystore*) from among the `resources` should be selected and the corresponding keystore is selected based on the profile chosen during compilation (the profile id is passed as an argument to the -P option). The selected keystore is then used by the source code to check for valid certificates.

3.5 Case Studies

We conduct a qualitative evaluation of our proposed APIs, seeking to answer the questions: *Can our semantic APIs be utilized to implement non-trivial client-server?* and *Can they help prevent the common misuses of cryptography?* We consider two realistic case studies. In Section 3.5.1 we discuss the implementation of a mobile payments app, and in Section 3.5.2 we explore a range of options for implementing a secure messaging service.

```
<profiles>
  <profile>
  <id>test</id>
  <build>
  <plugins>
     <plugin>
        ....
        <executions>
           <execution>
              <phase>test</phase>
              <goals>
                 <goal>run</goal>
              </goals>
              <configuration>
              <tasks>
                <delete file = "${project.build.outputDirectory}/keystore"/>
                <copy file = "src/main/resources/TestKeystore"
                     tofile = "${project.build.outputDirectory}/keystore"/>
              </tasks>
              </configuration>
           </execution>
        </executions>
     </plugin>
  </plugins>
  </build>
  </profile>
  <profile>
  <id>prod</id>
  <build>
  <plugins>
     <plugin>
        ....
        <executions>
           <execution>
              <phase>test</phase>
              <goals>
                 <goal>run</goal>
              </goals>
              <configuration>
              <tasks>
                <delete file = "${project.build.outputDirectory}/keystore"/>
                <copy file = "src/main/resources/ProdKeystore"
                     tofile = "${project.build.outputDirectory}/keystore"/>
              </tasks>
              </configuration>
           </execution>
        </executions>
     </plugin>
  </plugins>
  </build>
 </profile>
</profiles>
```

Figure 3.8: **A pom.xml file which selects different keystores for development and production environment.**

Security Checks

To prevent bypassing security checks that are used in development or test phase from permeating into production releases, we should use compile time checks for example separating the keystores for test and production phases.

3.5.1 Case Study 1: Mobile Money Application

Mobile money applications allow users to send and receive money without exchanging physical currency, which reduces the risk of theft. Reaves et al. analyzed seven mobile money apps for Android and reported severe vulnerabilities resulting from misuses of cryptography [93].

Security Objectives. Any mobile money application would require the user to register with the app and login to view account balances or make financial transactions. For this, the application is required to connect to the banking server and to authenticate this server. This can be achieved by ensuring proper certificate validation and verification checks are performed (the detailed TLS handshake protocol is reviewed in Section 3.4.1.1). Once a secure connection is established with the banking server, the sensitive information like credit/debit card numbers, account numbers, SSNs, passwords must be sent over the secure channel. Additionally, the user's data and preferences must be stored on the device but these files are usually a system level resource that can be easily accessed by other applications. As the data may include sensitive information related to financial transactions, it is important to store these files securely.

Using Semantic APIs. We can utilize the semantic APIs proposed to isolate the security related decision making from the application developer. We establish a secure channel to the banking server, while authenticating the server correctly, by invoking the `secureConnect` function, which implements all the necessary validation and verification checks. We can then send sensitive information over this channel by using the `secureSend` function. Internally, this function ensures that an authenticated connection is established and encrypts the data with secure implementations of standard algorithms before transmitting the data. In addition, the developer implements an `isConfidential` function, which flags all the financial information and the private user information as sensitive. While we employ self-signed certificates during development and testing, this is disabled at compile time when building the production release.

Avoiding Common Errors. Reaves et al. reported that the most common mistake in practice is to disable hostname and certificate validation to be able to use self-signed certificates [93]. This leaves the application vulnerable to man-in-the-middle attacks. With our API, developers do not have the opportunity to make this mistake, as they only invoke the `secureSend` and `secureConnect` functions. The necessary checks are performed by these functions, thus preventing the developer from bypassing these checks. The flexibility to test using self-signed certificates is ensured by using a build system as described in Section 3.4.3.

Another common mistake is not to check whether the TLS certificate presented by the server had been revoked. This check is performed automatically by `secureConnect`, using the revocation interface provided to the library developers. The library developers

can choose the method to check for revoked certificates. In the case of mobile applications downloading large CRLs is not a good choice, so we utilize OSCP stapling instead.

`secureSend` ensures that sensitive information is encrypted and sent to an authenticated endpoint. This is transparent to application developers, as the actual parameters, algorithms and keys used for encryption are not exposed. This prevents other mistakes such as the key publicly exchanged or using static keys.

Finally, because sensitive data is flagged by the `isConfidential` function, invocations of `send` or `write` on such data will fail. Sensitive data can be stored in logs and user preference files only by using `secureWrite`, and can be accessed only by legitimate users using `secureRead`.

Summary. This case study illustrates the separation of concerns provided by our APIs, which insulate the developers from the low level cryptographic operations, while still allowing them to decide which channels should be secure and which data is sensitive. The APIs also prevent the most common mistakes involved in the implementation of secure protocols and in storing data in local files.

3.5.2 Case Study 2: Secure Messaging

Secure messaging services aim to provide a secure alternative to sending unencrypted emails or text messages. In this scenario, the messages exchanged among users are encrypted end-to-end.

Security Objectives. Here we will classify the security goals as stated by Unger et al., who defined three security goals of secure messaging applications: trust establishment, message privacy and integrity (or conversation security) [113]. Trust establishment refers to ensuring that the secure communication is established with the intended party. This can be implemented with a key exchange, along with certificate validation. Conversation security refers to the privacy and integrity of the messages sent in the presence of an active network adversary. As illustrated in [113] it is impossible to accommodate all the security goals in a single application. This is due to the fact that many times the security goals (including usability and feasibility constraints) offer a trade-off between themselves. In fact, usually it is not feasible even to accommodate the most inclusive security features together due to the trade-offs between variety of security goals. For readers interested in the further discussion on this topic we redirect them to [113].

In this case study, we consider an example application which meets the following security goals, which provide an intuitive notion of security:

- Trust Establishment
  - Network MITM protection: Prevent man-in-the-middle attacks by network adversaries. This means that, at the time of connection establishment, no third party can claim the identity of the desired end point.
  - Key Revocation Possible: Support mechanism to allow easy revocation (and/or renewal) of keys (or credentials). This refers to the capability of removal of compromised secrets used for authentication.
- Conversation Privacy

– Confidentiality: Only the intended participants are able to read the message.

– Integrity: Any message modified in the transit will never be accepted by an honest party.

– Authentication: All participants are able to verify the message was sent by the claimed source.

These security guarantees suffice the required level of security by most of the common applications. Moreover, as illustrated in [113] these are also the security guarantees provided by most of the usable (and hence widely deployed) protocols and applications.

Using Semantic APIs. This level of security can be achieved by using the `secureSend` function. The developer can simply invoke the function and send the message to its destination, without explicitly choosing an encryption method and connection protocols. As all messages sent by the application are considered confidential, the insecure `send` function cannot be used.

In some cases, the applications may wish to use additional verification methods to ensure privacy from service providers or CAs. These additional measures (e.g., a secure multi-party computation algorithm called the Socialist Millionaire Protocol [113]) could be incorporated in our API by modifying the `secureConnect` function. For instance, this could be done by a library developer in a manner that is transparent to the application developer who invokes the `secureSend` or `secureConnect` functions. To implement this functionality, the library developer can invoke an additional function, beside the certificate validation checks already present. This new function would implement the checks

required by the Socialist Millionaire Protocol.

Avoiding Common Errors. Owing to the separation of concerns, the application developer is never exposed to low level decision making of selecting *secure* encryption algorithms or parameters. In fact, these can be specified in the configuration file as environment specific parameters which gives the flexibility to implement different levels of security in the different settings. This configuration file is used during the build phase and can also be updated without application developers' knowledge using the Regulator pattern. Thus, this approach eliminates the mistakes caused due to complicated and inconsistent checks required to establish a secure connection for various implementations (and various protocols). It also prevents using self-signed certificates or other security workarounds in the production environment. Moreover, the Regulator pattern allows the transparent integration of standards which again deters the attacks exploiting the use of broken algorithms or parameters.

Summary. Unlike popular cryptographic frameworks like JSSE, our APIs avoid exposing low level implementation details while allowing library developers to extend the existing functionality with new security protocols. This flexibility is key for the secure messaging use case. Additionally, the level of protection against cryptographic misuse would be difficult to achieve with a library that simplifies the cryptographic APIs too much. For example, NaCl [20] provides only a `crypto_box` function for performing authenticated encryption, but does not provide methods for establishing secure connections or for integrating external information. In consequence, NaCl does not provide protection

against the common mistakes involving certificate validation.

3.6 Discussion

Most of the documented misuses of cryptographic APIs can be explain by the inappropriate abstractions provided to developers who lack a background in security or cryptography. However, achieving an effective separation of concerns requires understanding of the security decisions that developers must make. For example, it is not reasonable to expect these developers to know in which circumstances the SHA-1 hash function can be used securely (it is currently not recommended for digital signature generation, but it is allowed for all other applications [14]), but we should expect them to determine the sensitivity level of the data handled by their code. Other security decisions may be less obvious. In particular, developers have a legitimate need to disable security checks during development and testing, and they must also be able to select the mechanism for retrieving information about revoked TLS certificates, as there is currently no agreement about what method is best and the choice is likely to be platform dependent [72].

We infer and classify the developer needs based on the misuse cases that have been documented so far. This is only a first step towards understanding how to help developers make fewer mistakes with cryptography, as other applications may give rise to additional security needs. However, by trying to address the needs we currently understand, we identify two challenges that are specific to our domain: how to incorporate external information, at run time, and how to define compile-time checks for excluding security workarounds needed during development. This highlights the fact that a solution to the

problem of cryptographic mistakes must go beyond a good API design. Additionally, providing developers with good documentation about the cryptographic framework is an important, but this is likely insufficient [5].

Another avenue for future research is to develop a method for evaluating the effectiveness of our solutions. A potential approach is to conduct an controlled experiment with two teams of programmers, similar to Yskout et al. [121]. The key challenge is to assess the impact of our solutions on the security of the resulting code (rather than on the programmer's productivity), as creating meaningful security metrics is difficult, in general [19]. Such an experiment would likely require a systematic way of finding the cryptographic vulnerabilities introduced by each programmer, for example by having a panel of experts separately inspect the code that participants write.

3.7 Conclusion

This chapter focused on direct risks from the perspective of addressing misuse of cryptographic APIs.

Modern cryptographic frameworks like Oracle JSSE, IBM JSSE, BouncyCastle, OpenSSL are widely used to develop secure applications. However, software developers who lack a background in security or cryptography often make mistakes when using these frameworks, and these mistakes usually introduce severe security vulnerabilities.

My work first sought to understand and categorize the five concrete needs of the developers who utilize cryptographic frameworks and identify four root causes for these mistakes: the lack of a separation of concerns, the diverse needs of developers, the fact

that security often depends on information external to the system and reliance on secure default values. We then used those insights to develop a new interface for third-party resource inclusion that is more resilient to developer errors as it allows developers to make effective security decisions but does not expose low level implementation details. These APIs can be implemented consistently on different platforms by using novel and known design patterns. In particular, we propose a Regulator pattern for incorporating external information from trusted sources, e.g. the revocation status for TLS certificates or knowledge about insecure cryptographic algorithms and key lengths. We also propose compile-time checks to isolate the certain workarounds to the development build environment, in order to address the legitimate need to disable security checks during development and testing. Finally, we discuss the research avenues opened by these ideas.

Collectively, this work shows that the direct risks can be mitigated by better understanding the *users' needs and limitations* on either side of the interface, and designing in a way that meets them. The end result, like with our semantic API, may not always be the *overall smallest* interface—traditional cryptographic APIs do not account for different classes of engineers—but it ought to result in that which presents the smallest necessary interface for each specific class of user. So doing allows developers to be able to more easily reason about what exactly the code they directly include is actually doing. As we will see in the next section, however, this is only half of the challenge, as sometimes the details of what the code is doing can be hidden through indirect or repeated inclusions.

Chapter 4: Sound Methodology for Downloading Webpages

In our paper [61], we demonstrate several measurement parameters that can have a significant effect on how much of the inclusion graph a tool is able to obtain. In particular, we show that the `UserAgent` string, number of times the page is loaded, specific Node.js event handlers, and the method by which the inclusion graph is constructed can lead to significant differences. Surprisingly, many studies fail to specify precisely what these parameters are. The papers that introduce Crawlium [10] and ZBrowse [67] specify their tools' network events, but do not investigate multiple page loads. In this chapter, we describe in detail the sound methodology to obtain the complete inclusion graph of a webpage at a given point in time accounting for the various parameters.

4.1 Overview

Today's web is highly dynamic, with extensive third-party inclusions [10, 59, 88] such as basic resources (e.g., `fonts.google.com`), user tracking (e.g., `scorecardresearch.com`), or advertising.

One can think of today's websites as complicated *inclusion graphs* [10], wherein the nodes are resources loaded from domain names, and there is a directed edge from resource r_1 to resource r_2 if r_1 caused r_2 to be loaded (e.g., if r_1 were a JavaScript file that generated

a GET request for an advertisement r_2).[1] It is important to many research questions to be able to obtain as full a picture of a webpage's inclusion graph as possible. For instance, explorations into third-party inclusions [10, 59, 88], malware [62], and website performance [79] are all sensitive to how complete a view of the graph they can obtain at any point in time.

This paper asks a straightforward question: *What is the best way to download a webpage so as to obtain as much of the page's inclusion graph as possible at any one point in time?* Getting a full snapshot of a webpage's inclusion graph can be very difficult, owing to the fact that: (1) The domains, resources, and edges that make up the website's inclusion graph can vary dynamically, even from back-to-back refreshes. (2) Resources are loaded in myriad ways, such as through static inclusions in HTML, dynamic calls in JavaScript, WebSockets, and so on.

We look at this question from two key perspectives:

What effect does the *choice of tools* have on the obtained inclusion graph? Early efforts used basic tools such as `curl` or `wget`, but neither of these support executing JavaScript, which has been shown to be deployed in more than 93% of the most popular webpages [83]. More recent measurement efforts tend to use headless versions of more full-fledged browsers, and programmatic front-ends such as Puppeteer [90] or Selenium [101] to drive them. It would seem at first glance that, so long as the headers and the backing browser are the same, the inclusion graphs would be the same, too.

To evaluate this hypothesis, we compare two recent, sophisticated tools that arose

[1] Arshad et al. [10] originally introduced these as inclusion *trees*, but we refer to them as inclusion *graphs* because we observe that nodes can commonly have more than one incoming edge.

from the research community: Crawlium [32] and ZBrowse [122]. We find that, surprisingly, both obtain parts of the resource graph that the other does not, even when providing the same headers and controlling for dynamic webpage content.

***How many times* should one refresh a page to obtain a more complete snapshot of a webpage's inclusion graph?** To answer this question, we repeatedly download websites and compare the nodes and edges in the inclusion graph with each refresh. We find that there is a surprisingly high variability in the dynamism of websites, with some (like `wikipedia.org`) being highly static over short periods of time, and others resulting in new content with virtually every refresh. The majority of websites are somewhere in the middle: that is, their inclusion graph can be captured in its entirety by reloading the page several times. We present and analyze an *adaptive* webpage loading strategy that fully obtains the inclusion graph without excessive reloads for any page.

We analyze both of these questions across the Alexa top-10,000 (and 10,000 sites selected randomly from the 10,001-1M Alexa-ranked sites), and using multiple user-agent strings, to rule out potential differences between desktop and mobile version of websites. Collectively, our results lead us to a set of considerations that researchers should have when obtaining and analyzing inclusion graphs. We will be making our code and data publicly available.

The rest of this chapter is organized as follows. In Section 4.2, we evaluate the effect that different tools can have by comparing the resource graphs obtained by two popular tools (Crawlium and ZBrowse), and find that neither of them gets the entire inclusion graph, but that they do complement one another. One tempting explanation for

these results is that tools differ simply because websites are highly dynamic; in Section 4.3, we show this not to be the case, but rather some tools *consistently* get parts of the inclusion graph that other tools do not. In Section 4.4, we analyze how many times webpages should be loaded to obtain a full view of the inclusion graph, and propose an *adaptive* downloading strategy. Finally, in Section 5.6, we conclude with a set of recommendations.

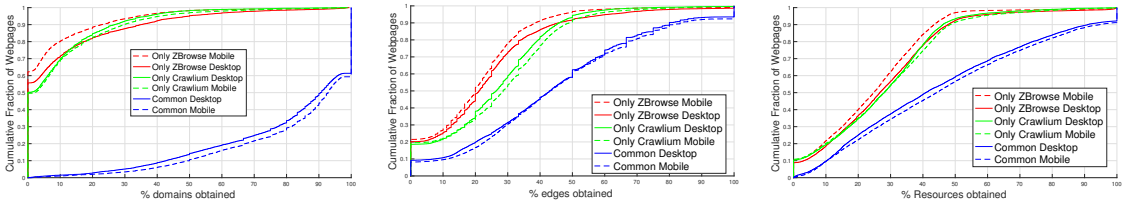
4.2 What Effect Do Tools Have?

In this section, we ask: does choosing a different automation tool result in a different inclusion graph, even if all other fields (headers and `UserAgent` strings) are kept the same?

Because we are focused on obtaining the *inclusion graph*, this precludes tools that obtain web content but do not record the precise provenance necessary to construct the inclusion graph. For this reason, we do not explore OpenWPM [42], Puppeteer, or Selenium. Certainly, these could all be modified to obtain the inclusion graphs, and so doing would be an interesting (and useful) endeavor. The goal of our work is not to exhaustively compare all tools, but to demonstrate that even seemingly minor differences in two tools can result in significantly different results.

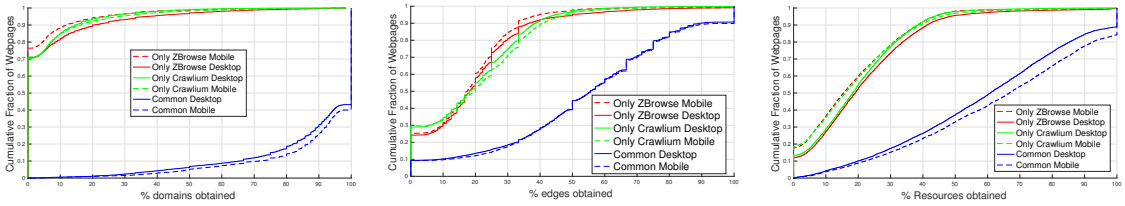
We focus our study on two seemingly similar tools that natively report on the inclusion graph: Crawlium [10, 15–17] and ZBrowse [59, 67]. We chose these specific tools because they are, to our knowledge, the state-of-the-art of methods for collecting an inclusion graph, and we decided to consider inclusion graphs as they represent a superset

of the information that one could extract from a webpage.



(a) Domains. (b) Edges. (c) Resources.

Figure 4.1: Comparison of the domains, edges, and resources obtained by Crawlium and ZBrowse when obtaining the **Alexa top-10k** sites. These plots compare when both *Desktop* and *Mobile* `UserAgent` strings are used.



(a) Domains. (b) Edges. (c) Resources.

Figure 4.2: This is the same as Figure 4.1, but focused instead on less popular sites (a random selection of **10k sites from the Alexa top 10,001–1M** most popular websites). Less popular sites tend to have more in common between the two tools.

4.2.1 Methodology

To compare what the tools obtain, we use them to download each webpage in the Alexa top-10k most popular websites, as well as 10k less popular websites chosen uniformly at random from the websites with Alexa rank between 10,000 and 1M. We use the Alexa ranking instead of the Tranco list, because Tranco list contains the most popular URLs that are visited, be it by human action or not, while Alexa ranking list contains the most popular sites that users explicitly go to. For instance, `googletagmanager.com` is ranked 15 in the Tranco list while it is not ranked anywhere in Alexa top 10,000 web-

sites. Alexa-ranked websites tend to include more third-party resources (including those that are Tranco-ranked), and thus we view Alexa as a sort of "worst case scenario" for crawling an inclusion graph.

These tools require protocols to be explicitly given (`https` versus `http`) and can fail on some pages without the `www` subdomain. To account for this, we try loading each page in the following order of prefixes, whichever succeeds first: `https://`, `https://www.`, `http://`, `http://www.`.

We will demonstrate in Section 4.4 that it is important to reload a webpage multiple times to ensure broad coverage of the page's inclusion graph. For the results in this section, we use the Adaptive strategy with $\delta = 3$: that is, for each individual page, and for each tool, we load the webpage repeatedly until there are three consecutive page loads that yield no new domains or edges, up to a maximum of 30 page loads. We disable caching between separate page loads, and we wait until the page has loaded completely (or timed out, at 2 minutes) before continuing.

To evaluate whether the tools' coverage differ for desktop versus mobile browsing, we run two separate trials with different `UserAgent` strings: one[2] (*Desktop*) that purports to be running Chrome in Mac OS X, and another[3] (*Mobile*) that claims to be Chrome on iOS. We performed several tests comparing desktop- and mobile-builds of the browsers, but observed no difference, so in our experiments we used the desktop-build but varied the `UserAgent` string.

After downloading each of the Alexa top-10k sites, we union together the resources

[2] `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36`

[3] `Mozilla/5.0 (iPhone; CPU iPhone OS 13_3 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) CriOS/69.0.3497.105 Mobile/15E148 Safari/605.1`

(complete URL) obtained, the corresponding fully-qualified domains and the directed edges between them representing the redirections/loading to construct a complete inclusion graph for each individual site. Unfortunately, Crawlium and ZBrowse both sometimes fail to download webpages; to permit a direct comparison in this section, we only compare the 8,221 pages among the Alexa top-10k sites for which both tools were successful in downloading them. We observed two broad reasons for failure: exceeding our two-minute timeout (reducing this failure would have required drastically increasing our time to collect data), and ephemeral errors in reaching the websites (which we verified by manually visiting the websites at the time). We compare the data only for those websites which have successful downloads for both tools and both `UserAgent` strings. We accordingly downsampled from the less-popular webpages to 8,221, as well.

For each of the webpages we could successfully download, we obtained four inclusion graphs, accounting for the two tools and the two `UserAgent` strings. This gave us a total of 65,768 inclusion graphs ($4 \times 8{,}221 = 32{,}884$ for the most popular, and an equal number for the less popular sites). For a particular website, we alternated between Crawlium and ZBrowse and ran both the tools until no new data was obtained. We did this simultaneously for both the mobile and desktop `UserAgent` strings. Thus, the data for a single page load of any particular website for both devices is obtained with a difference of less than our timeout of two minutes (typically much less) to enable thorough comparison.

4.2.2 Results

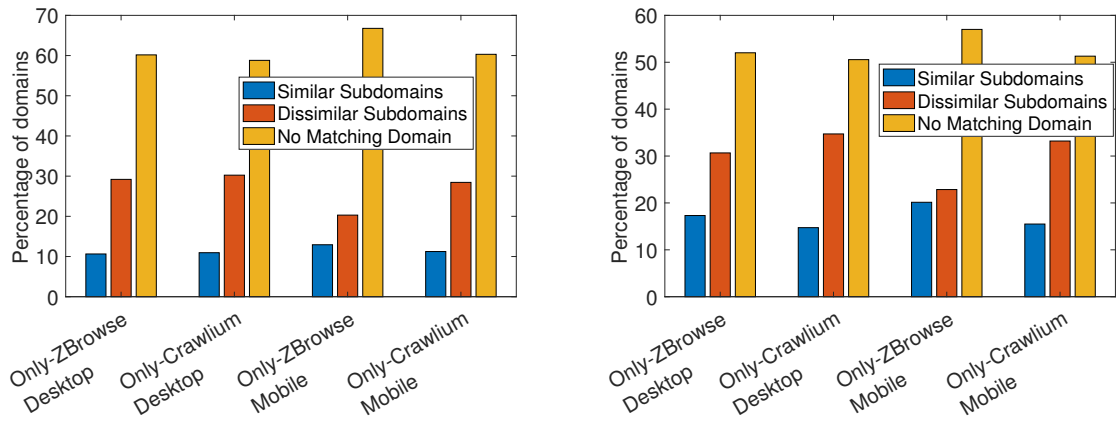
We begin by comparing the percentage of data that is obtained only by Crawlium, only by ZBrowse, and common to both. Figure 4.1 presents the comparison between the (a) fully qualified domains, (b) inclusion graph edges, and (c) resources obtained by Crawlium and ZBrowse for Alexa top-10k sites. Figure 4.2 presents the equivalent data for the less-popular sites. We observe that the most and least popular sites show similar trends, but that in general there is more agreement between tools for less popular sites. For the remainder of this section, we focus on the top-10k most popular sites, but our overall observations hold independent of popularity.

We investigate the three data types in turn.

Differences in domains Figure 4.1(a) shows that, for the median webpage, Crawlium and ZBrowse agree on over 90% of the domains for both desktop and mobile. About 40% of webpages in the Alexa top-10k return the exact same domains to both tools (this corresponds to the jump in the common line at x=100%). However, there is still a significant fraction of webpages where the tools differ considerably. For 20% of webpages, they disagree by 38% of the domains for desktop and 33% for mobile.

When they disagree, Crawlium tends to obtain more domain names than ZBrowse, as expected, but surprisingly, ZBrowse is still able to obtain many domains that Crawlium does not. This is surprising because Crawlium subscribes to more network events than ZBrowse.

We initially hypothesized that the tools' differences may be superficial, and attributable to load balancing, such as one tool obtaining `gtms01.alicdn.com` and the



(a) Alexa top-10k sites.

(b) Less popular 10k Alexa sites.

Figure 4.3: How the **domains** found by only one tool compare to the domains found by the other. The tools show no significant difference for less popular sites.

other obtaining `gtms03.alicdn.com`. To better understand the nature of the differences between the two tools, Figure 4.3 shows *how different* the domains are. To ensure consistency in the differences, the data is compared not for a single page load but for multiple page loads (until three consecutive page loads yield no new data). Each group of numbers in this plot correspond to the percentage of domains that are unique to the specific tool (and `UserAgent`), broken down into three categories:

First are the domain names that, although unique, are very similar to a domain name that the other tool obtained. More precisely, the tool finds a domain d for which there is another domain d' found by the other tool that shares the same effective second-level domain (E2LD, e.g., "`example.com`" in "`www.example.com`") *and* the subdomain has a Levenshtein distance of at most 3 (not including the separating period). For instance, this category would include prefix differences, for example `http://www.maxtv.cn` obtained by one tool, while both obtain `https://maxtv.cn`. The second category is ones where the E2LD matches but the subdomain differs by a Levenshtein distance of more than

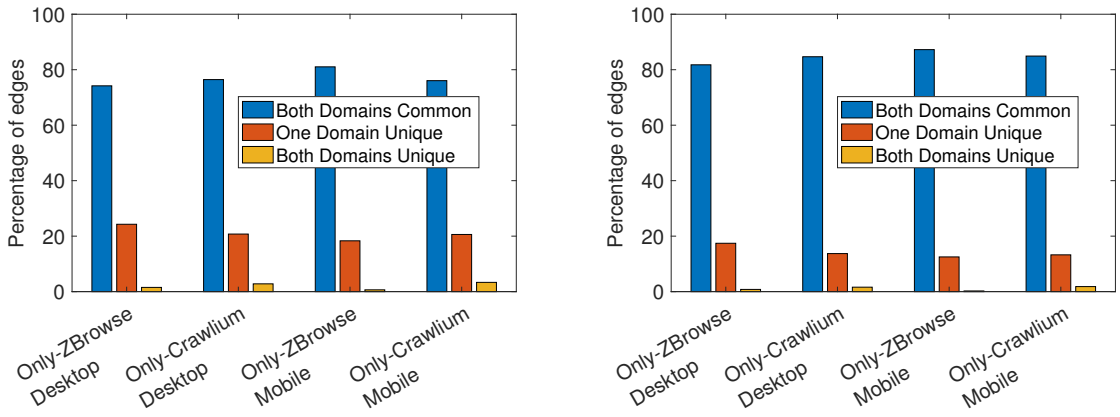
3. Finally, the third category are domains found by the given tool for which there is no matching E2LD found by the other tool.

Figure 4.3 shows that only a small fraction (9–11%) of differences are attributable to very similar domains. Rather, the bulk of the differences are due to finding *completely different* domain names.

Differences in edges Crawlium and ZBrowse disagree significantly more on edges than they do on domains. As shown in Figure 4.1(b), for the median website, the tools agree on only 43% of the edges (55% for less popular sites). Moreover, less than 8% of webpages yield the same exact edges in both tools. For 70% of webpages, ZBrowse regularly finds fewer unique edges in the inclusion graph than Crawlium, especially for mobile webpages. In the long tail, there are about 10% of pages for which ZBrowse finds considerably many more edges than Crawlium on the desktop. Much like with domain names, this indicates that, although Crawlium does generally outperform ZBrowse, there is no clear winner.

To better understand the nature of the differences, we investigated the two domains on either side of each unique edge. For an edge $d_1 \rightarrow d_2$ that is unique to one of the tools, there are three possibilities: either the other tool also had downloaded domains d_1 and d_2, or it had only one of the two, or it had neither.

Figure 4.4 shows the breakdown of the unique edges for both tools and `UserAgent` strings. Interestingly, more than 70% of the edges that are unique to the tools are due to the edges between domains both tools found in common. This is likely due to differences in how the two tools compute their inclusion graphs. Recall that ZBrowse uses Node.js's



(a) Alexa top-10k sites.

(b) Less popular 10k Alexa sites.

Figure 4.4: When a tool obtains a unique **edge**, how often both tools observe the edge's domains. The tools show only slightly greater agreement for less popular domains.

built-in method for obtaining the DOM tree, while Crawlium builds it from scratch by listening on all events. ZBrowse is less likely to discover edges with both unique domains when compared to Crawlium, owing in part to the fact that Crawlium obtains more domains that ZBrowse does not.

Once again, we conclude that Crawlium and ZBrowse offer complementary data: ZBrowse is more adept at identifying edges in the domains that the two tools share in common, but Crawlium obtains more domains, which allows it to find more edges, as well.

Differences in resources Finally, we turn to the differences in the resources the two tools obtain. Figure 4.1(c) shows that the two tools differ considerably in which resources they return, with the median *Desktop* webpage having 41% of the resources in common (45% for the median *Mobile* page, and 60% for less popular pages). Like with edges, slightly less than 10% of webpages had the same exact resources. This is to be expected based on the previous results; many websites load multiple resources from the same domains, so

when domains differ, resources will, as well.

4.2.3 Recommendations

Based on the above results, we make the following recommendations:

Report what specific events are triggered. Tools and papers that use headless browser APIs like Puppeteer should clearly articulate which events they trigger on.

Compare against other tools. Future tools should compare directly to one another, and report on the differences in the domains, resources, and edges the tools are able to obtain.

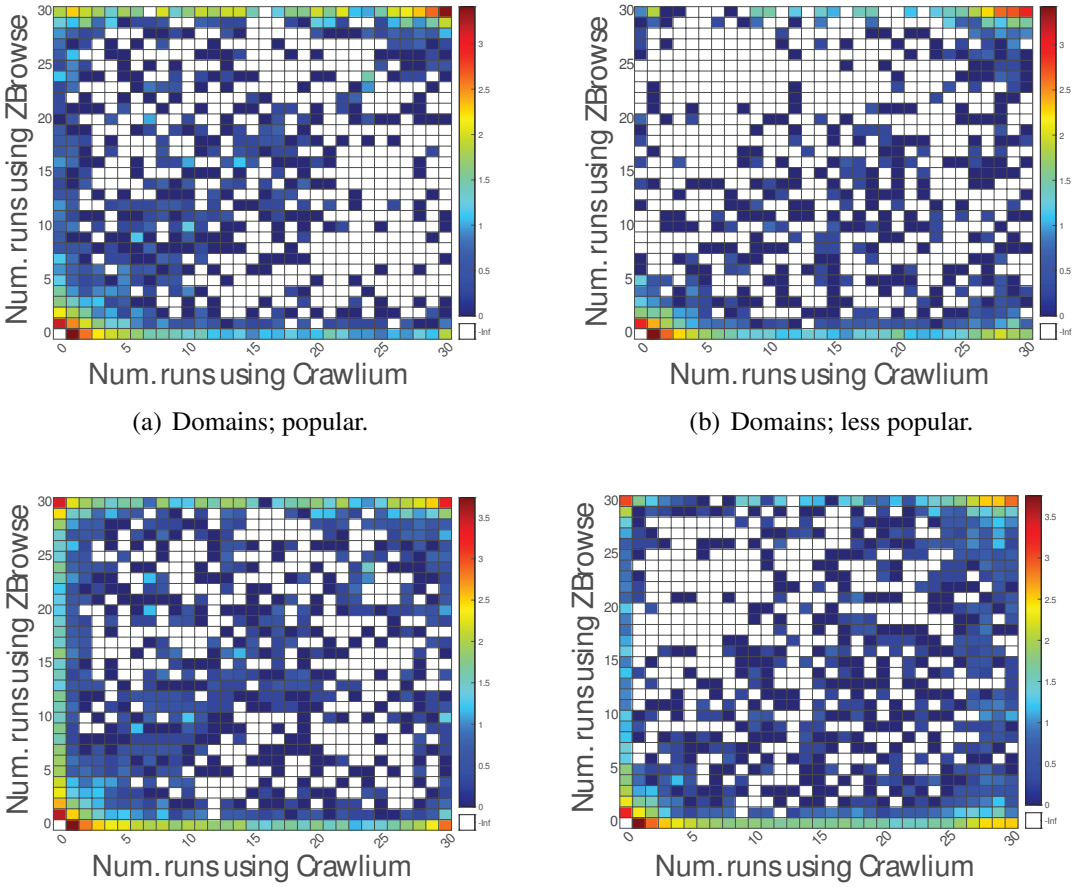
To maximize coverage, use complementary tools concurrently. If the goal is to maximize coverage of a website—that is, to obtain as many domains, edges, or resources as possible—then one should consider using two complementary tools concurrently.

> **Tool**
>
> Web crawling tools obtain data differently, specifically ZBrowse and Crawlium obtain significant data that one tool obtains but the other does not. To maximize coverage, these tools should be used concurrently.

In the remainder of this chapter, we use both Crawlium and ZBrowse concurrently.

4.3 Is Disagreement Caused by Dynamism?

Section 4.2 showed that different tools can result in different inclusion graphs. One tempting explanation for this is that the differences arise from the fact that webpages today are highly dynamic, rather than something inherent to the tools themselves. In this section, we show this not to be the case. Rather, we observe many instances in which one



(a) Domains; popular.

(b) Domains; less popular.

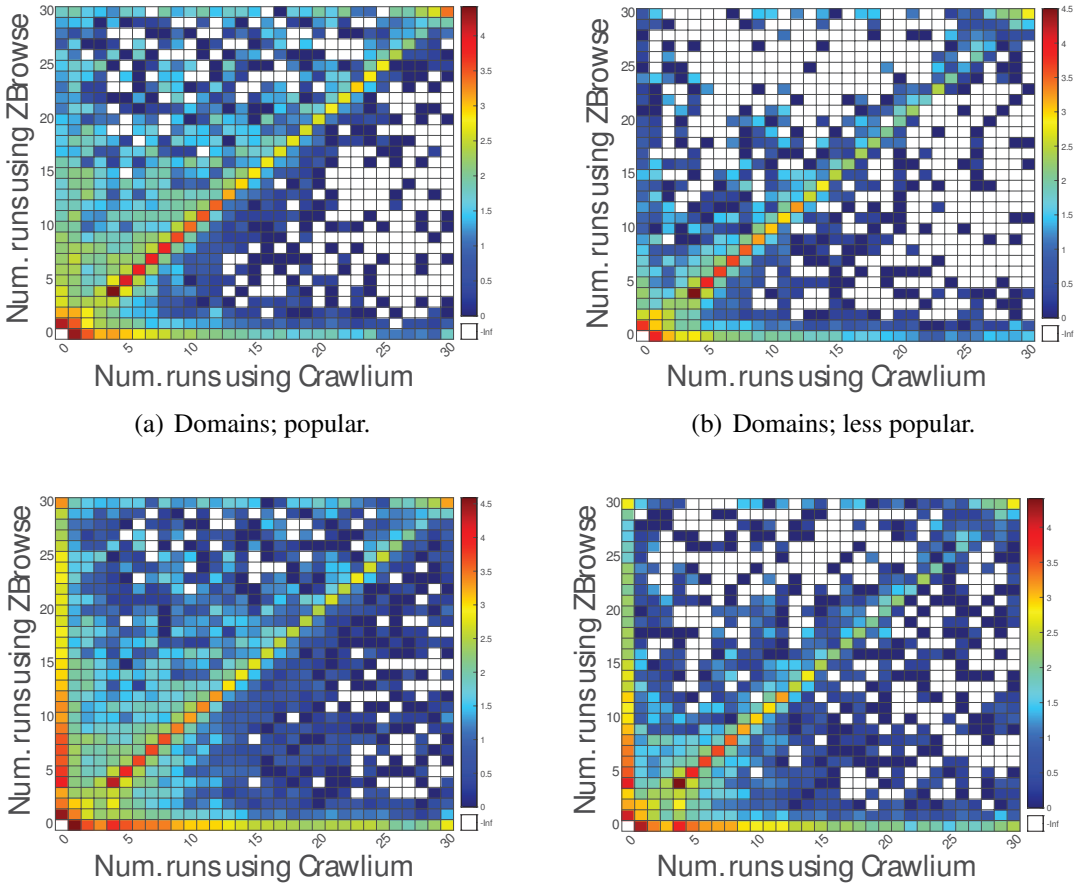
(c) Edges; popular.

(d) Edges; less popular.

Figure 4.5: The number of runs in which different domains and edges are obtained by Crawlium and ZBrowse, limited to the domains that **required all 30 page loads**. (Desktop only shown; Mobile results are very similar.)

tool *consistently* obtains a given domain, edge, or resource that the other tool rarely or *never* gets. For example `https://www.nytimes.com` utilizes all 30 page loads to obtain data. Crawlium obtains `https://stats.g.doubleclick.net` in all 30 page loads, yet ZBrowse does not obtain it once. Similarly, ZBrowse obtains `https://pixel.adsafeprotected.com` in all 30 page loads and Crawlium obtains it in only 23.

To study this more broadly across all of the domains we measured, we compute, for every domain D loaded from every Alexa site A, how many times D appeared in each



(a) Domains; popular.

(b) Domains; less popular.

(c) Edges; popular.

(d) Edges; less popular.

Figure 4.6: The number of runs in which different domains and edges are obtained by Crawlium and ZBrowse, covering all domains **regardless of the number of page loads**. (Desktop only shown; Mobile results are very similar.)

of the page loads of A. We compute this separately for both Crawlium and ZBrowse. To rule out any potential domain name differences caused by load balancing, we perform our analysis strictly over E2LDs: this should result in *greater* agreement between the two tools.

We plot these as heatmaps in Figures 4.5 and 4.6, where each value (x, y) is the number of (Alexa, domain) or (Alexa, edge) pairs that were obtained x times by Crawlium and y times by ZBrowse. In these plots, white cells correspond to zero, and all other cells

are colored from blue (low) to red (high) on a logarithmic scale. Note that the cells at $(0, 0)$ are always white, because we only compare domains that were loaded at least once by at least one of the tools. If the two tools were in complete agreement, then all of the values would be along the $y = x$ diagonal.

Figure 4.5 shows the results for the webpages that required all 30 page loads in our adaptive strategy. We note that there are strong concentrations at the top-right corner (when both tools get almost all of the data all of the time) and the bottom left corner (when one tool got a data item once, and the other tool never did). These alone would correctly account for typical webpage dynamism. However, for domains, there are also strong concentrations along bottom row ($y = 0$; when Crawlium gets a domain that ZBrowse never does) and the top row ($y = 30$, when ZBrowse consistently gets a domain that Crawlium does not). Likewise, for edges, we also see a strong concentration along the left column ($x = 0$, when ZBrowse gets an edge that Crawlium never does).

Figure 4.6 shows the results for all of the webpages, regardless of how many times they needed to reload. Here, we see a stronger concentration along the diagonal $y = x$, but note that the diagonal is effectively the union of the webpages "top right corner" (i.e., when both tools get the data item in *all* of the refreshes).

We provide several additional examples:

- `doubleclick.net` is loaded in all 30 page loads by 6 (and 4) Alexa websites using only Crawlium on Desktop (and Mobile) and on 6 other Alexa websites using only ZBrowse on both Mobile and Desktop.

- `googlesyndication.com` is obtained only by Crawlium on Mobile for 6 Alexa

websites.

- `spotxchange.com` is loaded in a single page load by 187 (and 455) Alexa websites using Crawlium only and by 514 (and 281) other Alexa websites using only Zbrowse on Desktop (and Mobile).

These trends cannot be explained by mere randomness in webpage content, and instead demonstrate that there are systemic differences between the two tools we have studied.

Dynamism

The heatmaps show that tools obtain data differently not merely due to randomness in webpage content, instead there are systemic differences between the two tools.

4.4 How Many Refreshes?

In this section we ask: how often does a page need to be loaded in order to obtain all of its resources and links at a given point in time? Barring dynamic content such as advertising, it would seem as though the answer should be *once*: downloading a webpage a single time *ought* to obtain virtually all of the content. We show this not to be the case.

4.4.1 Methodology

For this part of our study, we download individual websites many more times than in other portions of our study, and as a result we focus here on a smaller set of domains: the Alexa top-1,000 most popular websites. As in Section 4.2, we use the same order

of `https`, `http`, and `www`, and we use both the Desktop and Mobile `UserAgent` strings. Also, we download using both Crawlium and ZBrowse, and union their nodes and edges into a single inclusion graph for each download.

We download each webpage 30 times, in back-to-back succession, restarting the tool and clearing the cache and cookies each time. We chose this number because we felt it was likely much higher than necessary (or at least higher than most researchers would be willing to invest), and we found larger values to have marginal returns for every website in our initial test set. The union of all 30 of these inclusion graphs provides our most complete view of the given webpage that we have.

Our primary analysis in this section consists of computing how many of the consecutive page loads were necessary in order to obtain a given percentage of the domains (fully qualified domain names), edges (directed edges between domains), and resources (complete URL) from the graph of all 30 page loads.

4.4.2 Results

Of the Alexa top-1,000 webpages we crawled, 982 have domains and resources for both Desktop and Mobile; out of which 942 have edges on Mobile, and 930 have edges on Desktop. We report on the 982 domains for which both tools responded.

Table 4.1 shows the number of page loads necessary to obtain *column%* of the domain names from *row%* of the webpages in the Alexa top-1,000. For example, 22 page loads would yield at least 90% of the domains from at least 75% of the webpages. This

		Domains					
		≥1%	≥50%	≥75%	≥90%	≥95%	≥99%
Webpages	≥1%	1	1	1	1	1	1
	≥50%	1	1	1	1	3	7
	≥75%	1	1	11	22	26	29
	≥90%	1	7	18	26	28	30
	≥95%	1	10	20	26	28	30
	≥99%	1	12	21	27	29	30

Table 4.1: Number of page loads necessary to obtain *column%* of **domains** for *row%* of webpages from the Alexa top-1000, using a *Desktop* `UserAgent`.

		Domains					
		≥1%	≥50%	≥75%	≥90%	≥95%	≥99%
Webpages	≥1%	1	1	1	1	1	1
	≥50%	1	1	1	1	2	4
	≥75%	1	1	6	19	24	28
	≥90%	1	5	17	25	28	30
	≥95%	1	8	19	26	28	30
	≥99%	1	13	27	28	29	30

Table 4.2: Number of page loads necessary to obtain *column%* of **domains** for *row%* of webpages from the Alexa top-1000, using a *Mobile* `UserAgent`.

		Edges					
		≥1%	≥50%	≥75%	≥90%	≥95%	≥99%
Webpages	≥1%	1	1	1	1	1	1
	≥50%	1	1	1	3	8	16
	≥75%	1	2	12	22	26	29
	≥90%	1	8	19	26	28	30
	≥95%	1	10	20	26	28	30
	≥99%	1	14	25	28	30	30
		Edges					
		≥1%	≥50%	≥75%	≥90%	≥95%	≥99%
Webpages	≥1%	1	1	1	1	1	1
	≥50%	1	1	1	2	4	11
	≥75%	1	1	7	19	24	29
	≥90%	1	7	18	25	28	30
	≥95%	1	13	22	28	29	30
	≥99%	1	15	24	29	29	30

Table 4.3: Number of page loads necessary to obtain *column%* of **edges** for *row%* of webpages from the Alexa top-1000, using a *Desktop* (top) and *Mobile* (bottom) `UserAgent`.

table corresponds to using a Desktop `UserAgent`; we find very similar results when we use a Mobile one instead, in general requiring slightly fewer page loads.

The results for the number of page loads to obtain a given percentage of *edges* are very similar as seen in Table 4.2. As with domain names, we see that a surprisingly large number (26) is required for obtaining over 90% of the edges from over 90% of the webpages (Table 4.3).

		Resources					
		≥1%	≥50%	≥75%	≥90%	≥95%	≥99%
Webpages	≥1%	1	1	1	1	1	1
	≥50%	1	13	21	27	29	30
	≥75%	1	14	22	27	29	30
	≥90%	1	15	23	27	29	30
	≥95%	1	15	23	28	29	30
	≥99%	1	16	24	28	29	30
		Resources					
		≥1%	≥50%	≥75%	≥90%	≥95%	≥99%
Webpages	≥1%	1	1	1	1	1	1
	≥50%	1	13	22	27	29	30
	≥75%	1	14	22	27	29	30
	≥90%	1	15	23	27	29	30
	≥95%	1	15	23	28	29	30
	≥99%	1	16	24	28	29	30

Table 4.4: Number of page loads necessary to obtain *column%* of **resources** for *row%* of webpages from the Alexa top-1000, using a *Desktop* (top) and *Mobile* (bottom) `UserAgent`.

Finally, Table 4.4 shows how many page loads are necessary to obtain a given percentage of resources for Desktop and Mobile. We find that webpages' resources are by far more dynamic than the corresponding fully-qualified domain names from which they are loaded and the edges that connect them.

4.4.3 Adaptive Reloading

The above results indicate that the number of reloads necessary to obtain a large fraction of webpages' inclusion graphs can vary widely across webpages. Some require just a few reloads to get over 90% of the inclusion graph, while others require dozens of reloads. Picking a single number of reloads risks unnecessary overhead downloading webpages that do not need many, or risks under-sampling the webpages that do.

We next experiment with a simple adaptive reloading heuristic. The idea is to reload a webpage until there have been at least $\delta \geq 1$ consecutive reloads that yield no additional domains or edges beyond what has already been returned from previous loadings of the page. Note that this would require each webpage to be downloaded at least $\delta + 1$ times. For instance, with $\delta = 1$, websites like `wikipedia.org` that return the entire inclusion graph in a single page load would require two page loads.

Strategy	**Total # Loads**	**Avg. % Domains**	**Avg. % Edges**	**Avg. % Resources**
1 load (prior work)	982	76.4	73.0	24.7
Adaptive, $\delta = 1$	9781	95.2	93.4	51.5
Adaptive, $\delta = 3$	12,515	97.6	96.5	59.0
Adaptive, $\delta = 5$	14,357	98.1	97.2	63.8
Adaptive, $\delta = 10$	18,958	98.9	98.3	75.5
30 loads (all)	29,460	100	100	100

Table 4.5: Requisite page loads and amount of content received for different download strategies, when run against the 982 of the Alexa top-1000 websites that responded with a *Desktop* `UserAgent`.

Tables 4.5 and 4.6 present our results comparing a single page load (the standard in prior work) to this adaptive strategy with varying values of δ. for `UserAgent` strings purporting to be a Desktop and Mobile client. In the case of a Mobile client, the percentages of domains, edges, and resources are nearly the same, but Mobile tends to require roughly

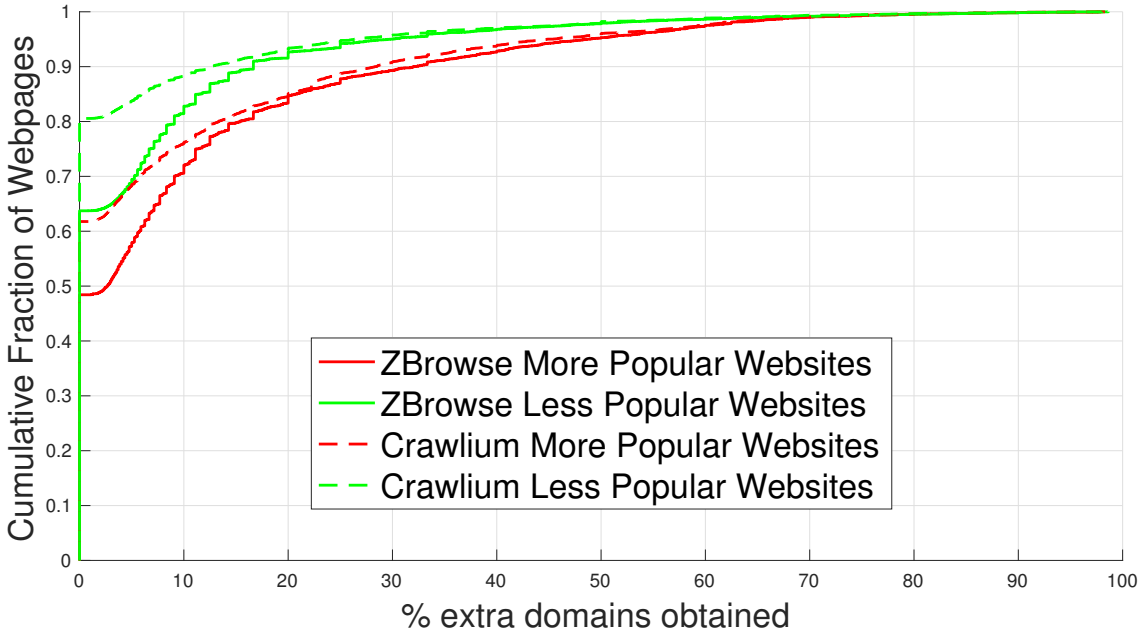
Strategy	Total # Loads	Avg. % Domains	Avg. % Edges	Avg. % Resources
1 load (prior work)	982	79.3	75.9	23.8
Adaptive, $\delta = 1$	8683	95.5	93.9	47.8
Adaptive, $\delta = 3$	11,205	97.5	96.5	54.9
Adaptive. $\delta = 5$	13,243	98.1	97.3	60.3
Adaptive, $\delta = 10$	17,921	98.7	98.3	72.3
30 loads (all)	29.460	100	100	100

Table 4.6: Requisite page loads and amount of content received for different download strategies, when run against the 982 of the Alexa top-1000 websites that responded with a *Mobile* `UserAgent`.

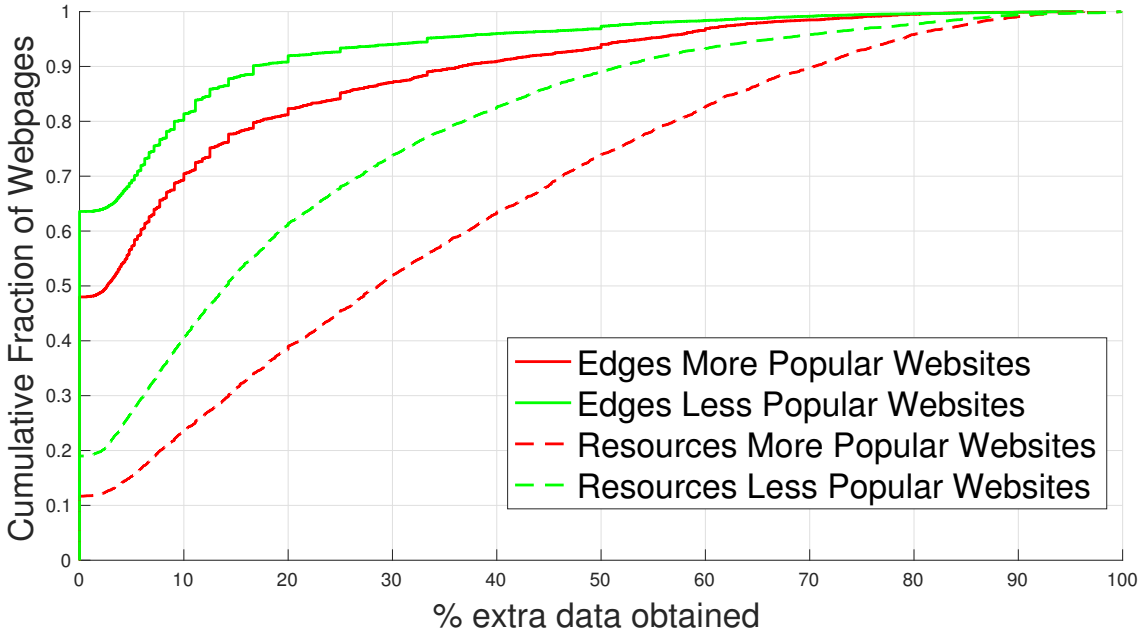
1000 fewer page loads for each of the Adaptive strategies.

These results show that the standard approach of performing a single page load fails to obtain on average 23.6% of the domain names and 27.0% of the edges in webpages' inclusion graphs. Our adaptive strategy is able to obtain significantly more content without having to exhaustively download the full 30 times. For instance, with 42.5% of the maximum number of page loads, the adaptive strategy with $\delta = 3$ misses only 2.4% of the domains and 3.5% of the edges, on average. There are diminishing returns for higher values of δ.

How much data comes after the first page load? Figure 4.7 shows the percent of new data (domains, edges and resources) obtained after the first page load using adaptive strategy with $\delta = 3$ for Alexa top-10k sites and randomly selected 10k less popular sites on desktop (the plots for mobile are very similar). We observe that, compared to less popular websites, more popular websites obtain more new domains, edges and resources in subsequent page loads. Also, as shown in Figure 4.7(a), ZBrowse obtains more new domains in subsequent page loads than Crawlium. Figure 4.7(b) shows the percentage of new edges and resources obtained using ZBrowse (the results were nearly identical for Crawlium).



(a) Domains.



(b) Edges and Resources.

Figure 4.7: Percentage of domains obtained by Crawlium and ZBrowse, and percentage of edges and resources obtained by ZBrowse in subsequent page loads for the Alexa top-10k sites and 10k sites from among Alexa rank 10,001 to 1 million using *Desktop* `UserAgent` string.

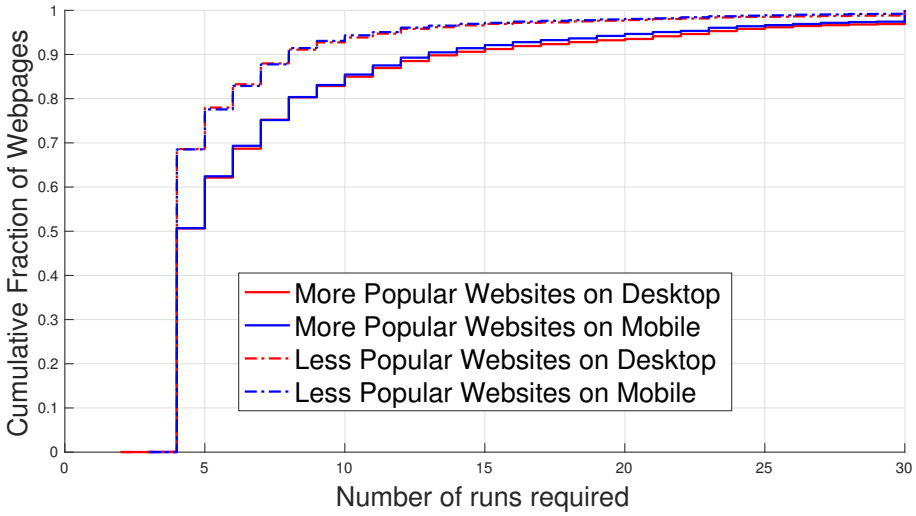


Figure 4.8: Number of page loads for adaptive strategy ($\delta = 3$)

How many page loads were needed in the rest of our study? Figure 4.8 shows the distribution of the number of page loads when using the adaptive strategy with $\delta = 3$ across websites we used in Sections 4.2 and 4.3. The majority of websites required four total downloads (meaning all of their content was obtainable in one); and there is a long tail. Only a small fraction require the maximum of 30 page loads—these are likely websites with so much dynamic content that they can never obtain a full snapshot of their content at any one point in time. We note also that Mobile versions of websites required slightly fewer page loads than their Desktop versions. Interestingly, less popular websites required fewer page loads across both Desktop and Mobile versions.

Collectively, these results shows that there is a wide variance in webpages' dynamism and complexity, and motivates moving away from one-size-fits-all approaches to measuring them.

Our adaptive strategy incurs significantly higher overhead compared to the standard single page load—even with $\delta = 3$, it requires nearly $12\times$ page loads on average. One

possible area for improvement would be to adaptively load not until there are *no* more changes (as we have done), but rather until the changes are below some threshold fraction of all of the content received thus far.

4.4.4 Recommendations

Based on the above results, we make the following recommendations:

Load webpages more than once. Almost all webpages today have too much dynamism to get a significantly complete inclusion graph from a single page load.

Avoid one-size-fits-all solutions. There is simply too much variability across webpages in terms of the number of page loads required to obtain a given percentage of their resources. Prefer reload strategies tailored to specific webpages. Barring any prior knowledge of a webpage, one can apply adaptive techniques like the one we have presented.

Refreshes

Due to webpage dynamism, a single page load obtains only about 75% domains and edges. To increase the amount of data obtained to more than 90%, and owing to the variability across webpages, adaptive number of page loads should be used.

4.5 Conclusion

Downloading accurate and complete inclusion graphs of webpages is an important building block towards understanding myriad phenomena such as advertising [62], malicious inclusions [10, 59, 88], and performance optimizations [79]. Surprisingly, there is

little consistency across various studies on how to crawl the web, and even sophisticated tools lack thorough comparisons to one another.

In this chapter, we have sought to take the first step towards an empirical foundation for crawling webpages. To this end, we compared two state-of-the-art tools, Crawlium [32] and ZBrowse [122], to understand *which* tool to use and *how many* times to reload a page. We found the tools to be complementary, and recommend that both tools' techniques should be used to download webpages. We also found that downloading webpages a single time (as is often the case) misses more than 25% of the domain names from more than 25% of webpages. We recommend an adaptive strategy that trades off overhead (more page loads) for more coverage.

Our results demonstrate several features that have significant impact on the inclusion graph (whether the `UserAgent` string is that of a mobile or desktop device, how many times the page is loaded, which events the tools listen for, and so on). Our hope is that our results will lead to more research exploring these various parameters and reporting on them so that others may evaluate and reproduce more accurately.

Chapter 5: Resource Hints or Resource Waste?

Web developers include *resource hints* in the webpage to have performance benefits by downloading the resources while loading the website. In this chapter, we describe the various ways of incorporating the resource hints and check if ad blockers block any of the resource hints from being downloaded.

To identify the list of resource hints that are included in webpages, we scrape the HTML of the webpages using beautiful soup in selenium. To download the HTML (or source code) of Alexa top-100,000 websites, either selenium or puppeteer can be used. To check if the resource hints are actually being loaded or not, we have to obtain the complete list of resources including JavaScripts that are downloaded when a webpage is accessed. In Chapter 4, we developed a sound methodology to download a complete map of the resource-level topology of a webpage at any given point in time. This same setup can be used to obtain the list of resources, but it has a high overhead since we require only the resources and not the order in which they are loaded (inclusion graph). Hence, we use Puppeteer to reduce the time overhead.

5.1 Overview

Reducing page load times is of critical importance to websites [41, 45]. Many sophisticated optimizations for web performance have been proposed [79, 80, 103].

In this chapter, we consider a very basic form of web optimization: small *resource hints* that web pages can include in the `head` of their HTML to instruct browsers to speculatively perform some part of fetching [50]. For instance, if a webpage knows that it will *eventually* ask the user to download a resource from `example.com`, then it can include a resource hint telling the browser to:

- ***DNS prefetch***: Speculatively perform the DNS lookup for `example.com`
- ***Preconnect***: Initiate the TCP/TLS connection early
- ***Prefetch* or *Preload***: Preemptively download a specific resource from `example.com`
- ***Prerender***: Download and start the execution and rendering of the resource.

HTML resource hints are easy to include, easy to reason about, and, *if used correctly*, have the potential to significantly decrease webpage load times. Sundaresan et al. showed that DNS resolution and TCP connection establishment can constitute 7% and 53% of page load times, respectively [109].

Conversely, if used *incorrectly*, then resource hints can be resource *waste*. If a browser is asked to DNS-prefetch a domain that the website ultimately never connects to, then it is a waste of the website's and browser's bandwidth. In the most extreme case, if the browser is asked to Prefetch, Preload, or Prerender a resource that is never used, it has

the potential to be a significant waste of client resources.

Given the ease of use and high potential of HTML resource hints, it seems reasonable to assume that popular websites—who are notoriously sensitive to latencies [41]—would be making effective (and correct) use of them. Our work shows otherwise.

In this chapter, we measure resource hint usage across the Alexa top-100k most popular websites. We find that a moderate percentage of the most popular websites use these hints, but a shockingly low number use them correctly. For instance, many websites (26.3%) use DNS prefetching, but 7% of those sites never direct the browser to actually load anything from any of the DNS-prefetched sites. Also, for the most popular browser, Chrome, to actually perform DNS prefetching in an HTTPS website, the website must explicitly turn DNS prefetching on, which 93.9% of them do not. We explore such misconfigurations, errors, and oversights across all five HTML resource hints, uncovering myriad ways in which their potential is going untapped and clients' resources are being wasted. We also observe that some resource hints bypass ad blockers, and measure how many domains among those loaded by the Alexa top-100k are being looked-up or connected to that would have otherwise been blocked.

5.2 Methodology

By referencing Alexa Top 1 Million list, we scraped the landing page of the top 100,000 websites in mid-April 2021 [6].

We ran a similar experiment last year (March–April, 2020), but at the time we used Python requests to pull HTML from websites [91]. However, further inspection

reveals that Python requests are insufficient for our study as they do not return JavaScript. Websites may use JavaScript load resources that were retrieved by resource hints. To correct for that, in this chapter, we used Puppeteer driving a headless Chrome browser to obtain both HTML and the resources that are dynamically loaded via JavaScript.

To determine the most suited methodology to list all the resources downloaded by a webpage, we followed techniques based on our tool described in Chapter 4. We conducted experiments for a small set of websites using Crawlium, ZBrowse, OpenWPM and Puppeteer. We observed that although the list of resources differs across different tools, the resources corresponding to the HTML resource hints varies minimally. This implies that a single tool would suffice to check if the resource hints are being downloaded by a webpage. Since the time overhead for Crawlium, ZBrowse and OpenWPM is more than 50% when compared to puppeteer, we use puppeteer to obtain both the list of resource hints on a webpage HTML and the list of resources loaded when the webpage is accessed. For the required number of page loads, we used adaptive $\delta = 1$ strategy (load until no new data is obtained in last page load) up to a maximum of 10 page loads to be able to obtain the complete list of resource hints that are used/unused by a webpage.

Some websites do not implement HTTPS. We visit pages in the corresponding order of URL prefixes: “https://”, “https://www.”, “http://”, “http://www.”. With this method, we are able to first extract pages from HTTPS domains, if possible.

DNS prefetching is bound by certain rules that websites must follow. DNS prefetch must be explicitly turned on in order to work for HTTPS websites. We search for the meta tag using the BeautifulSoup Python library to discover if websites are properly turning on prefetching [18]. BeautifulSoup facilitates data extraction from HTML files. To scan for

DNS prefetch control within HTTP headers, we simply use Python responses and retrieve the headers. JavaScript is not required to view the HTTP headers.

In order to DNS prefetch, preconnect, preload, prerender, or prefetch, websites must create a link tag with the targeted domain in the href field and their desired resource hint. An example for the syntax of dns-prefetch is `<link rel="dns-prefetch" href="example.com">`. Again, this data can be extracted using BeautifulSoup.

A key aspect of our study is to determine whether websites are actually using the links that they mention in resource hints. In order to do so, we count the occurrences of the hinted links to see if they are mentioned at least once. The links tags must first be removed before parsing the page source for later references as this may lead to false positives. In addition, the link must be properly escaped to allow for safe filtering within the HTML body.

5.3 Resource Use and Misuse

This section outlines results from measuring the deployment of resource hints. We determine overall usage for each resource hint and find corresponding domains that influence results. We also review how well websites are properly using DNS prefetching with the requirement of enabling it for HTTPS.

5.3.1 Resource Hint Invocations

We begin by investigating how often websites invoke each of the five resource hints (DNS Prefetch, Preconnect, Preload, Prefetch, and Prerender). Table 5.1 shows the col-

Resource hint	# Websites
DNS Prefetch	26,341
Preconnect	14,866
Preload	28,744
Prerender	186
Prefetch	10,186

Table 5.1: Number of websites from the Alexa top-100k that invoke each given resource hint at least once.

lective calls for each resource hint. We observe that Preload is the most common, followed closely by DNS Prefetch. Prerender is rarely used. We hypothesized that prerendering may primarily be utilized in mobile browsing. However, prior work suggests that resource hint usage between desktop and mobile browsers is similar [55]. We validated this by re-scraping websites with Selenium's mobile options, and confirmed that prerendering remains rare even when connecting to sites with mobile user-agents.

Figure 5.1 shows the resulting fraction for each resource hint. This figure (as well as Figures 5.2 through 5.6) orders the x-axis by the Alexa ranking, binned into buckets of size 1,000. Popular websites are generally more likely to call resource hints with the exception of DNS prefetch. Most resource hints are more likely to be used by more popular websites. Unlike all other resource hints, DNS prefetch is more likely to be included in less popular websites than it is in more popular sites.

5.3.2 Resource Hint Usage

Even if a website includes a resource hint, that does not mean that the rest of the page actually *used* that resource. Websites may include resource hints, but not reference them later, which will result in unnecessary calls from the browser and wasted resources.

Table 5.2 presents the overall number of links included in resource hints across

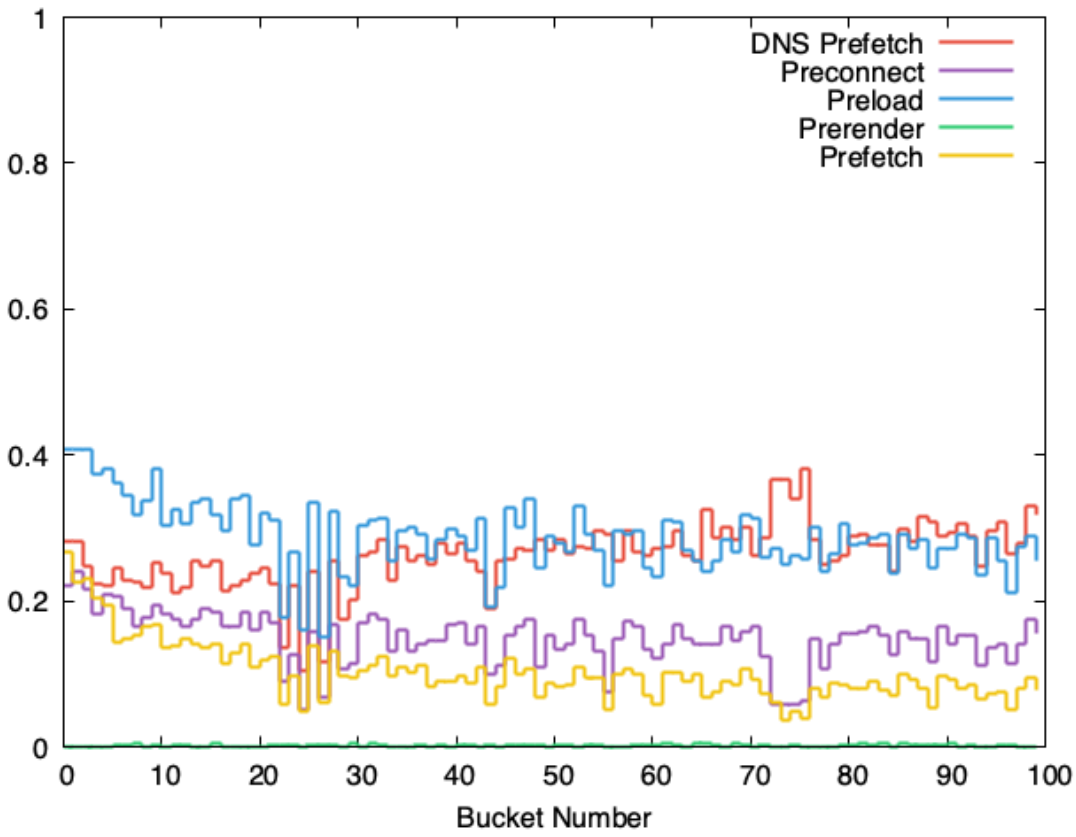


Figure 5.1: Fraction of Alexa top-100k sites that invoke each given resource hint (x-axis ordered by Alexa ranking, binned into buckets of size 1,000).

Resource Hint	**# Links**	**# Unused HTML only**	**# Unused HTML and JS**
DNS Prefetch	122,721	51,525 (42.0%)	41,038 (33.4%)
Preconnect	59,203	29,942 (50.6%)	16,534 (27.9%)
Prefetch	47,957	35,202 (73.4%)	17,311 (36.1%)
Prerender	339	62 (18.3%)	43 (12.7%)
Preload	126,255	34,496 (27.3%)	17,632 (14.0%)

Table 5.2: Aggregated number of links per resource hint, and how many go unused.

the entire Alexa top-100k, and how many of them go unused. This table also shows the importance of using a JavaScript-enabled browser when performing these experiments: limiting only to the resources used in the HTML would indicate a significantly higher rate of unused links. For the remainder of the chapter, we strictly refer to the results from

loading both HTML and JavaScript. That said, even accounting for resources loaded via JavaScript, a significant fraction of resources go unused. We find a surprising fraction of unused links; 33.4% of DNS prefetched links are never used (and, as we will see, even fewer are turned on). This is our first evidence of excessive misuse of resource hints.

To further investigate resource hint usage on a per-website basis, we consider three categories of usage: Full Usage (when *all* hinted resources are loaded at least once from within the webpage), Some Usage (when some are loaded), or No Usage (when none of the hinted resources are loaded). The sum of the three resulting fractions is 1.

The results for usage are presented in Figures 5.2 through 5.6. We analyze each of these in turn next. We later return to them in Section 5.3.3 when we observe the usage in relation to the specific second-level domains that are being referenced by resource hints.

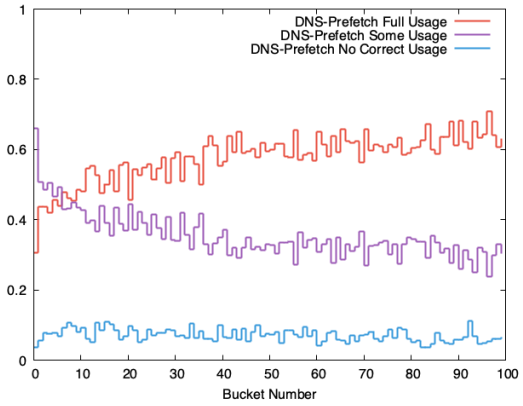


Figure 5.2: DNS Prefetch Usage (bucket size 1,000).

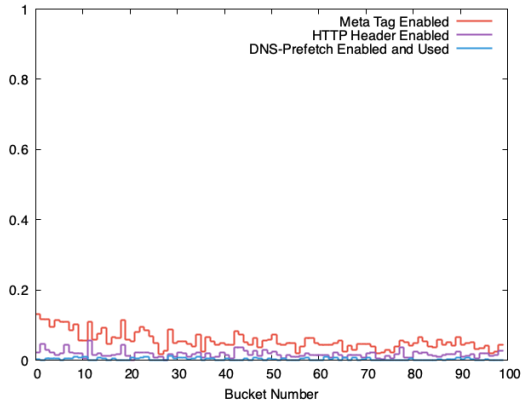


Figure 5.3: DNS Prefetch Usage ON.

DNS Prefetch As website rank decreases, there are more websites that use DNS prefetch fully as shown in Figure 5.2. Full-usage and some-usage display behavior that closely resembles an inverse function. This behavior may be attributed to more popular websites DNS prefetching more links in general, which may allow for greater room of error. We

find that as website rank decreases, the average number of links a single website DNS prefetches also decreases.

Recall that browsers require websites to explicitly turn on DNS prefetching for HTTPS. To track proper usage of DNS prefetch, we count how many website contains the meta tag to enable DNS prefetching, and how many enable DNS prefetching within HTTP headers. In order to fully properly DNS prefetch, the tag must be turned on and all of the referenced links should be loaded at least once to avoid wasted resources.

We find that websites barely turn DNS prefetching on and are even less likely to properly both enable DNS prefetching and use all referenced DNS prefetched links. More popular websites tend to marginally turn on DNS prefetching more frequently. Overall, these results indicate that a majority of websites are not properly using DNS prefetching and are in turn wasting resources. Tallying up the buckets, we find that only 1606 of the 26,341 (6.1%) of websites that DNS prefetch actually turn on DNS prefetching. 1431 turn on DNS prefetching using meta tags, while 175 turn it on in the HTTP headers. Without enabling DNS prefetching, users connecting to HTTPS sites will not even obtain the benefits of DNS prefetching. Even fewer websites turn on DNS prefetching and use all DNS prefetched links. Only 443 of 26,341 (1.7%) of websites meet both conditions.

Preconnect As shown in Figure 5.4, Preconnect has similar deployment success as DNS-prefetch: surprisingly, less-popular sites employ it more effectively.

Prefetch Figure 5.5 shows that Prefetch is used fully in over 80% of websites of all popularity in our experiments. Interestingly, the majority of Prefetch hints that go used are a result of resources loaded via JavaScript; concentrating only on HTML misses over

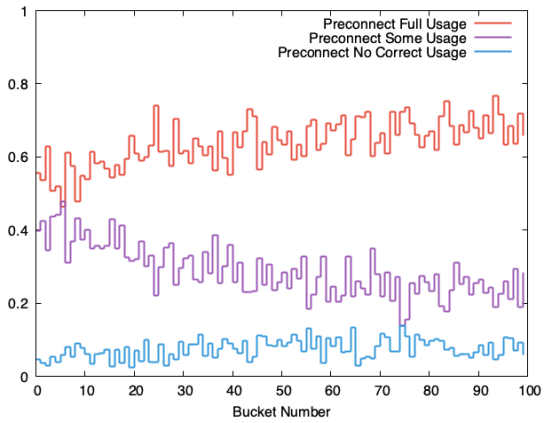


Figure 5.4: Preconnect Usage.

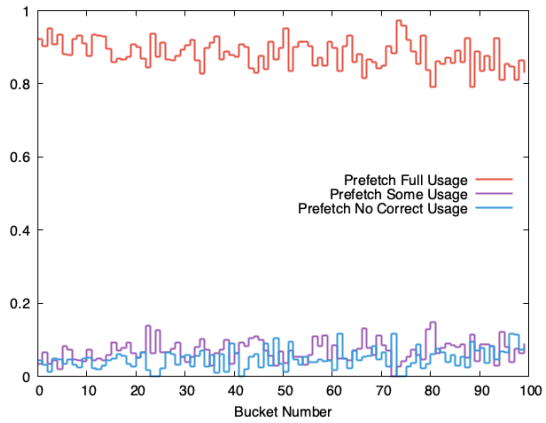


Figure 5.5: Prefetch Usage.

half of the resources. Recall that Prefetch states that websites should fetch content for *future* navigation; the fact that we are only crawling websites' landing pages may not be representative of non-landing pages.

Prerender Prerender has insufficient data to analyze. With only 0.19% of websites using prerender, our results do not display any significant findings. Prerender is very resource heavy, which may be a repelling factor for websites in deciding to use it. Incorrect usage of prerender will be more detrimental than beneficial to load speeds.

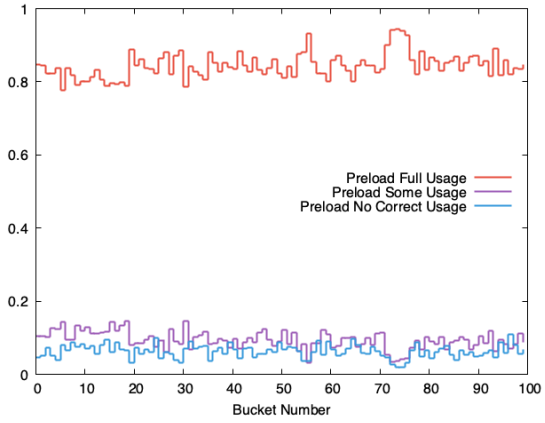


Figure 5.6: Preload Usage.

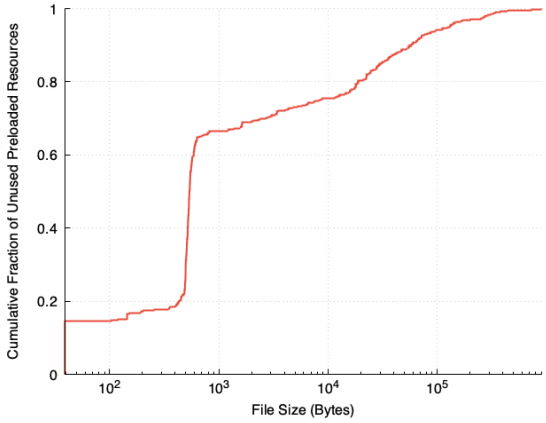


Figure 5.7: CDF of File Sizes of Unused Preloaded Links

Preload Figure 5.6 shows that websites are effective at fully using preloaded links;

across the Alexa top-100k most popular websites, more than 80% fully use the resources they preload. However, it is not perfect, and unfortunately preload has some of the most severe ramifications of being incorrect. Preload actually loads the resource, incurring a DNS lookup, TCP connection, and data transfer. To not actually use a preloaded resource is not only a waste of client resources, but it can harm download speeds by consuming bandwidth. To understand the costs of unused preloads, we plot in Figure 5.7 the CDF of resources that are called for preloading, but in the end unused. We find that the median is 539 bytes with a minimum of 39 bytes and maximum of 910 *KB*.

5.3.3 Resource Hint Links

We also analyzed what second-level domains are being called by resource hints and which ones are actually being used. In Figure 5.1, DNS prefetch was the outlier that was more likely to be used by less popular websites. We discovered that less popular websites are more likely to use s.w.org, a link that redirects to wordpress's main website wordpress.org, and fonts.googleapis.com, a website for free font files. The domain s.w.org accounts for 14,024 of all DNS prefetch calls and fonts.googleapis.com accounts for 9,689. These two websites combined consist of 23,713 of 122,721 (19.32%) of DNS prefetched links and may be a direct factor to the increase in DNS prefetch calls as website popularity decreases.

Preconnect's most referenced links are to several Google service (including fonts.gstatic.com, www.google-analytics.com, and www.googletagmanager.com). Many of these are also the most *unused* Preconnect links, but less popular Preconnect domains are also among

the most unused, including avatars.mds.yandex.net and ads.adfox.ru. However, these alone do not account for the fact that Preconnect's fully-correct usage increases as website popularity decreases. Rather, this appears to be due to the fact that less popular sites simply include fewer Preconnect hints—the Alexa top-10k collectively include 10,483, while the Alexa 90k–100k include only 5743—suggesting that they leave less room for error.

Preload displays some of the most consistent correct usage. This may be a result of Chrome warning developers of incorrect Preload usage. Resources that are preloaded, but not used within 3 seconds will trigger a warning in the Chrome Developer Tools console [44]. Another contributing factor for high full preload usage may be associated with adservice.google.com. 23,869 of the 126,255 (18.9%) preloaded links are for Google ad services tailored for their websites.

Prefetch's most common domain is tpc.googlesyndication.com, which is used to store Google ads that are served to users. It is prefetched 6727 times—66.0% of all prefetches in our dataset—and each time it is referenced, it is also subsequently used. As a result, it extensively contributes to the pervasively well-deployed Prefetch.

Collectively, these results indicate that there are multiple correlating factors towards the correct and incorrect application of resource hints. Common tools like Wordpress and ad services would serve their users well—and significantly decrease unnecessary overheads—by fixing their use of resource hints.

Resource Hint Usage

A moderate fraction of Alexa top-100k websites use resource hints and they're more common among more popular websites, except DNS-prefetch. However, many of the hinted resources are never actually loaded.

5.4 Circumventing Ad Blockers

In this section, we investigate whether websites could use resource hints for abuse. We demonstrate that, surprisingly, *ad blockers do not block all actions on resource hints' URLs* that they do block when the URLs are outside of resource hints. Also, we report on which URLs are being included in resource hints that are not being blocked by ad blockers (but otherwise would have been).

5.4.1 Ad Blockers and Resource Hints

In order to see how the average user would see their ad blocker interact with resource hints, we investigate the most popular ad blockers in Chrome and Edge. For Chrome it is uBlock Origin and Adblock Plus, and for Edge it is Adguard and Adblock. To test them, we create a website that includes two links with the resource hint we are testing. One link is to blocked website, and the other is to a non-blocked website. While running a Wireshark capture, we load the site and observe whether any packets showed a DNS request or a connect to those sites. We also look at the ad blockers' logs, and Chrome's Network tab in their developer tools.

We find that all of these ad blockers perform the same in regards to resource hints: The links in preload, prerender, or prefetch tags are all filtered normally. However, links

in DNS-prefetch or Preconnect tags are not noticed by the ad blockers at all. Those links do not appear in Chrome's list of resources on the Network tab of its developer tools, nor do they appear in any ad blocker logs.

While it is a good sign that ad blockers can filter preload and Prerender—the tags that load the most data—the fact that they do nothing about Preconnect or DNS-Prefetch is concerning. In both cases, the client's IP address is revealed to the destination (or its name server), making it possible to track users. One could even imagine encoding more fine-grained tracking information as subdomains (e.g., `<hash of referrer>.example.com`). To evaluate whether tracking may be happening, we next look at which specific URLs are being loaded, and whether they would otherwise have gotten blocked by ad blockers.

5.4.2 URLs that Bypass Blocking

The most commonly accessed domain is adservice.google.com, being requested over 21,000 times. The next most popular domain is googletagmanager.com, with only about 100 requests. While it is not surprising that ads are a common use for these resource hints, the fact that they are not blocked by ad blockers makes this more concerning.

Resource Hint	**Tested**	**In Block-list**
All	162,610	32,557 (20.02%)
DNS-Prefetch	19,437	4,529 (23.30%)
Preconnect	12,656	3,454 (27.29%)

Table 5.3: Number of resources that *would have* been blocked by an ad blocker.

Taking a closer look at DNS-prefetch and Preconnect, we want to analyze what kinds of links these tags were being used for. With a list of all the domains accessed by

each type of resource hint in the Alexa top-100k, we tested each domains against uBlock Origin, one of the more popular and rigorous ad blockers. The results of these tests are shown Table 5.3. We tested 162,610 domains accessed across all resource hints. This was then repeated with only the domains accessed with DNS-prefetch, and again with only the domains accessed with Preconnect.

Most of the total collection of links are from the use of the preload tag, but since that resource hint can be filtered by ad block, we are more concerned about DNS-prefetch and Preconnect. DNS-prefetch is used to access ad blocked sites about 23.3% of the time. Preconnect, while it is used significantly less than DNS-prefetch, accesses blocked sites 27.29%. Both of these tags access blocked sites at an overall higher rate than the other resource hints. While this does not prove that either resource is being used with ill intent, this high of a figure when for the tags that are not filtered by ad blockers is cause for concern.

Ad Blockers

DNS Prefetch and Preconnect are not blocked by ad blockers while the others are blocked. This results in more than 20% URLs being bypassed by current ad blockers.

5.5 Limitations

We only scraped and analyzed the landing pages of the Alexa top-100k, which may not be representative of less popular sites' use of resource hints. That said, the impact of wasted resource hints is amplified with these most popular sites.

In addition, we retrieve each domain's landing page. Called resource hints may be

cached and used later in future page navigation. As a result, our findings for resource hints are perhaps best seen as a lower bound on full usage and an upper bound on some and no correct usage.

5.6 Conclusion

A majority of websites do not utilize any sort of resource hint, meaning many website hosts are missing out on opportunities to improve page load speeds with the use of resource hints. In addition, even fewer websites use resource hints to their full potential. Resource hints will negatively impact performance if not used as resources will be fetched, but in the end wasted. An even smaller subset of these actually enable DNS prefetching for HTTPS. These websites wish to have the performance benefits of DNS prefetching, but have not implemented it properly. Enabling DNS prefetching is very simple and can be done by adding one line of HTML. However, we have found that only 5% of websites that DNS prefetch actually turn it on. Looking at how the websites using resource hints are actually using them, they are often used for advertisement, and it is possible that they are being used to circumvent ad blockers. While most resource hints are filtered by ad blockers, both DNS-prefetch and preconnect tags are not. An increased understanding of the benefits and dangers of resource hints is important for both ad blocking companies and for webpage creators.

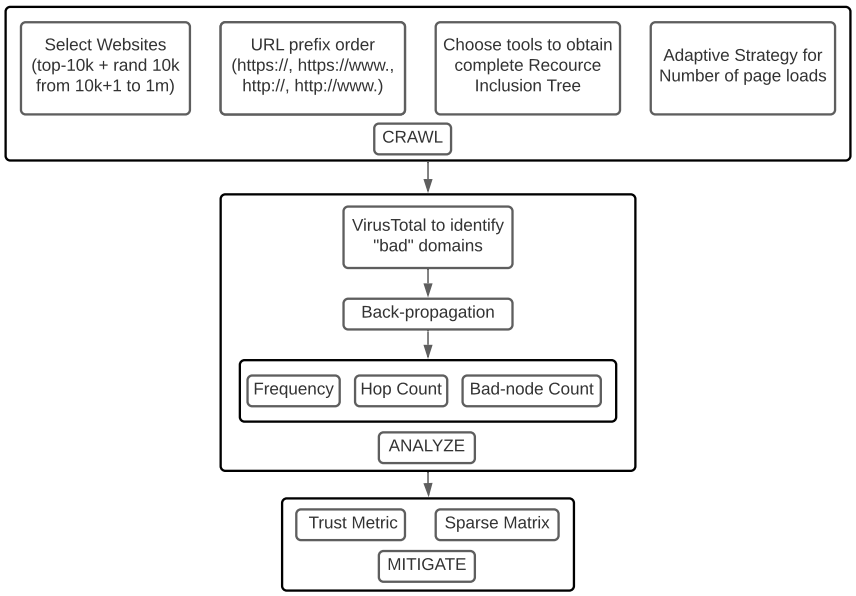
To facilitate future work in this space, we will be making all of our code and data publicly available.

Chapter 6: Measurement Study of the Malicious Web Topology

In the second part of my thesis, I focus on indirect risk—the risk that arises with the third-party resource potentially acting in an untrustworthy manner, even if it were invoked correctly. I study indirect risks by analyzing the topology of the popular Web. Users who visit benign, popular websites are unfortunately bombarded with malicious popups, malware-loading sites, and phishing sites. The benign website does not include the malicious content *directly*, but rather *indirectly*; it includes third-party resources which further include other third-party resources (and so on). At some point along this chain of inclusions, there is a breakdown of trust. At a high level, I seek to understand where this happens and how it can be mitigated.

In this chapter, I propose to study the topology of the Web in the context of trust relationships between the benign website and the malicious redirections. The first step was to obtain a complete map of a website's resource-level topology, which is surprisingly non-trivial. In Chapter 4, we empirically evaluate the methodology of downloading a webpage. Once the malicious web topology is obtained, we propose various metrics that can be used to analyze the sources of malicious activity and finally suggest possible ways that developers can use to mitigate such malicious activity. Figure 6.1 provides a hierarchical picture of the steps we follow to study the Web.

Figure 6.1: Block Diagram representing the Experimental Methodology



6.1 Overview

In recent times, online or web advertising has become an easy way to distribute web-based malware. Using web advertising to spread malware is called malvertising. Malvertising can be analyzed either by static analysis of the advertisement's source HTML code or initial HTTP response or by behavioral analysis of the advertisements. Drive-by downloads, click-jacking, third party advertisements and applications, content delivery networks, hidden iframes and malicious banners on webpages and website redirection are ways in which malvertising is seen to occur. All or most of these require users intervention to install malware. Ad injection is a technique in which users have ads imposed on them in addition to, or different from those that websites originally sent them which can be malicious sometimes without the users' knowledge.

Some ads can be introduced specifically to retrieve personal information. User data

is collected and profiled by advertising networks in order to serve personalized ads. This way of advertising is called targeted advertising, where ad networks display the ads based on users interests. Targeted advertising does improve users' online shopping experiences, but it sometimes has serious consequences, especially leakage of user sensitive information. A lot of recent work shows that in some cases, ads may be biased, showing better offers like job ads to specific set of users which could be based on personal characteristics such as gender. Browsing history is another key information that is used in targeted advertising. Thus, it is important for users to observe the websites that are accessed and loaded.

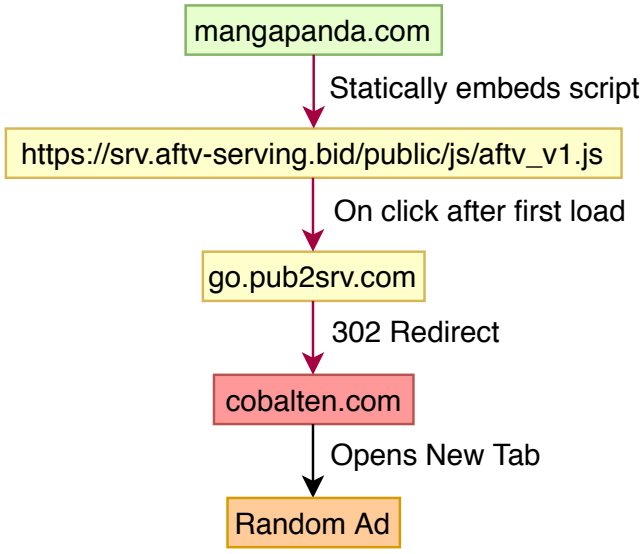
This chapter is driven by a basic question that has confounded us not only as researchers but as users: Why is it that when users visit a benign, popular website, they are often bombarded with pop-ups, malicious websites, and phishing attacks? Is it at the behest of the seemingly benign website, or is it due to some third-party provider hosting a resource downstream from the original site? *At what point is the trust the user has in the website violated?*

Much of the answer to this question rests in the fact that the web is a complex interconnection of resources, hyperlinks, and dynamically generated code. When a user visits a webpage, many first-party and third-party resources are loaded in the background, some of which are related to the visible content on the webpage, others of which are third-party services and advertisements. For instance, when a website is loaded, besides just the URL of the website, there are many other URLs, including resources corresponding to images, fonts, cookies and ads, for example *doubleclick.com*, *google-analytics.com*. It would be natural for a user to assume that when they visit a website, then all of the resources are

ultimately delivered with that site's permission. However, many resources can lead to suspicious or downright malicious content. Most video streaming and anime sites among top ranked Alexa websites load malware or phishing sites, sometimes preventing from accessing content on the webpage.

Consider the example of malicious activity on *mangapanda.com* in figure 6.2. When this website is loaded on any browser, we notice unexpected behaviour. A click on the webpage, not necessarily the portions including ads or other links, opens up a new tab with a redirect to random advertisements. A closer look at the code reveals that *mangapanda.com* embeds `https://srv.aftv-serving.bid/public/js/aftv_v1.js` in the source code, which loads *go.pub2srv.com* based on a particular variable, which in turn redirects to *cobalten.com*. *cobalten.com* adware is what opens up inappropriate, malicious or phishing domains that would possibly collect user information.

Figure 6.2: One of the paths when loading *mangapanda.com*



The central goal of this chapter is to better understand why these nefarious behaviors are happening on otherwise benign websites. We approach this problem as that of

trust inference on a graph. To this end, we conduct a simple but systematic and detailed analysis of the web topology by obtaining a graphical structure of the websites that are loaded one after the other when 20,000 different websites among the Alexa ranking last are accessed both on a desktop and a mobile. The graph we consider models individual webpage resources as nodes; if resource r ever causes resource s to be loaded, then there is a directed edge from r to s. We build up such a graph by downloading the Alexa top-10k most popular websites, and we use VirusTotal (VT) [2] to identify which nodes are malicious. Through a series of graph-based metrics and analyses, we investigate how trust in third-party resources breaks down, who the most common culprits are, and what we can do as a community to more readily identify those who are responsible for introducing malicious content to otherwise benign corners of the web.

This problem is made all the more challenging by the fact that, as we will show, sometimes, a website when opened on a mobile leads to a phishing site whereas it does not show such behavior when opened on a desktop. Such behavior is seen for example in the case of `https://gfycat.com/blackimpolitekawala`. To understand such behavior, we compare the desktop and mobile versions of a website. Key issues of concern are whether analogous functionality of security in desktop computers can be found, and if so, whether it is offered in a manner that parallels the desktop experience (i.e. to ensure understanding and usability).

Our results show that a large fraction of popular websites indirectly load malicious content, and that the web between good and malice can be large and complicated. Surprisingly, the results differ considerably for websites loaded with desktop versus mobile browsers; the mobile web has more malicious content, and a larger, more interconnected

network of third parties connecting to it. Unfortunately, the blocklists of today's ad blockers fail to capture many of the malicious content on either desktop or mobile platforms. We introduce a technique for analyzing the graph of third-party inclusions to identify which intermediate nodes can be added to blocklists, and show that it more effectively blocks bad content.

We summarize our contributions as follows:

- We find that there is a different malicious landscape between desktop and mobile settings.
- We provide an analysis of trust degradation within the internal resource topology of popular websites by using basic and effective graph minimization techniques and graph metrics to better understand the malicious topology of the web.
- We evaluate the effectiveness of ad blockers in blocking malicious domains, and introduce a technique for identifying domains to include in blocklists.
- We will be making our code and data publicly available.

6.2 Experimental Methodology

To identify the kind of behaviour seen in the case of *mangapanda.com* (Figure 6.2), we should first identify all the resources loaded when accessing a website. Manually digging into the resource topology is not feasible as we want to study a large number of domains.

For the large scale study on the web topology in the context of unknown malicious

behavior observed even when a trustworthy site is accessed, we use websites from the Alexa ranking list. This is a list of websites in the order of their popularity and is the most suited list of trustworthy websites that users visit. For our study, we consider a total of 20,000 websites. We select the Alexa top-10,000 websites (to capture the websites that most affect users) and a set of 10,000 websites chosen at random from Alexa ranking between 10,001 and 1 million (to capture an unbiased sample of more standard websites).

6.2.1 Data Collection

For each website, we collect the data following the experimental methodology described in Chapter 4. We collect the web crawl data using both Crawlium and ZBrowse. For the crawl data of a website, a single page load corresponds to the union of Crawlium and ZBrowse. We use the adaptive $\delta = 3$ strategy for the number of page loads, i.e., load until no new data is obtained by the union of Crawlium and ZBrowse in the last 3 page loads.

To evaluate whether the data differs for desktop versus mobile browsing, we run two separate trials with different `UserAgent` strings: one[1] (*Desktop*) that purports to be running Chrome in Mac OS X, and another[2] (*Mobile*) that claims to be Chrome on iOS. To avoid website changes over time, we obtain the data for desktop and mobile `UserAgent` strings simultaneously.

Unfortunately, Crawlium and ZBrowse both sometimes fail to download webpages because of either exceeding the timeout or the webpage not loading manually on any

[1] `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.149 Safari/537.36`

[2] `Mozilla/5.0 (iPhone; CPU iPhone OS 13_3 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) CriOS/69.0.3497.105 Mobile/15E148 Safari/605.1`

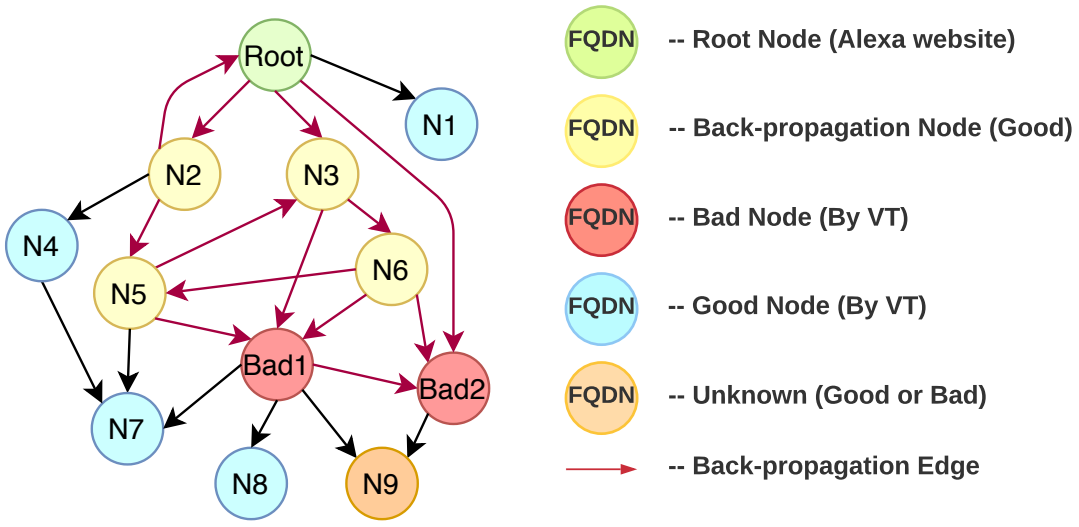
browser. To permit a direct comparison, we only compare the 9,781 pages among the Alexa top-10,000 sites for which either tools were successful in downloading them, i.e. the union of Crawlium and ZBrowse obtained crawl data. We accordingly downsampled from the less-popular webpages to 9,781, as well.

6.2.2 Conceptual Graph

In total, we crawled 19,562 Alexa-ranked websites; we refer to these collectively as the *Alexa websites*, to distinguish them from the third-party sites they load. We then construct a complete inclusion graph from the downloaded resources, the corresponding fully-qualified domain names (FQDNs), and the order in which the domains are loaded. In the graph, the nodes represent the fully-qualified domains and there's an edge from domain u to domain v if u directly loaded v, either by invoking the resource (e.g., `<img src="example.com/img.png">`) or by redirecting (e.g., an HTML 301). Figure 6.3 represents a conceptual graph enumerating the types of nodes and edges in the graph topology of a general Alexa website.

In the graph for an Alexa website which we call an *Alexa graph*, the root node is the Alexa website that is assumed to be benign for users to access. All the other nodes correspond to the domains that the Alexa website loads directly/indirectly. In our study the most significant nodes would be the "bad" nodes (Bad1 and Bad2 in Figure 6.3). Once the bad nodes have been identified (described next, in Section 6.2.3), the set of nodes that we focus on are the nodes labeled as back-propagation nodes in Figure 6.3. An edge is a *back-propagation edge* if it is on a path from the (good) root node to any of the bad nodes

Figure 6.3: Conceptual Graph Template showing the types of nodes and edges in a typical graph



(a) Template graph

(b) Types of Nodes

and the nodes on any of these paths are *back-propagation nodes*. These are important because they capture the domains that are involved—directly or indirectly—in causing bad content to be loaded onto good websites. Additionally, the graphs for some of the Alexa websites (*Alexa graphs*) are very large and most of the nodes are not connected to the bad nodes either directly or indirectly like in Figure 6.4. Thus, it is another advantage for minimizing the Alexa graph to the back-propagation graph.

Figure 6.4 shows the graph for *tribunnews.com* which contains 96 nodes and 138 edges. The node represented by green color is the root, *tribunnews.com*, the nodes represented by red color are the bad nodes flagged by VirusTotal (`https://mc.webvisor.org` and `https://search.spotxchange.com`) and the blue nodes are the intermediate nodes that are loaded by *tribunnews.com*. By just accessing the website, we do not observe any malicious behaviour, but the graph shows that ultimately two malicious domains are ac-

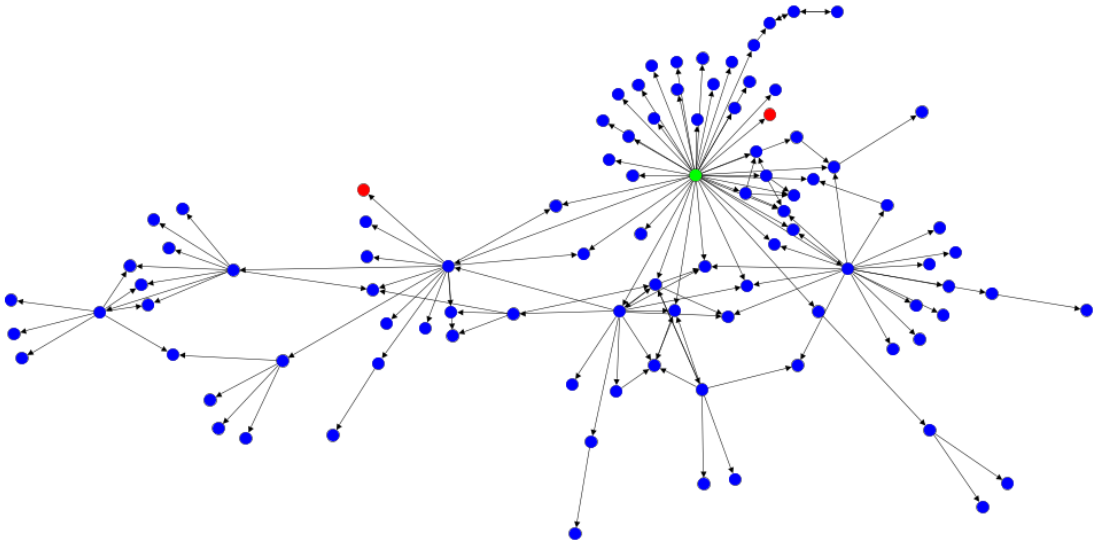


Figure 6.4: Example: Alexa graph for *tribunnews.com* highlighting the root and bad nodes cessed. One of them is directly loaded by the Alexa website, whereas the other domain is loaded indirectly.

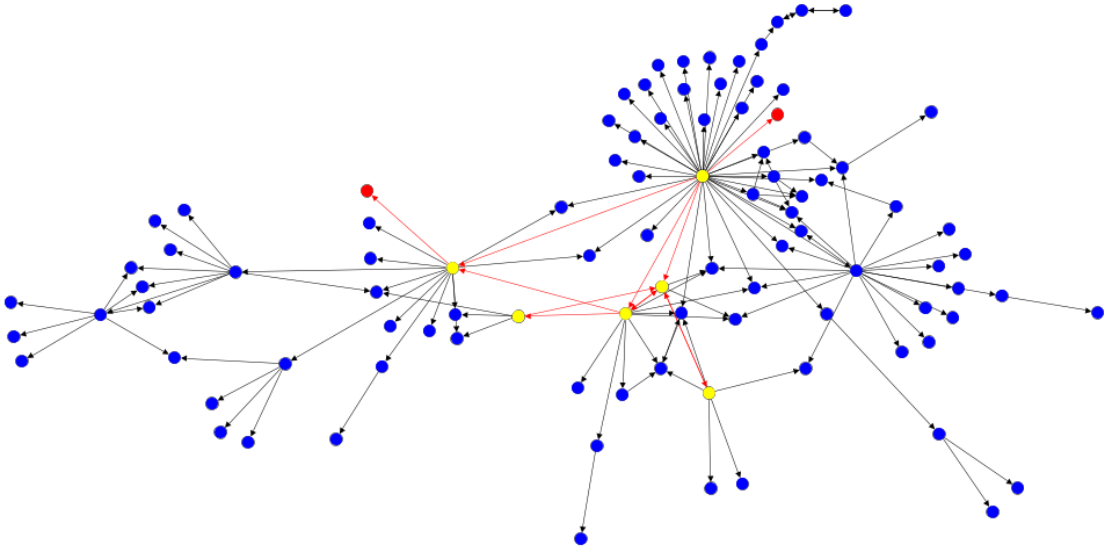


Figure 6.5: Alexa graph for *tribunnews.com* highlighting the nodes and edges in the back-propagation graph

When we perform back-propagation on the Alexa graph for tribunnews.com (Figure 6.4), the final step in back-propagation is seen in Figure 6.5 highlighting the back-

propagation nodes and edges. The back-propagation graph for *tribunnews.com* is shown in Figure 6.6.

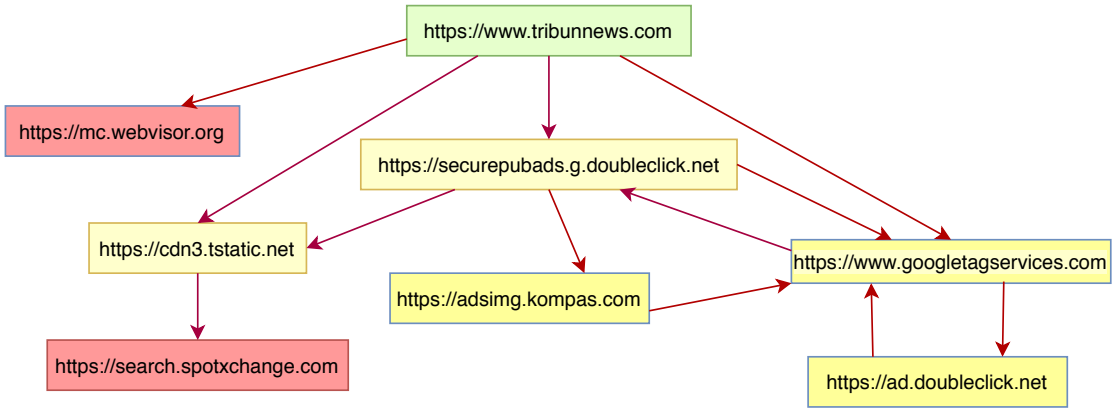


Figure 6.6: Example of the back-propagation graph for *tribunnews.com*

Alexa graphs—while they all have different root nodes—can share other nodes. For instance, many Alexa graphs include doubleclick.google.com. Thus, one could imagine taking the union of all of the Alexa graphs into a single, large *universal graph* U. This has some benefits—it allows us to easily see how central some third-party resources are—but it also runs the risk of introducing false positives, as we will see later. Thus, for some analysis, we will use the universal graph, and for others we will use the Alexa graphs.

6.2.3 VirusTotal

From the graph of each website, we first identify the domains that are "bad" like *cobalten.com* in Figure 6.2. In order to select the bad domains from the huge list of domains, we could use APIs like the Google SafeBrowsing API or the VirusTotal (VT) API. The VT API identifies a domain as a bad domain based on multiple (more than 50) VT partners and Google SafeBrowsing is one of them. Since VT responses are an aggregate of results from multiple VT partners instead of just one, we use the VT API to

define a "bad" domain. For a particular domain, each VT partner identifies it as one of the following.

- Clean site: no malware detected.
- Unrated site: the partner never reviewed the given site.
- Malware site: distributes malware.
- Phishing site: the site tries to steal users' credentials.
- Malicious site: the site contains exploits or other malicious artifacts.
- Suspicious site: the partner thinks this site is suspicious. Grey area.
- Spam site: involved in unsolicited email, popups, automatic commenting, etc.

There are scenarios where a domain can be classified as a malware site by one VT partner and as a phishing site by another.

In our study, we consider any classification other than Clean or Unrated site as "bad." But a question remains: should a domain be considered bad if only one VT partner identifies it as such? Zhu et. al. [125] identify that the VT labels vary over time and that there is a interdependence between VT partners, i.e. if one VT partner labels a domain as malicious, then few others label the domain as 'malicious as well, for example AVG and Avast. To incorporate these two factors, we identify a domain as "bad" if at least two independent VT partners identify it as such and the other domains are identified as "good" nodes.

6.3 Analysis Results

In this section, we analyse the Alexa graphs and the back-propagation graphs for the 19,562 websites. We study the possible reasons for the variations in the graphs and demonstrate the breakdown of trust along the long chains of third-party resource (back-propagation nodes) inclusions using graph analysis metrics. Our results show that some of these included resources load bad domains that are not blocked by current ad blockers. We explore the extent to which bad domains are blocked.

6.3.1 Factors Influencing the Topology

We observed that websites behave differently on mobiles when compared to desktops. To evaluate this, we study the differences in the union of all the Alexa graphs which we defined earlier as the "universal graph" U and the union of all back-propagation graphs which we call the universal back-propagation graph B obtained for the mobile and desktop versions. Additionally, we explore whether the malicious activity is consistent across websites that are visited more often and websites that are less frequently used.

User Agent: Mobile vs Desktop

Our analysis begins with a simple question: are users of mobile devices subjected to malicious websites differently than users of desktop devices? We answer this by observing differences in the inclusion and back-propagation graphs depending on the User Agent string used to obtain them. In Tables 6.1 and 6.2, we enumerate the total number of nodes, edges, bad domains and back-propagation edges present in U and B. From the

tables, we observe that there are some domains and edges that are unique to desktop or mobile environments for the same Alexa websites. We also see that the total number of domains, edges and unique bad nodes is more on desktop than on mobile, i.e. U is bigger for desktop User Agent String.

On the contrary, although the total number of domains, edges and bad domains are more for the desktop User Agent String, the number of back-propagation edges, are more for the mobile User Agent String, i.e. B is larger for mobile users. This can occur for one or both of the following reasons: (1) Alexa websites load more bad domains on average for mobile users, and/or (2) There are more levels of indirection (more back-propagation nodes) between the Alexa websites and the bad domains. We find that both are factors, although the latter is a less contributing factor when compared to the former.

The last row of Tables 6.1 and 6.2 shows the number of Alexa websites that load at least one bad domain. From this result, we see that 5% more Alexa websites load bad domains on mobile when compared to desktop. From our data, we observe that the maximum number of domains between a benign domain and a bad domain (hop count, explained in detail in Section 6.3.2) is more on mobile than on desktop, although this is a less contributing factor when compared to the number of Alexa websites that load bad domains. For example, on desktop, a back-propagation node is at most 10 hops away from any bad node and there are 2 such nodes loaded by less popular websites and 2 by more popular websites, whereas on mobile, there are nodes that are up to 13 hops away. Specifically, on mobile, among the domains loaded by more popular websites there are 4 domains that are 10 hops away, 1 that is 11 hops away and 1 that is 13 hops away. Similarly, among the domains loaded by less popular websites on mobile there are 7

domains that are 10 hops away, 4 that are 11 hops away and 1 that is 12 hops away.

	Only Desktop	Only Mobile	Common to Both
Alexa-ranked sites leading to bad nodes	100	167	755
Nodes (Total)	21,110	17,416	41,812
# Bad (raw)	166	343	2,892
# Bad (unique)	54	38	301
Edges (Total)	98,875	87,489	195,173
# Back-propagation	6,743	9,297	7,232

Table 6.1: Graph properties for *more popular* websites.

	Only Desktop	Only Mobile	Common to Both
Alexa-ranked sites leading to bad nodes	100	167	755
Nodes (Total)	5,440	4,330	26,131
# Bad (raw)	192	58	825
# Bad (unique)	22	8	255
Edges (Total)	42,034	27,469	117,394
# Back-propagation	1,568	3,118	2,444

Table 6.2: Graph properties for *less popular* websites.

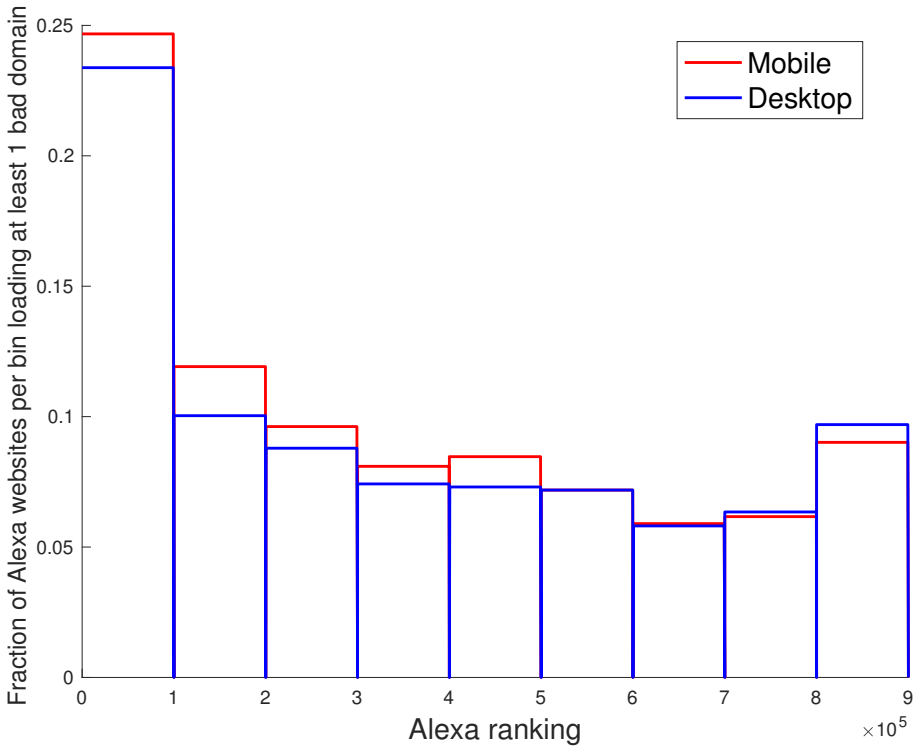
User Agent

Mobile users are more prone to malicious activity majorly because more Alexa websites (5%) load bad domains on mobile when compared to desktop. Sometimes these bad domains are farther from the Alexa websites in the mobile graph.

Website Popularity: More vs Less Popular Websites

In general, more Alexa domains load bad domains on mobile when compared to desktop. But, as the popularity of the Alexa website decreases, the number of Alexa websites loading bad domains is slightly more on desktop instead of mobile. This can be seen from Figure 6.7, which depicts the fraction of Alexa websites in every 100,000 websites that load at least one bad domain in the order of Alexa ranking.

Figure 6.7: Fraction of Alexa websites that load at least one bad domain as a function of Alexa ranking



To further analyse the differences due to Alexa ranking, we compare U and B for the 9,781 websites in Alexa top 10,000 websites and 9,781 websites selected at random from Alexa ranking 10,001 to 1 million. Tables 6.3 and 6.4, list the number of unique bad domains, domains, edges, and back-propagation edges on desktop and mobile. In each table, column 2 corresponds to the data that is obtained only by the websites in Alexa top-10,000 (More Popular), while column 3 corresponds to the data that is obtained only by the websites in Alexa rank 10,001 to 1 million (Less Popular) and the last column corresponds to the data that is obtained by both More Popular and Less Popular Websites.

From the Tables, we observe that the number of nodes and edges unique to More Popular Websites is nearly twice that of Less Popular Websites and although the difference between the number of bad nodes is not very high, the number of back-propagation

	More Popular Websites Only	Less Popular Websites Only	Common to Both
Nodes	56,059	24,708	6,863
# Bad (Unique)	280	202	75
Edges	284,900	150,280	9,148
# Back-propagation	13,527	3,564	448

Table 6.3: Graph properties when crawling with the *Desktop* `UserAgent` string.

	More Popular Websites Only	Less Popular Websites Only	Common to Both
Nodes	52,382	23,615	6,846
# Bad (Unique)	266	190	73
Edges	272,913	135,114	9,749
# Back-propagation	15,896	4,929	633

Table 6.4: Graph properties when crawling with the *Mobile* `UserAgent` string.

edges for More Popular Websites is more than thrice that of Less Popular Websites. This is because the number of Alexa websites in top 10,000 that load bad domains is more than 2,000 while the number of Alexa websites in rank 10,001 to 1 million is less than 1,000.

Website Popularity

More popular websites load 28% more unique bad domains when compared to less popular websites.

These results show that *where* a user goes (website popularity) and *how* (the user's device) both play a significant role in how much malice a user is subjected to when visiting otherwise benign websites.

6.3.2 Trust breakdown: What are the intermediary benign domains that should be held accountable?

Websites explicitly and implicitly trust third-party domains. These domains load bad domains directly or indirectly. Once we obtain the back-propagation graphs, we have all the domains that directly or indirectly load bad domains. Which among all of the back-propagation nodes are more likely responsible for introducing a malicious party onto a benign website *knowingly*? In this section, we investigate three different metrics that seek to identify whether a back-propagation node is intentionally serving a malicious role.

Frequency

Our first intuition is that domains that intentionally include malicious domains are likely to do so at a higher rate. We observe that few domains are loaded by multiple Alexa websites and most domains (87% domains that lead to at least one bad domain on desktop) are loaded by one Alexa website. For a detailed analysis, we define frequency of a node as the fraction of the number of Alexa graphs in which the node loads a bad domain to the number of Alexa graphs in which the node is loaded. The number of Alexa graphs in which the node loads a bad domain is equivalent to the number of back-propagation graphs in which the node is present. Thus, the *frequency* of domain D is:

$$\frac{\text{\# back-propagation graphs containing domain } D}{\text{\# Alexa graphs containing domain } D}$$

If the frequency is equal to zero, it implies that the node does not have a path to

any bad domain and a frequency equal to one implies that in each of the Alexa graphs containing the node, it has at least one path to a bad domain. In other words, a domain D with frequency one, implies that the D loads a bad domain everytime an Alexa website loads D. By definition, frequency of any bad domain would be one. Let us consider the back-propagation node `https://www.googletagmanager.com` which is loaded by 4893 out of Alexa top 10,000 websites. In 253 out of 4893 websites, it loads at least one bad domain. Thus, for this domain, the frequency is 0.05.

To obtain the frequency, it is essential *not* to use the"universal graph" because U would contain some additional paths which are not present while accessing any of the individual Alexa websites, but one cannot distinguish this when looking at the universal graph. For instance, let us consider an Alexa graph in which domain A loads domain B which does not load malicious domain C. In another Alexa graph, A is not present, and B loads C. Then, the universal graph would have an indirect path from A to the malicious domain C, although it does not occur in any of the Alexa graphs. Thereby implicating A as a potential cause for loading C, even though C was never loaded as a result of A in any of the Alexa graphs.

We plot the scatter plot and the cumulative distribution or each of the following 4 categories:

- Desktop, More Popular(Top10k) - For all the domains (62922) obtained by the 9781 out of Alexa top 10,000 websites on desktop
- Mobile, More Popular(Top10k) - For all the domains (59228) obtained by the 9781 out of Alexa top 10,000 websites on mobile

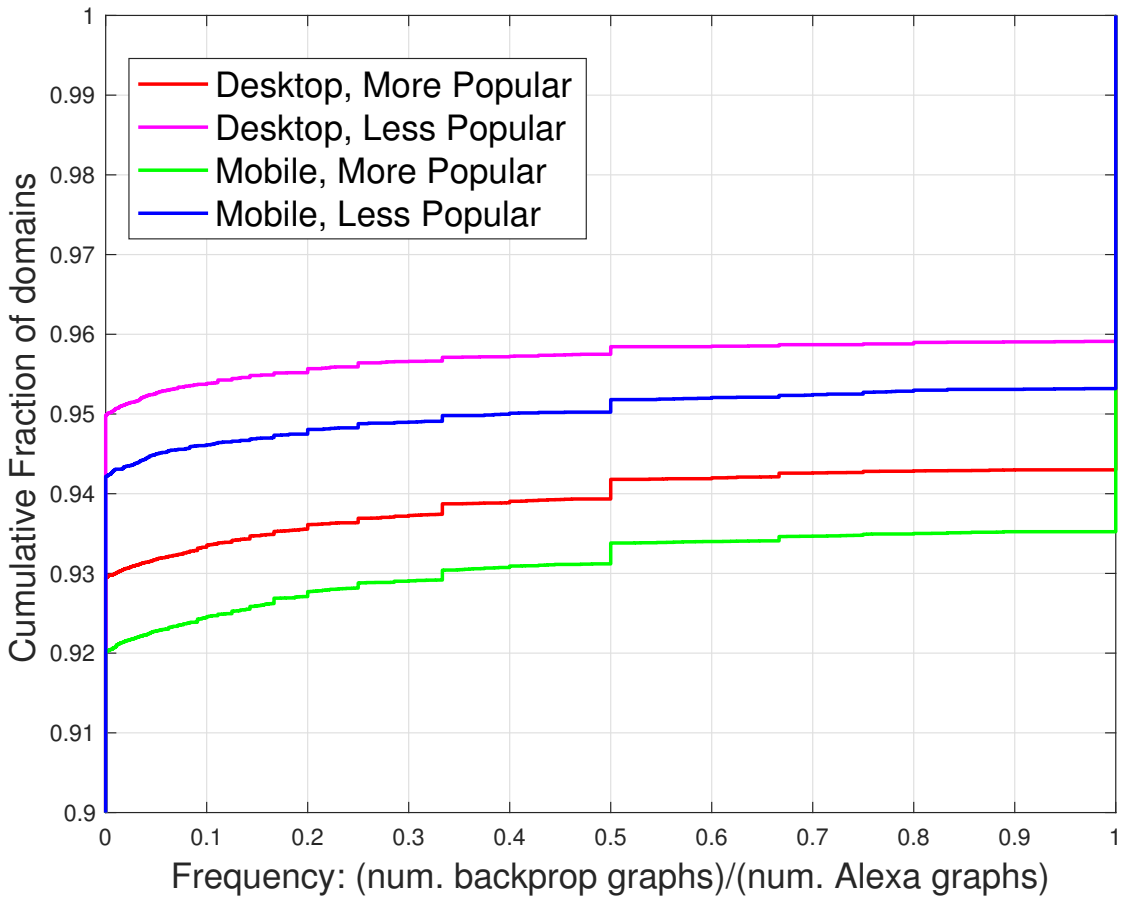


Figure 6.8: Plot of the frequency as a cumulative fraction of the number of domains.

- Desktop, Less Popular(Bot10k) - For all the domains (31571) obtained by the 9781 out of Alexa websites in 10,001 to 1 million on desktop

- Mobile, Less Popular(Bot10k) - For all the domains (30461) obtained by the 9781 out of Alexa websites in 10,001 to 1 million on mobile

Figure 6.8 represents the frequency of benign domains loading bad domains. From this plot, we observe that less than 8% of all the domains loaded, lead to at least one bad domain. We also observe that more than 4% and less than 6.5% domains load bad domains every time the domain is loaded. Among these domains, more than 90% domains are back-propagation nodes (not bad domains) that are loaded by more popular websites

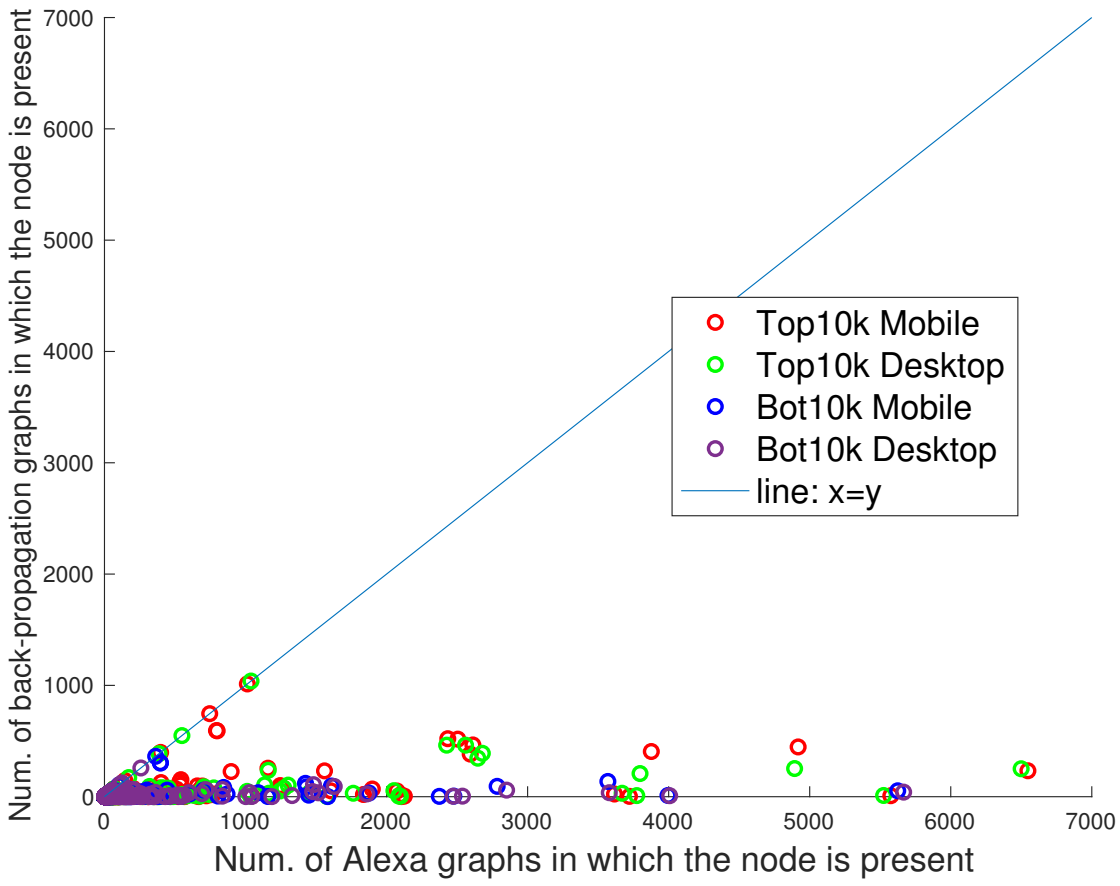


Figure 6.9: Scatter plot of domains present in at least one back-propagation graph representing the number of back-propagation graphs to the number of Alexa graphs

(about 83% back-propagation nodes loaded by less popular websites).

Figure 6.9 is the scatter plot of all the back-propagation nodes loaded by the 19,562 websites. For a particular domain (represented bya circle in the plot), the x-coordinate corresponds to the number of Alexa graphs that contain the domain and y-coordinate corresponds to the number of Alexa graphs in which the domain loads a bad domain. Figure 6.10 represents the scatter plot for the domains that are loaded in at least 10 Alexa graphs, which contributes to around 10% of the domains that load bad domains. Among these domains, more than 40% of the domains load bad domains with a frequency ≥ 0.1 and up to 23% of the domains load bad domains with a frequency ≥ 0.25. These domains

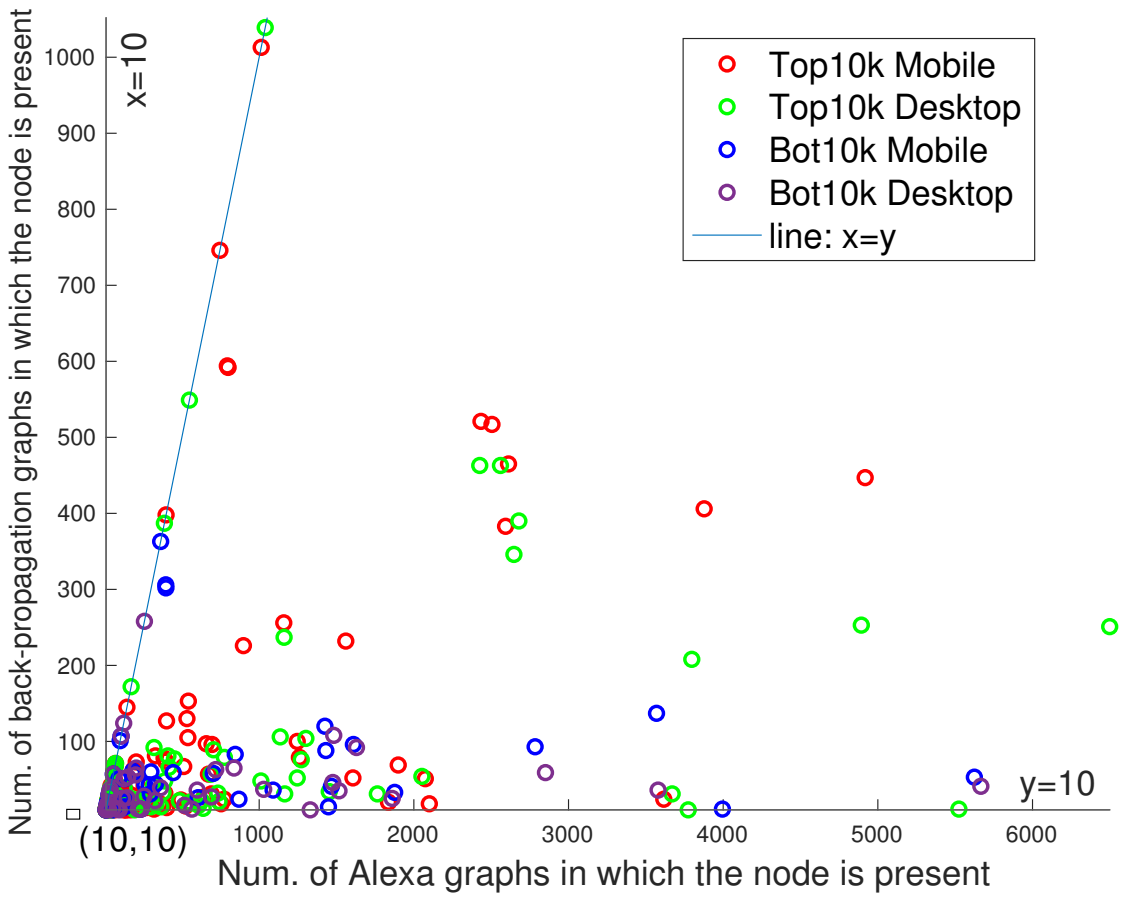


Figure 6.10: Same plot as in Figure 6.9, but limited to only the nodes present in at least 10 Alexa graphs

that frequently load bad domains are responsible for significant malicious activity on the Web.

From the scatter plots, we also observe that the maximum number of back-propagation graphs containing a single bad domain (`https://certify.alexametrics.com`) is 1039 and 1013 graphs on desktop and mobile, respectively. Out of the back-propagation nodes that load bad domains whenever loaded, around 97% of the domains are loaded only by one Alexa website and only 2 domains are loaded by at least 10 more popular Alexa websites, while only 1 domain is loaded by at least 10 less popular Alexa websites.

In the scatter plots, the line $y = x$ shows the set of domains that are on a back-

propagation graph every time they are included, lending strong evidence that they are indeed responsible for ultimately loading bad content. The domains that are closer to the line $y = x$ load bad domains more frequently and thus are more *knowingly* responsible for loading bad domains.

Frequency

A significant number of back-propagation nodes (up to 23%) load bad domains frequently (at least 25% of the time the back-propagation node is loaded).

Bad-Node Count

From the results of the frequency metric, we observed that a single back-propagation node loads bad domains across multiple Alexa graphs. A question arises as to whether the back-propagation node loads the same bad domain across all the Alexa graphs or different bad domains each time. For example, domain `https://sync.go.sonobi.com` loads only one bad domain, `https://ap.lijit.com` in 17 different Alexa graphs, whereas `http://pagead2.googlesyndication.com` loads 6 different bad domains in 9 Alexa graphs. To explore this in detail, we define *bad-node count* for a back-propagation node, D as the unique number of bad domains that are ultimately loaded by the D. By definition, for domains that do not load any bad domain, the bad-node count is zero.

Figure 6.11 represents the distribution of number of unique bad domains loaded by all the back-propagation nodes. Since, we are considering domains that load at least one bad domain, the minimum bad-node count is one. From the Figure, we can see that nearly 53-60% of the doamins load a single bad domain and less than 5% of the domains load more than 10 different bad domains. The highest value of bad-node count is for domain

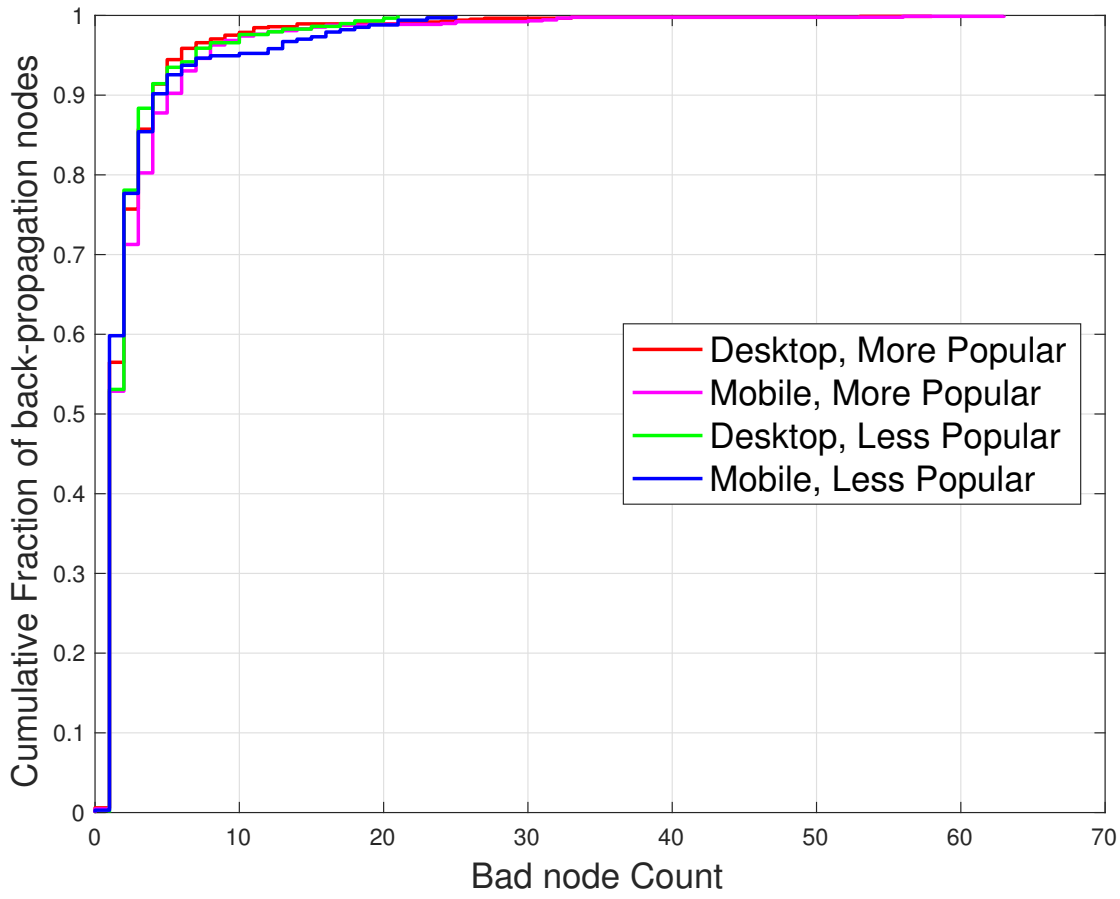


Figure 6.11: Distribution of the bad node count as a cumulative fraction of the number of benign domains.

`https://tpc.googlesyndication.com` which loads 58 and 63 different bad domains when loaded by more popular websites on desktop and mobile, respectively, while it loads 21 and 25 different bad domains when loaded by less popular websites on desktop and mobile, respectively.

When we take a closer look at the bad domains, we see that there are some bad domains that are loaded more commonly. For example, `https://ap.lijit.com` and `https://certify.alexametrics.com` that are loaded by about 44% of the back-propagation nodes. We also observe that these predominantly occurring bad domains are loaded by the back-propagation nodes independent of their frequency. Thus, from the data, we observed

that although majority back-propagation nodes load a single bad domain, they often tend to load one of these predominantly occurring bad domains.

Bad-Node Count

While majority of the domains load a single bad domain whenever accessed by Alexa websites, there are few domains that load up to 63 different bad domains. Also, most domains load one or more of the few predominantly occurring bad domains.

Hop Count

By looking at the back-propagation graphs, we see that back-propagation nodes load bad domains either directly or indirectly. Sometimes, a back-propagation node loads the same bad domain differently when loaded by different Alexa websites, i.e., different levels of indirection. For example, in Figure 6.6, the root node (`https://www.tribunnews.com`) loads the bad domain, `https://search.spotxchange.com` via `https://cdn3.tstatic.net` directly or through `https://securepubads.g.doubleclick.net`. To understand this graphical nature, we define *Hop count* for a back-propagation node D and a bad domain B, as the minimum number of nodes in the path starting from D to B, excluding the bad domain. For the same example as above, the hop count for the domain `https://www.tribunnews.com` and bad domain `https://search.spotxchange.com`) is 1. The results of bad-node count demonstrate that some back-propagation nodes load more than one bad domain. Thus, we consider the hop count for a pair (back-propagation node, bad domain) as there would be multiple such pairs for a particular domain. By definition, we hypothesise that lower the hop count more responsible the domain for malicious activity.

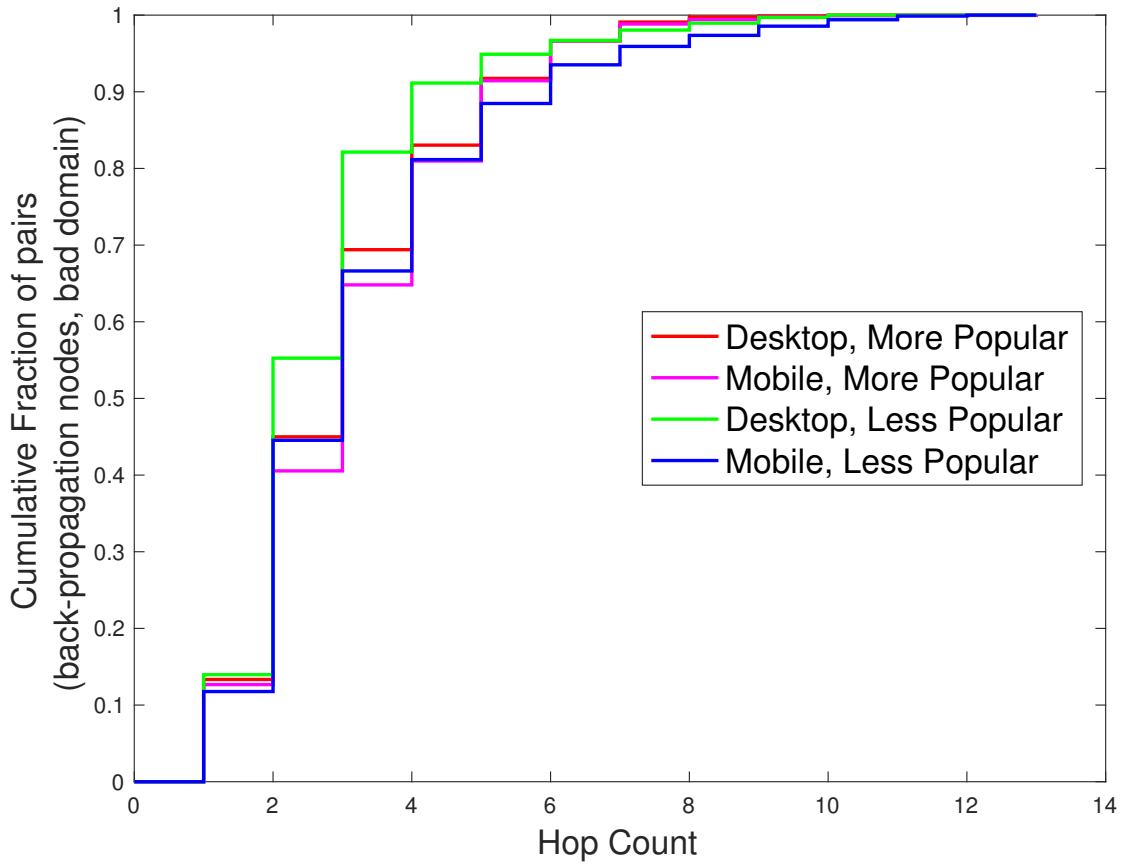


Figure 6.12: Distribution of the hop count as a cumulative fraction of the number of (domain, bad domain) pairs.

Figure 6.12 represents the distribution of hop count for all (back-propagation node, bad domain) pairs. The least value for hop count for a (back-propagation node, bad node) pair is one. We observed that the highest hop count was 13 for (`https://exchange.adtrue.com`, `https://certify.alexametrics.com`) loaded by Alexa website ranked 2245, `tuko.co.ke` on mobile device. Considering the back-propagation node, `https://www.googletagmanager.com`, which loads 24 bad domains in 253 Alexa graphs. This back-propagation node contributes to 24 pairs in the distribution. 4 of the bad domains are one hop away and the highest hop count for this back-propagation node is 5. From Figure 6.12, we observe that maximum number of pairs (around 30%) have a hop count 2, while the mean is close to 3 and only

about 12% of the domains load bad domains directly. This implies that majority back-propagation nodes (55% to 75%) are 2 or 3 hops away from bad domains.

Hop Count

While a few benign domains (less than 15%) load bad domains directly, the majority of the domains load bad domains at a hop distance of 2 and 3. Thus implying that most bad domains are not directly loaded, instead they are a result of implicit trust on the domains that are directly loaded.

Using graphical analysis metrics, we observe that there are significant back-propagation nodes that frequently load bad domains and few of these bad domains are loaded much more often than most other bad domains. Additionally, we see that these bad domains are more often indirectly loaded than directly being loaded. These results illustrate the breakdown of trust along the chain of domains loaded and help identify the domains that are likely responsible for loading bad domains. Blocking or minimizing the use of these back-propagation nodes can significantly reduce malicious activity on the Web, which we demonstrate in Section 6.4 of this chapter.

6.3.3 AdBlockers: Can we evaluate the effectiveness of various blocklists?

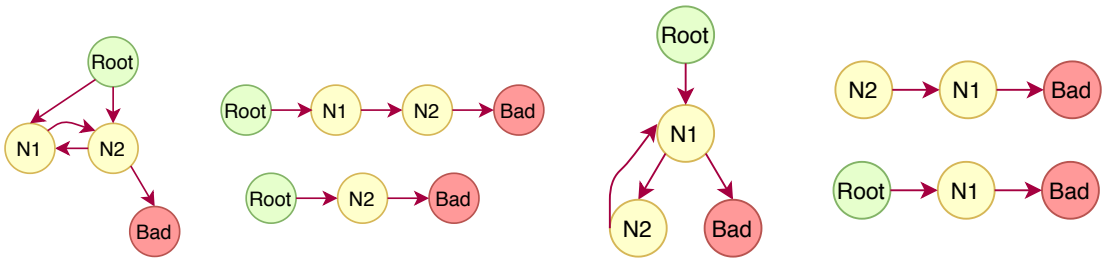
Many users like to use Adblockers for a convenient browsing experience. Adblockers use blocklists, containing a list of domains that are considered malicious, to block malware. But, these blocklists do not block all bad content. To study these lists, we compare the blocklists with the list of bad domains we have identified using VT. In this section, we demonstrate that there are domains that are not blocked by a blocklist but are

identified as "bad" using VT. For a detailed look at the blocklists used by Adblockers, we look at the Pete Lowe's list, which is a default list used by uBlock Origin browser extension. Pete Lowe's blocklist has been maintained for 20 years and is highly downloaded (5-7 million downloads per day) even on mobile devices to be added to Adblockers. Additionally, we also looked at the popular EasyList, but we observed that none of the domains were blocked by Easylist. Hence, we focus our study on Pete Lowe's blocklist.

To understand the functioning of adblockers, we define a *path* by enumerating the nodes from the root to the leaf in the order of directed edges. We define a *blocklisted node* as a node which is present in a blocklist. If a path contains a blocklisted node, N_b, all the domains from N_b to the leaf in the path are blocked. If a leaf node is in the blocklist, only that domain is blocked. Going back to the graph template in Figure 6.3, if "Bad2" alone is present in the blocklist, only the nodes on paths containing "Bad2" are blocked, thus, "Bad1" is not blocked. Instead if the "Bad1" was present, both the bad nodes would be blocked even if "Bad2" were not present in the blocklist.

We list the paths to the bad nodes in the back-propagation graphs, by enumerating the nodes backwards from the bad domains up to the root node or a back-propagation node with incoming edges from a node that's already in the path. Since the graphs are cyclic, while listing the paths, we ensure that a node is not repeated in a path more than once, i.e. nodes in a cycle are counted once, for example Figure 6.13(a). Few paths start with a cycle instead of the root node, as seen in the example in Figure 6.13(b) where N2 is not a root node; instead it has one incoming edge from N1. Two paths are considered different if they differ by at least one node.

We categorize the paths into the following two types based on the presence of a



(a) Example 1 (b) Example 2

Figure 6.13: Examples enumerating paths to a bad node in graphs with cycles ensuring that the nodes in the cycles are counted only once.

blocklisted node in the path.

- Blocked Path - there exists at least one blocklisted node in the path
- Unblocked Path - there is no blocklisted node in the path

We categorize all the "bad" domains based on the paths to these bad domains obtained from the graphs of the 19,562 Alexa websites:

- Directly Blocked - bad domain that is present in the blocklist and has at least one incoming edge
- Completely Indirectly Blocked - bad domain that is not present in the blocklist and there is no unblocked path to it
- Partially Indirectly Blocked - bad domain that is not present in the blocklist and there exists at least one unblocked path to it
- Not Blocked - bad domain that is not present in the blocklist and there is no blocked path to it

	Only Desktop	Only Mobile	Common to Both
# Bad Domains	54	38	301
Directly Blocked	10	8	33
Completely Indirectly Blocked	5	6	8
Partially Indirectly Blocked	9	16	24
Not Blocked	46	37	130

Table 6.5: Bad domains classification for More Popular Websites

	Only Desktop	Only Mobile	Common to Both
# Bad Domains	22	8	255
Directly Blocked	2	2	11
Completely Indirectly Blocked	5	8	7
Partially Indirectly Blocked	8	4	16
Not Blocked	27	14	142

Table 6.6: Bad domains classification for Less Popular Websites

From Tables 6.5 and 6.6, we can see that only about 1.2% and 5% of the bad domains loaded by more and less popular websites, respectively, are directly blocked by Peter Lowe's blocklist. Only around 12% of the bad domains are blocked indirectly either partially or completely. Nearly 50% (60%) of the bad domains loaded by more (less) popular websites, are not blocked in any way by the Peter Lowe's blocklist.

We provide some examples:

- `https://crcdn01.adnxs.com` is a bad domain that is loaded only on mobile device and is present in the Pete Lowe's blacklist.

- `https://crptgate.com` and `https://etahub.com` are two of the bad domains that are not blocked in mobile but are partially indirectly blocked in desktop. This

is because the paths to the bad domains are different on desktop and mobile.

Ad Blockers

Nearly 50% and 60% of the bad domains loaded by more and less popular websites, respectively are not blocked in any way by the Pete Lowe's blocklist. A good blocklist should ensure that more bad domains are blocked and the number of unblocked paths to the bad domains is small.

6.4 Proposed Mitigation Strategies

So far, we discussed that the use of various benign domains leads to the possibility of landing to an unsafe site unintentionally owing to 25% of Alexa top-10,000 websites and up to 10% of less popular websites being affected by malicious activity. Here, we propose the possible ways of mitigating the prevalence of bad domains. This can be done by notifying the developers of the possible vulnerable domains (after assigning a trust metric to the domains) so that the use of these domains can be minimized in the future and add them to blocklists in order to reduce malicious activity when users access the websites.

6.4.1 Trust Metric

From the results of trust breakdown in Section 6.3.2, we see that there are three different metrics, namely frequency, hop count and bad-node count that can be attributed to a back-propagation node. In this section, we use these metrics to define a trust metric for a back-propagation node. Using the trust metric, developers can determine if the node is safe to use for future web development. We define two different trust metrics.

Scalar Trust Metric

We define trust metric such that the domains loading bad domains have a low trust metric compared to domains that do not load bad domains. For a bad domain, the trust metric should be 0, whereas for a benign domain that does not load any bad domains, the trust metric should be 1. A simplistic approach to assign trust metric can be using the frequency attribute. A domain that more frequently loads bad domains should be assigned a low value of trust metric. In figure 6.3, N6 loads two bad nodes directly, whereas N5 loads one bad and one good node. We should be able to determine using the basic trust metric that N5 is comparatively less harmful compared to N6. Thus, we define the trust metric as:

$$\text{scalar trust metric} = 1 - \frac{\text{Num. of times the domain loads a bad domain}}{\text{Total num. times the domain is loaded by all Alexa websites}}$$

Threshold	% back-propagation nodes	% bad domains
0.5	8.84	9.86
0.6	13.02	11.27
0.7	18.37	13.80
0.8	25.81	18.59
Threshold	% back-propagation nodes	% bad domains
0.5	11.52	11.80
0.6	13.99	12.39
0.7	20.58	16.52
0.8	29.63	28.61

Table 6.7: For different threshold on Scalar trust metric, the percentage of back-propagation nodes with metric less than the thershold and corresponding percentage of bad domains loaded by more popular domains on desktop (top) and mobile (bottom).

Using this metric, all back-propagation nodes with scalar trust metric above a threshold are safe to use. Tables 6.7 and 6.8 contain the percentage back-propagation nodes and the corresponding percentage of bad domains that can be avoided by not using the back-propagation nodes. Column 2 represents the percentage back-propagation nodes present in at least 10 Alexa graphs and have scalar trust metric less than different threshold values.

Threshold	% back-propagation nodes	% bad domains
0.5	9.71	5.42
0.6	12.57	6.14
0.7	14.86	7.58
0.8	20.57	8.30
Threshold	% back-propagation nodes	% bad domains
0.5	12.37	7.22
0.6	14.36	7.60
0.7	19.80	9.13
0.8	25.74	10.65

Table 6.8: For different threshold on Scalar trust metric, the percentage of back-propagation nodes with metric less than the thershold and corresponding percentage of bad domains loaded by less popular domains on desktop (top) and mobile (bottom).

Using 0.5 as the threshold, we observed that around 9% back-propagation nodes loaded on desktop and 12% back-propagation nodes loaded on mobiles have the metric value less than 0.5. We also observe from the tables that for any threshold fewer bad domains loaded by less popular websites are blocked when compared to more popular domains.

Vector Trust Metric

The scalar trust metric uses only frequency, to incorporate hop count and bad-node count, we define a vector trust metric of length 15. Let us denote the i^{th} element of the vector as V_i. For $i = 1, ..., 13$, V_i is equal to the number of bad nodes that are i hops away from the benign node. V_{14} corresponds to the number of back-propagation graphs and V_{15} corresponds to the number of Alexa graphs in which the domain is present. For example, `https://googleads.g.doubleclick.net` is 2 hops away from 6 different bad nodes, 3 hops away from 10 different bad nodes, 4 hops away from 5 different bad nodes and 8 hops away from 1 bad node. Although, this domain loads multiple bad domains, it loads these bad domains only 208 out of 3797 occurrences. From this, the trust metric vector for this back-propagation node would be [0, 6, 10, 5, 0, 0, 0, 1, 0, 0, 0, 0, 0, 208, 3797].

Using the last two elements, we can compute the scalar trust metric. The other 13 elements are representative of *how many* and *how far* the bad domains are loaded. This is useful especially when the scalar trust metric is high. Considering the examples `https://served-by.pixfuture.com` with vector trust metric [0, 1, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 6, 15] and `https://cdn.pixfuture.com` with vector trust metric [0, 0, 1, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 6, 15]. Both these domains load the same number of bad domains with the exact same frequency. Observing the paths from these domains to bad domains, we see that `https://served-by.pixfuture.com` loads `https://cdn.pixfuture.com`, which in turn loads 3 different bad domains. Thus, `https://cdn.pixfuture.com` is comparatively more responsible for loading bad domains. This is reflected from the vector trust metric.

Also, there are cases where a back-propagation node, D1 always loads another back-propagation node D2 whenever a bad domain, B is ultimately loaded, while it loads other benign domains (say D3) when it does not load any bad domains. In such cases, the domain D2 not only loads the bad domain B, but also loads other bad domains when loaded by domains other than D1. This can be seen in the example where domain, `https://play.aniview.com` (vector trust metric [0, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 8, 13]) always loads `https://player.aniview.com` (vector trust metric [2, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 27, 41]) whenever (3 different) bad domains are loaded (it loads other domains which lead to benign paths only). `https://player.aniview.com` loads 2 other bad domains other than the 3 bad domains that are loaded when `https://play.aniview.com` loads `https://player.aniview.com`.

The examples show that the vector trust metric includes additional details, thus

enhancing the identification of the domains that are likely more responsible in loading bad domains. To obtain the list of such domains using the vector trust metric, we first add all the back-propagation nodes which are loaded by at least 10 websites and at least 50% of the time, i.e., domains with $V_{14} \geq 10$ and $\frac{V_{14}}{V_{15}} \geq 0.5$. For domains with fraction $\frac{V_{14}}{V_{15}}$ between 0.2 and 0.5, we add domains recursively such that $V_i * \frac{V_{14}}{V_{15}} > t_i$, threshold t_i increases as i increases. We use zero for t_1 and $t_i = t_{i-1} + 0.05$ for $i = 2, ..., 13$. The results of our algorithm are shown in Table 6.9. Comparing these results to the results of using scalar trust metric, we can see that avoiding the use of fewer back-propagation domains leads to more bad domains being prevented.

	% back-propagation nodes	% bad domains
Desktop; Popular Websites	20.93	22.82
Mobile; Popular Websites	24.90	38.05
Desktop; Less Popular Websites	14.28	9.58
Mobile; Less Popular Websites	20.29	15.60

Table 6.9: The percentage of back-propagation nodes and corresponding percentage of bad domains minimized using vector trust metric.

Using either the scalar or vector trust metric, we can obtain a list of back-propagation nodes, which when added to a blocklist can block some bad domains from being loaded. But, this does not necessarily ensure that these bad domains are not loaded by other back-propagation nodes that are not in the blocklist. For a more sophisticated list, we should take into account the interactions between the back-propagation and bad nodes. We describe this next.

Trust Metric

Using only the graphical analysis metrics to compute a trust metric for domains in order to block domains based on a threshold on the computed trust metric is ineffective at blocking bad domains.

6.4.2 Optimization Problem

In Section 6.3.3, we analysed that less than 5% of all the bad domains flagged by VirusTotal are directly blocked by the Peter Lowe's blocklist. The blocklist contains some back-propagation nodes like `https://googleads.g.doubleclick.net` although it is not a bad node. This node indirectly blocks the 22 bad nodes that it loads. From the previous analysis, we observe that even after indirectly blocking, more than 50% of the bad nodes are still not blocked.

To mitigate malicious activity, we should increase the number of bad domains blocked, by adding minimum number of domains to the blocklist. Here, a question that arises would be, *why not add all the bad domains to the blocklist which would imply that all bad domains are directly blocked*. But, the problem is that attackers may generate bad domains or use registered benign domains which are no longer in use (residual trust [69]) to inflict malicious activity. Bad domains are more closer to the leaf than the root node and domains closer to the root require more established identities, for instance, `doubleclick.net` requires a business arrangement. This makes it easier to generate bad domains. Thus, just blocking the bad domains would not serve the purpose. Instead, if the back-propagation nodes that are more responsible for loading bad domains are added to the blocklist, even if the back-propagation nodes load new bad domains, the newly included bad domains would also be blocked without the necessity to add the new bad domains to the blocklist.

In this section, we analyse the paths to the bad nodes that are not blocked by the Peter Lowe's blocklist and identify the list of back-propagation nodes that can be added

to the list to indirectly block more bad nodes. This is an optimization problem which when solved would increase the percentage of bad domains that are blocked. From our analysis results of bad-node count, we observe that many back-propagation nodes load bad domains from among a few predominantly occurring bad domains, some of which are not directly blocked by Pete Lowe's blocklist. These bad domains can be blocked by directly adding them to the blocklist, but for the other bad domains we can add the back-propagation nodes that load one or more of them. From our web topology data, we observed that multiple back-propagation nodes load a single bad domain. If only a subset of these back-propagation nodes are added, the bad domain is partially blocked. The optimization algorithm should ensure that bad domains are completely indirectly blocked.

To solve the problem, we first compute a binary matrix, M that captures the relationship between the back-propagation nodes and bad domains. The rows in the matrix correspond to back-propagation nodes (total n_{bp}) and each column corresponds to a bad domain (total n_{bad}). The element $M_{i,j}$ in the matrix would be 1 if back-propagation node i loads bad domain j and the value would be 0 otherwise. The optimization problem would then be to add the minimum benign domains (rows) such that maximum number of bad domains (columns) would be blocked. Based on the web topology graph, M is a sparse matrix. From M, we compute a row vector v with element v_j equal to the number of back-propagation nodes that load bad domain j. We then compute a diagonal matrix, N with the diagonal elements equal to the elements in v.

Let us consider binary column vectors x and y to represent the back-propagation nodes and bad domains, respectively. The elements in x and y are either 0 or 1. Element

x_i is 1 if back-propagation node i is added to the blocklist and is 0 otherwise. Similarly, element y_j is 1 if the bad domain is blocked and 0 otherwise. Equation (1) represents the optimization problem for blocking k bad domains. This mathematical formulation corresponds to a *Sparse Binary Integer Linear Programming Problem* which is NP-complete.

$$\begin{aligned} \min_{x_i} & \sum_{i=1}^{n_{bp}} x_i \\ \text{s.t.} \quad & M^T \cdot x \geq N \cdot y \\ & \sum_{j=1}^{n_{bad}} y_j \geq k \\ & N = diag(M^T \cdot (11 \cdots 1)) \\ & x_i, y_j \in \{0, 1\} \end{aligned} \tag{6.1}$$

There are several heuristic methods available to solve an integer programming problem, but due to the size of the matrices, the available solutions are not suitable. Another drawback with the binary optimization formulation is that the optimization needs to be solved separately for each value of k separately making it impractical for use on realistic web topology data. Further, the back-propagation nodes obtained from the optimization solution can be very different for close of values of k making it hard to identify incremental blocklists. In order to overcome these drawbacks, we develop a greedy iterative methodology.

Optimization Problem

Maximizing the bad domains blocked by adding minimum back-propgation nodes to a blocklist is an NP-complete problem

6.4.3 Greedy Iterative Algorithm

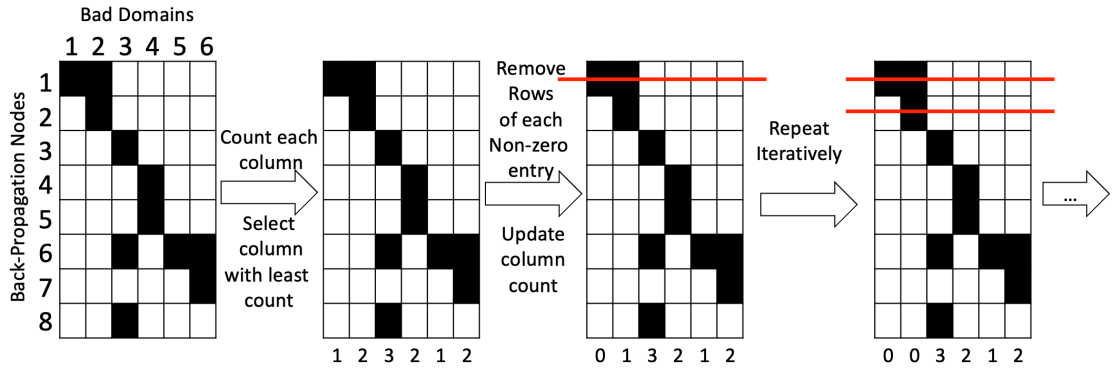
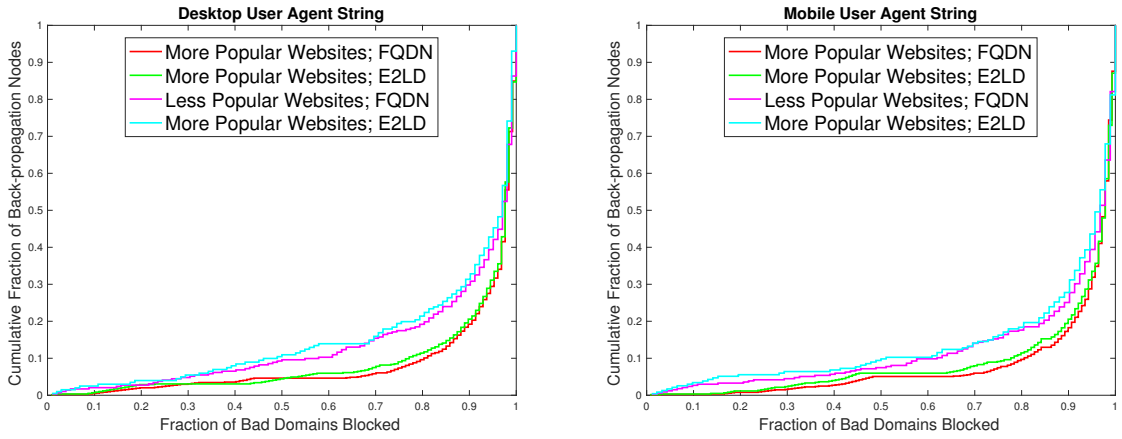


Figure 6.14: Heuristic Algorithm to identify the list of back-propagation nodes to be added to a blocklist in order to indirectly block bad domains

In our algorithm, illustrated in Figure 6.14, we recursively include back-propagation nodes to the blocklist and remove from the matrix, the bad domains that are completely blocked by the added node. The algorithm determines the sequence of back-propagation nodes in the descending order of importance. In this example, there are 8 back propagation nodes and 6 bad domains with the filled in cells of the sparse matrix corresponding to a path between a back-propagation node and a bad domain. The first step is to add the column of the matrix and identify the bad domain that is connected to the minimal number of back-propagation nodes. In this example, the bad domain is B1. The back-propagation nodes corresponding to this bad domain are then added to the blocklist and the sparse matrix is updated by removing the entries of the corresponding rows (bad domains). This procedure is then repeated iteratively on the remaining bad domains and back-propagation nodes till all the bad nodes are blocked. As we are estimating the most effective bad domain to block at each iteration using only the remaining sparse connections at that iteration, it is a greedy approach. Thus, the proposed algorithm is a greedy

iterative methodology to identify the optimal sequence of the back propagation nodes to be appended to the blocklist to achieve a required number of bad nodes to be completely blocked. For this specific example, the sequence of the back propagation nodes are 1, 2, 6, 7, 3, 8, 4, 5.

By definition of blocklists, if a domain is present in the blocklist, all its subdomains would be blocked. To implement this, we obtain the matrix for the effective second level domains (E2LD's) corresponding to fully qualified domains names (FQDNs) of the back-propagation nodes. Trivially, we only use those domains that are not currently in present in the Peter Lowe's blocklist.



(a) Desktop. (b) Mobile.

Figure 6.15: Distribution of the fraction of bad nodes that are blocked as a cumulative fraction of back-propagation nodes added to the blocklist.

Figures 6.15(a) and 6.15(b) show the distribution representing the fraction of bad domains that would be blocked as a cumulative fraction of back-propagation nodes that are added to the blocklist. The distributions include both the FQDNs and E2LDs of the back-propagation nodes. We observe that for the same fraction of back-propagation nodes E2LDs block more bad domains as compared to FQDNs. Converting from FQDNs

to E2LDs would reduce the total back-propagation nodes. Thus, fewer E2LDs block more bad domains as compared to the corresponding FQDNs. From the plots, we can conclude that adding about 15% back-propagation nodes would completely block more than 70% bad domains. We also observe that, for the same number of back-propagation nodes added to blocklist, the number of bad domains blocked is more on mobile when compared to desktop.

Blocking back-propagation nodes would not only block bad domains, but also benign domains that are loaded by the blocked back-propagation nodes. To incorporate the constraint that the number of benign domains blocked is low, we could compute another matrix representing the paths between the back-propagation nodes and benign domains. This is a challenge since the total number of benign domains loaded is very high (about 50,000 for more popular websites and 20,000 for less popular websites). An alternative is that M can be modified to include other graphical metrics like hop count. Using hop count, blocking nodes with lower values for hop count could ensure that fewer benign domains are blocked. Incorporating hop count for the graph of more popular websites on desktop, we observed that by blocking 10% back-propagation nodes, we can block 80.16% bad domains while simultaneously blocking 24.74% benign domains that do not load any bad domain.

Greedy Iterative Algorithm

Representing the interactions between the back-propagation and bad nodes as a sparse matrix and using it in our greedy algorithm significantly improves blocking bad domains.

This optimization problem is a trade-off between the number of bad domains blocked

vs the number of benign domains blocked. Our algorithm is a first step towards optimal blocking of bad domains, such that the number of bad domains blocked is maximized while simultaneously ensuring that number of benign domains blocked is minimal. We hope this work would unfold more avenues for effective blocking of bad domains.

6.5 Conclusion

The web is a complex interconnection of resources, hyperlinks, and dynamically generated code. We observed that when a user visits a webpage, many first-party and third-party resources are loaded in the background, some of which are related to the visible content on the webpage, others of which are third-party services and advertisements. Prior work [59] shows that nearly 50% of the websites implicitly and explicitly trust third-party resources. Our aim is to study *At what point is the trust the user has in the website violated?* Towards this end, we conduct an extensive study on Alexa top-10,000 websites (to capture the websites that most affect users) and a set of 10,000 websites chosen at random from Alexa ranking between 10,001 and 1 million (to capture an unbiased sample of more standard websites).

We observe that *where* a user goes (website popularity) and *how* (the user's device) both play a significant role in how much malice a user is subjected to when visiting otherwise benign websites. Mobile users and more popular websites are more prone to malicious activity majorly because 5% more Alexa websites load bad domains on mobile when compared to desktop. Sometimes there are more levels of indirection between the Alexa websites and the bad domains. The results also show that more than twice the more

popular websites load an overall 23% more bad domains when compared to less popular websites.

The results of our study show that 25% of Alexa top-10,000 websites and up to 10% of less popular websites being affected by malicious activity. We analyze the fully qualified domains corresponding to the resources loaded by these websites based on three different graph analysis metrics. From the results of these metrics, we observe that up to 23% domains loaded by various websites, frequently load bad domains and these bad domains are more often indirectly loaded than directly being loaded. Minimizing the use of these domains can significantly reduce malicious activity on the web. We propose trust metrics based on the graph analysis metrics to obtain a list of domains that web developers can avoid using in order to minimize malicious activity on websites. Some of these include ad networks. Thus, the trust metric can help developers better understand which ad networks are safer to use.

Many users use Adblockers to block malicious activity while navigating various websites. We analyzed the effectiveness of the Peter Lowe's blocklist which is a default list used by uBlock Origin and is highly downloaded even on mobile devices. Our results show that nearly 50% and 60% of the bad domains loaded by more and less popular websites, respectively are not blocked in any way by the Pete Lowe's blocklist. We propose to add a subset of the back-propagation nodes that load bad domains to the blocklist. This subset can be obtained based on the optimization problem that minimizes the number of back-propagation nodes blocked to indirectly block a pre-specified number of bad domains while simultaneously ensuring that the number of benign domains blocked is minimal. Based on a greedy iterative methodology to solve this problem, we observe that

by adding about 15% of the domains can block more than 70% (85%) bad domains loaded by more (less) popular websites completely.

Our results show that currently, the web is prone to significant malicious activity due to the long chain of indirect resources downloaded by websites. This can be partly owed to the unavailability of an interface for web developers (and maintainers) to understand the third-party resources that websites can be delegated to. Our work aims to provide such an interface to users and developers by analysing the graph theoretical approaches to analyse the data.

Chapter 7: Conclusions

Developers are often unaware of the risks involved in the inclusion of third-party resources compromising not only the security, but sometimes even the performance. In my thesis, I aim to understand these risks and based on the results, I conclude that the potential risks in third-party resources inclusions can be measured and mitigated. To better understand the risks, we first classified the risks into Direct and Indirect Risks. Direct risk is the risk that comes with invoking the third-party resource incorrectly—even if the third party is otherwise trustworthy—whereas indirect risk is the risk that comes with the third-party resource potentially acting in an untrustworthy manner—even if it were invoked correctly.

We study security related direct risks from the perspective of addressing misuse of cryptographic APIs. Our work first sought to understand and categorize the needs of the developers and the root causes of errors. We then used those insights to develop a new interface for third-party resource inclusion that is more resilient to developer errors. Collectively, our results show that the direct risks can be mitigated by better understanding the *users' needs and limitations* on either side of the interface, and designing in a way that meets them. The end result, like with our semantic API, may not always be the *overall smallest* interface—traditional cryptographic APIs do not account for different classes of

engineers—but it ought to result in that which presents the smallest necessary interface for each specific class of user. So doing allows developers to be able to more easily reason about what exactly the code they directly include is actually doing.

Besides security related issues, sometimes even the performance may be affected due to ineffective invocation of resources. We study this in the context of HTML resource hints. We observe that a majority of websites do not utilize any sort of resource hint and even fewer websites use resource hints to their full potential. An even smaller subset of websites actually enable DNS prefetching for HTTPS, this improper implementation would negatively impact performance. While most resource hints are filtered by ad blockers, both DNS-prefetch and preconnect tags are not. Our results show that an increased understanding of the benefits and dangers of resource hints is important for both ad blocking companies and for webpage creators.

Reasoning about what resources are directly included—and how—is only half of the challenge, as sometimes the details of what the resources are doing can be hidden through indirect or repeated inclusions. We study this in the context of resource inclusions in websites. We observed that when users visit some popular and less popular websites known to be benign either on desktop or mobile, they are affected by unwanted sometimes malicious activity. We use graph-based analysis on these websites to identify at what point the users' trust in the websites is violated. Our results show that understanding the domains that are trusted by the websites, but load bad domains, could provide users and developers with an insight on the malicious activity of the web. This would help users improve their browsing experience by better blocking malicious activity and it would help web developers and maintainers to avoid the use of the domains that knowingly load bad

domains.

From the results of both direct and indirect risks, we believe that a solution to improve performance and security is to provide a better interface for developers and providing more avenues to inform users of the risks. Whether invoking third-party resources directly or indirectly, the interface should have the ability to restrict and have some notion of trust. For example, information about which cryptographic libraries or what ad networks would be safer to use and what domains should be blocked.

While it is of critical importance to present users and developers with information about the risks of third-party inclusion, it is currently quite difficult to *obtain* much of the information in the first place. One of the primary challenges we faced through much of this dissertation was the lack of established, sound methodologies for measuring third-party inclusion. For instance, it is surprisingly difficult to fully extract what third-party resources are being loaded by a website, and similarly surprising that prior work had not established sound methodologies for doing so. The methodologies I have presented in this dissertation can serve as somewhat of a guide towards how to gather and understand data pertaining to the risks of third-party resources.

Bibliography

[1] Adblock plus. `https://adblockplus.org/`.

[2] Virus total. `https://www.virustotal.com/`.

[3] Cryptography.io - cryptographic standard library. `https://cryptography.io/en/latest/`, 2013.

[4] *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[5] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *IEEE Security & Privacy*, 2016.

[6] Alexa – top sites. `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`.

[7] Athanasios Andreou, Giridhari Venkatadri, Oana Goga, Krishna P. Gummadi, Patrick Loiseau, and Alan Mislove. Investigating ad transparency mechanisms in social media: A case study of facebook's explanations. 01 2018.

[8] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for dns. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[9] Sajjad Arshad, Amin Kharraz, and William Robertson. Identifying extension-based ad injection via fine-grained web content provenance. 2016.

[10] Sajjad Arshad, Amin Kharraz, and William Robertson. Include Me Out: In-Browser Detection of Malicious Third-Party Content Inclusions. 2016.

[11] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In Gail C. Murphy and Guy L. Steele Jr., editors, *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 1–13. ACM, 2015.

[12] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367, 2016.

[13] Albert-László Barabási, Réka Albert, and Hawoong Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: statistical mechanics and its applications*, 281(1-4):69–77, 2000.

[14] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Technical Report 800-131A Revision 1, NIST Special Publication, 2015.

[15] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. A Longitudinal Analysis of the ads.txt Standard. 2019.

[16] Muhammad Ahmad Bashir, Sajjad Arshad, William Robertson, and Christo Wilson. Tracing information flows between ad exchanges using retargeted ads. 2016.

[17] Muhammad Ahmad Bashir, Sajjad Arshad, and Christo Wilson. "Recommended For You" A First Look at Content Recommendation Networks. 2016.

[18] beautifulsoup4. `https://pypi.org/project/beautifulsoup4/`.

[19] Steven Bellovin. On the brittleness of software and the infeasibility of security metrics. *IEEE Security & Privacy*, (4), 2006.

[20] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology–LATINCRYPT 2012*, pages 159–176. Springer, 2012.

[21] Joshua J. Bloch. How to design a good API and why it matters. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 506–507. ACM, 2006.

[22] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *J. Comput. Secur.*, 13(3):347–390, May 2005.

[23] Ethan Bommarito and Michael Bommarito. An empirical analysis of the python package index (pypi). *arXiv preprint arXiv:1907.11073*, 2019.

[24] Reinhardt Botha, Steven Furnell, and Nathan Clarke. From desktop to mobile: Examining the security experience. *Computers & Security*, 28:130–137, 05 2009.

[25] José González Cabañas, Ángel Cuevas, and Rubén Cuevas. Unveiling and quantifying facebook exploitation of sensitive personal data for advertising purposes. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 479–495, Baltimore, MD, 2018. USENIX Association.

[26] Carlos Castillo, Debora Donato, Aristides Gionis, Vanessa Murdock, and Fabrizio Silvestri. Know your neighbors: Web spam detection using the web topology. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 423–430, New York, NY, USA, 2007. ACM.

[27] Frank Chang, Sam Kierniski, and Andrew Dubatokawa. Using gan's for detecting malware and malicious scripts. *Neural Networks & Machine Learning*, 1(1):2, 2017.

[28] Yizheng Chen, Panagiotis Kintis, Manos Antonakakis, Yacin Nadji, David Dagon, Wenke Lee, and Michael Farrell. Financial lower bounds of online advertising abuse. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 231–254, Berlin, Heidelberg, 2016. Springer-Verlag.

[29] Yizheng Chen, Yacin Nadji, Rosa Romero Gómez, Manos Antonakakis, and David Dagon. Measuring network reputation in the ad-bidding process. In *DIMVA*, 2017.

[30] Edith Cohen and Haim Kaplan. Proactive caching of dns records: Addressing a performance bottleneck. *Computer Networks*, 41(6):707–726, 2003.

[31] The MITRE Corporation. Common weaknesses enumeration. `https://cwe.mitre.org`.

[32] Crawlium (DeepCrawling): A Crawling Platform Based on Chrome (Chromium). `https://github.com/sajjadium/Crawlium`, 2020.

[33] N. Delessy-Gassant, E. B. Fernandez, S. Rajput, and M. M. Larrondo-Petrie. Patterns for application firewalls. In *Pattern Languages of Programs Conference (PLoP)*, 2004.

[34] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. Free for all! assessing user data exposure to advertising libraries on android. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* [4].

[35] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[36] DNS Prefetching. `https://dev.chromium.org/developers/design-documents/dns-prefetching`.

[37] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 257–268, New York, NY, USA, 2018. ACM.

[38] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In *Proceedings of the Internet Measurement Conference*, Vancouver, Canada, Nov 2014.

[39] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[40] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[41] Yoav Einav. Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, 2019.

[42] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. 2020.

[43] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 49–60. ACM, 2013.

[44] Fast load times: Techniques for improving site performance. `https://developers.google.com/web/fundamentals/performance/resource-prioritization`.

[45] Dennis F. Galletta, Raymond Henry, Scott McCoy, and Peter Polak. Web Site Delays: How Tolerant are Users? *Journal of the Association for Information Systems*, 5(1), 2004.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[47] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly 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*, pages 38–49. ACM, 2012.

[48] David Gibson, Jon Kleinberg, and Prabhakar Raghavan. Inferring web communities from link topology. In *Proceedings of the ninth ACM conference on Hypertext and hypermedia: links, objects, time and space—structure in hypermedia systems: links, objects, time and space—structure in hypermedia systems*, pages 225–234. Citeseer, 1998.

[49] Dan Goodin. https://arstechnica.com/information-technology/2018/01/malvertising-factory-with-28-fake-agencies-delivered-1-billion-ads-in-201 2018.

[50] YoavIlya Grigorik. Resource Hints, W3C Working Draft. https://www.w3.org/TR/resource-hints/, 2019.

[51] Saikat Guha, Bin Cheng, and Paul Francis. Challenges in measuring online advertising systems. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 81–87, New York, NY, USA, 2010. ACM.

[52] Md Ahsan Habib and Marc Abrams. Analysis of sources of latency in downloading web pages. In *WEBNET*, 2000.

[53] M. Hafiz. A collection of privacy design patterns. In *Pattern Languages of Programs Conference (PLoP)*, 2006.

[54] Munawar Hafiz, Paul Adamczyk, and Ralph E. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, 2007.

[55] Katie Hempenius. Resource hints. *The 2019 Web Almanac*, 2019.

[56] Thomas Heyman, Koen Yskout, Riccardo Scandariato, and Wouter Joosen. An analysis of the security patterns landscape. In *Third International Workshop on Software Engineering for Secure Systems, SESS 2007, Minneapolis, MN, USA, May 20-26, 2007*, page 3. IEEE Computer Society, 2007.

[57] Lin-Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged SSL certificates in the wild. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 83–97, 2014.

[58] Sebastian Hunt and David Sands. On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90, January 2006.

[59] Muhammad Ikram, Rahat Masood, Gareth Tyson, Mohamed Ali Kaafar, Noha Loizon, and Roya Ensafi. The chain of implicit trust: An analysis of the web third-party resources loading. In *The World Wide Web Conference*, pages 2851–2857, 2019.

[60] Soumya Indela, Mukul Kulkarni, Kartik Nayak, and Tudor Dumitraş. Helping johnny encrypt: Toward semantic interfaces for cryptographic frameworks. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 180–196, 2016.

[61] Soumya Indela and Dave Levin. Sound methodology for downloading webpages. In *Proceedings of IFIP/IEEE Traffic Measurement Analysis Conference (TMA)*, 2021.

[62] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. ADGRAPH: A Graph-Based Approach to Ad and Tracker Blocking. 2020.

[63] T. Johnson and P. Seeling. Desktop and mobile web page comparison: characteristics, trends, and implications. *IEEE Communications Magazine*, 52(9):144–151, Sep. 2014.

[64] Christopher Kern. Preventing security bugs through software design. `https://www.usenix.org/conference/usenixsecurity15/symposium-program/presentation/kern`, August 2015.

[65] Srinivas Krishnan and Fabian Monrose. Dns prefetching and its privacy implications: when good things go bad. 2010.

[66] Srinivas Krishnan and Fabian Monrose. An empirical study of the performance, security and privacy implications of domain name prefetching. 2011.

[67] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, J Alex Halderman, and Michael Bailey. Security challenges in an increasingly tangled web. 2017.

[68] C. Lever, R. Walls, Y. Nadji, D. Dagon, P. McDaniel, and M. Antonakakis. Domain-z: 28 registrations later measuring the exploitation of residual trust in domains. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 691–706, May 2016.

[69] C. Lever, R. J. Walls, Y. Nadji, D. Dagon, P. McDaniel, and M. Antonakakis. Dawn of the dead domain: Measuring the exploitation of residual trust in domains. *IEEE Security Privacy*, 15(2):70–77, March 2017.

[70] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 674–686, New York, NY, USA, 2012. Association for Computing Machinery.

[71] Dwayne Litzenberger. Pycrypto - the python cryptography toolkit. `https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.AES-module.html`.

[72] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the web's pki. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 183–196. ACM, 2015.

[73] Steve Mansfield-Devine. When advertising turns nasty. *Network Security*, 2015:5–8, 11 2015.

[74] Max-Emanuel Maurer, Doris Hausen, Alexander De Luca, and Heinrich Hussmann. Mobile or desktop websites?: Website usage on multitouch devices. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*, NordiCHI '10, pages 739–742, New York, NY, USA, 2010. ACM.

[75] Wei Meng, Ren Ding, Simon P. Chung, Steven Han, and Wenke Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* [4].

[76] Dimitris Mitropoulos, Panos Louridas, Vitalis Salis, and Diomidis Spinellis. Time present and time past: analyzing the evolution of javascript code in the wild. In *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2019.

[77] Jack Morse. `https://mashable.com/2018/04/03/congratulations-pop-up-ads-malware/#74djBJTktGqw`, 2018.

[78] Winai Nadee and Korawit Prutsachainimmit. Towards data extraction of dynamic content from javascript web applications. In *International Conference on Information Networking (ICOIN)*, 2018.

[79] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. 2016.

[80] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. 2019.

[81] Lily Hay Newman. `https://www.wired.com/story/pop-up-mobile-ads-surge-as-sites-scramble-to-stop-them/`, 2018.

[82] T Nidd, Thomas Kunz, and Ertugrul Arik. Prefetching dns lookup for efficient wireless www browsing. In *Proceedings of Wireless*, volume 97, pages 409–414.

[83] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. 2012.

[84] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E. Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J. Murdoch. Adblocking and counter blocking: A slice of the arms race. In *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*, Austin, TX, 2016. USENIX Association.

[85] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[86] Andy Patrizio. *Windows Server vulnerability disclosed by NSA*, 2020.

[87] Angelisa C. Plane, Elissa M. Redmiles, Michelle L. Mazurek, and Michael Carl Tschantz. Exploring user perceptions of discrimination in online targeted advertising. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 935–951, Vancouver, BC, 2017. USENIX Association.

[88] Karlis Podins and Arturs Lavrenovs. Security implications of using third-party resources in the world wide web. In *IEEE Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2018.

[89] Prabaharan Poornachandran, N Balagopal, Soumajit Pal, Aravind Ashok, Prem a U, and Manu R. Krishnan. *Demalvertising: A Kernel Approach for Detecting Malwares in Advertising Networks*, pages 215–224. 11 2017.

[90] Puppeteer: Headless Chrome Node.js API. `https://github.com/puppeteer/puppeteer`, 2020.

[91] Python requests. `https://pypi.org/project/requests/`.

[92] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* [4].

[93] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin RB Butler. Mo (bile) money, mo (bile) problems: analysis of branchless banking applications in the developing world. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 17–32, 2015.

[94] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 110–120, New York, NY, USA, 2016. ACM.

[95] T. Saridakis. Design patterns for fault containment. In *Pattern Languages of Programs Conference (PLoP)*, 2003.

[96] Bruce Schneier. Schneier on security. `https://www.schneier.com/crypto-gram/archives/1998/1015.html#cipherdesign`, 1998.

[97] Kyle Schomp, Mark Allman, and Michael Rabinovich. Dns resolvers considered harmful. 2014.

[98] M. Schumacher. Firewall patterns. In *EuroPLoP*, 2003.

[99] Markus Schumacher. Security patterns and security standards. In *EuroPLoP*, pages 289–300, 2002.

[100] C. Seifert, I. Welch, and P. Komisarczuk. Identification of malicious web pages with static heuristics. In *2008 Australasian Telecommunication Networks and Applications Conference*, pages 91–96, Dec 2008.

[101] Selenium. `https://www.selenium.dev/`.

[102] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. Why johnny still can't encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security*, pages 3–4. ACM, 2006.

[103] Ankit Singla, Balakrishnan Chandrasekaran, P Godfrey, and Bruce Maggs. Towards a speed of light internet. *arXiv preprint arXiv:1505.03449*, 2015.

[104] Sarah Sluis. `https://adexchanger.com/publishers/forced-redirect-ads-cost-publishers-money-blocking/`, 2018.

[105] P. Sommerlad. Reverse proxy patterns. In *Pattern Languages of Programs Conference (PLoP)*, 2003.

[106] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* [4].

[107] K. E. Sorensen. Session patterns. In *Pattern Languages of Programs Conference (PLoP)*, 2002.

[108] C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Core Series. Prentice Hall PTR, 2005.

[109] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, and Nazanin Magharei. Measuring and mitigating web performance bottlenecks in broadband access networks. 2013.

[110] Google Security Team. keyczar - easy-to-use crypto toolkit. `https://github.com/google/keyczar`, 2008.

[111] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 151–167, Washington, DC, USA, 2015. IEEE Computer Society.

[112] Liam Tung. *Windows 10 security: Are ads in Microsoft's own apps pushing fake malware alerts?*, 2019.

[113] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. Sok: Secure messaging. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 232–249. IEEE, 2015.

[114] Bart van Delft, Sebastian Hunt, and David Sands. Very static enforcement of dynamic policies. *CoRR*, abs/1501.02633, 2015.

[115] Natalija Vlajic, X.Y. Shi, Hamzeh Roumani, and Pooria Madani. Resource hints in html5: A new pandora's box of security nightmares. 2017.

[116] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. 2013.

[117] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with wprof. 2013.

[118] Alma Whitten and J Doug Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *USENIX Security Symposium*, volume 348, pages 169–184, 1999.

[119] X-DNS-Prefetch-Control. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-DNS-Prefetch-Control`.

[120] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Pattern Languages of Programming Conference (PLoP)*, volume 51, page 61801, 1997.

[121] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. Does organizing security patterns focus architectural choices? In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 617–627. IEEE, 2012.

[122] ZBrowse. `https://github.com/zmap/zbrowse`, 2017.

[123] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollsén, and Martin Lopatka. The representativeness of automated web crawls as a surrogate for human browsing. In *Proceedings of The Web Conference 2020*, WWW '20, page 167–178, New York, NY, USA, 2020. Association for Computing Machinery.

[124] Liang Zhang, Dave Choffnes, Tudor Dumitras, Dave Levin, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of SSL certificate reissues and revocations in the wake of Heartbleed. In *Proceedings of the Internet Measurement Conference*, Vancouver, Canada, Nov 2014.

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