BUILDING TRUST IN THE ONLINE ECOSYSTEM THROUGH EMPIRICAL EVALUATIONS OF WEB SECURITY AND PRIVACY CONCERNS

A Dissertation
Presented to
The Academic Faculty

By

Dhruv Kuchhal

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Cybersecurity and Privacy
College of Computing

Georgia Institute of Technology

August 2023

BUILDING TRUST IN THE ONLINE ECOSYSTEM THROUGH EMPIRICAL EVALUATIONS OF WEB SECURITY AND PRIVACY CONCERNS

The more I read, the more I acquire, the more certain I am that I know nothing.

Voltaire

This dissertation is dedicated to my parents, for whom my education always came first.

ACKNOWLEDGMENTS

First and foremost, I want to express my deepest gratitude to my advisor, Frank Li, for his exemplary mentorship, endless patience, and steadfast support throughout my Ph.D. journey. He has been my greatest cheerleader, and I feel immensely fortunate to have him on my side. This dissertation would not have been possible without his timely, thorough, and insightful feedback. Over the last three years of working together closely, his profound expertise, knowledge, and ability to provide me with constructive criticism have motivated me tremendously in becoming the researcher that I am today.

I am thankful to my collaborators Adam Oest and Muhammad Saad for enriching my research with their complementary perspectives. Working with Adam's team at PayPal over the last two summers has been an amazing opportunity that has greatly influenced my approach to studying real-world security. I have learned a lot from observing the various challenges and opportunities of applying security research in industry.

I would also like to thank Georgia Tech and the School of Cybersecurity and Privacy for providing an engaging academic environment, resources, and the generous financial support that enabled my research. I am grateful for Alexandra Boldyreva's prompt guidance in navigating the Ph.D. program, and all the dedicated staff for their assistance throughout.

As an international student in the U.S., I treasure the friendships that feel like family to me. I am grateful for the amazing support system of family at GT: Swetha, Saurabh, and Sashank, as well as Aviral and Aditya outside GT who always root for me. Having their company, love, and support, has made this journey delightful.

I owe a huge debt of gratitude to Ponnurangam Kumaraguru and Michelle Mazurek, who welcomed me into their research groups when I was just an aspiring undergrad with lofty ambitions. Elissa Redmiles, the most amazing mentor one could ask for, inspired me to pursue a Ph.D. and helped me achieve it. Pursuing a Ph.D. was beyond my wildest imagination, and it became a reality because of them.

This dissertation is dedicated to my parents, who have painstakingly instilled the importance of education in me. Without my mom's ever-present and sage counsel, I would certainly not have reached this point in my life and career. I am deeply grateful for their unconditional love and support. Last but not least, remembering my grandfather, who believed in me in ways no one else did, and taught me to live by the mantra "work *is* worship".

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Security and privacy concerns for the Web can manifest in practice due to inadvertent misconfigurations, or intentionally be considered an acceptable risk to promote better usability or compatibility. Our community needs to monitor when these concerns become realistic threats that erode trust in the ecosystem, so that appropriate defenses can be adopted to mitigate the threats while minimizing the decline in usability. To take a meaningful next step towards improving the state of trust and safety for users on the Web, it is imperative to first bridge the gap between theory and practice by corroborating with evidence the extent to which such weaknesses exist on the Web today. This dissertation demonstrates how large-scale empirical studies help uncover such gaps in real-world implementations.

Trust and safety go both ways between users and online platforms. To study the security and privacy concerns for platforms, I present measurement techniques to (i) analyze the practical security provided by passwordless authentication to securely authenticate users when deployed in the real world, and (ii) evaluate the efficacy of YouTube's anti-abuse measures to protect their content from manipulation by malicious actors in terms of organically produced fake engagement. On the other hand, for users to trust online services with their data, they too expect a certain level of privacy when online. To that end, my work explores the privacy implications of (i) local network communications by popular websites, and (ii) invasive access to a user's Web activity on popular Android apps.

Through the studies presented in this dissertation, I find that measurement methods, such as the ones I present, are effective at highlighting the gaps between secure configurations that exist in theory, and real-world implementations which seldom follow best practices. Across various contexts, I learnt that the gaps exist because Web services optimize for lower user friction, without taking full cognizance of the risks involved. Ultimately, I demonstrate that for broader adoption of recommendations made by security practitioners in theory, our community needs increased operational insights of real-world systems.

CHAPTER 1
INTRODUCTION

The ubiquity and complexity of today's digital ecosystems underscore the significant role they play in our personal lives, business operations, and society as a whole. In an era where data breaches, manipulation of public discourse via social media, and disregard for the privacy of users have become commonplace, the efficacy of security, privacy, and anti-abuse measures on the Web demand comprehensive scrutiny. In this pursuit, this dissertation provides a thorough empirical examination of real-world implementations, with a specific focus on authentication, online abuse, and Web privacy.

Implementations of real-world systems on the Web are constantly evolving, with new technologies and protocols emerging regularly, often with a focus on enhancing security and privacy. However, the adoption of new technologies does not always necessarily favor the most secure or privacy-respecting behavior, particularly when incentives misalign. Therefore, it is crucial for practitioners to continually monitor the effective security and privacy of a system, especially as they consider the development and adoption of next-generation protocols. The identification and rectification of these vulnerabilities is important not just for the intrinsic value of improved security and privacy, but also because they serve to increase trust within the ecosystem. As individuals, businesses, and governments become increasingly dependent on digital platforms, the need for mutual trust in the integrity, security, and privacy between platforms and users becomes ever more critical. With an aim to inspire improvements in theory and practice, I present empirical perspectives about concerns spanning both the platforms and users. Considering platforms, I evaluate real-world implementations of an upcoming authentication protocol and anti-abuse measures at a major Web service, in chapter 3 and chapter 4 respectively. In chapter 5 and chapter 6, I investigate privacy concerns of users visiting the Web using desktop browsers and non-browser mobile apps.

Theoretical recommendations, while valuable, do not suffice in isolation. Without empirical evaluations of the effectiveness of real-world implementations, we cannot achieve a complete understanding of the systems' effectiveness and vulnerabilities, nor can we come up with the next generation of guidelines. These evaluations are essential for ensuring that the systems evolve to meet the shifting landscape of threats and challenges, and this thesis aims to underscore their necessity. Ultimately, this thesis contributes empirical approaches to bridge the gap between theoretical Web security and privacy concerns and their practical manifestations, building towards a safer and more trustworthy online ecosystem. Below I will briefly introduce each of the studies, the technical contributions made by each, and the unique challenges that they overcome.

1.1 FIDO2-based Passwordless Authentication

Authentication plays a critical role in asserting the user's identity to Web services and establishing trust. FIDO2 is a suite of protocols that combines the usability of local authentication (e.g., biometrics) with the security of public-key cryptography to deliver passwordless authentication. It eliminates shared authentication secrets (i.e., passwords, which could be leaked or phished) and provides strong security guarantees assuming the benign behavior of the client-side protocol components. However, when this assumption does not hold true, such as in the presence of malware, client authentications pose a risk that FIDO2 deployments must account for. FIDO2 provides recommendations for deployments to mitigate such situations. Yet, to date, there has been limited empirical investigation into whether deployments adopt these mitigations and what risks compromised clients present to real-world FIDO2 deployments, such as unauthorized account access or registration. In chapter 3, I fill in the gap by: 1) systematizing the threats to FIDO2 deployments when assumptions about the client-side protocol components do not hold, 2) empirically evaluating the security posture of real-world FIDO2 deployments across the Tranco Top 1K websites, considering both the server-side and client-side perspectives, and 3) synthesizing the mitigations that the ecosystem can adopt to further strengthen the practical security provided by FIDO2. Through my investigation, I identify that compromised clients pose a practical threat to FIDO2 deployments due to weak configurations, and known mitigations exhibit critical shortcomings and/or minimal adoption. Based on my findings, I also propose directions for the ecosystem to develop additional defenses into their FIDO2 deployments.

1.2 YouTube View Fraud

In addition to secure authentication, anti-abuse measures are crucial for maintaining a trustworthy online environment for users. As engagement-driven social media platforms take center stage in modern discourse, concerns regarding their ability to defend against abuse are worth investigating. Unfortunately, engagement metrics are often susceptible to manipulation and expose the platforms to abuse. Video view fraud is a unique class of fake engagement abuse on video-sharing platforms, such as YouTube, where the view count of videos is artificially inflated. There exists limited research on such abuse, and prior work focused on automated or bot-driven approaches. In chapter 4, I explore organic or human-driven approaches to view fraud, conducting a case study on a long-running YouTube view fraud campaign operated on a popular free video streaming service, 123Movies. Before 123Movies users are allowed to access a stream on the service, they must watch an unsolicited YouTube video displayed as a pre-roll advertisement. Due to 123Movies' popularity, this activity drives large-scale YouTube view fraud. In this study, I reverse-engineer how 123Movies distributes these YouTube videos as pre-roll advertisements, and track the YouTube videos involved over a 9-month period. For a subset of these videos, I monitor the view counts and metrics for their respective YouTube channels over the same period. My analysis reveals the characteristics of YouTube channels and videos participating in this view fraud, as well as the efficacy of such view fraud efforts.

1.3 Local Network Communication by Popular Websites

Trust between users and online platforms goes both ways. As much as online platforms need to establish trust in a user's online identity and engagement, users too expect a certain level of privacy when online. In chapter 5, I explore privacy implications of local network communications by popular websites. Modern webpages are amalgamations of resources requested from various public Internet services. In principle though, webpages can also request resources from localhost and devices in the LAN, providing a degree of internal network access to external entities. Prior work has demonstrated how this access can be used for supporting Web attacks, particularly for profiling and fingerprinting users. I empirically investigate if and how popular websites are interacting with their visitors' localhost and LAN resources, and compare the behavior observed to that from known malicious websites. I crawl and monitor the network requests made by the landing pages of domains in the Tranco top 100K domains as well as $\sim$145K websites that are known to be related to malware, phishing, or abuse. For both popular and malicious sites, I find over 100 sites in each category making requests to internal network destinations, including several highly-ranked sites. My findings establish empirical grounding on the intentional and unintentional localhost and LAN network activities of real-world websites.

1.4 Third-Party Web Content in Android Apps

More than 65% Web traffic comes from mobile devices, according to some estimates [1]. Mobile browsers are not the only source for the traffic. Many mobile apps also display Web content, either by showing a Web page in full-screen as the app itself (i.e., hybrid app), or by opening Web links from within an app. The WebView class in Android has been a common way to show Web content in an app, but it has many security and privacy issues, especially when displaying third-party content. Custom Tabs (CTs), introduced in 2015, are the recommended alternative to show third-party content safely and efficiently. In chapter 6, I present a large-scale empirical study to examine if the top ∼146.5K Android apps use WebViews and CTs appropriately. I find that ∼55.7% of apps use WebViews, mainly to display ads, and ∼20% use CTs, primarily to integrate Facebook's Login SDK. I also find that popular SDKs, such as Google Firebase have switched to CTs, but some others, such as Stripe that should switch have not done so yet. Moreover, I semi-manually analyze the top 1K apps to identify apps which circumvent Android's default behavior of opening third-party URLs in a browser. I discover 10 apps that use WebViews to show external Web pages inside their app. I do a careful manual analysis of these 10 apps using detailed measurements, and reveal unsolicited manipulation of WebViews to enable behaviors such as payments, injecting ads and conducting network measurement from end-user devices. Ultimately, my work takes a first step towards an improved understanding of how mobile apps display third-party Web content, and its real-world security and privacy implications.

CHAPTER 2
RELATED WORK

This section describes the prior literature for the different empirical studies presented in this dissertation.

2.1 FIDO2-based Passwordless Authentication

Since FIDO2's official launch in 2018, a number of studies have investigated the usability and security properties of its protocols as a secure replacement for password-based authentication. On the usability side, several works [2, 3, 4, 5] have conducted user studies to understand how users engage with FIDO2 authentication workflows. Overall, while these works have identified valuable usability properties complemented by a general willingness among users to adopt FIDO2, a few works have also identified user concerns and misconceptions. A common user concern was identified as complexity in account recovery under FIDO2, which can drive users back to using passwords [4, 5].

On the security side, Barbosa et al. [6] provide a cryptographic security analysis of both FIDO2's WebAuthn 1 and CTAP2 protocols, confirming the authentication security of WebAuthn. Among the four security notions proposed in [6], CTAP2 was observed to have the weakest security model. Bindel et al. [7] expanded on this initial analysis to consider WebAuthn2 and CTAP 2.1, demonstrating provable security for both protocols. Meanwhile, Guan et al. [8] and Jacomme and Kremer [9] both applied ProVerif to formally analyze the assumptions that FIDO2 require for its security properties to hold. Similar to prior efforts, they confirmed the security properties of FIDO2 under explicit assumptions but identified that these properties do not necessarily hold with malicious client components. These works demonstrate FIDO2's security, but only under the assumptions of FIDO2's threat model, which does not consider malicious client components. Indeed, several works [10, 11] have identified social engineering-based attacks on FIDO that exploit weaknesses not accounted for by FIDO assumptions.

In my work, I empirically evaluate the consequences of malicious client components. While prior work demonstrated such a threat to FIDO2 theoretically exists, in my work, I empirically evaluate how this threat manifests in practice.

2.2 YouTube View Fraud

The YouTube platform has been widely studied over the past decade, from analysis of user-generated video content [12] to identification of topic-based communities [13]. More recently, several studies have also detected and characterized various kinds of abuse permeated via YouTube. For example, researchers have investigated the marketing of fraudulent products and services on YouTube [14, 15].

Most relevant to my study, several prior works have considered the manipulation of video engagement metrics. Dutta et al. studied collusion networks that deliver fake collusive likes and subscribers to videos and channels, although they did not consider view fraud [16]. Li et al. investigated a behavior-based clustering approach to detecting automated fake engagement on YouTube [17]. Miriam et al. evaluated the detection systems of five online video platforms (including YouTube, Vimeo, and Dailymotion) and found that YouTube better detected fake views than other platforms [18]. They also observed that YouTube penalizes its videos' public view counters after detecting view fraud activity.

However, there has been little empirical investigation into real-world video view fraud campaigns. What limited work does exist considered detecting and characterizing view bots on live streaming platforms like Twitch [19]. Thus, my work expands upon the literature by providing empirical grounding on video view fraud in practice, in particular considering an organic form of view fraud, rather than the automated view fraud considered in prior work.

2.3 Local Network Communication by Popular Websites

Here I summarize the prior work on developing Web-based methods for discovering, fingerprinting, and attacking LAN devices, along with the prior work empirically evaluating the security and privacy behaviors of websites.

2.3.1 Web-based LAN Attacks

Web-based methods for identifying, profiling, and attacking devices on a user's LAN have existed for years. Over a decade ago, Lam et al. [20] demonstrated how a malicious webpage could scan for and exploit vulnerabilities on LAN devices, potentially propagating worms and profiling users. Grossman and Niedzialkowski [21] similarly presented a Web-based method for scanning the local network using the `onerror` JavaScript handlers of image resources. Stamm et al. [22] leveraged this scanning method to hijack the DNS configurations of local routers. More recently, Gallagher [23, 24] refined Web-based LAN scanning by using WebSockets and Web Workers, and Lee et al. [25] did likewise leveraging HTML5 AppCache to identify cross-origin resource statuses. Acar et al. [26] further built upon these advancements to demonstrate how a webpage can discover and interact with local IoT devices that expose HTTP interfaces, potentially triggering malicious commands as well as fingerprinting the network. Beyond these academic studies, several different Web-based network and port scanners have been developed, demonstrating the feasibility of Web-based targeting of LAN devices [27, 28, 29, 30, 31, 32, 33].

These works highlight the potential threats that websites could pose if given internal network access. However, we currently lack awareness of whether websites actually leverage this access in practice. My study establishes empirical ground on the localhost and LAN network behaviors of modern websites.

2.3.2 Measurements of Website Security and Privacy Behavior

Numerous empirical studies have evaluated the security and privacy behaviors of websites. Prior work has examined how websites have deployed various security mechanisms. For example, Felt et al. [34] tracked the deployment and configuration of HTTPS across the Web. Similarly, Stark et al. [35] evaluated the adoption of certificate transparency on websites. Both Calzavara et al. [36] and Roth et al. [37] examined the content security policies deployed by websites, evaluating their effectiveness in practice.

On the privacy side, existing studies have analyzed the real-world data collection and Web tracking mechanisms deployed by websites. As an example, Englehardt et al. [38] developed an open-source Web privacy measurement tool and used it to measure different types of tracking on the top 1 million websites. Similarly, Acar et al. [39] uncovered how websites were using surreptitious techniques for profiling and tracking users longitudinally, including through canvas fingerprinting, evercookies, and cookie syncing. Snyder et al. [40] investigated the browser features used by popular websites for advertising and tracking purposes.

As with these prior studies, my work seeks to understand the security and privacy behaviors of websites. Expanding beyond the existing explorations though, I explore how websites interact with localhost and LAN network services, and the security and privacy implications of this behavior.

2.4 Third-Party Web Content in Android Apps

While WebView-based mobile apps afford a seamless and enriched user experience with relatively low development demands, they bear the risk of potential security and privacy concerns. Apps utilizing WebViews can inject JS code into webpages, enable bidirectional communication between native Java objects and webpages through JS Bridges, and control network requests. Over a decade ago, Luo et al. introduced the threat model for WebViews, illustrating how they could serve as attack vectors for webpages within malicious apps or for malicious webpages embedded in benign apps [41]. Since then, numerous static analysis approaches have been proposed to detect data leaks and code injection attacks through JS Bridges [42, 43, 44]. Additionally, vulnerabilities and bugs introduced by interactions with the JS Bridge [45, 46, 47] and native app event handlers [48] have been identified. WebViews have also been manipulated to expand traditional Web-based attacks, such as phishing [49] and browser fingerprinting [50]. Moreover, malware has been discovered exploiting the capabilities of WebViews to evade detection [51]. As apps have become more complex, the threat landscape has also evolved. Recently, Zhang et al. examined how identity confusion in WebView-based apps can expose privileged APIs of one app component to an unrelated component [52].

WebViews possess such expansive access rights primarily because they are intended to present trusted first-party content. However, there's no restriction that prevents them from loading third-party content. Tuncay et al. suggested a policy-based strategy, modeled after the Same-Origin-Policy, which would permit granular control over Web origins within WebViews [53]. Nevertheless, in practice, Zhang et al. identified instances of inter-party manipulation of Web resources in WebViews [54]. The majority of these manipulations served benign purposes like customizing Web services or enabling hybrid functionality. Yet, there were a handful of malicious instances aimed at stealing cookies or user credentials. It is worth noting here that Zhang et al.'s approach uses static analysis techniques to generate the URLs passed into the WebView. This means their analysis is constrained to URLs that already exist in the app in some form prior to its execution. Therefore, their findings do not extend to In-App Browsers (IABs) implemented with WebViews. In other words, if a user clicks a third-party link sent by a friend via a social media app and that link opens in a WebView, this behavior is not included in their study because the URL was not originally present in the app.

In order to mitigate these risks, Android encourages employing CT for any non-first-party or untrusted Web content [55]. Beer et al. recently proposed that the callback mechanism of CTs could be exploited as a cross-site oracle [56], but their potential attack would still have much less impact than the ones demonstrated for WebViews. To date, there has been limited empirical investigation into the real-world adoption of CT. Zhang et al. studied IAB implementations in 25 popular apps from a usable security perspective [57]. Their findings indicate that IABs leveraging CT generally exhibit secure behaviors, whereas every single IAB utilizing WebViews presented at least one insecure behavior. Through my research, I offer an extensive characterization of WebView and CT adoption, drawing from a large-scale evaluation of popular Android apps. Furthermore, I contribute to a comprehensive understanding of privacy invasive behavior within WebView-based IABs found in a smaller sample of widely used apps.

CHAPTER 3
SECURITY POSTURE OF REAL-WORLD FIDO2 DEPLOYMENTS

FIDO2 is a second-generation suite of protocols supporting passwordless authentication. It relies upon public-key cryptography, where the cryptographic keys, their access control, and cryptographic operations are ideally secured by a hardware-based Trusted Execution Environment (TEE). For access control to cryptographic keys, FIDO2 relies on the authentication performed locally at the user's device, supporting methods such as biometrics (e.g., fingerprint or facial scanning), and local PINs. FIDO2 has been designed and formally verified [7, 6, 8, 9] as providing strong authentication security guarantees (with prior work also demonstrating promising usability outcomes [2, 3, 4, 5]), all without the use of any shared authentication secrets (i.e., passwords). FIDO2's promise has driven real-world adoption, with support by major browsers (e.g., Chrome, Firefox, Edge, Safari), OSes (e.g., Windows, Linux, macOS, iOS, Android), and online services (e.g., Microsoft, Google, Facebook, PayPal, and eBay).

By avoiding shared secrets, FIDO2 directly eliminates the threat of credential phishing and data breaches: two of the primary credential theft vectors [58]. However, malware remains a notable threat, which has also plagued password authentication for decades (e.g., keyloggers are often used to steal passwords).

As FIDO2 becomes more prominent, attackers will naturally be incentivized to prioritize this remaining attack surface. The documentation of FIDO2 [59] also recognizes the threat of malware and thus recommends configurations that help mitigate the impact of malware. However, to date, there has been limited investigation into whether FIDO2 deployments adopt these mitigations, and what risks compromised clients pose in practice.

FIDO2's strong security guarantees also have the potential to bias the risk assumptions made by practitioners, as these guarantees depend fundamentally on the integrity of the local client operations, and may not completely hold if client-side components are malicious, compromised (as proven by formal analysis of the protocol [8, 9]), or misconfigured. Moreover, FIDO2 does not inherently eliminate the risk of social engineering: in section 3.6.2, we show that a practical social engineering attack could compromise a sensitive action such as an online payment transaction. This motivates the need for additional controls alongside FIDO2 authentication—such as in-browser integrity checks and authenticator attestation—to adequately secure such interactions.

In this work, we investigate the existing configurations of FIDO2 deployments, the degree to which they adopt the recommended mitigations, and the extent to which malicious client software and social engineering attacks can impact them. We consider the case where legitimate clients are compromised, both when the compromise occurs during FIDO2 registration and when it occurs only during authentication after a legitimate user has successfully registered.

We cover the background for FIDO2 in section 3.1 and section 2.1, describe our threat model in section 3.2, and our measurement method in section 3.3. In section 3.4 section 3.6, we provide three key contributions across the aforementioned scenarios:

1. We systematize the threats to FIDO2 from malicious client components, expanding beyond prior work and FIDO2 documentation. As part of our threat characterization, we identify and disclose a unique social engineering attack and browser vulnerability through which user-level malware can trick users into authenticating sensitive actions without their knowledge.

2. Using the FIDO2 documentation, we synthesize the potential mitigations that FIDO2's stakeholders can implement to reduce the risk of unauthorized account access posed by compromised clients and further strengthen FIDO2's security in practice.

3. We collect a snapshot of FIDO client authentication telemetry from a large financial service provider, and longitudinal measurements of FIDO Alliance's own database of

authenticator metadata, to characterize the available FIDO authenticators. Building upon our characterization, we analyze the configurations of FIDO2 deployments in the Tranco Top 1K websites and evaluate their susceptibility to the threats we discuss.

Through our investigation, we find that real-world FIDO2 deployments are broadly not configured to identify compromised FIDO2 clients, and hence are vulnerable to potential attacks. For some threats, improved configurations and adopting recommended mitigations could help secure FIDO2 deployments; for other issues, we identify the lack of or significant shortcomings with existing mitigations. Grounded in these findings, we identify long-term directions that various stakeholders of the FIDO2 ecosystem can pursue for broader impact, as well as immediate measures that FIDO2 deployments can adopt to reduce their risk exposure. Ultimately, our study is a first step towards developing defenses for practical attacks on FIDO2 deployments, which is particularly salient at this time as the protocol becomes increasingly adopted.

3.1 Background on FIDO2

FIDO2 provides public-key cryptographic protocols for authentication, using client-side FIDO2 components that interact with an online service supporting FIDO2, called the Relying Party (RP). At a high level, when a user wishes to authenticate with an RP via FIDO2, they use the client components to execute a registration protocol with the RP that results in the generation of a public/private cryptographic key pair (the FIDO2 credentials) that is stored on the client, where the public key is shared with the RP. This credential is bound to the specific user and RP, and access control to the private key is maintained by the client components, permitting key use only upon successful user authentication on the local device (e.g., via biometrics or a PIN). For subsequent authentication attempts with the RP, the client components perform a challenge-response cryptographic protocol with the RP, after locally authenticating the user. In addition to the traditional challenge-response protocol, FIDO2 also allows for RPs and clients to build custom functionality through FIDO2 extensions [60], where extension data is linked to and stored securely with the FIDO2 credential.

Client-side components of the family of FIDO protocols consist of the Authenticator, the Authenticator Specific Module (ASM), and the FIDO client. The Authenticator is the component that stores the cryptographic key materials and supports cryptographic operations, while the ASM provides a standardized software interface to the Authenticator. FIDO clients are the software clients that interact with the Authenticator and user agents during authentication. There is a wide variety of FIDO authenticator implementations, including software emulations of physical authenticators that are called virtual authenticators. Virtual authenticators by design do not provide any security guarantees, as they are primarily designed for testing and development use-cases.

The FIDO Alliance has published three specifications: the deprecated *Universal Second Factor (U2F)*, which describes the framework for using FIDO authenticators as a second factor [61], *Universal Authentication Framework (UAF)*, which is a framework for passwordless authentication [62], and the *Client to Authenticator Protocols (CTAP)*, which specifies how OSes/browsers communicate with a compliant authenticator [63]. CTAP's latest version, CTAP2, is complementary to the W3C's Web Authentication (WebAuthn) specification, which defines an API for Web applications to use FIDO protocols [64]. WebAuthn supports communication by Web applications with both RPs and authenticators, whereas CTAP2 provides an interface between applications and authenticators. Together, CTAP2 and WebAuthn are known as FIDO2.

In this study, we focus on the two FIDO2 protocols in the UAF framework, as U2F is deprecated. However, since FIDO2 is backward compatible with U2F, it can still be used for multi-factor authentication (MFA), which we consider in scope for our study.

RP security mechanisms: When an authenticator is registered on an RP, it generates a new key pair (the "FIDO credential") and transmits the public key and the authenticator's unique model ID (AAGUID), among other information, to the RP. The authenticator signs this output with its attestation private key, which is part of a key pair specific to the device model burned into the device at the time of manufacturing. The attestation certificates associated with the attestation keys are also attached with the output, which chain to a trusted root certificate, conveying the provenance of the authenticator to the RP. This process cryptographically proves that a user has a specific model of authenticator when they register [65].

An RP is responsible for maintaining a FIDO Server that stores the public key credentials of users, and a database of authenticator metadata and trust anchors [59] described as follows:

- Authenticator metadata can be derived from FIDO Alliance's Metadata Service (MDS), which contains standardized information describing the characteristics of an authenticator. RPs can use this information to make interoperability or risk decisions. For example, it includes an authenticator's FIDO certification level, which indicates its attack-resistance [66]. There are five possible certifications, in increasing order of attack-resistance, namely L1 [67], L1+ [68], L2 [69], L3 [70] and L3+ [71]. In brief, the authenticators which are resistant to malware and more sophisticated OS, circuit, or chip-level attacks, owing to the presence of TEEs, qualify for L2 or above certification, whereas authenticators lacking TEEs qualify for an L1/1+ certification. The certification is awarded by the FIDO Alliance based on their proprietary security evaluation tests. We note here that all authenticators of L2 or higher consist of hardware-based TEEs, but not all authenticators consisting of hardware-based TEE are awarded L2 or above.

- Trust anchors can be derived from two primary sources, the MDS as well as known root CAs. For example, trusted certificate chains for Android's SafetyNet attestation can be found in MDS, whereas Apple publishes its own root CA for its devices [72].

After establishing the legitimacy of the authenticator based on trust anchors, the RPs can calculate the risk associated with an authenticator based on its security characteristics and decide whether to allow its registration [59].

At registration, **RPs can verifiably ascertain an authenticator's legitimacy** and the risk associated with it.

Client security mechanisms: FIDO UAF's design ensures that the FIDO credentials (i.e., cryptographic keys) can only be accessed by the user associated with the credential (through local user authentication on the user device), and by the same client application and ASM that performed the initial registration operation [73]. Conceptually, UAF specifies that when a new credential is generated, authenticators bind the private authentication key with a key provided by the ASM called the KHAccessToken, which is generated as a function of the AppID, PersonaID, ASMToken, and CallerID. AppID is an identifier for a particular user application (e.g., Android application); PersonaID is an identifier generated by the ASM to differentiate credentials registered on the same RP; ASMToken is a random ID unique to the ASM; CallerID is an identifier unique to the FIDO Client. KHAccessToken ensures that malware on the client device is unable to access keys previously registered by a legitimate application [74] (so long as the attacker lacks root OS access, in which case they can spoof these ID values). FIDO2's CTAP2 implements the equivalent of KHAcessToken via an authenticator activation PIN (authenticatorClientPIN) [63, 59].

3.2 Threat Model

Given our focus on understanding the impact of compromised FIDO2 client components, our threat model is centered on an attacker who is able to infect the client with malware. Such malware can be delivered remotely and at scale, permitting widespread attacks. This threat model is realistic as in practice, keyloggers have been widely used to steal user passwords [58]. We consider both a weaker threat model of malware with only user-level access (e.g., a malicious user application or browser extension) and a stronger threat model of malware that achieves elevated root access (such as through exploiting kernel vulnerabilities [75]). For user-level malware, we assume the integrity of OS-level security protections.

We note that a malware-infected client is already in a grave situation, as the attacker may have remote access to their victim's active sessions. However, if the attacker is able to leverage malware to bypass a FIDO2 authentication attempt, this affords significantly expanded capabilities through circumventing additional defenses (e.g., step-up authentication). Such capabilities can allow attackers to execute sensitive authentication-protected actions, such as changing account recovery settings to permanently take over an account or initiate arbitrary financial transactions (rather than hijacking only ones that a user legitimately initiates). As FIDO2 becomes more popular, attackers will naturally have even more incentive to exploit it with malware, especially without the ability to rely on phishing and credential leaks.

Malware on the client is a strong threat model which can already result in significant harm, including remote access to the user's active sessions online. **However, we focus on malware that enables an attacker to bypass a FIDO2 authentication attempt**, allowing an attacker to execute any sensitive authentication-protected action.

Our threat model does not include attacks on the FIDO2 protocol itself, which we assume is secure (aligning with prior work [7, 6, 8, 9]). We also assume that authenticators using TEEs that are certified by the FIDO Alliance at L2 or above provide a secure execution environment (per the certification criteria), such that cryptographic keys and operations cannot be leaked or tampered with. However, for authenticators of lower or no security certifications, we do not make such assumptions (as they have not been certified as providing such security properties). In addition, we do not investigate attack vectors beyond those of malicious software on the user device, including network-based attacks (FIDO2 and existing protocols such as TLS protect against such threats) and attacks directly against the RP (e.g., DoS attacks). Finally, we do not consider FIDO2 privacy concerns.

Prior works have considered similar threat models. For example, Eskandarian et al. proposed an architecture to run Web applications securely within a compromised browser and compromised OS [76]. However, the design changes proposed are not practical enough to implement at scale. On the contrary, FIDO2 is fast gaining adoption as it offers a wide range of authenticators at different levels of security and interoperability.

3.3 Measurement Method

We begin by discussing how we empirically evaluate real-world FIDO2 deployments, studying the ecosystem through the lens of each of its participants: (i) online services (RPs) in the Tranco Top 1K which deploy FIDO2, (ii) FIDO Alliance's MDS which plays a central role in providing metadata and trust anchors for authenticators, and (iii) authenticators observed on mobile devices in the wild. We describe the various datasets we collect, which we use to assess the vulnerability of RPs to malware-based threats.

3.3.1 Relying Parties (RPs)

Identifying RPs deploying FIDO2

Evaluating the behavior of RPs requires identifying the websites supporting FIDO2 logins, creating user accounts on those sites, and registering authenticators (typically in the account settings) to observe the RP's FIDO2 configurations. This process is challenging to fully automate, given the broad diversity in website designs and authentication workflows. Our initial exploration involved crawling the landing sites of the Tranco Top 100K domains (using the snapshot of February 26, 2022). We searched the JS resources loaded by each site for the presence of the call made to the Credential Management API (`navigator.credentials.create`) to create a WebAuthn credential (`publicKey`) [77]. We found the pair of strings present in JS resources loaded by 135 sites, out of which for 82 sites, it was found in enterprise Identity and Access Management SDKs that the sites had imported [78, 79]. Manual analysis of the remaining revealed that barring a few sites (e.g., GitHub), the strings were found in imported JS SDKs without the WebAuthn functionality being utilized by the site. Moreover, popular sites such as Google and PayPal, which we knew supported FIDO2 [80, 81], were not detected by this method. We also attempted automated search engine queries for FIDO2 support on a site, but our manual analysis often led to false positives, such as documents or posts on a site discussing FIDO2, despite the site itself not deploying it (e.g., a Q&A page discussing FIDO2 on a commercial site). Prior work also observed similar issues when investigating MFA based on security keys [82, 83].

Therefore, we use a manual approach for identifying RPs and creating accounts and apply a semi-automated approach for assessing RP FIDO2 configurations. Given the significant manual effort required, we choose to search for FIDO2 support among the domains listed in the Tranco Top 1K domains. First, we visit each domain in a browser and manually inspect the landing page for account creation/login pages. If a site supports public account creation, we then manually look for evidence of FIDO2 support on either the account creation pages or through the links on the first page returned by a Google search (using the query *[domain] "webauthn" OR "passwordless" OR "u2f" OR "uaf" OR "fido2" OR "security key"*). We note that for all sites we found supporting FIDO2, the top search result provided the relevant information on FIDO2 support.

Table 3.1 provides statistics on each stage of this search process. From the initial 1K domains, we found 585 domains with account login or signup pages (note, some domains did not host a website and thus did not load). Of these domains, we identified 85 supporting FIDO2's WebAuthn, many of which mapped to the same RP (e.g., multiple Microsoft domains all use the same RP). In total, we aggregated the FIDO2-supporting domains into 40 distinct RPs. However, not all of these RPs support consumer-facing account creation (e.g., enterprise or financial services), inhibiting our investigation of their configurations. Ultimately, we could create accounts on and evaluate 29 RPs across the Tranco Top 1K.

Table 3.1: Measurement Statistics for FIDO2 RPs evaluated.

Measurement Stage	#
1. Domains in Tranco Top 1K	1000
2. Domains/sites that load in a browser	841
3. Sites with account login/signup pages	585
4. Sites that support FIDO2 WebAuthn	85
5. Distinct RPs for FIDO2 sites	40
6. *Distinct RPs we can evaluate*	*29*

RP-Requested Authenticator Properties

To evaluate the requirements each RP imposes on authenticators registering on their platform, we manually create an account on each RP's website (arbitrarily choosing one site if an RP is associated with multiple), initiate authenticator registration, and parse the requirements indicated by the RP as part of the authentication challenge request (note here that we do not need to complete the registration). Specifically, the RP's client-side script calls `navigator.credentials.create()`, passing along data fields indicating the RP's criteria for allowed authenticators, the attestation type required, and the extensions supported [77]. In our manual analysis, we found that the client-side code calling the API was often obfuscated, inhibiting static analysis. Instead, we used Chrome's in-built JS debugger [84] to set a breakpoint at the function call, so we could manually inspect the request's parameters when we trigger a registration.

Table 3.2 lists the 29 RPs evaluated, the authenticators allowed, and the attestations required by each RP.

RP Allowlisting of Authenticators

Next, we investigate whether RPs place further restrictions on the authenticators they allow on their platforms by allowlisting certain authenticator/AAGUIDs based on metadata from MDS, as recommended by FIDO to maintain an acceptable level of assurance [59]. To that end, we attempt to manually register a credential on each of the 29 RPs found

Table 3.2: Authenticator Configurations requested by FIDO2 RPs.

(a) For RPs using FIDO2 for Passwordless Authentication.

Domain Rank (↓)	RP/Service	RP Authenticator Preferences			RP Acceptance of Virtual Authenticator Configurations			
		Authenticator Attachment	Attestation Requirement	User Verification	USB	BLE	NFC	Internal
26	Microsoft	platform	direct	✗	✗			
74	PayPal			discouraged	✗			✓CTAP2 ✗U2F
85	eBay	cross-platform		required	✓CTAP2, ✗U2F			
128	Mail.ru	platform	none	✗	✓CTAP2, ✓U2F			
517	BestBuy			required	✗			✓CTAP2 ✗U2F

(b) For RPs using FIDO2 for Multi-Factor Authentication.

Domain Rank (↓)	RP/Service	RP Authenticator Preferences			RP Acceptance of Virtual Authenticator Configurations			
		Authenticator Attachment	Attestation Requirement	User Verification	USB	BLE	NFC	Internal
1	Google	cross-platform	direct	✗	✓U2F, ✗CTAP2			
3	Facebook			✗	✓U2F ✓CTAP2			
6	Twitter		none	discouraged				
14	Yahoo		direct	preferred				
28	GitHub	✗	none	discouraged				
32	AWS	cross-platform	direct	preferred	✓U2F, ✓CTAP2			✗
35	WordPress	✗			✓U2F ✓CTAP2			
56	Chaturbate	cross-platform	✗	discouraged				
67	Dropbox	✗	none	preferred				
80	Cloudflare							
151	GoDaddy		✗	discouraged				
162	Aliyun	✗	direct					
192	Shopify	platform	✗					
202	AoL	cross-platform	direct	preferred				
277	Zoho							
296	Binance	✗						
321	Roblox		✗	discouraged				
419	Stripe	cross-platform			✓U2F, ✓CTAP2			✗
490	BofA		direct		✓U2F, ✓CTAP2			
553	NVIDIA	✗	none	required	✗			
607	GitLab	✗		discouraged	✓U2F ✓CTAP2			
656	Namecheap	✗	direct					
938	BitBucket	cross-platform						
984	Norton	✗	none	✗				

in section 3.3.1, using Chrome's built-in virtual authenticator [85] under different configurations. While there are other ways to instrument custom authenticators, such as Google's OpenSK [86], our goal is to choose an authenticator with the weakest security guarantees. Since Chrome's virtual authenticator is designed for testing WebAuthn applications, it stores the credentials (including private keys) in plaintext and allows one-click export of sensitive credentials. At registration, it provides an RP with a self-signed certificate issued by ‘Chromium Authenticator Attestation’. Since it is not registered with the MDS, its attestation does not chain to any trust anchor. Thus, RPs should not allow such an authenticator to register in practice, as an attacker could easily implement a malicious virtual authenticator, or attack credentials registered via a virtual authenticator if used in real-world settings.

In our experiment, we attempt to register the virtual authenticator under different configurations, including different protocols (CTAP2 or U2F) and transports (USB, BLE, NFC, or Internal). Whenever the RP required user verification, we enabled user verification in the virtual authenticator (which does not involve real user verification). In Table 3.2, we list whether we successfully registered a virtual authenticator under each configuration, for all RPs.

3.3.2 FIDO Metadata Service (MDS)

Given the central role that the MDS plays in maintaining trust and safety in the FIDO ecosystem, we conducted over a year long longitudinal study into the operation of the MDS, from September 16, 2021, to April 13, 2023. We recorded daily snapshots of the MDS, via their APIv3 [87], as they regularly update it with new entries. Our last snapshot listed a total of 160 authenticators, a 90% increase from the first snapshot. Out of the 160 authenticators, 120 authenticators were certified at Level 1, and only 7 at Level 2; the remaining are not yet certified. We identified that 117 authenticators support mere user presence (e.g., pressing a button) as a user verification mode, 64 authenticators support a local passcode that is collected outside the authenticator boundary, and 58 authenticators support no user verification at all. Note that an authenticator can support multiple user verification modes. We further characterize the MDS metadata provided about authenticators in Table 3.3.

Table 3.3: Change in FIDO MDS authenticator metadata statistics over time.

Authenticator Attribute		# authenticators in MDS snapshot	
		16 Sept. 2021	13 April 2023
Total # authenticators		84	160
# FIDO2 authenticators		34	98
FIDO Certification	**Not Certified**	15	33
	Level 1	64	120
	Level 2	5	7
UAF Version	**v1.0 (2014)**	49	107
	v1.1 (2017)	32	47
	v1.2 (2020)	3	6
Attestation Supported	**Full**	62	135
	Self-signed	18	18
	Full + Self-signed	2	5
	Attestation CA	2	2
Key Protection	**Hardware & Secure Element**	38	90
	Hardware & Secure Element & Remote Handle	21	27
	Hardware	4	17
	Software	9	10
	Hardware & TEE	8	10
	Hardware & Remote Handle	4	6
Transaction Confirmation Display	**None**	66	136
	Any	18	21
	HW	0	3
Extensions Supported	**None**	79	138
	hmac-secret	2	19
	credProtect	0	5
	fido.uaf.android.key_attestation	3	3
	txAuthSimple	0	2
	loc	0	1
User Verification Methods	**Presence**	79	117
	Passcode (external)	10	64
	None	8	58
	Fingerprint	31	50
	Passcode	18	27
	Faceprint	9	10
	Voiceprint	1	5
	Eyeprint	4	5
	Patterm	1	3
	All	2	2
	Handprint	1	2
	Location	0	1

3.3.3 Real-World User Authenticators

While MDS lists available authenticators and their properties, it does not provide any information about the distribution of authenticators used in the wild. To gain visibility into real-world authenticators, we partner with a large financial services RP to collect authenticator registration telemetry recorded during a FIDO2 passwordless authentication pilot involving live user traffic. The RP chose to run the pilot only for sessions from mobile browsers, and therefore the authenticator data we collect is limited to mobile devices. To the best of our knowledge, our study is the first to study a significant sample of real-world authenticators found on users' mobile devices from the lens of a large RP. Prior work conducted a user study on FIDO2 with 29 participants (with 18 iPhones and 11 Android devices) [2].

Using the RP's authentication server logs, for all users who were enrolled into passwordless authentication, we extracted the WebAuthn response payloads received from their authenticator, on two days chosen at random during the pilot. We collected a total of 126,608 payloads on July 20, 2022, and 74,270 payloads on August 23, 2022. We then parsed the payloads to extract the corresponding attestation data, which provides information on the authenticator itself. We call this dataset the *bulk WebAuthn data*. Due to technical limitations of the server's logging mechanism, we could only retrieve the first 4KB of the payload, and some response payloads were truncated and could not be parsed. We identified that authenticators attested by Android SafetyNet issue WebAuthn responses with payload exceeding 4KB, so we attribute the responses with truncated payloads as being SafetyNet-attested. To validate these assumptions about payload truncation, we collected a separate set of WebAuthn response data containing full payloads for 14,616 sessions randomly chosen in the last week of July 2022. We call this dataset the *sampled validation WebAuthn data*.

From the 200,878 WebAuthn responses in the entire bulk WebAuthn data, we were able to parse $\sim 72\%$ of responses as coming from Apple-attested authenticators, $\sim 1\%$ without any attestation, and 1 self-attested response. About 0.5% of registrations had Apple's attestation but also an empty AAGUID – we suspect that these are Apple Passkeys [88], based on the session HTTP user agents indicating iOS 16 devices and information from Apple developer forums [89]. For the other responses with Apple's attestation, the AAGUID was set to `f24a8...`, but we note that this AAGUID was not listed in the MDS. The anomalous self-attested authenticator was associated with sessions using an HTTP User Agent (potentially spoofed) indicating a Chrome browser on a Mac OS X device. As a sanity check, we verified that the HTTP User Agent for the remaining $\sim 27\%$ truncated responses did indicate Android devices. We did not observe any other attestations.

Using the sampled validation WebAuthn data, we observed that the responses consisted of $\sim 72\%$ attested by Apple, $\sim 27\%$ attested by Android SafetyNet, and $\sim 1\%$ without attestation. The stark similarity in the distributions of Apple-attested and unattested responses between the sampled and the bulk dataset gives us further confidence that the $\sim 27\%$ truncated responses in the bulk dataset should be correctly attributed to Android SafetyNet attestation.

3.3.4 Ethical Considerations

Our measurements evaluate FIDO2 deployments and their susceptibility to attacks. In conducting our measurements, we only test benign FIDO2 authentication attempts using different configurations on a test account, without inducing a high load on the online services and without causing any harm to the service or any real users. For identified issues, we have or are in the process of vulnerability disclosure to the relevant stakeholders in a position to employ remediations. The partner-RP telemetry we analyzed did not contain any personally-identifying information.

3.4 Practicality of Malware Threats

FIDO2 assumes a TEE environment for providing its security guarantees and defending against malware attacks. Here, we analyze our data on authenticator characteristics to identify whether this assumption holds true in practice for available authenticators and to determine whether malicious client components pose a realistic threat to FIDO2.

Using the latest snapshot of MDS (from section 3.3.2), we find that only 7 out of 160 (4%) authenticators have L2 certification, which the FIDO Alliance has defined as offering malware resistance (with no authenticators certified at a higher level). As noted in section 3.1, it is possible though for an authenticator to have secure hardware but still not be certified as malware-resistant (L2). To determine an upper bound on the population of such authenticators, we analyzed the distribution of authenticators by the security properties of their keystores. We find that $\sim 93\%$ of authenticators are listed as backed by either TEE, hardware, or secure element keystores, while the remaining $\sim 7\%$ authenticators are backed by software-only keystores. Therefore, at least the $\sim 7\%$ authenticators are likely vulnerable to malware, while the others may provide some malware resistance, based on other properties such as mode of attachment.

96% of authenticators available today did not receive FIDO Alliance's malware-resistance certification, and thus are potentially vulnerable to malware-based attacks.

Beyond considering the distinct authenticators available, we also evaluate hardware-backed authenticators in our real-world user authenticator data from section 3.3.3 (recall though that our client authenticator data is strictly from mobile clients of our partner RP). Specifically, for authenticators attested by Android's SafetyNet, the attestation information contains a field labeled `evaluationType` which indicates whether SafetyNet's device integrity evaluation is based on hardware-backed security features (`evaluationType` = `HARDWARE_BACKED`). We found that ∼94% of attestations in the SafetyNet WebAuthn sample were hardware-backed, indicating that a non-trivial minority of authenticators lack hardware support.

Furthermore, for hardware-backed SafetyNet attestations, ∼0.25% indicated compromise (SafetyNet's device integrity check failed) and ∼13% of the non-hardware-backed

SafetyNet attestation indicated compromise. For the devices that were detected as being compromised, SafetyNet further specified that $\sim$49% were detected having an unlocked bootloader, $\sim$10% a custom ROM, and $\sim$5% having both. Due to the sensitivity of customer data, our insights from this data are limited to aggregate statistics, however, our partner RP did indicate that they flagged evidence of suspicious scripted activity for the sessions from these compromised devices. They noted a distinctly high volume of logins on multiple accounts from these devices, many using proxy IP addresses. **Thus, we observe that FIDO2's assumption of hardware-backed security does not hold for many authenticators, and we already find evidence of malicious activity on compromised devices.**

3.5 Registration-Phase Attacks

As FIDO2 gains adoption, it is important to consider the threats when an authenticator is first registered, as this authenticator would be trusted for subsequent login attempts to the account. In this section, we expand upon existing FIDO2 documentation [59, 90, 64] to systematize realistic threats to FIDO2 deployments involving malicious authenticator registration. We consider two attack scenarios, malicious authenticators that are registered to a legitimate user's account, and a legitimate but vulnerable authenticator registered to the user's account which may be subsequently compromised. For each threat, we discuss what mitigation actions could be taken, and then empirically evaluate the extent to which such mitigations can or have been adopted by real-world FIDO2 deployments.

3.5.1 Malicious Authenticator

Here we consider the threats where the attacker aims to register their malicious (i.e., attacker-controlled) authenticator to a user's account. This allows the attacker to authenticate into the account at will, effectively taking over the account. To gain access to a user's account, the attacker could either utilize existing account takeover techniques to gain access themselves, or they could deploy malware to the user's device which already has the access required.

Traditional Account Takeover

Attack Description: An attacker can exploit non-FIDO2 credentials, such as passwords (e.g., via phishing), to take over an account and register a malicious authenticator. Pre-hijacking attacks (e.g., Sudhodanan et al. [91]) could also be leveraged to register a malicious authenticator and gain persistent access to a user's account.

Measurement of Mitigation Adoption: Given the importance of initial authenticator registration, FIDO2 best practices recommend that RPs employ user verification methods, such as challenge-response verification to a user's phone or email, to ensure that the actual user is registering an authenticator. Prior work has shown that similar login challenges are effective in limiting account takeover itself [90, 92]. However, while registering our authenticators at our evaluated RPs (section 3.3.1 and section 3.3.1), we found that out of the 29 RPs analyzed, only Aliyun, Binance, Stripe, and Bank of America (BofA) challenged our attempt to register an authenticator. They required us to enter a One-Time Password (OTP) sent over either SMS or email, and BofA additionally required the PIN of the debit card registered on the account. We note that while other services may have made risk-aware decisions not to verify our identity, prior work has demonstrated that such user verification challenges should augment risk-aware authentication to minimize the risk of account takeover [92]. Thus, RPs currently do not widely deploy such measures to limit malicious authenticator registration through traditional account takeover.

Phishing FIDO2 via User-Level Malware

Attack Description: A user could inadvertently install a malicious user-level (i.e., non-root) application that misrepresents itself as the target RP's official application. Similar to a phishing site, the application could look and behave the same way as its legitimate counterpart. Once the user attempts to log into the RP through the malicious app, the attacker has access to the user account. If the attacker can directly register their own malicious authenticator for the user, then this case falls back to that of section 3.5.1.

However, some services may require user verification when registering a new authenticator. In that case, the malicious application must trick the user into registering the malicious authenticator while thinking they are registering their real authenticator (or wait for the user to conduct an action that would require the same form of user verification). When this occurs, the application could intercept the legitimate FIDO2 registration request and instead register a malicious virtual authenticator embedded in the application itself (with the user verifying the malicious authenticator registration, thinking they are registering their actual authenticator or doing the user verification for a different action). Once registered, the malware could report the FIDO2 credentials (visible to the malware in plaintext) back to the attacker, allowing the attacker to clone a similar virtual authenticator with the same stolen credentials and remotely compromise the user's account. Since FIDO2's threat analysis does not account for virtual authenticators, this attack is not discussed in existing documentation.

From our characterization of RP FIDO2 configurations (in both section 3.3.1 and section 3.3.1), we observe that only 14/29 RPs request attestation of any kind, and the remaining RPs cannot ensure that they have trustworthy information on the authenticators being used by clients. Furthermore, we find that 27/29 RPs – including financially sensitive RPs such as BofA, PayPal, Binance, Stripe, and eBay, allow even a virtual authenticator to be registered. Thus the vast majority of evaluated RPs are vulnerable to such an attack.

While RPs are broadly not allowlisting trusted authenticators, we explore the extent to which they could, using our real-world user authenticator data (section 3.3.3). From the distribution of attestation types that we observed from client authenticators (all on mobile clients of our partner RP), as listed in Table 3.4, we find that the vast majority provide attestation linked to either an Anonymization CA (Apple Anonymous; $\sim 71\%$) or attestation root certificates found in MDS (Android SafetyNet; $\sim 27\%$). Thus, for at least mobile devices, attestation information from most authenticators can be verified and used for allowlisting.

However, there are $\sim 1\%$ of client authenticators that do not provide any attestation (and 1 self-attested authenticator), for which the RP has no trusted information. In such cases, RPs could disallow these unattested authenticators at the risk of impacting legitimate users or permitting such devices but remain exposed to this attack. We briefly note that for accounts linked to these non-attested authenticators, our partner RP indicated that their risk systems identified significantly higher volumes of risky transactions (attempts to steal funds) than typical for the average account, as well as evidence of attempted money laundering through newly created accounts. We can also explore suspicious activity on authenticators attested by Android SafetyNet, as SafetyNet provides information on the calling/authenticating application in its attestation response [93]. From our sample of WebAuthn responses attested by Android SafetyNet, we found that $\sim 0.1\%$ of attestations did not include the calling application's certificate, and another $\sim 0.1\%$ had a certificate different from that of Chrome on Android, the calling application it claimed. SafetyNet also indicated that these sessions came from rooted devices. Thus, we have preliminary evidence that suspicious activity from authenticators lacking proper attestation may already be occurring in practice, exposing RPs that permit such behavior.

FIDO2 Hijack by Root Malware

Attack Description: Overprivileged devices, such as rooted Android devices or jailbroken iOS devices, allow malware to circumvent built-in security mechanisms enforced by the OS [94, 95]. A root-level malware can intercept and respond to a FIDO2 registration request from a malicious virtual authenticator. Unlike the malware attack described earlier, here the user can still interact with the legitimate RP application, but their FIDO2 operations are hijacked at the root level. While a strong attack, this threat is still realis-

Table 3.4: Distribution of WebAuthn attestations observed at registration.

Attestation Type	**Apple Anonymous**		**None**		**Android SafetyNet**	**Self**
AAGUID (in MDS ✓/✗)	`f24a8`... (✗)	**No AAGUID Provided (✗)**		`b93fd`... (✓)		`adce0`... (✗)
No. of registrations (07/20/22 — 08/23/22)	71% — 71%	0.5% — 0.4%	0.6% — 0.9%	0.5% — 0.4%	27% — 27%	1 — 0

The following are the complete AAGUIDs of the authenticators observed in the Table:

- **Apple:** f24a8e70-d0d3-f82c-2937-32523cc4de5a.
- **Android:** b93fd961-f2e6-462f-b122-82002247de78.
- **Packed (suspected Apple Touch ID):** adce0002-35bc-c60a-648b-0b25f1f05503.
- **Chrome's Virtual Authenticator:** 01020304-0506-0708-0102-030405060708.

tic as a non-trivial population of mobile devices is rooted [96] (as we also confirm in our measurements), and rooted devices are frequent compromise victims [97].

Measurement of Mitigation Adoption: This attack can also be mitigated by RPs verifying attestation and allowlisting trusted AAGUIDs [65], which prevents the attacker from registering a malicious authenticator. As uncovered above (in section 3.5.1), we found that such mitigations are limited in practice. The 27/29 sites that lack trusted authenticator allowlisting are similarly vulnerable to this attack.

27/29 RPs can increase their risk awareness by formulating policies to allowlist secure and trusted authenticators.

3.5.2 Vulnerable Authenticator

FIDO2 authentication is available on a wide variety of devices, and the authentication security depends on characteristics of the device security, such as the availability, modality, and accuracy of user verification, matcher protection, and private key management protection. Some authenticators may be weak/vulnerable and later compromised by an attacker after FIDO2 registration. Here we consider threats where the attacker targets a vulnerable authenticator.

Local Authentication Downgrade

Attack Description: The level of assurance an authenticator provides for a user's identity varies with the modality of local verification used. It can range from biometrics (e.g., fingerprint scan) to knowledge-based factors (e.g., PIN) to mere user presence, or even no verification at all. FIDO defines stringent biometric performance requirements that a FIDO-certified authenticator needs to adhere to [98], while prior work has shown that knowledge-based factors such as PINs can often be observed or easily guessed [99], such as by malware on the device. Therefore, the security posture of a user's account could significantly weaken if their authenticator allows for weak user verification methods (i.e., PIN), which malware could successfully bypass. Similar social engineering downgrade attacks have been demonstrated previously at the RP level [11].

Measurement of Mitigation Adoption: To mitigate this attack, FIDO2 provides the User Verification Method Extension (`uvm`) which enables the RP to know which verification methods (factors) were used for an operation [64, 100]. In case a PIN needs to be used, such as when other modalities fail (e.g., physical obstruction for biometrics), WebAuthn is expected to soon implement an extension currently supported by CTAP called `minPinLength` which conveys the minimum PIN length value of the authenticator to the RP [101, 63]. Together, this information allows an RP to implement policies/requirements on user verification methods, and/or adjudicate risk appropriately during an authentication attempt. However, from analyzing the MDS data, we see that only 1 and 8 (out of a total 160) authenticators in the MDS currently support `uvm` and `minPinLength`, respectively. Our RP partner also did not support any extensions during the pilot, so we could not measure whether clients support these extensions. Overall, we conclude that the authenticator ecosystem does not yet widely support this mitigation.

In our RP characterization (section 3.3.1 and section 3.3.1), only 3/29 RPs required user verification, allowing authenticators on other RPs to return a success without any local verification of the user's identity. No sites we studied declared support for the aforementioned extensions, with only 9/29 RPs supporting any FIDO2 extension. Thus, RPs are not checking for robust user verification methods, and are vulnerable to local authentication downgrade attacks.

Compromise of Vulnerable Authenticators

Attack Description: For a legitimate authenticator, if user verification can be circumvented by malware, or its attestation or private keys can be compromised, either remotely or with physical access to the authenticator, the authenticator is known to be vulnerable [102]. An attacker can exploit such a vulnerability to compromise the authenticator's security and potentially take over an account.

Measurement of Mitigation Adoption: FIDO2 recommends that RPs monitor the MDS for security notifications declaring such vulnerabilities in authenticators registered with FIDO [59]. Ideally, an RP should also periodically keep monitoring if vulnerabilities are reported for previously registered authenticators, and in the case a vulnerability is discovered, treat the authentication attempts from those authenticators as risky, until patched.

In our longitudinal measurement of the MDS starting September 2021 (in section 3.3.2), spanning over a year, we did not record any security notifications for any authenticator. Yet, we are aware of existing authenticator vulnerability. For example, Shakevsky et al. demonstrated an IV reuse attack on Android's hardware-backed Keystore in Samsung's flagship smartphones, leading to the compromise of private FIDO2 credentials, and a bypass of FIDO2-based authentication [103]. Following their disclosure, Samsung published CVE-2021-25490 with High severity and issued a patch in October 2021. However, neither *Samsung Pass*, Samsung's identity management service, nor *Android with SafetyNet* – both authenticators registered with FIDO – reflected a vulnerability in the MDS. In fact, as of May 2023, *Samsung Pass*'s MDS entry has not been updated since 2018, and *Android with SafetNet*'s entry's last update was in 2020 – both when they were first registered.

As an increasing number of devices and services bank upon the security guarantees provided by TEEs, other TEE implementations have also been similarly targeted [104, 105]. The lack of updates signals that the MDS may not be a trustworthy source of information to make accurate decisions about vulnerable authenticators.

Credentials stolen from Virtual Authenticator

Attack Description: Users could install an application (or a Chrome extension [106]) that includes an authenticator and allows the user to manage (export and sync) their FIDO2 credentials. While some users might choose this for its usability, it provides little security guarantees, as the FIDO credentials reside in user space and can be stolen by malware.

Stolen credentials can be easily seeded in a cloned virtual authenticator for the attacker to gain access.

Measurement of Mitigation Adoption: RPs should disallow virtual authenticators. As seen in section 3.5.1, 27/29 RPs permitted virtual authenticator, and are exposed to this threat.

Authenticators need to implement FIDO2 extensions (e.g., `uvm`) which enable RPs to **ensure trusted user verification**. RPs also need to **actively track vulnerabilities in authenticators**, such as by monitoring public CVEs.

3.6 Authentication-Phase Attacks

In this section, we investigate how malicious software on the client can affect FIDO2 authentication assuming that a malicious or vulnerable authenticator is not registered to the user's account. In this setting, the attacker is not able to compromise the authenticator (unlike in section 3.5), and thus can only target the FIDO2 authentication phase. We consider first the case where malware attempts to leverage existing legitimately-registered credentials to authenticate, which is only possible with a limited set of authenticator properties. Outside of those conditions, malware cannot directly utilize existing credentials for authentication. Instead, malware must involve the user through a social engineering attack that results in the user authenticating during insecure situations. Towards that, we identify a unique social engineering attack where malware can trick users into authenticating sensitive actions without them realizing. For both categories of authentication-phase attacks, we discuss what mitigations exist to address them and the extent to which real-world FIDO2 deployments are configured to do so.

3.6.1 Targeting Existing Legitimate Credentials

Attack Description: As discussed in section 3.1, FIDO2 has built-in protection (KHAccessToken/authenticatorClientPIN) to prevent a user-level malware from accessing keys previously registered by a legitimate application, and thus user-level malware is not able to interfere with the FIDO2 authentication phase.

For root-level malware, as long as the FIDO2 Client resides as a Trusted Application in the TEE, OS integrity checks will fail when the FIDO2 client attempts to access the keystore, denying the malware access to the credentials. However, in the case of roaming authenticators (e.g., a USB security key) where the FIDO2 Client is a root application on the OS, root malware could bypass local user verification (by spoofing a successful verification) to the authenticator and raise the request to the authenticator when it is plugged in. If there is no user verification at the roaming authenticator itself (e.g., button push, biometric validation), the malware could successfully trigger an authentication using the user's FIDO2 credentials, without them realizing it.

Measurement of Mitigation Adoption. To address this threat, RPs could declare that they require `platform` authenticators during the authenticator registration protocol (through one of the parameters passed during credential creation), disallowing roaming authenticators. If an RP does want to allow roaming authenticators (e.g., to increase user adoption or reduce user friction), it should use MDS metadata, which lists the user verification methods supported by authenticators, to make a risk-aware policy/decision based on the strength of the user verification method (or lack thereof). For example, the RP could disallow roaming authenticators with weak user verification, or challenge authentication attempts from them.

From our analysis of RPs (Table 3.2), only 5/29 RPs declare a preference for `platform` authenticators, of which two (Mail.ru and Shopify) allowed registering a virtual authenticator with external transports. Thus, most RPs are potentially exposed to this attack on users with roaming authenticators which lack user verification.

We investigated how common such roaming authenticators were. In our April 13, 2023, MDS snapshot, we found that 139/160 authenticators were capable of acting as roaming authenticators. Of those, only 4 can locally authorize an authentication without requiring any user verification at all. Thus, the vast majority of roaming authenticators (even with user verification as simple as user presence) could mitigate this issue in practice, limiting the exposure of RPs.

Root-level malware can bypass FIDO-based authentication on external authenticators, which lack on-device user verification.

3.6.2 Social Engineering Attack to Authenticate Sensitive Action

Here, we consider the scenario where the FIDO Client resides as a Trusted Application in the TEE, so the malware (regardless of user-level or root-level) cannot directly trigger an authentication. Instead, it must leverage a social engineering approach to cause an attacker-desired authentication, as FIDO2's workflow involves a human-in-the-loop.

Prior work has proposed such a social engineering attack on FIDO2. Jubur et al. demonstrated an attack where multiple near-concurrent and indistinguishable 2FA authentication prompts can trick the user into authorizing the attacker-initiated attempt [10], similar to "MFA fatigue" and "push phishing" attacks [107, 108]. However, their attack requires a synchronized timing of when the user would attempt to log in, and hence the authors consider their attack a targeted one rather than a scalable one. Without an attack that could be executed automatically and at scale, one could argue that this is a limited real-world threat. However, we investigate and identify a unique social engineering attack that can be executed in an automated fashion by malware on the client device, thus raising this threat's practicality.

We observe that in practice, real-world implementations of FIDO2 lack explicit user consent for a specific action, as originally recommended by FIDO [109]. As a result, users lack clarity about what specific action they are authenticating. In addition, online services apply Risk-Backed Authentication (RBA) systems that users lack transparency into, and thus users are unable to accurately predict when they may be asked to (re-)authenticate [82, 110]. We combine these two observations to construct a unique social engineering attack that tricks a user into authenticating an attacker-initiated sensitive action (which the user does not realize is the action being authenticated), at the same time as when the user has initiated a non-sensitive action (but does not realize that the action does not actually require re-authentication). Sensitive actions could vary from RP to RP – we broadly define it as an action that a user performs on their account which requires re-authentication or a step-up/challenge authentication due to its sensitive nature. For example, financial and online banking sites often consider financial transactions to new parties as sensitive, and many other online services designate access to or change of personal profile settings as sensitive (e.g., changing a recovery email address). As a real-world example, Bank of America requires user verification when transferring funds above a threshold, and supports FIDO2 authentication for user verification [111].

Attack Setup/Assumptions: Conceptually, our attack makes no assumption about the platform and authenticator properties. Given the KHAccessToken/authenticatorClientPIN protection discussed in section 3.1, user-level malware, which would have a different (untrusted) AppID (and hence a different KHAccessToken/authenticatorClientPIN), would not be able to trigger an authentication using existing FIDO2 credentials. However, in the Web context, browser extensions are different because they act under the same AppID as the credential-registering application (that of the browser), and have the same scope of control as the actual user. Also, given that malicious browser extensions are already known vectors for social engineering attacks [112], we choose to construct our attack via a Chrome extension. (Note that as root-level malware can bypass the KHAccessToken/authenticator-ClientPIN protection, spoofing the AppID, it can conduct a similar attack.)

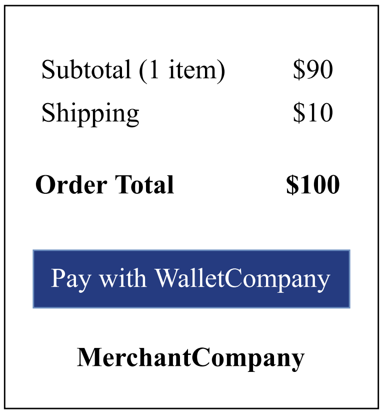
The attack setup requires the user to have FIDO2 passwordless authentication enabled on their account at a target RP. When they interact with the RP in a browser and perform an action that the user might consider sensitive (requiring re-authentication), but in fact is not, the malicious extension will simultaneously initiate a sensitive action on the RP in the background. The user will be tricked into authenticating the attacker's sensitive action mistaking it for a re-authentication challenge for their own action. Note that as our attack assumes malware on the device, the attacker already has access to an authenticated session. However, our focus is rather on attacking the step-up authentication challenge/RBA.

For our attack, the threat model assumes that the authentication prompt raised by the OS, as seen in Figure 3.1d, cannot be compromised. However, root-level malware can compromise such prompts, and trick the user into authenticating an attacker-controlled session for an RP different from the one that the user is interacting with.

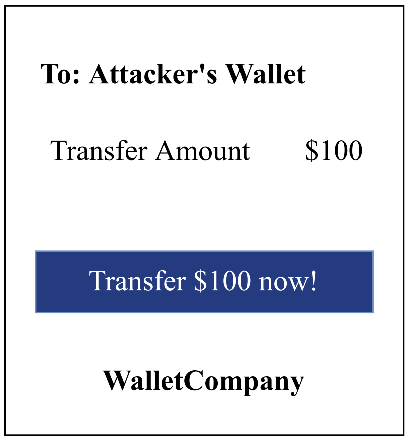
Attack Description: To describe our attack concretely, we assume WalletCompany to be an example target RP deploying FIDO2 for passwordless authentication. WalletCompany provides checkout functionality at merchant sites as well as person-to-person (P2P) transfer of funds. We choose the user-initiated action to be a checkout on MerchantCompany, a merchant site, and the attacker-initiated sensitive action to be P2P transfer of funds to the attacker's wallet on WalletCompany. The attack should also work on other RPs for other combinations of user-initiated sensitive actions.

Once the malicious Chrome extension is installed (Figure 3.1a), it waits for the user to visit the merchant site to make a transaction. This could be achieved by inserting a content script that records all elements which are clicked. When the user checks out with WalletCompany, as seen in Figure 3.1b, the click is detected and the extension spawns a background Chrome process to initiate a transfer of funds to the attacker's own wallet (Figure 3.1c) – since the Chrome instance has an active session on WalletCompany, no authentication is required by WalletCompany at this stage. However, the P2P transfer of funds to a previously unseen wallet is typically a sensitive action and would trigger re-authentication. The user will see a WebAuthn authentication prompt within seconds of attempting to check out with WalletCompany on MerchantCompany, as seen in Figure 3.1d. Since RBA sys-

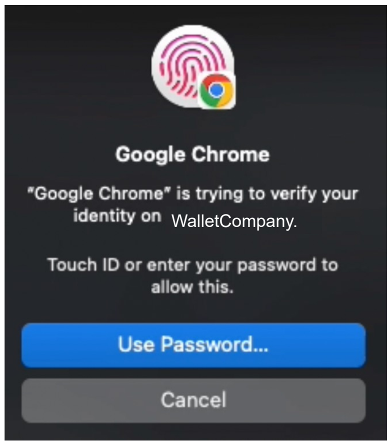
(a) User downloads and installs a malicious browser extension [112].



(b) With an active session on WalletCo., user attempts to make a transaction on another merchant site, say MerchantCo.



(c) Extension detects the transaction and attempts attacker-initiated action (e.g., transferring funds to the attacker's wallet) in a background tab. This triggers a FIDO2 authentication prompt.



"Google Chrome" is trying to verify your identity on WalletCompany.

Touch ID or enter your password to allow this.

(d) User is tricked into authenticating the attacker's transaction, mistaking it as authenticating their own transaction (which did not actually require authentication).

Figure 3.1: Demonstration of our social engineering attack on FIDO2 authentication.

tems are neither transparent nor well-understood [82], the user will be tricked into believing that the re-authentication prompt is for their checkout on MerchantCompany. Once they authenticate, the transfer of funds would be authorized, and the user would not notice anything suspicious because their checkout would have succeeded anyways.

Attack Proof-of-Concept (PoC): For simplicity, we demonstrate our concrete example attack using PayPal and eBay, which are leading mobile wallet and e-commerce companies respectively – via a malicious browser extension on Chrome 103.0.5060.53 on a 2019 MacBook Pro running OS X 12.4. We enabled passwordless authentication in one of the author's PayPal account, registering with Mac's Touch ID via Chrome. We also log into PayPal before the attack executes, establishing an active authenticated session. We will use the terms WalletCo. and MerchantCo. for PayPal and eBay respectively, to describe the attack as it can be generalized over other similar applications.

To demonstrate that an authentication prompt can be raised by a background process spawned by an extension, we present a PoC Chrome extension in Listing 3.1. The extension requires permissions `tabs` and `scripting` to open the target RP's site in a tab and execute JS scripts in its context, as well as `host_permissions` to our target RP (WalletCo.). The extension opens WalletCo.'s WebAuthn login page in a new background window and simulates a login button click to raise a WebAuthn authentication prompt to the user. The user provides their biometrics, authorizing the attacker-controlled session. Other than the prompt, the user never sees anything and their only action is providing their biometrics. The attack is automated and takes only seconds to execute (e.g., time for the page load and for the user to provide biometrics).

Measurement of Mitigation Adoption: To our knowledge, there are only a limited number of existing mitigations to this attack, so RPs and users are broadly vulnerable. However, we believe there are tractable directions for better mitigations.

For e-commerce specifically, the W3C Working Group has developed Secure Payment Confirmation [113], which is available as the `payment` extension in WebAuthn and is fully supported by Chrome. It provides transaction confirmation functionality to financial services RPs, with permission to perform registration and authentication ceremonies on behalf of the RP on merchant sites. The transaction confirmation includes information such as `payeeName`, `payeeOrigin`, `currency` and `value`, among other things – thus mitigating the potential for identity confusion in the context we demonstrated. However, in our real-world measurement of RPs (section 3.5), we did not find any RP supporting the `payment` extension. In our MDS snapshot, we found that only 15% authenticators supported a transaction confirmation display.

```
// trigger when the extension is installed/loaded
chrome.runtime.onInstalled.addListener(async () => {
    // open new Chrome browser window
    let window = await chrome.windows.create({
        url: 'https://www.walletcompany.com/signin-webAuthn' ,
        focused: false,
        state: 'minimized'
    });
    // wait for the page to load
    setTimeout(() => {
        chrome.scripting.executeScript({
            target: {
                tabId: window.tabs[0].id
            },
        // instrument an authentication attempt
            function: () => {
                document.getElementById('logIn_start').click();
            }
        });
    }, 1000); // wait 1 second
});
```

Listing 3.1: `background.js` for the Chrome extension PoC that visits WalletCo.'s WebAuthn login page in a new background window, and simulates a click on the login button, to raise a WebAuthn authentication prompt to the user.

In the absence of protocol-level protection, user agents could also mitigate such an attack by employing certain sanity checks before raising a WebAuthn request to the OS. For example, our attack can be mitigated if Chrome restricts `navigator.credentials` API requests to be made only by a page that is in the user's focus, and if RPs specify that user verification by the authenticator is either required or preferred (and not discouraged). To our knowledge, existing browsers do not implement such policies.

Additionally, in line with Prakash et al.'s proposal to modify the authentication prompt design [114] to counter Jubur et al.'s attack, RPs could send out-of-band notifications, such as a push notification to the user's mobile device, indicating the transaction details. We are not aware of any RPs currently doing so.

Attack Disclosure: We disclosed this attack via Google's Bug Bounty Program and recommended that Chrome restrict `navigator .credentials` API usage to the page in focus to prevent such misuse. Google's response was that while confusion with WebAuthn interactions across tabs is a known issue, our attack demonstrating the bypass of step-up authentication is plausible, and a fix for it will be prioritized [115]. They noted that they would need to work with RPs to gracefully handle the case of restoring a set of tabs on a Chrome restart, as some of them might initiate re-authentication from the background.

The **lack of trusted transaction confirmation information can be exploited by social engineering attacks** to trick the user into authenticating actions they do not intend to. Improved UI and user education could be stopgap solutions until the ecosystem matures to defend against such attacks.

3.7 Concluding Remarks

In this work, we evaluated the security posture of real-world FIDO2 deployments across the Tranco Top 1K, particularly focusing on the impact of compromised clients. Here we draw on our findings to synthesize key takeaways, as well as propose high-level recommendations for FIDO2 deployments.

Compromised clients are a realistic and salient threat to FIDO2 deployments.

In section 3.4, we found that 96% of authenticators present in FIDO MDS are not certified to be malware-resistant, so a significant population of existing authenticators could potentially be compromised by a motivated attacker. From real-world authenticator telemetry (via our partner RP), we found evidence already of system integrity compromise on $\sim$0.25% of hardware-backed clients and $\sim$13% of non-hardware-backed clients. Our evaluation of RPs in section 3.5 and section 3.6 revealed how RPs have not yet adopted recommended mitigations to combating compromised clients, making them potential targets. In fact, our partner RP already has observed online abuse from active FIDO-authenticated sessions on compromised devices. Thus, our study highlights that compromised clients are a salient threat that must be accounted for by FIDO2 deployments. As FIDO2 gains adoption, attackers will be even further incentivized to utilize this attack surface.

Improvements to mitigations are needed. Through our study, we uncovered various shortcomings with the existing FIDO2 mitigations to compromised clients. For example, a key recommendation is to maintain an assurance level for registered authenticators by allowlisting either known authenticator AAGUIDs or authenticators with particular attributes. However, in section 3.5.1, we found that nearly all (27/29) RPs do not enforce such a policy. To the best of our knowledge, Microsoft is the only RP that mentions a "Key restrictions policy" for passwordless authentication [116] (while BofA states that they require a FIDO2-certified authenticator, but we found that in reality, they do not verify if an authenticator is actually certified). While some RPs may intentionally allow all authenticators to encourage broader adoption[1], security-sensitive RPs (e.g., BofA, PayPal, Stripe) should ideally implement such a policy and adhere to strong authentication requirements (e.g., the European Union's Second Payment Services Directive [119]).

Another key recommendation is that RPs make risk decisions based on authenticator metadata available from the FIDO MDS. However, we found the MDS to be incomplete, as the metadata for popular authenticators (e.g., Apple devices) was missing. We also

[1]Passkeys are user-friendly FIDO2 credentials [117] designed to sync across devices. They are not device-specific, and as such, currently do not support attestation [118].

identified in section 3.5.2 that the MDS is not promptly updated with authenticator vulnerability information. These findings motivate the need for better MDS maintenance so that it provides reliable authenticator information to support the FIDO2 ecosystem.

We found that limited mitigations exist for malware-driven social engineering attacks, such as the one described in section 3.6.2, which can exploit the human-in-the-loop even if trusted authenticators are registered. The FIDO Alliance had initially proposed gathering explicit user consent for actions (i.e. 'Transaction Confirmation' [109]), which would have mitigated such attacks by allowing an RP to confirm that the transaction is exactly what the user intended. Subsequently, the W3C Working Group proposed the extension `txAuthSimple` in WebAuthn's first specification [120], allowing RPs to specify a prompt to be shown to users on a trusted display (e.g., the prompt in Figure 3.1d). However, it was removed from WebAuthn's second specification, due to a lack of client implementation support [121]. However, our results highlight the dire need for such mechanisms to be widely supported, allowing RPs to securely convey to users what they are authorizing. (In fact, some developers have similarly expressed their concerns with `txAuthSimple`'s removal and advocated for it to be officially supported [121].) In the meantime, UI/UX improvements (e.g., out-of-band notifications) can help users understand the action they are authorizing, reducing the impact of social engineering attacks.

Risk policies can harden FIDO2. For securing traditional password-based authentication, online services have employed risk-based authentication (RBA) models for years [82]. Given the continued threat of compromised clients, as online services move towards FIDO2-based authentication, we believe that they will benefit from similar risk-based policies built on FIDO2-specific features.

A key challenge for RPs during FIDO2 authentication is the reduced risk telemetry, which hampers their ability to detect anomalous login attempts. Risk signals from password authentication, including autofill/typing behavior, incorrect password attempts, and mouse movements [122], lack equivalents for FIDO2 authentications. Instead, FIDO2-

specific signals are needed, such as round-trip communication time with the authenticator. For instance, an RP might decide to allow authenticators without attestation requirements to reduce user friction, but challenge authentication attempts with anomalous round-trip communication times with the authenticator. While Whalen et al. argued that a successful FIDO2 credential creation represents a cryptographic attestation of human signals [123], identity confusion attacks such as the one demonstrated in section 3.6.2 violate that assumption (even if using a secure authenticator).

Interestingly, we have already observed some evidence of RPs modifying the risk profile for users who authenticate via FIDO2, primarily considering such user authentications to be less risky. Namecheap indicated that users registering a FIDO2 authenticator would never see a CAPTCHA [124], while BofA said that a FIDO2 authenticator can be used as an alternative to SMS-OTPs to verify high-value transactions [111]. Namecheap's waiver of bot defense is similar in concept to Cloudflare's Cryptographic Attestation of Personhood recently proposed by Whalen et al. [123], which uses FIDO2's attestation to distinguish human traffic from bot traffic. However, we found in section 3.5.2 that as Namecheap and BofA both allow registering virtual authenticators, FIDO2 authentications are not inherently less risk, and these RPs remain exposed to automated abuse and malware threats.

We suggest that existing authentication risk models [122, 125, 126, 127] should consider more risk signals to verify if a FIDO2 authentication attempt was made by the right user. FIDO2 has an extension interface that allows custom extensions for specific use-cases [60]. Future work could explore an extension that runs in the trusted FIDO2 Client and fingerprints the user's authenticator interaction, to tell apart a real user from automated activity. The FIDO2 Client's privileged access could also fingerprint the user's presence via kernel-level events [128], device features [129, 130], or hardware features [131]. Such risk models could help strengthen FIDO2 deployments in practice, and as a result increase mutual trust between platforms and users.

CHAPTER 4
INVESTIGATING FAKE ENGAGEMENT ABUSE ON YOUTUBE

As social media takes center stage in modern discourse, protecting online platforms from abuse and manipulation is essential. Social media platforms, such as YouTube, Facebook, and Vimeo, are driven by engagement metrics, including the view count of videos and the number of likes on posts. Manipulation of these metrics is a form of online abuse.

Video view fraud is one distinct class of fake engagement abuse, which entails artificially inflating the view count of videos (i.e., when the total view count exceeds the number of solicited views by legitimate users). Beyond the content popularity and visibility manipulation also offered by other fake engagement vectors, video view fraud uniquely provides a direct monetization mechanism as many platforms (including YouTube) pay video creators based on video views (specifically, platforms share a portion of the revenue derived from ads displayed when users view a video). Furthermore, the bar for perpetrating video view fraud is low; while other fake engagement efforts require logged-in users to execute an action (e.g., liking or commenting on content, or sending friend requests), video view fraud simply requires an arbitrary user to visit a webpage, where the video can start playing.

While other forms of fake engagement have been studied previously [17, 16, 18, 19], there exists limited empirical characterization of real-world video view fraud. The prior work that does exist [19] focuses on bot-driven automated view fraud. In this study, we expand our understanding of video view fraud by investigating organic or human-driven approaches. Organic view fraud relies on real human users (rather than bots) to generate views, presumably because manually-generated fake views may be more challenging to detect and block. Such activity is still classified as view fraud as users do not intentionally request to watch the videos [132].

In this work, we conduct a case study of a large-scale, long-running, organic YouTube view fraud operation on one of the most popular free video streaming services [133], 123Movies[1] [134]. Before users can watch a stream on 123Movies, a YouTube video is displayed as a pre-roll advertisement and automatically played, thus generating a fake view for that video. Given 123Movies' immense popularity, this activity results in large-scale YouTube view fraud. In our study, we reverse-engineer how 123Movies distributes YouTube videos as pre-roll ads, and collect the YouTube videos it distributes over a 9-month period. For a subset of these videos, we gather detailed metadata about video characteristics (e.g., view count, content category), and track their dynamics over time. We similarly collect metadata about the YouTube channels associated with the videos and analyze their participation in the scheme.

Our analysis sheds light on the characteristics of this view fraud ecosystem. We find hundreds of thousands of videos associated with tens of thousands of channels participating in this view fraud effort. These videos and channels skew heavily towards music, gaming, and entertainment content, hinting at certain industries that may engage in such activity. In fact, we observe extremely popular music videos (with millions of views) involved. We also assess the outcome of participating in view fraud, observing that while videos gain views in the short term, the majority of videos eventually lose a substantial amount of views (sometimes even within a week), presumably due to YouTube's detection and removal of abusive activity. Thus, the long-term success of this abuse seems to be limited (although our constrained visibility into YouTube and the view fraud campaign's operations prohibits establishing a definitive relationship between the view fraud and outcomes). By analyzing the channels that participate in view fraud, we identify that most participate only once or over a short period of time (e.g., a few days), suggesting that the participating channels also observe limited returns on investing in view fraud. Nonetheless, this view fraud operation has survived for years, raising questions about the economics at play and mechanisms that can combat view fraud, beyond detecting and removing fraudulent views.

[1]In a 2018 investigation, the Motion Picture Association of America (MPAA) found 123Movies to be the world's "most popular illegal site" serving 98 million visitors a month [133].

Ultimately, this work provides insights on an organic form of video view fraud, expanding on the limited prior work. Our findings provide further empirical grounding on attacker behavior and the outcomes of online abusive activities in practice.

4.1 Background

Here we provide background about YouTube view fraud, as well as on related prior work.

4.1.1 YouTube View Fraud

On YouTube, a video's view count is its currency. More views can translate to higher video rankings in search results (potentially attracting more organic visits), as well as represent broader popularity and approval for the content creator (particularly relevant for content creators in marketing themselves [135]). Furthermore, video creators make money based on the ad impressions generated while users view their videos. Thus, content creators strive for higher view counts for their videos.

123Movies, a popular free video streaming service, facilitates artificially inflating YouTube video view counts. Before 123Movies displays a video stream to a user, it overlays a pre-roll ad on the stream (similar to legitimate ad-supported video platforms, such as YouTube and Dailymotion). However, unlike pre-roll ads on legitimate platforms, 123Movies' pre-roll videos are regular YouTube videos, not ads. As shown in Figure 4.1, users must watch a YouTube video for at least 30 seconds before clicking through to the stream, which is the duration that YouTube requires a video to be played before recording a view [136]. While organically driven, these YouTube video views are classified as view fraud, as content creators pay an ad network to obtain views for their videos [137], and the ad network in turn pays 123Movies to show its users the videos [138].

4.2 Method

Here we describe our study's method for investigating real-world YouTube view fraud.

4.2.1 Data Collection

To understand the YouTube view fraud occurring through 123Movies, we collect the IDs of the videos involved and their metadata, including information about the associated YouTube channels.

Collecting Videos Receiving Fake Views.

To start, in July 2020, we reverse-engineered how YouTube videos are delivered as pre-roll ads on 123Movies. Using `mitmproxy` [139] (as 123Movies uses HTTPS), we monitored the plaintext Web traffic when accessing streams on 123Movies[2]. We identified that when a stream is requested, as shown in Figure 4.1, 123Movies interacts with multiple ad networks to request different types of ads (e.g., popups and banners). We traced the requests that fetched YouTube videos to the API endpoint `deliver.vkcdnservice.com`. We believe that this endpoint belongs to an ad network named `AdSpyGlass` [140], as `deliver.vkcdnservice.com` redirects to `AdSpyGlass.com` when loaded in a browser. Using the Wayback Machine [141], we find that `AdSpyGlass.com` has existed since at least April 28, 2019. Thus, we believe that this service has been operational for years (as it remained online at the time of this writing).

When a video is requested from the `AdSpyGlass` endpoint, it responds with the URL of another service endpoint, which actually provides the final YouTube video link. We observe that across all responses, only a small set of secondary service endpoints are used, which we refer to as *subservices*. We monitored 123Movies through July-September 2020 for active subservices, and found three in total. We observed *greedseed*[3], and *xtremeserve*[4] in July, and *socpanels*[5] in August. To collect the YouTube videos involved in this view fraud, we automatically milked all three subservices for YouTube videos, using a Python

[2]We used `mitmproxy` as 123Movies deploys anti-debugging techniques on its Web pages, which prevent stream loads when detecting that a browser's debugger is in use.

[3]`greedseed.world/vpaid/getVideo.php`

[4]`xtremeserve.xyz/add.php`

[5]`socpanels.thevideome.site/feed`

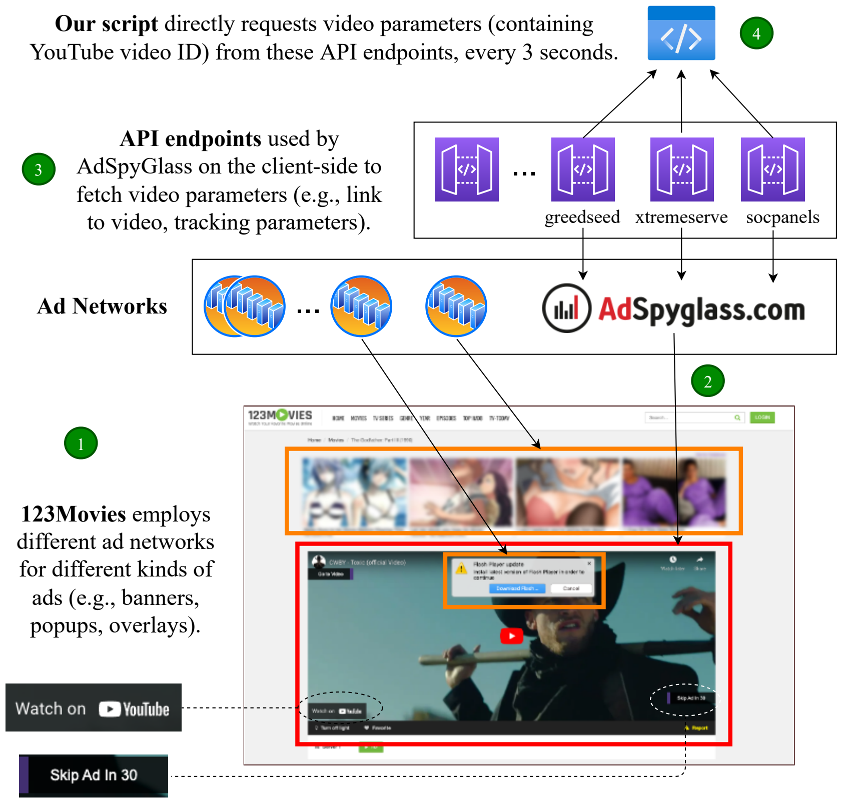

Our script directly requests video parameters (containing YouTube video ID) from these API endpoints, every 3 seconds.
4
3
API endpoints used by AdSpyGlass on the client-side to fetch video parameters (e.g., link to video, tracking parameters).
greedseed
xtremeserve
socpanels
Ad Networks
AdSpyglass.com
2
1
123Movies employs different ad networks for different kinds of ads (e.g., banners, popups, overlays).
123MOVIES
Watch on YouTube
Skip Ad In 30

Figure 4.1: Reverse-engineering ad-delivery and collecting ad data from 123Movies.

script that queried each subservice for a video every 3 seconds[6], and recording the IDs of the YouTube videos returned. To avoid detection and rate limiting, we distributed the milker's requests across a pool of 20 proxy servers (all on our university network). Over a 9-month period (from July 2020 to March 2021), we collected ∼45k unique videos from `greedseed`, ∼151k from `xtremeserve`, and ∼86k from `socpanels`. In Figure 4.2, we plot the temporal distribution of newly collected videos for the three subservices. While `xtremeserve` remained online throughout our data collection, providing new videos at a consistent rate, `greedseed` and `socpanels` ceased operations by October 2020.

[6]We determined the querying rate through manual experimentation, aiming to avoid rate limiting and overloading the API endpoints.

Collecting Video Metadata

To better investigate the gathered videos involved in the 123Movies view fraud operation, we used YouTube's Data API v3 [142] to periodically collect metadata snapshots for these videos and their channels, starting from when we first observed a video distributed by a subservice up until March 2021 (or until taken down). The metadata we acquired includes the video and channel titles, author, language, topics, and video and channel statistics (e.g., numbers of views and subscribers). Initially, we collected weekly snapshots starting July 12, 2020, but later transitioned to daily snapshots starting August 3, to monitor video view counts at a finer granularity. As we started collecting videos from `socpanels` on August 8, we recorded daily metadata snapshots for all `socpanels` videos. Since YouTube rate limits access to its Data API per API key, we generated 20 API keys (the maximum number of keys allowed by Google per account) to maximize our daily quota. This quota amount allowed us to access daily snapshots for the first ∼150k videos gathered (which we reached in September 2020), which were associated with ∼75k channels. Every 5 days, we also collected metadata snapshots for other videos hosted by these channels which we did not record as involved in this view fraud effort (∼818k videos). Table 4.1 summarizes our metadata dataset.

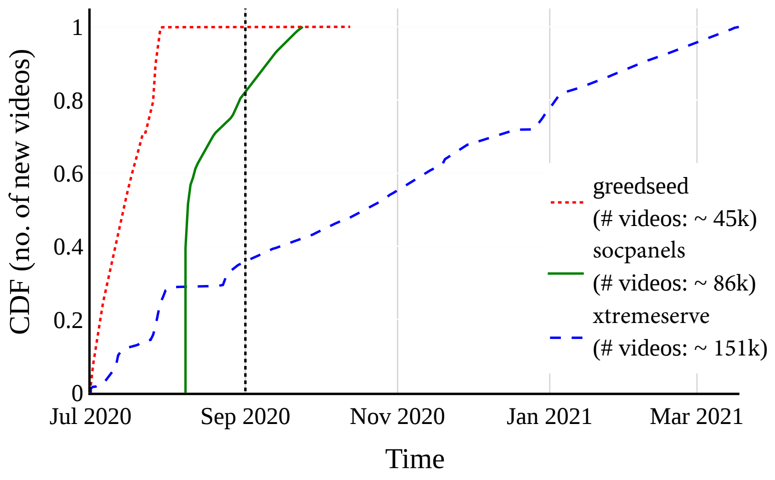


Figure 4.2: Temporal distribution of new videos recorded as participating in view fraud.

Table 4.1: Statistics about the metadata snapshots collected from YouTube's Data API.

Metadata	**Snapshot Frequency**	**# Videos**
Advertised YouTube videos	1 day	~150k
Channels of advertised videos		~75k
Non-advertised videos	5 days	~818k

4.2.2 Limitations

Our study takes a step forward in exploring the ecosystem that facilitates YouTube view fraud. However, we lack full visibility into the ecosystem, which leads to several important limitations:

- Our data collection is centered on `123movies2020.org`, a mirror of 123Movies. We note that 123Movies operates by redirecting multiple domains to a working mirror [135], so beyond direct visitors, our mirror likely received visitors from other domains (e.g., `123movies2020.email`). However, there may exist other mirrors of 123Movies or similar streaming sites that function differently, and for which our findings may not hold. Similarly, as described in section 4.2.1, we manually monitored 123Movies and initially discovered three active subservices. However, more such subservices may exist that behave differently. Nonetheless, we believe that our large-scale data provide useful insights into an organic view fraud operation.
- We ultimately are unable to distinguish fake views from real organic ones, and it is possible that the YouTube videos we monitor are involved in other view fraud operations. This lack of visibility prevents us from establishing causal relationships between the view fraud we observe and video outcomes, although we explore correlations to the extent possible.
- There are inherent delays between when a video is first advertised on 123Movies as part of the view fraud campaign, when we first milk it, and when we record its first metadata snapshot. These delays limit our observation of initial video dynamics, which we further discuss in section 4.3.3.

4.3 Findings

Here, we analyze our collected datasets to answer three research questions about the investigated organic view fraud ecosystem:

RQ1: What kind of videos receive the fake views?

RQ2: How extensively do videos/channels participate in view fraud?

RQ3: How effective is the view fraud?

4.3.1 RQ1: What kind of videos receive the fake views?

Content. We first analyze the general themes of the videos receiving fake views, using the topic tags in the video metadata annotations from YouTube. The distribution of topics among videos, as depicted in Figure 4.3, shows that the largest portion of videos covers music and entertainment topics (∼35% of all videos). This observation corroborates prior reports of the music, media, and entertainment industry's participation in view fraud [135]. Lifestyle and gaming were also popular video topics, each involving approximately 16% of videos.

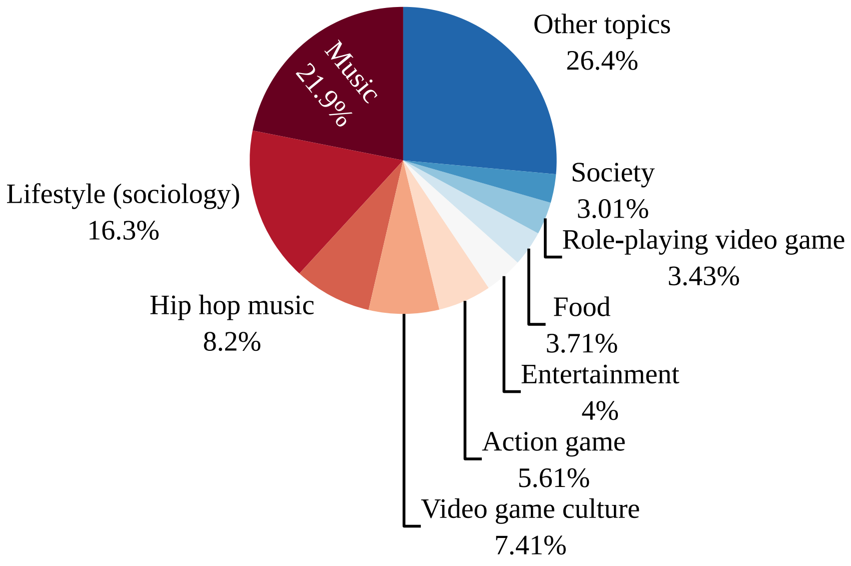


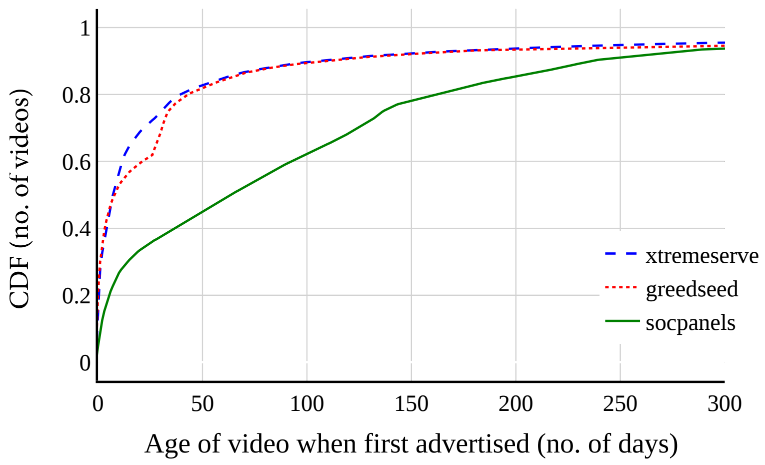
Figure 4.3: Distribution of topic tags of the YouTube videos.

While YouTube requires content creators to indicate when videos contain "paid promotions", their API does not disclose this information publicly and affiliate marketing disclosures on YouTube have previously been documented as rarely enforced [143]. As a result, we are unable to assess if videos contained commercial content.

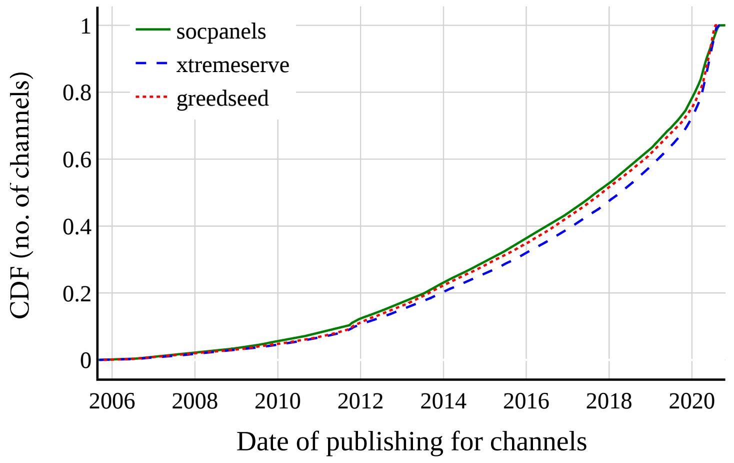
Age. To further characterize the videos receiving fraudulent views, we studied the age of the videos and channels involved, as shown in Figure 4.4. In Figure 4.4a, we plot the number of days between when a video was published on YouTube and when we first recorded it as advertised in this view fraud ecosystem. (Here, our results are upper bounds on video age, as we may not have observed initial advertisement for videos already being advertised at the start of our data collection.) We see that $\sim$80% of videos collected from `xtremeserve` and `greedseed` and $\sim$45% of those from `socpanels` were published within 50 days of first observed advertisement. Meanwhile, as seen in Figure 4.4b, the median year of first video publication for YouTube channels participating in the view fraud was 2018, with only $\sim$20% of channels publishing for the first time after 2020 (for all subservices). These findings indicate that while the videos involved in view fraud tend to be relatively new, the participating channels are often long-established, with some over a decade old.

Popularity. We additionally investigate the initial popularity of view fraud videos by evaluating the number of views in our first metadata snapshot for each video (taken within a day of being first milked). We plot this distribution in Figure 4.5, seeing that over 65% of videos already had more than a thousand views upon our first metadata snapshot, with a noticeable inflection point at approximately a thousand views for `greedseed` and `socpanels`. We hypothesize that these videos (represented near the inflection point in Figure 4.5) might have had a negligible number of views initially, and gained at least a thousand views upon being advertised on 123Movies (before our first metadata snapshot).

For each subservice, we also look at the top 3 most popular videos (see Table 4.2) and the top 3 channels by the number of videos advertised on 123Movies (see Table 4.3). Inter-



(a) CDF of the age of view fraud videos when we first observed the videos advertised.



(b) CDF of dates when channels participating in view fraud published their first videos.

Figure 4.4: Age of videos and channels involved in view fraud.

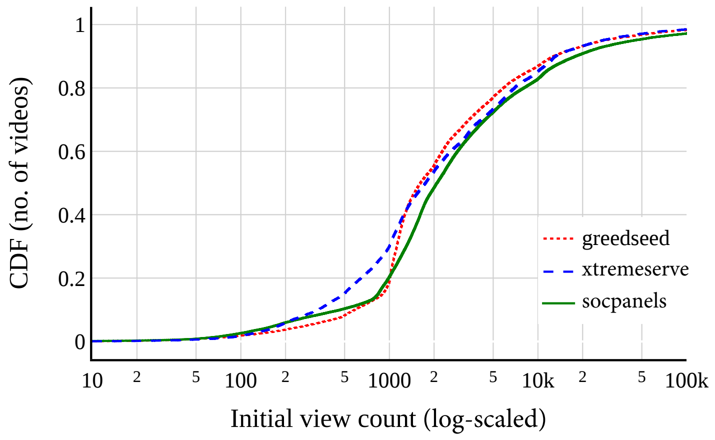


Figure 4.5: CDF of the view count of videos from their first recorded metadata snapshot.

estingly, we see that the top videos receiving view fraud include “Despacito” by “LuisFonsiVEVO”, which was the most viewed video on YouTube overall from 2018-2020 [144]. Another popular video was “Never Gonna Give You Up” by “RickAstleyVEVO”, a 1987 song that is part of a popular Internet meme known as “Rickrolling” [145]. These videos are already popular, with hundreds of millions of views, and we are unclear about their motivations for participating in view fraud.

4.3.2 RQ2: How extensively do videos and channels participate in view fraud?

Participation across subservices. In our dataset, we observed three separate subservices that `AdSpyGlass` relied upon to distribute YouTube videos for advertisement on 123Movies. We first evaluate the extent to which videos were distributed by different subservices, as depicted in Figure 4.6. We find limited overlap in the videos milked from different subservices; no more than ∼6% of videos from one subservice was observed for another subservice, and less than ∼0.5% of videos was seen for both other services. Channels also exhibit limited overlap across subservices, with only 5% of channels observed with videos advertised by two subservices, and only 0.2% of channels observed for all three subservices.

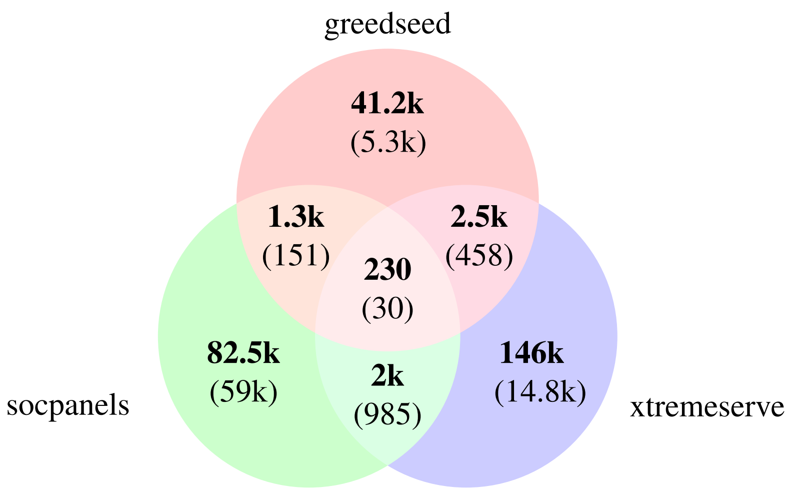


Figure 4.6: Overlap in the videos provided by the three different subservices.

Table 4.2: Top videos by the view count, for each subservice.

Subservice	**Video Title**	**# Views (First Seen) ↑**	**Δ # Views (Last Seen)**	**Channel Name**	**Topic**	**Duration (mins)**	**Published (Year)**
socpanels	Despacito	6.9B	351M	LuisFonsiVEVO	Music	4.7	2017
	BOOMBAYAH	930M	207M	BLACKPINK		4	2016
	Never Gonna Give You Up	735M	158M	RickAstleyVEVO		3.6	2009
xtremeserve	Saara India!	22M	7.3M	T-Series	Music	2.7	2020
	How To Get Your First 1000 Subscribers FAST!	5.7M	0.25M	Chaos	Video Games	14	2016
	black 45	2.5M	-647	Ramneek Sidhu	N/A	45	2018
greedseed	Mera Mera	1M	29k	Aussie Records	Music	4	2020
	AfrotroniX-Solal	0.9M	0.12M	AfrotroniX		3.4	2020
	One More Day-Jiang Tao	0.8M	0.25M	MIRROR		3.5	2019

Table 4.3: Top channels by the number of videos, for each subservice.

Subservice	**Channel Name**	**# Videos (Advertised) ↑**	**# Videos (Total)**	**# Subscribers**	**Key Topic**	**Country**	**First Published**
socpanels	kim lee	124	309	1240	Affiliate Marketing	Singapore	2018
	countrycampingkorea	118	482	7.4k	Entertainment	Korea	2019
	Ben Tre Oi	102	1210	44k	Lifestyle	Vietnam	2013
greedseed	IndieGamerRetro	27	1300	47k	Video Games	USA	2006
	Emma Moore	22	205	3	Lifestyle	N/A	2020
	Benigna Kennel Fiona Tjhin	18	811	17.6k	Pets	Indonesia	2013
xtremeserve	The Hu Music	64	66	2120	Music	N/A	2015
	Vic Stefanu - Amazing World Videos	50	3.3k	202k	Tourism	USA	2010
	Alioth Club	33	53	478	Society	N/A	2020

View Fraud Duration per Video. We investigate the duration for which videos are advertised as part of the view fraud operation. In Figure 4.7, we plot the proportion of videos that remain advertised over different durations, for each subservice. We observe differences across subservices. For `socpanels`, ∼80% of videos were advertised for over a month, whereas half of `xtremeserve` and `greedseed` videos were advertised for only a day. We note that for videos already being advertised when we first started milking a subservice, our observed duration lower bounds the true duration. However, only small portions of the videos collected for `xtremeserve` and `greedseed` were from the initial milking periods (as seen in Figure 4.2), so our high-level observations hold. (For `socpanels`, a larger portion of videos was collected at initial milking, but the advertising duration is already significantly longer than with the other two subservices.) We hypothesize that the differences across subservices may have arisen due to different service offerings. For example, `socpanels`' longer video advertising could be because it offers monthly view fraud services, instead of a daily offering. Alternatively, if `socpanels` provided guarantees on video view growth, videos may remain actively advertised by `socpanels` for longer periods (see section 4.3.3). However, we lack further visibility into the different subservices' operations to validate our hypothesis.

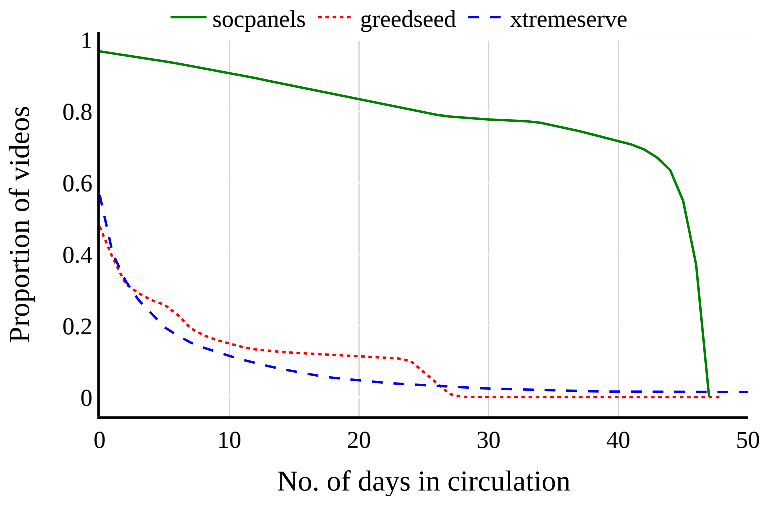


Figure 4.7: Proportion of videos that remain in circulation over time.

View Fraud Participation per Channel. To understand how extensively channels participate in view fraud, we analyze the number of videos we observed receiving view fraud per channel, as depicted in Figure 4.8. We find that over 70% of channels only advertised one video on 123Movies. In Figure 4.9, we also plot the duration over which channels advertised new videos. We see that more than 90% of channels only advertised new videos over a period of up to 10 days. We note that channels possibly participated in this view fraud campaign prior to our data collection, and our results are lower bounds. However, given the short periods that channels were observed actively participating with new videos, we believe it is unlikely that we missed observing significant amounts of prior channel activity. Thus, channels appear to only participate in this view fraud activity to a limited extent, either involving few videos or participating only over a brief duration. This observation naturally raises a question about the efficacy of this ecosystem, our final research question.

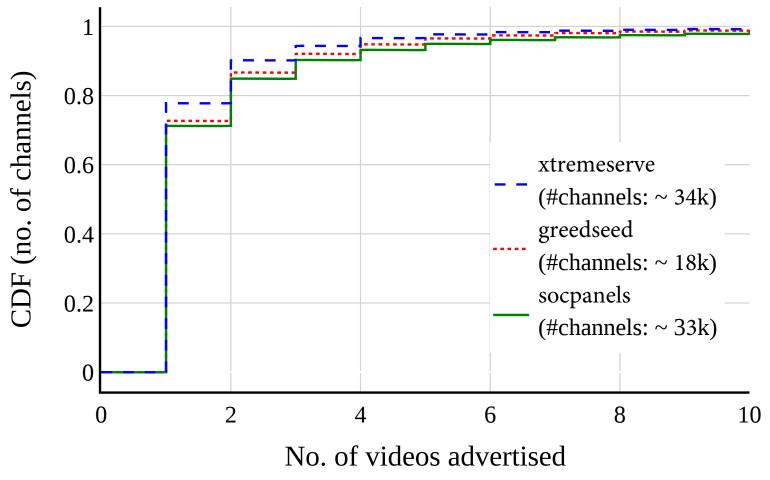


Figure 4.8: CDF of #videos advertised per channel, for each subservice.

4.3.3 RQ3: How effective is the view fraud?

Finally, we aim to evaluate the efficacy of the view fraud. To do so, we require analyzing videos for which we observed the beginning of their advertisement in the view fraud campaign. We identify such videos through multiple filtering steps.

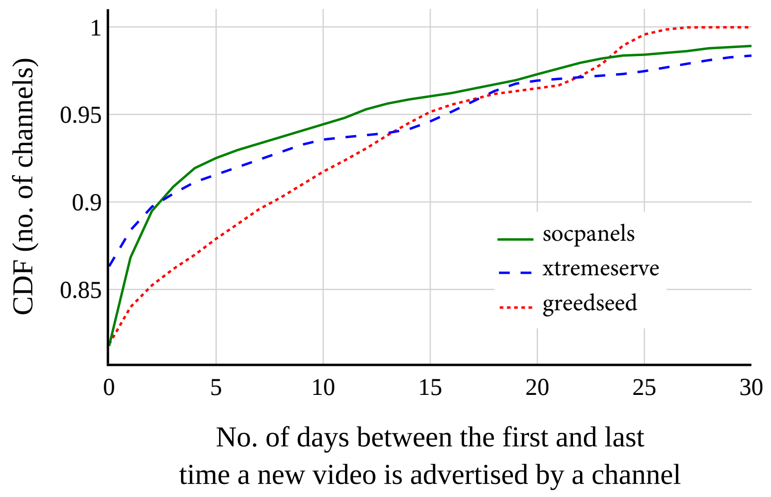


Figure 4.9: CDF of #days over which a channel advertises new videos.

First, we filter out the small portion of videos that appear across multiple subservices, to avoid convoluting the influences of different subservices. We then filter out videos that were already advertised when we began milking a subservice, as we may not have observed the start of their involvement in view fraud. For each subservice, we analyzed the growth in videos collected and identified the inflection point when the rate of new video collection reached a steady state, reflecting the natural rate of new videos being distributed by a subservice (rather than the gathering of already advertised videos that we had simply not yet observed). This initial data collection period lasted for 3, 11, and 15 days for `xtremeserve`, `greedseed`, and `socpanels`, respectively, and we filtered out the videos observed during those periods.

Finally, we further filter out videos where their first metadata snapshot was taken more than 12 hours after first observed advertisement, as the initial view counts recorded later than that duration may be too inaccurate. We explored other thresholds but chose the 12 hours window to balance the accuracy of the initial view count observed with the size of the remaining unfiltered video population across subservices. Our remaining unfiltered video population consists of 13.3k videos for `xtremeserve`, 4.7k for `greedseed`, and 3.8k for `socpanels`.

In our analysis, we consider two penalties that YouTube applies to discourage view fraud. First, YouTube discounts views that they identify as fake engagement [18]. Second, YouTube takes down videos that violate its Terms of Service, including those that artificially manipulate engagement metrics [146].

Net Change in View Counts

We analyze the unfiltered video population (for which we observe their initial advertisement by a subservice and obtain a timely first metadata snapshot), comparing the net change in view counts for these videos conditioned on (i) time and (ii) the initial view count.

View Count Change over Time. In Figure 4.10, we depict how the video view counts change over time by plotting the distributions of the view count changes at each 2-day interval across the first 20 days after the initial video advertisement, as well as the *all time* distribution, which compares the first and last snapshot per video across our full measurement period (using the last valid snapshot, if a video is taken down during our measurement). For each time interval, we display separate boxplots for each of the three subservices, as well as one for all other videos of channels participating in view fraud, where these videos were not observed as advertised.

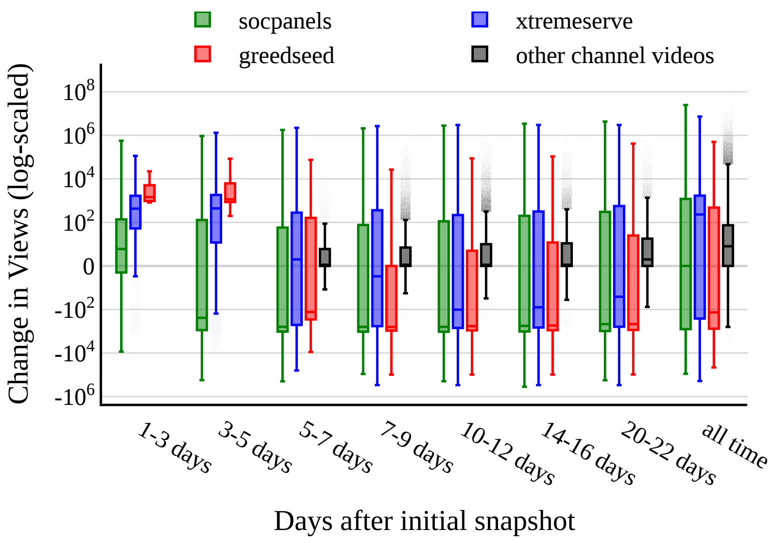


Figure 4.10: Boxplots of the net view count changes for videos, over different time periods.

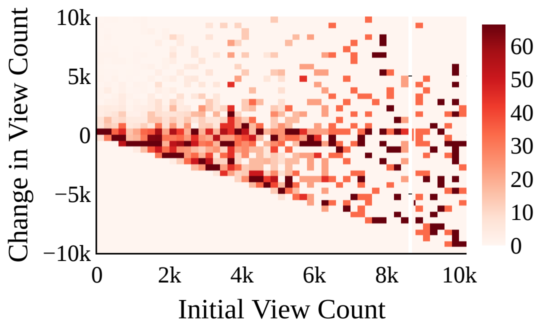
We see that in the first few days after their initial metadata snapshot, the majority of videos gain views (sometimes in the thousands) for all three services. However, within a week, the median video has negative net growth in view count, across all three subservices. Ultimately, the median long-term change in view count is 0, 232, and -136 for `socpanels`, `xtremeserve`, and `greedseed`, respectively. We also find that the median view count change for other channel videos that were not observed as advertised is small but positive (8). However, a quarter of these other channel videos had net negative view count changes, suggesting that they too were involved in other view fraud efforts (or alternatively are being falsely penalized by YouTube). These results suggest that while this view fraud campaign might be effective in the short term, YouTube is able to largely mitigate its visible impact in the long term.

View Count Change by Initial View Count. We next assess the distribution of view count changes, conditioned on the videos' initial view counts. In Figure 4.11, we plot heatmaps that visualize the distribution of view count changes (across our full measurement period) for videos with up to 10k initial views (which is over 80% of videos for all three subservices, as seen in Figure 4.5).

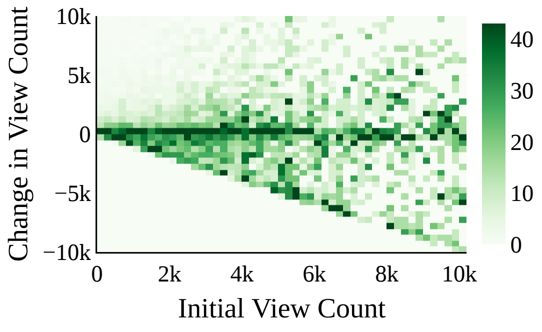
We observe that for all three subservices, the majority of videos experience net negative view growth, with the highest densities of videos near the $y = 0$ line, which indicate limited net change in view count, as well as the $y = -x$ line, which indicates that all initial views observed were removed. This result reinforces our conclusion that YouTube is able to mitigate view fraud in the long term. We note that a small portion of videos does experience sizable view count growth, which could be false negatives in YouTube's detection or cases where these videos attract real organic viewers.

Video Takedowns

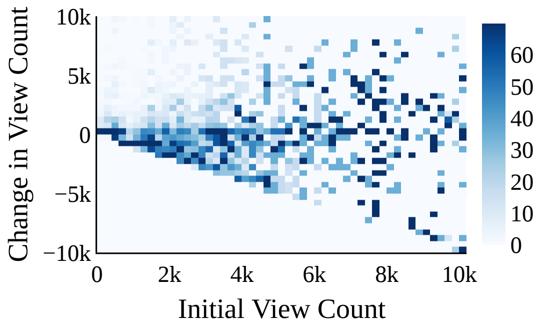
When a YouTube video is taken down, its metadata snapshot indicates that the video is not found or unavailable. We note that a video may be taken down for various reasons



(a) Subservice: `greedseed`



(b) Subservice: `xtremeserve`



(c) Subservice: `socpanels`

Figure 4.11: Distribution of net view count changes for videos, for each subservice.

beyond artificially manipulating engagement metrics [146], such as copyright/trademark infringements or inappropriate content, although its metadata does not indicate the specific takedown justification. Nonetheless, we believe that it is valuable to characterize the takedown of the videos observed participating in view fraud.

Figure 4.12 plots the proportion of videos that remain online over time after the video is first advertised. We find that in total, less than 5% of `greedseed` and `xtremeserve` videos were taken down within 50 days, whereas for `socpanels`, nearly 8% of videos were removed within 10 days of being advertised (with about 5% removed within the first day). We hypothesize that `socpanels` videos may have been more heavily penalized due to a higher rate of video distribution; during our monitoring of 123Movies, we observed that `AdGlassSpy` returned `socpanels` videos more frequently than for the other two subservices. For all other videos of the channels participating in the view fraud that were not observed as advertised, we found only 5 were taken down in total (out of 818k). Thus, videos receiving view fraud are significantly more likely to be removed, although only a small minority of videos receive such penalties, and YouTube appears to primarily rely on eliminating fake views instead of taking down videos.

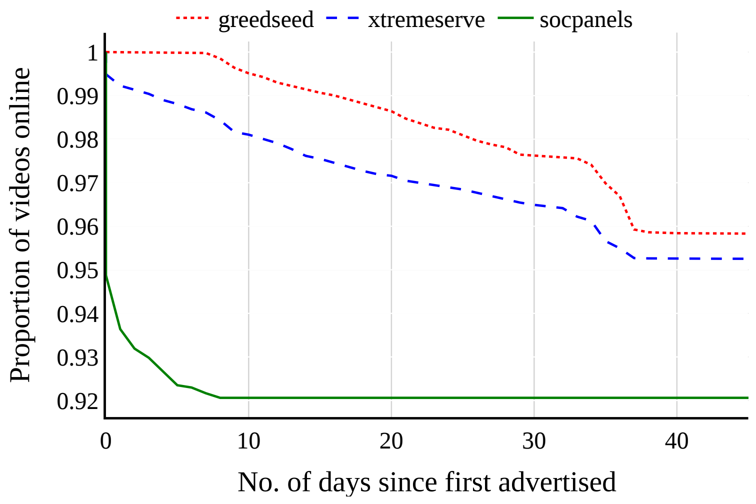


Figure 4.12: Proportion of videos receiving view fraud over time after first advertisement.

4.4 Concluding Remarks

In this study, we conducted an empirical investigation of a large-scale organic YouTube view fraud campaign. We monitored the campaign's operations over time to characterize the participants in this ecosystem, their behaviors, and the outcomes of the view fraud. What we found was an expansive ecosystem with hundreds of thousands of videos from tens of thousands of channels.

Our investigation into the success of view fraud efforts suggests that benefits are primarily short-term, and that YouTube is able to quickly detect and remove many of the fake views. We also observed that this operation has only a few "repeat customers", who typically participate only for a brief period of time, perhaps recognizing the poor return on investing in the view fraud. This brings into question how this operation continues to thrive over years. It seemingly exhibits "snake oil" properties, where the promised outcome (i.e., growth in views) is not truly delivered. Like with many snake oil scams, we hypothesize that this ecosystem survives by luring in new unsuspecting participants. However, future work can build on our initial results to study how participants are drawn into this scheme and the economics at play. It is interesting to note that over half of participating channels have over 1k subscribers, which is one of the qualifications required for receiving ad revenue from videos via the YouTube partner program [147]. It is possible that ad revenue from videos could change the cost-benefit trade-off for participants (such as through arbitrage, where fake views are cheaper than the ad revenue received), although the lack of long-term participants suggests otherwise.

Overall, the view fraud campaign remains persistent, despite YouTube's ability to detect and remove fake views. Thus, further work is needed on other methods to combat this abuse vector. For example, we observed that our studied view fraud operation relied on link redirection when loading the YouTube videos (such as using Twitter's `t.co` link shortener and Google Plus's redirection link[7]). This redirection is done presumably to obfuscate the HTTP referer, which would reveal the true location of where the YouTube videos are displayed (i.e., 123Movies) and potentially lead to easier detection and filtering by YouTube. Therefore, one signal for detecting videos involved in such campaigns could be the frequency with which a video is accessed from such redirection links. Other socio-technical directions may also need to be considered, such as more punitive penalties to discourage such view fraud activity or legal action. Ultimately, we consider our study to be a step forward in better understanding video view fraud in practice, informing future exploration.

[7]`https://plus.google.com/url?<youtube-link>`

CHAPTER 5
PRIVACY INVASIVE LOCAL NETWORK COMMUNICATIONS ON WEBSITES

Websites today stitch together Web resources from numerous sources. Beyond its own first-party content, a website fetches various third-party resources such as JavaScript (JS) libraries, CSS style sheets, images and videos, and even other webpages via inline frames. Typically, we think of these third-party resources as hosted by public Internet services, including other Web servers, content distribution networks, and API endpoints. However, nothing in modern Web browsers prevents a website from also attempting to communicate with internal network services, such as those on the browser's localhost and other devices within the Local Area Network (LAN), the addresses as defined in RFC1918 [148]. For example, a website could include JS code that initiates a request to a service running on a visitor's localhost. As a consequence, a browser affords a degree of internal network access to external entities.

Why would websites need to communicate with local destinations, given the uncertainty and diversity of local devices and network services? One legitimate reason might be for a website to coordinate with an affiliated native application (e.g., visiting a video meeting URL opens the native video conferencing software). However, prior work [20, 21, 22, 24, 25, 26] has identified that in theory, such access could also be used for malicious purposes. These studies have developed proof-of-concept demonstrations of user and LAN device fingerprinting, as well as network attacks on internal services. Particularly with user profiling and tracking, websites and advertisers have escalated to ever more complex and surreptitious methods for gathering user identifiable information [39], and one can plausibly envision the adoption of these internal network based approaches. The proliferation of consumer IoT devices [149] further exposes users to potential security and privacy violations.

In this work, we explore whether websites are using local network communications in practice. We empirically evaluate the local network behaviors of both popular and malicious websites at scale. We crawl and monitor the requests made by landing pages of the Tranco top 100K domains [150], performing two sets of measurements, half a year apart. We similarly crawl and monitor the requests made by $\sim$145K malicious websites drawn from abuse, malware, and phishing blacklists. Across these websites, we assess whether they generate locally-bound traffic, what the nature of the local traffic is, and why they may be communicating with internal network services. As different OSes support varying network services, a website's locally-bound traffic may depend on the underlying host OS. Thus, we also explore how websites' localhost and LAN behavior may differ across three popular desktop OSes: Windows, Ubuntu, and Mac OS X.

In total, we detect landing pages of over 100 benign domains and over 150 malicious websites communicating with localhost services or private network IP addresses. While this population may be relatively small, indicating that this behavior is not widespread (yet) among websites, it is a non-trivial number of sites exhibiting rather unexpected network activity. Furthermore, several of the observed websites are highly ranked with millions of users, including 19 sites ranked within the top 10K. We identify that for many sites, the internal network activity is not uniform across OSes, particularly skewing towards activity exclusively on Windows. Additionally, we note the extensive use of WebSockets for initiating such communication, which bypasses the Same-Origin Policy. These characteristics potentially suggest the intentional targeting of certain platforms.

To understand why these websites may be contacting services hosted on the local network, we manually analyze their behavior. For popular websites, we find that over 40% of them explicitly conduct host profiling. Upon deeper investigation, we determine that this profiling is performed for fraud protection and bot detection, rather than explicit advertising or user tracking purposes (although this approach could be readily adapted for such uses). We also observe a large set of top websites whose local network activity arises due to remnants of website development and testing. These cases should be benign, but highlight a class of Web developer errors that should be straightforward to detect and remediate. Finally, we also identify a legitimate scenario for localhost communication where a top website communicates with its affiliated native application. Efforts to protect users from malicious local network traffic generated by a website must preserve these valid use cases.

For malicious sites, we do not find evidence of internal network attacks. Rather, the observed local network traffic mirrors that of top sites. We detect that some phishing sites have cloned their Web interfaces from a legitimate site, where the legitimate site itself is conducting localhost profiling. As a consequence, the phishing sites host the same JavaScript libraries as the target site, and thus also generate the same local network traffic. For the other malicious sites, we find that the local network activity reflects the same sorts of Web developer errors seen on top sites, indicating that either the attackers made similar errors while developing their attack sites, or that perhaps more likely, these sites are compromised and the local traffic reflects the developer errors exhibited on the original benign sites. Overall, we do not attribute malicious intent to the local network traffic witnessed on these malicious sites.

Ultimately, our study provides the first large-scale measurement of the internal network behaviors of modern websites, uncovering both intentional and unintentional causes of locally-destined traffic. We conclude by discussing the implications of our findings on Web security and privacy, as well as potential directions for advancing user online safety.

5.1 Method

In this section, we describe our method for examining if and how websites interact with users' localhost and LAN resources. We consider two sets of websites, one consisting of popular sites, and the other consisting of known malicious webpages (e.g., phishing and malware sites).

5.1.1 Data Collection

Measurement Populations. To study popular websites, we focus on the top 100K domains in the Tranco top list [150]. This set of domains affords a large-scale exploration of top websites that users visit. We conduct two sets of measurements of the landing pages of the top 100K domains – one using the Tranco snapshot taken on June 3, 2020, and the other taken on March 11, 2021. To evaluate the local network behavior of malicious webpages, we collect $\sim$145K known malicious URLs that were actively listed on one of several blocklists: SURBL (abuse, malware, and phishing sites) [151], Abuse.ch's URLHaus (malware sites) [152] and PhishTank (phishing sites) [153]. As these blocklists often list multiple malicious URLs mapping to the same domain, we only select one malicious URL per domain to increase our measurement's coverage of malicious domains. We monitored and crawled webpages on the blocklists from March – April 2021.

Measurement Setup. As depicted in Figure 5.1, for both the landing pages of top websites and malicious webpages, we fetch and render (with JavaScript enabled) the target webpage using a full browser instance running within a virtual machine (VM), while monitoring the network requests generated by the website. Our Web crawler starts a full Google Chrome (v84) instance with a clean profile (incognito mode) via Chrome's command-line interface[1]. As we are exploring how a website may interact with localhost and LAN resources, which are external to the browser and may vary depending on the underlying OS, we visit all websites across three popular desktop OSes: Windows 10, Linux (specifically Ubuntu 20.04), and Mac OS X (v10.15.6). We conduct the Windows 10 and Linux Web crawls within VMWare VMs, running on a server within Georgia Tech's network (with firewall disabled). For the Web crawl on Mac OS X, we perform the crawl directly on a MacBook Air laptop residing on a residential Comcast network in Atlanta.[2] Throughout

[1]We avoid using more popular crawling tools, such as Puppeteer [154] and Selenium [155], as a website can detect their presence [156] and change its behavior – potentially contaminating our measurement.

[2]As Mac OS X is licensed only for use on Apple hardware, we conducted our Mac-based crawls directly on a Mac laptop.

our measurements, we disable Chrome's Safe Browsing feature, which blocks network requests to destinations on Google's Safe Browsing blocklist [157]. This configuration is necessary to ensure that Safe Browsing does not interfere with our measurements, such as blocking the browser instance from visiting a malicious page.

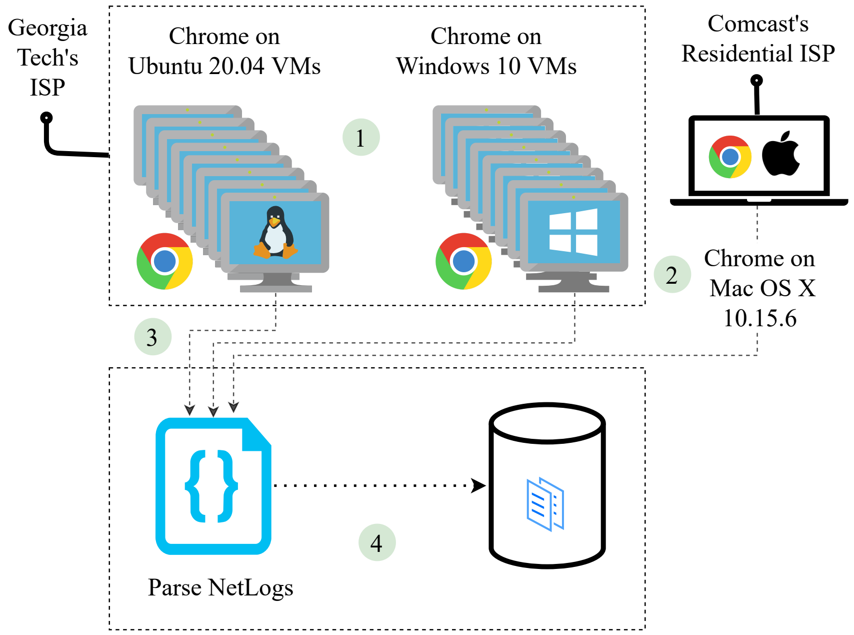

Georgia Tech's ISP
Chrome on Ubuntu 20.04 VMs
Chrome on Windows 10 VMs
Comcast's Residential ISP
1
2
Chrome on Mac OS X 10.15.6
3
4
Parse NetLogs

Figure 5.1: Our data collection method for measuring websites' local network activity.

Web Telemetry. Across all three OSes, for each target webpage, we load the page and monitor its network activity over a 20 second period, to permit time for execution of JS code and loading of dynamic resources. To select this delay threshold, we randomly sampled 100 websites (landing pages of domains from the top 100K list) and noted the time that it takes to completely load the page. We found that more than 98% of all requests (to any resource, local or otherwise) were made within the first 15 seconds, with the majority being made within the first 5 seconds. Prior research also suggests that users often spend 10–20 seconds on a webpage before navigating elsewhere [158]. Therefore, to measure the most common effect of these websites on users, we chose a delay threshold of 20 seconds. Since our analysis of our first crawl of top websites (in section 5.2.2) suggested that this threshold suffices in detecting the majority of localhost and LAN activity, we continued with the

Table 5.1: Summary of localhost and LAN requests found for malicious webpages.

Domain Type (Malicious Category)	**# Sites**	**Data Sources (% Contribution)**	**Crawl Success Rate**			**Activity Detected (# Sites)**					
						Localhost			**LAN**		
			W	**L**	**M**	**W**	**L**	**M**	**W**	**L**	**M**
Malware	103541	Abuse.ch (99%), SURBL (1%)	61%	65%	65%	72	83	75	8	7	7
Abuse	24958	SURBL (100%)	95%	97%	93%	0	0	0	1	1	1
Phishing	16426	PhishTank (85%), SURBL (15%)	73%	76%	69%	25	41	9	0	0	0

threshold in the second crawl.

For gathering network telemetry generated by a webpage while it loads, we record the network logs generated by Google Chrome's network logging system [159], which comprehensively logs all network events (i.e., any network requests sent and responses received) on Chrome's network stack. Among other data, this telemetry includes the following for each event:

- `time`: specifies the timestamp for the network event.
- `type`: specifies the type of network event as defined by Chrome. For example, `URL_REQUEST` events indicate GET/POST requests.
- `source`: specifies the entity that generated the event. When a new network request is initiated, it is assigned a new source ID (in serial order). Subsequent dependent events (e.g., responses) are assigned the same source ID, allowing the events within a network flow to be logically grouped together.
- `phase`: specifies the phase of the network event (either `BEGIN`, `END`, or `NONE`).

We parse and store the network logs in a database for efficient querying. From this telemetry, we can comprehensively observe whenever a webpage initiates a request destined for localhost or an IP address in the private IP address range, what precisely that destination is (e.g., the full URL), and what entity generated this request (the Chrome browser itself also generates network traffic, which we filter out based on the network event source). Additionally, we also consider websites that redirect to a local destination, since in theory, websites can send a request to a local resource, even if they can never receive the response.

Throughout our crawl, before visiting a webpage, we first check for network connectivity by pinging Google's DNS server (`8.8.8.8`). This ensures that we crawl a site only when the measurement infrastructure has Internet connectivity, and thus we can differentiate between website load failures and network issues on our end.

Ethics. As we conduct a large-scale Web measurement, we take care to minimize the load induced on the websites we visited. For each OS, we visit each site only once, and we start measurements on each OS at different times. Thus, we visit each site up to three times without concurrent visits, which should be a manageable load for websites.

5.1.2 Data Characterization

Our final datasets consist of 11 TBs of telemetry data collected from the following measurements:

1. Tranco Top 100K domains (as of June 3, 2020) on all three OSes, measured from July 24 to September 25, 2020.
2. Tranco Top 100K domains (as of March 11, 2021) on Windows and Linux[3], measured from March 22 to May 1, 2021.
3. $\sim$145K known malicious webpages on all three OSes, measured from March 22 to May 1, 2021.

We observe a $\sim$75% overlap between the two Tranco top list snapshots we used, indicating that the majority of top domains in our dataset were measured twice half a year apart. In total, we successfully loaded and gathered telemetry from $\sim$90% of domains in both sets of Tranco top list measurements. This success rate is commensurate with that observed by the creators of the Tranco top list [150]. For the measurement of malicious webpages, we successfully gathered telemetry from $\sim$70% of webpages. We further manually investigated a random sample of webpages that were not successfully crawled and confirmed that they remained inaccessible, giving us confidence that our collected data should comprehensively cover the accessible websites.

Table 5.2 shows our crawl success rate statistics, including the top errors when failing to load a domain's landing page. Nearly 90% of errors in all three crawls were due to DNS name resolution failures. Manually examining a random sample of these DNS errors, we

[3]We encountered logistical issues preventing us from conducting the second Tranco top list measurement on Mac OS X, related to our need to execute the crawl on a bare-metal environment.

observed that the domains indeed lacked name resolutions, but we could identify subdomains (e.g., through search engines) for which name resolutions existed. We note that in many cases, the domains are CDNs or API endpoints. Recall that our periodic connectivity test did not reveal any network outages on our end, indicating that these errors did not arise due to network connectivity issues with our measurement setup. We also explored the rank of the domains that we failed to crawl and observed that they were evenly distributed among the top 100K, indicating that the errors do not skew towards high or low-ranked domains.

5.1.3 Limitations

The data collected in this study affords a large-scale evaluation of website-generated network traffic. However, there are several important limitations to our methodology.

- Our measurements are conducted using Google Chrome and three desktop OSes, which are widely available and popular. However, websites may behave differently on other browsers and OSes (such as mobile OSes). Future work can extend our method to evaluate these other platforms.

- We only fetch the landing pages of top domains, and our crawler does not simulate user interactions with the webpages. As a consequence, our data does not capture the behavior of internal website pages, nor activity triggered by user actions. Thus, the number of domains for which we identify local network activity is a lower bound of the true number of websites exhibiting such behavior. We leave a broader exploration of these dimensions for future work.

- We monitor a webpage's behavior for 20 seconds. This threshold was determined based on an experiment we performed, as described in section 5.1.1, as well as prior research on user browsing behavior [158]. Our subsequent analysis (in section 5.2.2) suggests that this threshold suffices in detecting the vast majority of website localhost

Table 5.2: Web Crawl Statistics.

Type of Crawl	**OS**	**# successful loads**	**# failed loads**	**Error Type**				
				NAME_NOT_ RESOLVED	***CONN_ REFUSED***	***CONN_ RESET***	***CERT_CN_ INVALID***	***Others***
Top 100K: 2020	Windows	89744 (89.7%)	10256 (10.3%)	9179 (89.4%)	355 (3.5%)	248 (2.4%)	236 (2.3%)	238 (2.3%)
Top 100K: 2021		91765 (91.7%)	8235 (8.2%)	7287 (88.5%)	239 (3.0%)	230 (2.8%)	251 (3.1%)	228 (2.8%)
Top 100K: 2020	Mac	89819 (89.9%)	10181 (10.1%)	9001 (88.4%)	345 (3.8%)	193 (1.9%)	226 (2.2%)	416 (4%)
	Linux	90175 (90.2%)	9825 (9.8%)	8612 (87.6%)	335 (3.4%)	247 (2.5%)	235 (2.4%)	396 (4%)
Top 100K: 2021		91719 (91.7%)	8281 (8.3%)	7309 (88.3%)	272 (3.2%)	126 (1.5%)	248 (3%)	326 (3.9%)
Malicious	Windows	100317 (68.6%)	45864 (31.4%)	40715 (88.7%)	1475 (3.2%)	530 (1.1%)	1341 (2.9%)	1803 (3.9%)
Malicious	Mac	103154 (70.6%)	43027 (29.4%)	37310 (86.7%)	1488 (3.4%)	523 (1.2%)	1314 (3%)	2392 (5.5%)
Malicious	Linux	106078 (72.6%)	40103 (27.4%)	34723 (86.5%)	1346 (3.3%)	521 (1.3%)	1313 (3.2%)	2200 (5.5%)

and LAN activity. However, we would fail to detect such behavior on websites where the activity commences beyond our 20 second threshold.

- Due to logistical constraints, our Web crawl on Mac OS X was conducted on Comcast's residential network, whereas our Windows and Linux Web crawls were performed on Georgia Tech's network. In theory, crawl results could differ due to varying network configurations or network-dependent (including a network's geolocation) website behavior. However, we did not identify significant network-based discrepancies in our Web crawl telemetry – neither during our manual investigations of websites nor during our data analysis.

5.2 Findings

Here, we analyze our datasets on network activity generated by websites, to answer three research questions about the activity destined to a visitor's local network (i.e., localhost or LAN).

- **RQ1:** Which websites are generating local network traffic?
- **RQ2:** What are the characteristics of local network traffic?
- **RQ3:** Why are websites making local network requests?

For localhost activity, we detect requests generated by websites that are destined for either the localhost domain or loopback IP addresses (i.e., `127.0.0.1` for IPv4 and `::1` for IPv6). To identify LAN activity, we identify requests made by websites to IP addresses in the IANA-reserved private IP address ranges for IPv4 and IPv6 [148]. We did not observe any localhost or LAN network traffic over IPv6 though, so subsequent discussion is exclusively for IPv4.

For landing pages of Tranco top 100K domains, we first present the results from the dataset measured in 2020, and then compare our findings to those from the 2021 top 100K measurements. As mentioned in section 5.1.2, domains in the Tranco snapshots used for the two measurements overlap extensively, and many findings remained consistent. Thus, for the 2021 top 100K measurements, we focus on highlighting the changes observed. We additionally characterize our observations about malicious webpages in comparison to the top sites.

5.2.1 RQ1: Which Websites are Generating Local Network Traffic?

During our first measurement of sites from the Tranco top 100K in 2020, we found a total of 107 sites making localhost requests (as listed in Table 5.3 and Table 5.11) and 9 sites generating LAN requests (as shown in Table 5.4), with no overlap between the two sets of sites. These sites account for 0.12% of the ones successfully crawled, indicating that local network activity on websites is not currently widespread. However, such activity does occur on a non-trivial set of websites.

Similarly, in our 2021 top 100K measurements, we observed 82 sites making localhost requests (as listed in Table 5.5) and 8 sites generating LAN requests (as shown in Table 5.6), with again no overlap between the two sets of sites. Out of the 82 sites generating localhost requests, 19 sites were crawled in 2020 but were not observed as generating such traffic, whereas 21 sites were not crawled in 2020 (as their domains were not listed in the top 100K). The remaining sites exhibited the same behavior in both measurements. For the sites generating LAN requests, only one site was found performing LAN requests in both 2020 and 2021, while the others from 2020 stopped in 2021.

Behavior Across OSes. We observe that the set of websites generating localhost traffic is not uniform across OSes. As shown in Figure 5.2a, during the 2020 top 100K crawl, we found 92 sites (86%) generating localhost requests when on Windows, compared to 54

Table 5.3: Summary of website localhost requests (2020 crawl).

Domains marked with an asterisk (*) did not make localhost requests during the 2021 top 100K crawl.

(a) For Fraud Detection.

Rank (↓)	Domain	Protocol	Ports	Paths	OS		
					W	L	M
105-45156	ebay.{at, ca, ch, co.uk, com, com.au, com.my, com.sg, de, es, fr, ie, in, it, nl, ph, pl, us}	wss	3389, 5279, 5900-03, 5931, 5939, 5944, 5950, 6039-40, 63333, 7070	/	✓		
1251	fidelity.com				✓		
1289-7907	citi{.com, bank.com, bankonline.com}*				✓		
5680	marktplaats.nl*				✓		
7441	betfair.com				✓		
13119-57251	tiaa{.org, -cref.org}*				✓		
13901	2dehands.be*				✓		
25990	santanderbank.com				✓		
29104	ameriprise.com				✓		
34251	commoncause.org*				✓		
45228	ctfs.com*				✓		
50853	2ememain.be*				✓		
90641	highlow.net				✓		
97182	metagenics.com				✓		

(b) For Bot Detection.

Rank (↓)	Domain	Protocol	Ports	Paths	OS		
					W	L	M
8608	sbi.co.in*	http	4444, 4653, 5555, 7054-55, 9515, 17556	/	✓		
25881	cnes.fr*				✓		
27491	din.de*				✓		
32114	csob.cz*				✓		
48803	anaf.ro*				✓		
55267	data.gov.in*				✓		
55852	allegiantair.com*				✓		
58948	tmdn.org*				✓		
65955	beuth.de*				✓		
99638	bank.sbi*				✓		

sites (50%) for both Linux and Mac. Note though that the set of 54 sites for Linux and Mac are not equivalent, with 5 sites generating activity on Mac but not Linux, and 5 other sites active for Linux but not Mac. Overall, only 41 sites (38%) behaved equivalently on all three OSes. While few sites produced localhost traffic exclusively on Mac (5 sites) and Linux (2 sites), 48 sites (45%) did so on Windows 10, which suggests a degree of targeting

Table 5.3: Summary of website localhost requests (2020 crawl) contd.

(c) To communicate with Native Applications.

Rank (↓)	Domain	Protocol	Ports	Paths	OS		
					W	L	M
5370	faceit.com	ws	28337	/	✓	✓	✓
23219	cponline.pw (-)	ws	6463-72	/?v=1	✓	✓	✓
29301-77550	samsungcard{.com, .co.kr}	wss	10531, 31027, 31029	/	✓	✓	✓
		https	14440-9	/?code=*&dummy=*	✓	✓	✓
36141	gamehouse.com*	http	12071-72, 17021, 27021	/v1/init.json?api_-port=*&query_id=*	✓	✓	✓
47690	games.lol	ws	60202	/check	✓	✓	
57008	zylom.com		12071, 17021	/v1/init.json?api_-port=*&query_id=*	✓	✓	✓
74089	iwin.com		2080-82	/version?_=*		✓	✓
77134	screenleap.com (-)	http	5320	/status, /*/up	✓	✓	✓
88902	acestream.me (-)		6878	/webui/api/service	✓	✓	✓
91904	trustdice.win		50005, 51505, 53005, 54505, 56005	/, /socket.io	✓	✓	✓
98789	runeline.com (-)	ws	6463-72	/?v=1	✓	✓	✓

(d) For unknown reasons.

Rank (↓)	Domain	Protocol	Ports	Paths	OS		
					W	L	M
244	hola.org	http	6880-9	/*.json	✓	✓	✓
21246	wowreality.info		1080, 1194, 2375, 2376, 3000, 3128, 3306, 3479, 4244, 5037, 5242, 5601, 5938, 6379, 8332, 8333, 8530, 9000, 9050, 9150, 9785, 11211, 15672, 23399, 27017	/	✓	✓	✓
62048	svd-cdn.com		6880-9	/*.json	✓	✓	✓
78456	usaonlineclassifieds.com*	ws	2687, 26876	/	✓		
84569	usnetads.com*				✓		

towards Windows users (explored further in section 5.2.3).

In our 2021 top 100K measurements, we observed that behavior on Windows and Linux was similar, with 47 sites behaving equivalently on both OSes. For malicious webpages, as seen in Figure 5.2b, we also find that localhost traffic is not observed uniformly across OSes, although here, Linux elicited localhost behavior on more webpages. Similarly, we found that LAN activity on websites is also not consistent across OSes (although the population sizes are small enough to prevent identifying clear OS skew).

Table 5.4: Summary of LAN requests made by websites.

Rank (↓)	**Domain**	**Protocol**	**Local IP Address**	**Port**	**Paths**	**OS**		
						W	**L**	**M**
4381	gsis.gr	http	10.193.31.212	80	/system/files/2020-06/*.png	✓	✓	✓
19523	farsroid.com	http	10.10.34.35	80	/		✓	
35262	saddleback.edu	https	10.156.2.50	443	/*.ico			✓
46972	skalvibytte.no	http	10.0.0.200	80	/wordpress/wp-content/uploads/2020/04/*{.jpg,.mp4}	✓	✓	✓
56325	unib.ac.id	http	192.168.64.160	80	/wp-content/uploads/2019/10/*.jpg	✓	✓	✓
61554	adnsolutions.com	http	10.0.20.16	80	/wp-content/uploads/2018/11/*.jpg			✓
65302	tra97fn35n5brvxki5-sj8x5x34k2t4d67j883fgt.xyz	http	10.10.34.35	80	/		✓	
73062	zoom.lk	https	192.168.0.208	443	/wp_011_test_demos/wp-content/uploads/2017/05/*.jpg			✓
91632	1-movies.ir	http	10.10.34.35	80	/	✓	✓	✓

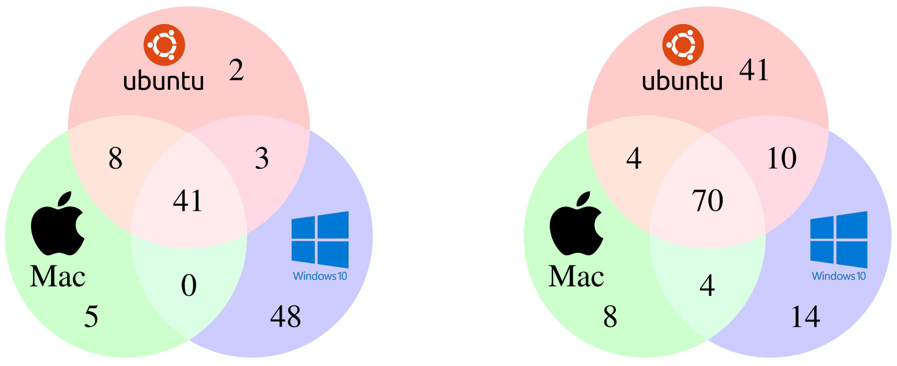
Table 5.5: Summary of website localhost requests (2021 crawl).

(a) For Fraud Detection.

Rank (↓)	Domain	Protocol	Ports	Paths	OS	
					W	L
2912	cibc.com	wss	3389, 5279,	/	✓	
8173	betfair.com (+)		5900-03,		✓	
10679	highlow.com		5931, 5939,		✓	
28370	moneybookers.com		5944, 5950,		✓	
31170	ebay.com.hk		6039-40,		✓	
64012	marks.com		63333, 7070		✓	

(b) To communicate with native applications.

Rank (↓)	Domain	Protocol	Ports	Paths	OS	
					W	L
592	iqiyi.com	http	16422-23	/get_client_ver?*	✓	✓
7664	qy.net				✓	✓
10966	qiyi.com				✓	✓
12350	iqiyipic.com				✓	✓
15581	ppstream.com				✓	✓
34989	ppsimg.com (+)				✓	✓
44280	soliqservis.uz (+)	wss	64443	/service/cryptapi	✓	✓
75083	nfstar.net (+)	http	28317, 36759	/get_thunder_version/	✓	✓
80108	9ekk.com (+)				✓	✓
87274	somode.com (+)				✓	✓
82814	mcgeeandco.com (+)	https	4000	/socket.io/?	✓	✓
86605	71.am (+)	http	16422-23	/get_client_ver?*	✓	✓
94270	didox.uz (+)	wss	64443	/service/cryptapi	✓	✓
96284	gnway.com (+)	ws	38681-87	/	✓	



(a) 2020 top 100K crawl (b) Malicious webpages

Figure 5.2: Overlap in localhost activity for sites across OSes.

Website Rankings. We also investigate whether the population of websites that make local requests is skewed based on site popularity. In Figure 5.3, we plot the CDFs of the

(c) Summary of website localhost requests (2021 crawl) contd.

(d) As a result of developer errors.

Rank (↓)	**Domain**	**Protocol**	**Ports**	**Paths**	**OS**	
					W	**L**
5154	phonearena.com	http	1500	/floor-domains	✓	✓
5331	madmimi.com		5555	/2.1.2/sockjs.min.js	✓	
14951	nursingworld.org		80	~4af7b9/globalassets/images/*.jpg	✓	
21280	ums.ac.id			/ums-baru/wp-content/*	✓	✓
25940	zee.co.ao (+)			/industrialwp/wp-content/*	✓	✓
37323	raovatnailsalon.com (+)	https	443	/raovatnailsalon/wp-content/*	✓	✓
42107	panduit.com	http	4502	/apps/panduit/clientlibs/*.js	✓	
45497	internetworld.de	https	443	/	✓	✓
47861	mcknights.com		9988	/livereload.js		✓
50650	san-servis.com	http	80	/vina/vina_febris/images/*	✓	✓
54756	postfallsonthego.com (+)			/magazon/magazon-wp/wp-content/uploads/*	✓	✓
55755	wealthcareportal.com (+)			/NonExistentImage48762.gif	✓	✓
55477	lited.com		110066	/getversionjpg?hash=*	✓	
68872	workpermit.com	https	6081	/news-ticker.json	✓	✓
75989	ethiopianreporterjobs.co (+)		443	/wp-content/uploads/*	✓	✓
77974	macroaxis.com (+)	http	8080	/img/icons/search.png	✓	✓
83256	adfontesmedia.com (+)		8888	/adfontesmedia/wp-content/uploads/*	✓	✓
84378	charityvillage.com (+)			/core/js/api/web-rules	✓	✓
90632	showfx.ro (+)	https	443	/wordpress/x-street/wp-content/*	✓	✓
98402	xaydungtrangtrinoithat.com (+)			/wp-content/uploads/*	✓	✓

ranks of the domains whose landing pages were found to be actively generating traffic to localhost in our 2020 top 100K measurements, across the three OSes. We observe fairly linear CDF curves, indicating that our detected domains are spread uniformly throughout the top 100K domains. Notably, there are highly-ranked sites (with millions of users) that display localhost behavior, such as those listed in Table 5.7. In our 2021 top 100K measurements, we see a similar distribution, as visible in Figure 5.4. As listed in Table 5.4 and Table 5.6, the ranks of the domains whose landing pages were found making requests to LAN destinations are also similarly distributed across the top 100K, with the highest site ranked at 4381 in 2020 and 4847 in 2021. None of the domains belonging to malicious webpages were found in the Tranco top 100K.

Table 5.6: Summary of LAN requests (2021 crawl).

Rank (↓)	Domain	Protocol	Local IP Address	Port	Paths	OS	
						W	L
4847	blogsky.com	http	10.10.34.34	80	/	✓	✓
23723	jollibeedelivery.qa		192.168.8.241	5000	/MyPhone/c2cinfo	✓	✓
47356	unib.ac.id	https	192.168.64.160	443	/wp-content/uploads/2019/10/*.jpg	✓	
61472	bahrain.bh		192.168.110.72		/matomo/*.js	✓	✓
69494	auda.org.au		10.50.1.242	8450	/libraries/slick/slick/*.gif	✓	✓
73274	mre.gov.br		192.168.33.187	443	/modules/mod_acontece/assets/*		✓
95595	haiwaihai.cn	http	172.16.0.4	1117	/UpLoadFile/20160801/*.jpg	✓	✓
96554	techshout.com	https	192.168.0.120	443	/wp_011_gadgets/wp-content/uploads/*	✓	✓

Table 5.7: Top domains whose landing pages made localhost requests.

Rank (↓)	Windows	Rank (↓)	Linux and Mac
104	ebay.com	243	hola.org
243	hola.org	5369	faceit.com
429	ebay.de	7699	zakupki.gov.ru*
536	ebay.co.uk	17826	rkn.gov.ru*
932	ebay.com.au	19243	cruze...sulvirtual.com.br*
1250	fidelity.com	21245	wowreality.info
1288	citi.com*	22729	smartcatdesign.net
1843	ebay.it	23218	cponline.pw*
2200	ebay.fr	24739	gamezone.com
2394	ebay.ca	26399	filemail.com

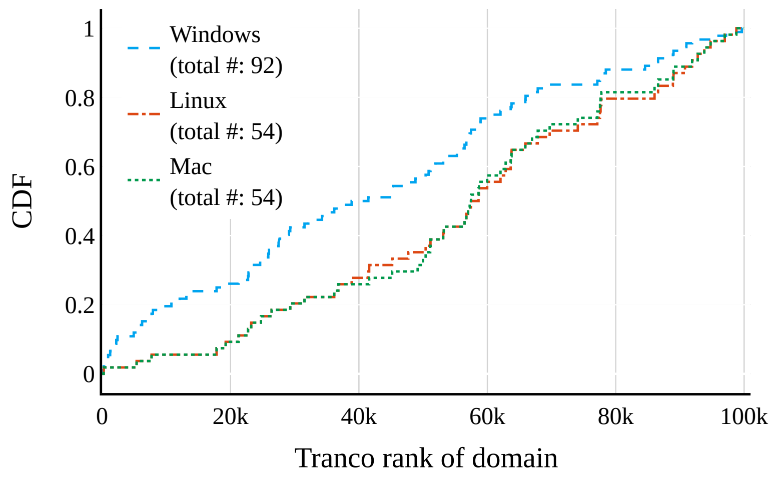


Figure 5.3: CDFs of the domain ranks for top sites generating localhost traffic (2020 crawl).

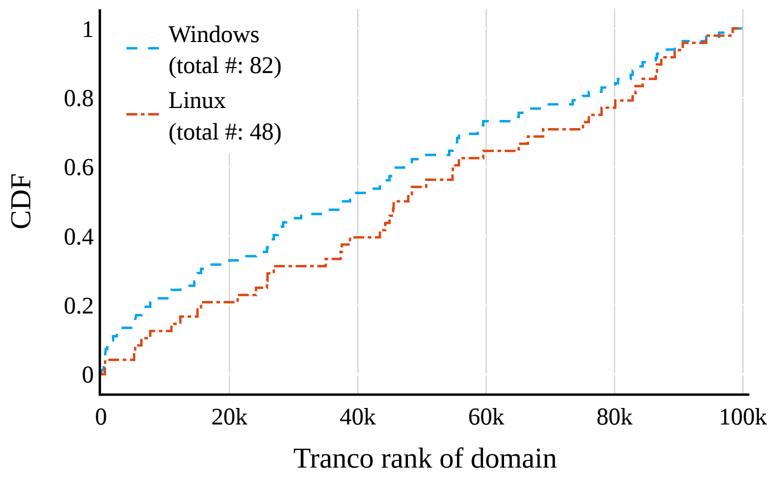


Figure 5.4: CDFs of the domain ranks for top sites generating localhost traffic (2021 crawl).

5.2.2 RQ2: What are the Characteristics of the Local Network Traffic?

Here we consider what protocols and ports are used by websites for requests to the local network, and when requests are generated.

Protocols and Ports. Figure 5.5a depicts the protocols and ports used in the localhost requests we observed in our 2020 top 100K measurements, across the three OSes. (Data on individual sites is listed in Table 5.3 and Table 5.11.) We observe that the majority (about 60%) of localhost requests on Windows were made using the Secured WebSockets (WSS) protocol, to a set of 14 ports. The use of WebSockets is interesting as it is not bound by the Same-Origin Policy, potentially allowing bidirectional network communication. In comparison, only a quarter of requests on Windows were sent over HTTP, with an eighth sent over HTTPS (primarily to the default HTTP(S) ports). Linux and Mac exhibited the opposite pattern, with 86% of requests sent over HTTP and HTTPS (primarily to the default ports) for both OSes.

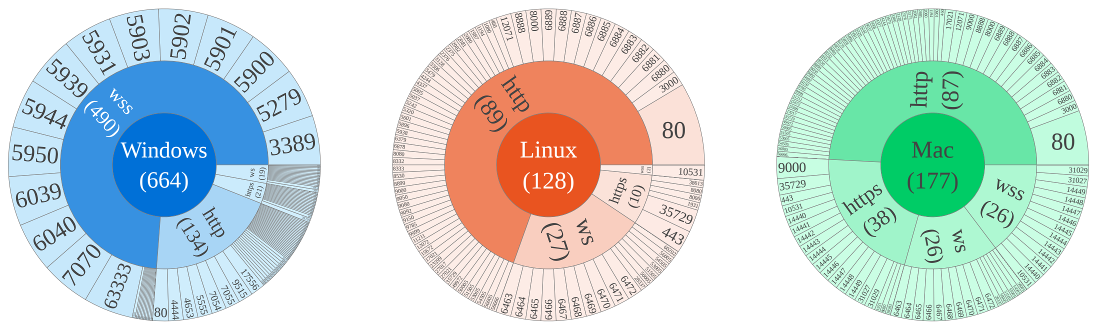
In the 2021 top 100K crawl, as well as the crawl of malicious webpages (shown in Figure 5.6 and Figure 5.5b, respectively), we see largely a subset of the ports and protocols observed with the 2020 top 100K crawl. We see fewer ports and protocols in our later measurements due to some changes in the types of local network activity exhibited by websites, as examined further in subsection 5.2.3.

For LAN traffic, as shown in Table 5.4 and Table 5.6, all requests from top 100K sites (in both the 2020 and 2021 crawls) were made to either HTTP or HTTPS, using the standard ports. This held true for most malicious webpages (as seen in Table 5.8), except for one site requesting HTTP on port 1080.

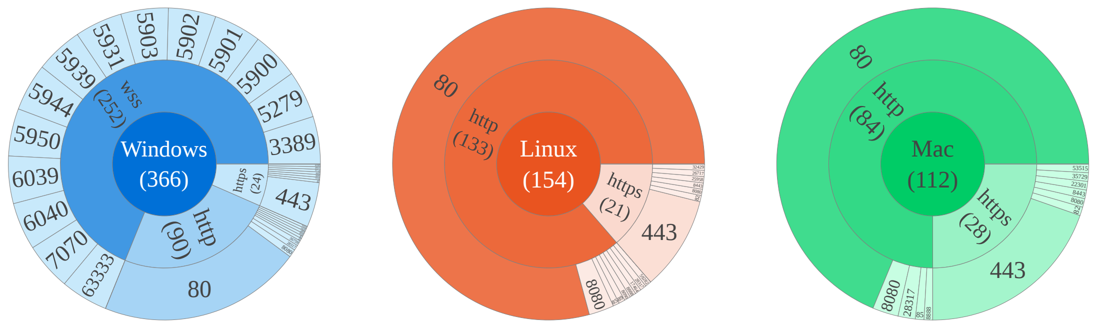
Request Timing. Figure 5.7a depicts the CDFs of the time between when a landing page is successfully fetched and when we begin observing localhost requests, across all OSes, for our 2020 top 100K measurements. We observe that for Linux and Mac, over half of

Table 5.8: Summary of LAN requests found for malicious webpages.

Malicious Category	**Domain**	**Protocol**	**Local IP Address**	**Port**	**Paths**	**OS**		
						W	**L**	**M**
Malware	test.laitspa.it	http	10.2.70.15	80	/*.css	✓	✓	✓
	wangzonghang.cn		192.168.0.226	1080	/wp-content/themes/*	✓		✓
	{www.}crasar.org		192.168.1.8	80	/crasar/wp-content/themes/*	✓	✓	✓
	mihanpajooh.com		10.10.34.35		/	✓	✓	
	ahs.si	https	192.168.33.10	443	/wp-content/uploads/2019/12/*.png	✓		✓
	fixusgroup.com		172.26.6.230		/wp-content/uploads/2020/02/*.png	✓	✓	✓
	zoom.lk	http	192.168.0.208	80	/wp_011_test_demos/ wp-content/uploads/2017/05/*.jpg	✓	✓	✓
Abuse	001tel.com		172.16.205.110		/usershare/*.js	✓	✓	✓



(a) 2020 top 100K crawl



(b) Malicious webpages

Figure 5.5: Protocols and ports used for website requests to localhost, across OSes.

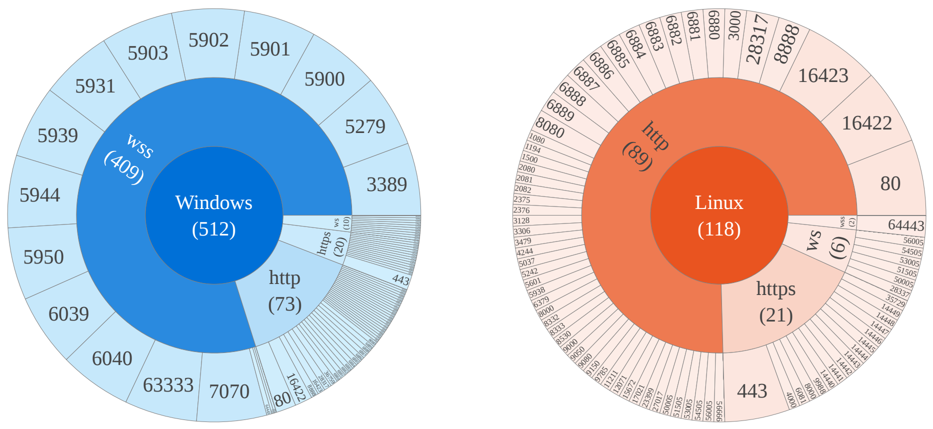
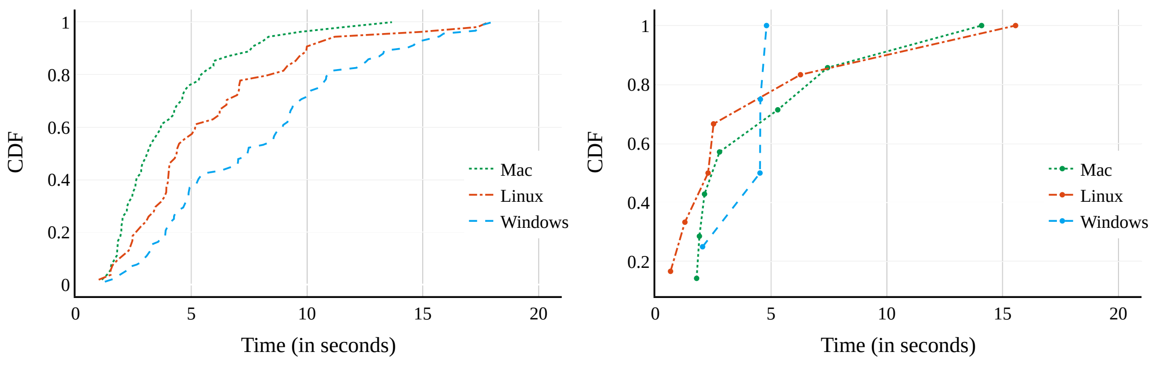


Figure 5.6: Protocols and ports used for website requests to localhost (2021 crawl).

the localhost requests are initiated within 5 seconds after the landing page is fetched. For Windows, the median gap is 10 seconds. The maximum delay is 14 seconds for Mac and 17 seconds for both Linux and Windows. Similarly, Figure 5.7b shows the time delay between when a landing page is fetched and when we begin observing LAN requests, across OSes, for our 2020 top 100K measurements. Here, the median delay for all three OSes is within 5 seconds. The maximum delay is 5 seconds on Windows, 15 seconds on Mac, and 16 seconds on Linux. These delays reveal that in many cases, the localhost and LAN traffic generate by websites do not immediately manifest. Our observations remain roughly consistent over our 2021 top 100K crawl and our crawl of malicious webpages, as shown in Figure 5.8 and Figure 5.9.



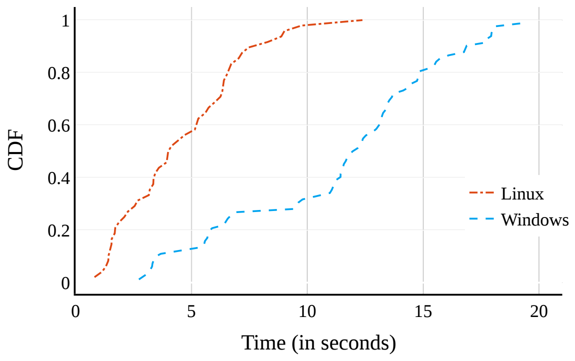
(a) Requests to localhost. (b) Requests to LAN addresses.

Figure 5.7: CDFs of time delays for the first local network request (2020 crawl).

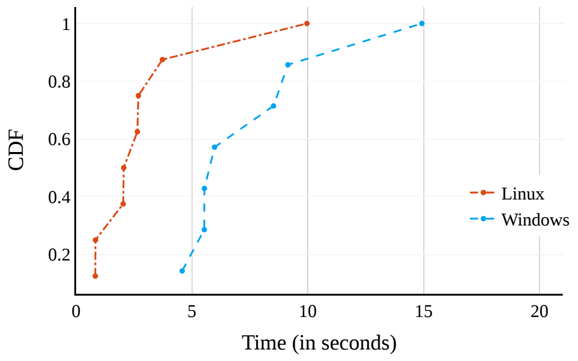
5.2.3 RQ3: Why are Websites Making Local Network Requests?

To understand why websites may be generating local traffic, we first manually investigate each website exhibiting local network activity in our 2020 top 100K measurements, identifying which elements of the webpages are originating the localhost and LAN requests in an attempt to determine the underlying reason. We subsequently also compare the observations with those from the 2021 top 100K crawl as well as the crawl of malicious webpages.

Through our exploration of the 107 websites found making localhost requests in our 2020 top 100K measurements, we identify four main classes of behavior: 1) 36 websites were scanning localhost ports for fraud detection, 2) 10 websites were conducting bot de-

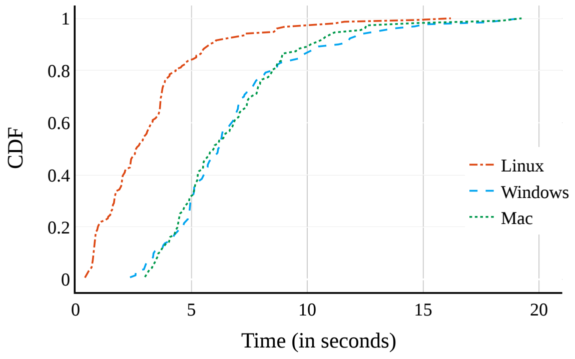


(a) Requests to localhost.

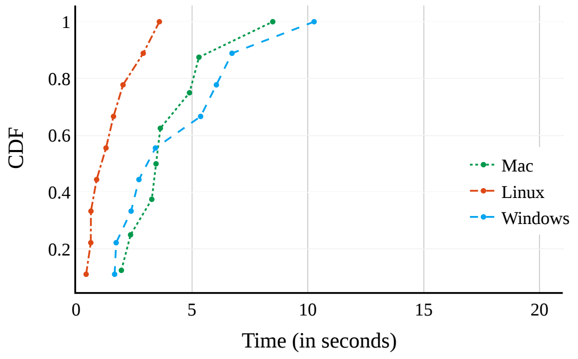


(b) Requests to LAN addresses

Figure 5.8: CDFs of time delays for the first local network request (2021 crawl).



(a) Requests to localhost.



(b) Requests to LAN addresses

Figure 5.9: CDFs of time delays for the first local network request (malicious site crawl).

tection, 3) 12 websites were found communicating with native applications, and 4) 44 websites requested localhost resources likely due to developer error. For the 5 remaining websites, we could not ascertain the cause of the localhost requests. We have listed the characterization of these domains in Table 5.3 and Table 5.11.

Among the 9 LAN-requesting websites in our 2020 top 100K measurements, listed in Table 5.4, we believe that 6 websites were doing so due to developer error. For the remaining three (which all requested the HTTP root directory), we were unable to clearly identify an underlying reason.

We observe a strict subset of these classes of local network behavior for our 2021 top 100K crawl and the crawl of malicious webpages. Thus in this section, as we discuss each behavior class, we first discuss results from our 2020 top 100K crawl and then draw comparisons between the different datasets.

Fraud Detection

As shown in Table 5.3, for the first set of rows where the localhost network activity reason is *Fraud Detection*, we encountered 36 websites in our 2020 top 100K crawl that make the same set of WebSocket Secure (WSS) requests to 14 distinct localhost ports, all using the same URL path and operating only on Windows 10.

We determine that these websites were deploying a fraud detection script from LexisNexis called ThreatMetrix [160], which claims to help the websites distinguish between trusted transactions and fraudulent behavior. On each of these websites, we identified that all of the localhost requests were made by a dynamically-generated JavaScript blob object. We observe that these blobs are generated by an external JavaScript resource loaded from either a subdomain (e.g., `regstat.betfair.com` for `betfair.com`) or similar-appearing domain (e.g., `ebay-us.com` for `ebay.com`). The JavaScript blobs collect telemetry from the localhost WSS requests, and upload the results back (in an encrypted format) to the external JavaScript-hosting site (e.g., `ebay-us.com`). Conducting WHOIS lookups on these domains and their IP addresses, we find that these domains all belong to the ThreatMetrix Inc. organization. As advertised on their website [160], ThreatMetrix provides their customers with protection against fraud using network-based heuristics. This purpose aligns with our observation that all but one (i.e., `commoncause.org`) of the sites deploying ThreatMetrix are e-commerce websites.

We observe that the localhost ports scanned by ThreatMetrix are primarily the standard ports for various remote desktop software for Windows, as shown in Table 5.9. We map ports to applications based on information from IANA's Service Name and Transport Protocol Port Number Registry [161] and SANS Internet Storm Center's TCP/UDP Port Activity Database [162]. We hypothesize that ThreatMetrix is attempting to determine if the website visitor's host machine (focusing specifically on Windows hosts) may be under remote control, potentially correlating with fraudulent activity. As the ThreatMetrix script uses the WebSocket protocol for initiating requests, which is not bound by the Same-Origin Policy and thus permits reading data from the request responses, the script may also be gathering more extensive information about the network services active on each port (e.g., server version and configuration).

Table 5.9: Services or malware operated over localhost ports found to be scanned.

Port (↓)	**Service/App**	**Use Case**
3389	Windows Remote Desktop	Fraud Detection
4444	Malware: CrackDown, Prosiak, Swift Remote	Bot Detection
4653	Malware: Cero	
5555	Malware: ServeMe	
5279	Unknown	Fraud Detection
5900-03	Remote Framebuffer (e.g., VNC)	
5931	AMMYY Remote Control	
5939	TeamViewer	
5944	Unknown (likely VNC)	
5950	Cisco Remote Expert Manager	
6039-40	X Window System	
63333	Tripp Lite PowerAlert UPS	
7054-55	QuickTime Streaming Server	Bot Detection
7070	AnyDesk Remote Desktop	Fraud Detection
9515	Malware: W32.Loxbot.A	Bot Detection
17556	Microsoft Edge WebDriver	

We briefly note that as our measurement method only examines website landing pages, ThreatMetrix may be more broadly deployed on the internal pages of other websites. Indeed, a recent blog post [163] identified several websites using ThreatMetrix specifically on login pages. In comparison to our study, which conducts a large-scale comprehensive and systematic investigation of website localhost and LAN activity, the blog post's investigation was an ad-hoc and manual one that considered a few websites suspected to be using ThreatMetrix. The set of landing pages we identified as using ThreatMetrix is a superset of the ones found in the blog post.

In our 2021 top 100K measurements, we continue to observe this class of behavior for popular websites. We do note changes in the websites deploying ThreatMetrix though (as indicated in Table 5.3 and Table 5.5), as some websites stopped (e.g., `citibank.com`) while some other previously crawled sites began exhibiting its fraud detection localhost traffic (e.g., `cibc.com`). Interestingly, we also recorded a set of phishing sites making localhost requests matching with ThreatMetrix's behavior (as seen in Table 5.10). While investigating these phishing pages, we identify that they impersonate legitimate websites using ThreatMetrix. We believe that the attackers have cloned the Web interface code of the legitimate impersonated websites (e.g., `customer-ebay.com` cloning `ebay.com`), inadvertently also copying over the ThreatMetrix JavaScript library (resulting in the phishing page generating localhost requests). Prior work has documented similar cloning behavior for phishing websites [58].

Bot Detection

From the *Bot Detection* section of Table 5.3, we see that 10 sites in our 2020 top 100K measurements made HTTP requests to the same set of 7 ports on localhost, using the same URL path and only on Windows 10.

We identify that this activity is initiated by BIG-IP ASM Bot Defense, a bot detection service developed by F5 Inc. [164]. On each site, the localhost requests are initiated by a JS script hosted at the relative path `/TSPD`, which is the standard path used by BIG-IP ASM Bot Defense [165, 166]. Note here that this bot defense uses HTTP, as opposed to WSS as done by ThreatMetrix (from section 5.2.3). Despite the application of the Same-Origin Policy to these HTTP requests, this bot defense presumably is able to deduce whether or not a localhost port is active. The JS script is heavily obfuscated, but we hypothesize that this inference is based on a timing side-channel. A request to an active localhost port returns quickly (even if the response cannot be read), while a request to an inactive port will time-out.

Investigating further, we discover that 4 out of the 7 ports scanned are notably used

Table 5.10: Summary of localhost requests found for malicious webpages.

(a) Type of Malicious webpage: Malware.

Domains	Protocol	Ports	Paths	OS		
				W	L	M
(79 domains, omitted for brevity)	http(s)	80/443	/*/wp-content/*	✓	✓	✓
acffiorentina.ru	http	8080	/socket.io/socket.io.js	✓	✓	✓
elilaifs.cn		28317, 36759	/get_thunder_version	✓	✓	✓
boatattorney.com	https	35729	/livereload.js	✓		✓
jdih.purworejokab.go.id	http	80	/website-bphn-bk/*	✓	✓	✓
metolegal.com			/metolegal/wp-includes/js/*	✓	✓	✓
ppdb.smp1sbw.sch.id			/ppdbv3/ro-error/*		✓	
scopesports.net			/scope/xpertspanel/*		✓	
tonyhealy.co.za			/	✓	✓	✓
oceanos.com.co			/wp-oceanos/*	✓	✓	✓

(b) Type of Malicious webpage: Abuse.

Domains	Protocol	Ports	Paths	OS		
				W	L	M
autorizador5.com.br, classyfashionbd.com, coralive.org, saudiwallcovering.com	http(s)	80/443	/*/wp-content/*	✓	✓	✓

(c) Type of Malicious webpage: Phishing.

Domains	Protocol	Ports	Paths	OS		
				W	L	M
ebaybuy.com.buying-item-guest.com, 100-25-26-254.cprapid.com, advancedlearningdynamics.com, smarturl.it, customer-ebay.com, {www.}citibank.gulajawajahe.my.id, o2-billing.org, samarasecrets.com, sic-week.000webhostapp.com, signin01.kauf-eday.de, hotelmontiazzurri.com, mahdistock.com, adesignsovast.com	wss	3389, 5279, 5900-03, 5931, 5939, 5944, 5950, 6039-40, 63333, 7070	/	✓		
ag4.gartenbau-olching.de, grp02.id.rakutan-co-jpr.buzz	http	80	/	✓	✓	
ag4.gartenbau-olching.de						✓
rakuten.* (8 domains), www.ip.rakuten.1ex.info, rakuteni.co.jp.ai12.info, www.ip.rakuten.rbimomro.icu					✓	
elmagra.net			/dashboard-v1/*	✓	✓	
etoro-invest.org			/StudentForum//*	✓	✓	✓
amazon.co.jp.* (12 domains)			/robots.txt		✓	
survivalhabits.com		44056	/NonExistentImage33090.gif	✓	✓	
evolution-postepay.com	https	5140	/NonExistentImage19258.gif	✓	✓	
postepaynuovo.com		62389	/NonExistentImage55353.gif	✓	✓	✓
sbloccareposte.com	http	44938	/NonExistentImage37362.gif	✓		
verificapostepay.com	https	49622	/NonExistentImage20705.gif	✓		✓
aladdinstar.com		8443	/images/*.png	✓	✓	✓

by well-known malware (as shown in Table 5.9). The other ports are for Microsoft Edge WebDriver, used for browser automation, and the QuickTime Streaming Server, which historically has been heavily exploited [167]. This set of ports suggests that BIG-IP ASM Bot Defense potentially searches for (i) indicators of host compromise, and (ii) the presence of browser automation.

We find that BIG-IP ASM documentation indicates that this product enables user fingerprinting [168]. The F5 Inc. website also claims that they provide their customers, including government agencies, with protection against botnets using network-based heuristics [169]. Aligned with this description, the sites that we observed deploying this bot detection script belonged to government-related entities (e.g., central banks and websites hosting government data).

Interestingly, we do not observe sites making bot detection requests during our 2021 top 100K crawl as well as the crawl of malicious webpages. Upon inspecting the websites that originally generated such requests in the 2020 top 100K crawl, we do confirm that these websites no longer host the BIG-IP ASM Bot Defense JS script (although we are uncertain of the impetus for this change).

Native Applications

In our 2020 top 100K crawl, we identified 12 websites that were communicating with native applications associated with the website itself, as listed under *Native Applications* in Table 5.3. Except for one site, all of these sites behaved uniformly across OSes. Below we detail the localhost communications that websites initiate with native applications.

- *E-commerce*: `samsungcard.co.kr` redirects to `samsungcard.com`, an e-commerce website. We found that `samsungcard.com` looks for the presence of INCA Internet's nProtect Online Security [170], an endpoint security solution which claims to protect user transactions with financial services against fraud, and communicates with a digital signature software called AnySign [171]. Our telemetry observed `samsungcard.com` requesting localhost ports 14440-9 over HTTP via a JS file that specified "nProtect Online Security" in its header, and requests to other ports over WebSockets, via a different JS file with "AnySign for PC" in its header.

We note that while the end goal of this localhost communication is presumably similar to the fraud detection goals of ThreatMetrix (in section 5.2.3), the localhost communication is used to interact with a native application which performs the fraud detection, rather than the communication being done as part of the fraud detection method (as with ThreatMetrix).

- *Gaming*: `faceit.com`, `gamehouse.com`, `games.lol`, `zylom.com`, and `trustdice.win` are gaming websites that conduct localhost communication to detect their native gaming clients. `faceit.com` and `games.lol` use the WebSockets protocol, whereas the others communicate using HTTP.
- *Communication*: `cponline.pw` and `runeline.com` redirect to channel invitation pages for Discord, an IM/VoIP application [172]. These webpages initiate localhost communication with Discord's native client. Similarly, `screenleap.com` makes localhost requests to its own screen sharing product. Given the recent extensive use of video conferencing software clients such as Zoom [173] and BlueJeans [174], we expect that we are missing other communication websites performing localhost requests, as their activities are not exhibited on landing pages.
- *Media*: `acestream.me` initiates localhost communication with the Ace Stream client, a live-streaming media platform.

Given the close relationship between the website and the native app (e.g., a gaming website contacting its associated gaming client), we assume benign intent here, although we note that the website may be detecting if a visitor has already installed the associated application, and adjusting its behavior (e.g., advertising) accordingly. Additionally, this class of localhost communication highlights a legitimate use case that should be preserved if attempting to defend against malicious forms of Web-based localhost traffic.

In addition to these 12 websites, we found another 14 websites in our 2021 top 100K measurements similarly attempting to communicate with their native apps (as listed in Table 5.5). Only one website was found no longer making these localhost requests in our second crawl. For malicious webpages, we identified one similar case with `elilaifs.cn`. Inspecting the webpage indicates that it imports the JS library of Thunder [175], a download manager, which attempts to communicate with its native application.

Developer Errors

Beyond the intentional locally-bound traffic that websites make, we find that a large class of local resource requests likely arise due to developer error. For our 2020 top 100K measurements, we believe this situation occurred on 44 (out of 107) websites requesting localhost resources and 6 (out of 9) sites requesting LAN resources, indicating that it is relatively widespread among our observed sites. Table 5.11 and Table 5.4 characterize these websites in detail.

In these cases, we observe localhost and LAN requests for various files (e.g., images, videos, JavaScript libraries) that were likely hosted at a local server during Web development. For example, a number of websites request a URL path that contains `/wp-content /uploads/`, suggesting that the requested resource is a file uploaded to the local WordPress installation. Upon manual investigation, we do not find any evidence suggesting intentional localhost or LAN communication across these websites, and we believe that these resource requests are unintentional and remnants of when the websites were being developed and tested. Here, we also discuss these cases in more detail.

- *Accessing files hosted on a local file server*: As seen in the first set of rows in Table 5.11 for localhost requests and Table 5.4 for LAN requests, we find that 57% (25/44) and 67% (6/9) of websites exhibiting developer errors request files (e.g., images) from a server on localhost or LAN, respectively, all using HTTP(S).

  Interestingly, there are cases where this behavior differs across OSes.

We hypothesize that these discrepancies arise in part because websites may present themselves differently for different OSes (for example, based on the user-agent string), and certain website elements that generate the unintentional resource fetch are not included when on certain OSes.

- *Penetration testing*: `rkn.gov.ru` fetches the JavaScript file `xook.js`, a file used as part of OWASP's Xenotix Exploit Framework, which Web developers use for vulnerability detection and penetration testing [176].
- *LiveReload.js*: Several websites fetch the file `livereload.js`, which is for the LiveReload.js tool that helps Web developers dynamically monitor changes during Web development [177].
- *Redirect*: `romadecade.org` and `fincaraiz.com.co` redirect to `http://127.0.0.1/`. We are uncertain of the reason.
- *SocksJS-Node*: We observe five websites initiating localhost HTTP(S) requests where the URL path is `/sockjs-node/info`. This path is associated with SockJS-node, which is the Node.js server-side component of the SockJS JavaScript library [178]. This library provides a WebSocket-like communication interface between a server and a browser even for legacy browsers without WebSocket support. Thus, we believe that the website developers were using SockJS during development and testing, which involved deploying a localhost SockJS-node server, and this artifact remained in the published website. Interestingly, this activity was observed only when on a Mac.
- *Other local services*: We observe 7 websites initiating localhost HTTP(S) requests likely for interacting with local network services during website development and testing. For example, `zakupki.gov.ru` updates user activity state at a localhost service.

Similar cases were found in both the 2021 top 100K crawl (as shown in Table 5.5 and Table 5.6) and our crawl of malicious webpages (Table 5.10 and Table 5.8). For mali-

Table 5.11: Websites with developer errors that resulted in localhost requests (2020 crawl).

(a) To access a local file server.

Rank (↓)	Domain	Protocol	Port	Paths	OS		
					W	L	M
22730	smartcatdesign.net	http	8888	/wp-content/uploads/2018/06/*.jpg	✓	✓	✓
36786	uinsby.ac.id	http	80	/eduma/demo-1/wp-content/uploads/sites/2/2017/11/*.jpg	✓	✓	✓
38865	upbasiceduboard.gov.in (-)	http	1987	/TeacherRecruitment2018/images/*.jpg	✓	✓	
41468	walisongo.ac.id	https	80	/wordpress/wp-content/uploads/2015/07/*.jpg	✓	✓	
41596	classera.com	http	8080	/wp-content/uploads/2020/04/*.png		✓	✓
45177	weavesilk.com*	http	80	/Silk%20Static/*{.mp4, .ogg}	✓	✓	✓
50390	upsen.net (-)	http	80	/6/10/*.js	✓	✓	✓
51910	dsb.cn*	http	80	.*jpg	✓		
56450	sin-tech.cn (-)	http	9999	/admin/kindeditor/attached/image/20191017/*.jpg	✓	✓	✓
56730	nwolb.com*	https	36762	/*.gif	✓	✓	✓
57467	cryptopia.co.nz*	http	49972	/*.ico	✓	✓	✓
63636	weijuju.com* (-)	http	9092	/image/page/index/*.png	✓	✓	✓
63770	tdk.gov.tr*	http	80	/magazon/magazon-wp/wp-content/uploads/2013/02/*.ico	✓	✓	✓
65915	shqilon.com (-)	http	80	/stop/*.html	✓	✓	✓
66891	aau.edu.et*	http	80	/graduation/wp-content/uploads/2020/06/*.png	✓		
67851	sirrus.com.br	http	80	/sitesirrus/wp-content/uploads/2017/07/*.png	✓	✓	✓
69708	unionbankph.com*	http	8888	/socket.io/*.js	✓	✓	✓
77636	qubscribe.com (-)	https	443	/wp-content/uploads/2019/03/*.png		✓	✓
77761	persian-magento.ir (-)	http	80	/graffito/images/sampledata/*.png	✓	✓	✓
86045	serymark.com (-)	http	80	/sm/wp-content/uploads/2017/06/*.png	✓	✓	✓
88997	ghana.com (-)	https	8080	/gdc/wp-content/themes/consultix/images/*.png	✓	✓	✓
92768	gomedici.com	http	3000	/assets/*.png		✓	✓
93798	xaipe.edu.cn (-)	http	80	/*.html		✓	✓
94771	health.com.kh (-)	http	8899	/newhealth/wp-content/uploads/2018/01/*.png	✓	✓	✓
96981	urkund.com (-)	http	4337	/wp-content/uploads/2019/07/*.png		✓	✓

(b) Due to a pen testing tool.

Rank (↓)	Domain	Protocol	Port	Paths	OS		
					W	L	M
17827	rkn.gov.ru (-)	http	5005	/xook.js	✓	✓	✓

(c) Because of `LiveReload.js`.

Rank (↓)	Domain	Protocol	Port	Paths	OS		
					W	L	M
19244	cruzeirodosulvirtual.com.br*	http	460	/livereload.js	✓	✓	✓
53124	melissaanddoug.com*	https	35729	/livereload.js	✓	✓	✓
53216	airfind.com*	https	35729	/livereload.js	✓	✓	✓
58629	hollins.edu	https	35729	/livereload.js	✓	✓	✓
59978	amitriptylineelavilgha.com (-)	http	35729	/livereload.js	✓	✓	✓

cious pages, we believe that either attackers made such developer errors on their own attack sites, or more likely, the attackers compromised benign websites exhibiting these errors. In

Table 5.11: Websites with developer errors that resulted in localhost requests (2020 crawl) contd.

(d) Because of a redirect.

Rank (↓)	Domain	Protocol	Port	Paths	OS		
					W	L	M
51142	romadecade.org (-)	http	80	/	✓	✓	✓
63644	fincaraiz.com.co*	http	80	/	✓		

(e) Because of SocksJS-Node.

Rank (↓)	Domain	Protocol	Port	Paths	OS		
					W	L	M
49144	lyfdose.com	http	9000	/sockjs-node/info?t=*			✓
49990	klik-mag.com	https	9000	/sockjs-node/info?t=*			✓
51101	acedirectory.org	https	9000	/sockjs-node/info?t=*			✓
57249	veteranstodayarchives.com	https	9000	/sockjs-node/info?t=*			✓
66971	smartsearch.me	https	9000	/sockjs-node/info?t=*			✓

(f) Other local services.

Rank (↓)	Domain	Protocol	Port	Paths	OS		
					W	L	M
7700	zakupki.gov.ru (-)	https	1931	/record/state	✓	✓	✓
24740	gamezone.com	http	8000	/setuid	✓	✓	✓
26400	filemail.com	http	56666	/	✓	✓	✓
31518	interbank.pe	http	9080	/avisos-portal	✓	✓	✓
58708	fsist.com.br (-)	http	28337	/getCertificados	✓	✓	✓
62852	spaceappschallenge.org	http	8000	/graphql		✓	✓
90791	fromhomefitness.com (-)	https	8000	/app/getLicenseKey		✓	

total, we attribute more than 90% of the localhost activity on malicious webpages to this behavior class.

While presumably benign, these local network requests indicate that some functionality on the site is incorrect, as the webpage attempts to load a resource that does not actually exist. We also note that the requests can reveal aspects of the site development process, such as a particular vulnerability testing JavaScript framework used by `rkn.gov.ru`. Ultimately, these requests reflect a class of Web developer errors that can impact the functionality of websites but should be easy to identify and remedy, such as through searching for localhost or private IP address requests in the website code base or examining the requests made when loading the website.

Unknown Cases

We did not identify plausible explanations for the 5 websites listed under *Unknown* in Table 5.3. There were 3 websites (`farsroid.com`, `tra...fgt.xyz` and `1-movies.ir`) generating LAN traffic that we could not confidently classify. These sites return a `403 Forbidden` page with an iframe element sourced at `http://10.10.34.35:80`. Recently, Raman et al. [179] observed such iframe sources occurring during website censorship in Iran. We note that `farsroid.com` and `1-movies.ir` resolve to Iranian IP addresses, but `tra...fgt.xyz` did not. We suspect that the observe behavior is related to censorship, although we remain uncertain of the exact situation, and are unclear why this particular LAN address is used.

5.3 Discussion

Here, we synthesize our results and their implications on Web security and privacy.

5.3.1 Website Anti-Abuse

From our measurements, we discovered that a number of sites leverage Web-based localhost scanning to defend themselves against abuse. These sites are primarily e-commerce and government-related ones and include highly-ranked sites for organizations such as eBay and Fidelity Investments. However, the websites do not conduct local network scanning themselves, but rather (as uncovered in section 5.2.3 and section 5.2.3) they rely on third-party services (i.e., ThreatMetrix and BIG-IP ASM). A 2018 ThreatMetrix brochure [180] claimed to protect \$170 billion worth of e-commerce transactions in a year, preventing \$19.5 billion in fraud. They advertised an extensive database of 1.4 billion unique online identities collected from their clients operating in the e-commerce space. Given the prevalence of online abuse across various platforms beyond just e-commerce, such as on social media and communication sites, we may observe an expansion of Web-based localhost scanning for anti-abuse on other sites.

Our analysis in section 5.2.3 and section 5.2.3 identified that these anti-abuse measures focus on detecting indicators of host compromise or remote control. We are unable to assess the reliability of these signals for anti-abuse. However, we hypothesize that attackers could evade this detection with relative ease by modifying the ports they operate on. For example, attackers could configure a remote control server on a bot to run on a non-standard port, and malware could dynamically adjust the ports it listens on. This evolution would likely result in an arms race that seems tilted in the attacker's favor due to information imbalance. As any Web-based localhost scanning activity would be visible to website visitors as well as attackers, attackers can directly observe and counter the anti-abuse strategies deployed. In contrast, attackers lack direct visibility into the operations of server-side anti-abuse mechanisms. Thus, we question whether Web-based localhost scanning for fraud and bot detection will remain viable in the long term.

5.3.2 Web Tracking

We did not uncover explicit evidence of user tracking through Web-based localhost or LAN scanning, a positive finding regarding Web privacy. However, we did observe clear host profiling as part of fraud and bot detection, which can naturally be extended for user fingerprinting and tracking (as demonstrated by existing proof-of-concept techniques [20, 26]). Similar to other Web fingerprinting methods that have been used in practice, such as detecting the browser extensions installed [181], Web trackers could distinguish users based on their localhost network services and LAN devices. Prior work [39] revealed how websites often deploy surreptitious methods for Web tracking. With ongoing efforts to curtail the amount of data that websites can obtain on users, including Google's Privacy Sandbox project [182], Web trackers may be forced to resort to novel tracking mechanisms such as those based on local network scanning.

Further incentivizing this new tracking strategy is the increasing amount of information that can be derived from Web-based localhost and LAN scanning. On user machines, numerous modern native applications (such as those discussed in section 5.2.3) host localhost network services, and their diversity and popularity have grown (particularly gaming and video conferencing clients during the COVID-19 pandemic [183, 184]). Similarly, user home networks contain growing numbers of devices [149], particularly IoT devices with exposed network interfaces that can be readily detected [26]. Thus, Web-based local scanning should provide significant amounts of information about users, which also serve as high entropy features for fingerprinting and tracking them. We discuss defending against such website behavior next.

5.3.3 Defending Against Malicious Web-Based Local Traffic

We did not discover any websites generating malicious attack traffic to localhost or LAN network services, as hypothesized by prior work [20, 26, 22]. Nonetheless, browsers should ideally protect users from both malicious attacks and Web tracking based on locally-bound network traffic.

The browser security community has already begun working on mitigating the unintentional exposure of local network resources to browsers. In March 2021, the Web Platform Incubator Community Group (WICG) [185] proposed modifications to the Fetch API [186], where resources loaded from a public IP space are allowed to fetch resources from private/local IP spaces only if: 1) the public resource was loaded over a secure channel (i.e., `https` or `wss`), and 2) a CORS preflight request to the local network origin is successful [187]. These CORS preflight requests should include the *Access-Control-Request-Private-Network=true* header, and browsers should only permit access to the local network resource if the same header is in the responses.

We believe this proposal is a promising step in the right direction, as such an opt-in model for access to a user's local network resources would support legitimate use cases (e.g., native application communication), while barring other unintentional exposure. However, the real-world security and privacy impact of such a proposal depends on its implementation and adoption, by both browser vendors and local network services (e.g., native applications). As of this writing, Google Chrome has initiated a deprecation trial to adopt the proposal [188]. In the interim, before broad adoption by local network services and native applications, browsers could alert the user before initiating locally-bound requests, similar to existing browser permission prompts for other access requests. This human-in-the-loop approach could help prevent unauthorized local communication and so far, has already been adopted by the Brave browser [189].

We hope that our findings from this study help inform and drive the further development of potential defenses against malicious Web-based local traffic.

5.3.4 Developer Errors

A large number of websites were found making local network requests likely due to developer errors. As discussed in section 5.2.3, these requests reflect broken functionality on the sites as well as potential information leakage on the Web development and testing process. Therefore, we recommend that Web developers check for such local network behavior through either analyzing the website code base or examining network traffic generated by the website during testing. We note that while assessing a website during testing, different user-agents should be evaluated, as we observed different behavior across OSes even for developer errors (likely due to the error occurring in OS-specific portions of the website code). We expect that these errors should be easy to remedy, as the request destinations can be updated to the correct public servers, or the traffic-generating code can be removed altogether.

5.4 Conclusion

In this study, we conducted a large-scale empirical investigation of if, how, and why modern websites interact with network services on a browser's localhost and LAN. Crawling both the landing pages of the Tranco top 100K domains [150] as well as 145K malware, phishing, and abuse websites, using Google Chrome on three popular desktop OSes (Windows, Linux, and Mac), we uncovered hundreds of websites generating requests to internal network destinations, including highly-ranked sites within the top 10K. We found that local network activity was not uniform across OSes, with the most activity exhibited on Windows. We also observed extensive use of WebSockets for locally-destined connections, which notably is not bound by the Same-Origin Policy.

By investigating the detected top websites in detail, we identified four common causes for the local traffic: fraud detection, bot detection, native application communication, and developer errors. Notably, the fraud and bot detection techniques conducted host profiling. While we did not uncover explicit user tracking, the adaptation of these host profiling techniques for such purposes may be on the horizon. Defending against such malicious Web-based local traffic will require preserving the legitimate use case of native application communication. Meanwhile, the class of developer errors that we exposed affects website functionality but should be easy for Web developers to discover and remedy.

For malicious websites, we did not detect internal network attacks. Rather, 90% of malicious webpages with local network activity exhibited local traffic matching the developer errors observed on top websites, suggesting that either attackers make the same mistakes when implementing their attack websites, or perhaps more likely, the attackers compromised benign websites exhibiting these errors. We also found an interesting case of phishing webpages cloning the Web interfaces of legitimate websites that generate local traffic for fraud detection, thus inadvertently inheriting the same local network behavior.

Ultimately, our work establishes empirical grounding on the intentional and unintentional localhost and LAN network activities of real-world websites. Future work can expand upon our initial investigation. As discussed in section 5.1.3, our examination of websites only considered the landing page, but we expect that internal pages, such as those for account creation or login, may also generate localhost and LAN network traffic. Additionally, our measurement method (as detailed in section 5.1) focused on a desktop environment. Subsequent studies can build upon our work by evaluating websites on mobile platforms and inspecting website internal pages. Our study also uncovered host profiling purportedly for fraud and bot detection. These techniques can be studied in more depth to characterize their effectiveness in practice and assess the potential risk that they are adapted for user tracking purposes. Finally, browser mechanisms can be developed to detect and prohibit malicious local network requests from websites while preserving legitimate use cases, thus better protecting user security and privacy on the Web.

CHAPTER 6

INVESTIGATING THIRD-PARTY WEB CONTENT IN ANDROID APPS

Mobile devices are ubiquitous in today's digital world. According to a June 2023 estimate, more than 65% of Web traffic originates from these devices [1]. This includes not only users browsing the Web via mobile browsers, but also various mobile apps that display Web content. For example, hybrid mobile apps that render a Web page in full-screen mode within an app, in-app advertisements that show free content alongside ads, and Web links that users follow from within an app. Traditionally, such functionality has been achieved using the `android.webkit.WebView` class, or simply 'WebView'. WebView is a component of Android's View class that allows developers to embed Web pages into their app's interface [55]. Android suggests using it for trusted first-party content, as it gives the app more control over the webpage, such as executing Javascript (JS) code, intercepting requests, and enabling interaction between the webpage and the app's native code [190]. This enables hybrid app experiences that can be useful in many contexts [191, 192]. However, nothing prevents it from loading untrusted third-party content, which exposes a large attack surface that has been extensively studied. Custom Tabs (CT), on the other hand, is a feature designed for displaying third-party Web content securely and efficiently. It has been available in Android devices since 2015 [193]. With CTs, developers can provide a customized browser experience within their app using Android browsers such as Chrome. It loads the webpage with access to the user's default browser cookies but does not interfere with the page itself [190, 194]. CT offers the latest browser experience, with a secure User Interface (e.g., SSL lock icon), without requiring extra development effort. As shown in Figure 6.1, it has also been found to load pages twice as fast as WebView in tests [193]. Considering the advantages of CT and drawbacks of WebViews, CT should be preferred for displaying third-party Web content in Android securely and privately. However, there is limited empirical research on whether this is reflected in real-world practices.

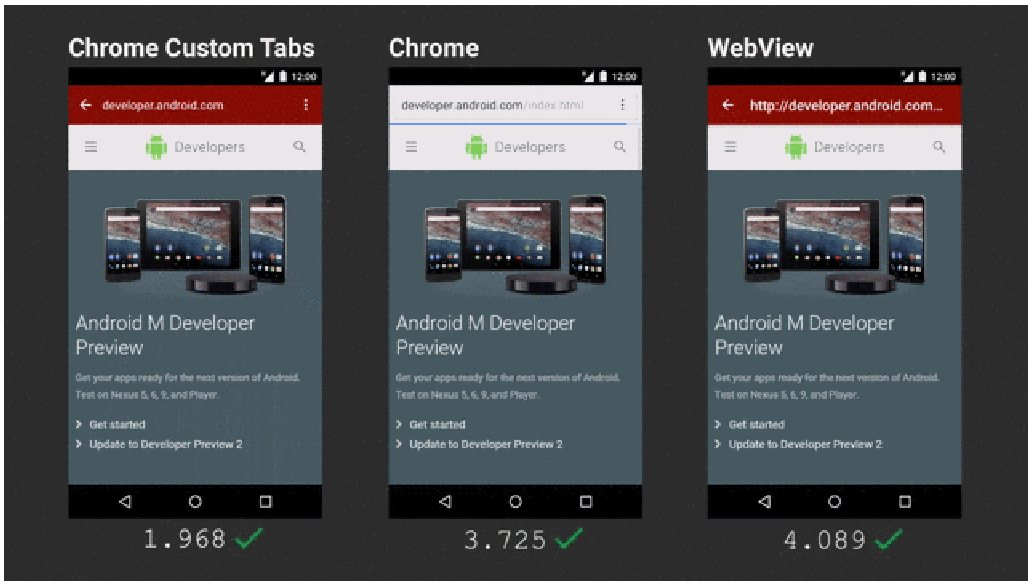

Chrome Custom Tabs
Chrome
WebView
developer.android.com
developer.android.com/index.html
http://developer.android.com...
Developers
Android M Developer Preview
Get your apps ready for the next version of Android. Test on Nexus 5, 6, 9, and Player.
Get started
Update to Developer Preview 2
1.968
3.725
4.089

Figure 6.1: Page load test in Chrome CT, Chrome, and a WebView within an app [193].

In this work, we conduct measurements on popular Android apps to provide novel insights into the state of Web security in mobile apps. We analyze ∼146.5K apps, each with over 100K users, and find ∼55.7% apps using WebViews, ∼20% using CTs, and ∼15% using both. To infer use-cases, we detect 125 popular SDKs using WebViews, 45 SDKs using CTs, and 34 SDKs using both. Our results indicate that WebViews were mainly used for advertising purposes, with 46 SDKs supporting this use-case in ∼39K apps. In contrast, CTs were predominantly used for social media integration, with 6 SDKs enabling this use-case in nearly 23.8K apps. We combine our measurements with the extensive prior work on the insecurity of WebViews to propose evidence-based recommendations for both app and SDK developers. Overall, we find that even though many popular SDKs have migrated to CTs, there are still many widely-used SDKs that have not yet migrated.

Moreover, we conduct a semi-manual analysis of the top 1K apps with the highest number of downloads, identifying cases where apps override Android's default behavior of opening third-party URLs in the browser. We found 11 apps from the top 1K that masked URLs as buttons with hyperlink-like text, which upon clicking launched an In-App Browser

(IAB). We define an 'In-App Browser' as any non-browser Activity that can navigate to an arbitrary URL. 10 out of the 11 apps implemented a WebView-based IAB, which we examined further with comprehensive measurements. Through a rigorous manual analysis, we uncover various behaviors such as facilitating payments, ad injection and crowdsourcing network measurement from end-user devices.

Ultimately, we take the first step towards improving transparency regarding how mobile apps display third-party Web content, and the real-world security and privacy implications that follow.

6.1 Method

In this section, we describe our method for assessing the usage of WebViews and Custom Tabs (CTs) within Android apps. First, we outline our systematic approach to a large-scale analysis using static techniques, aimed at discerning patterns and deriving insights at an ecosystem level. Subsequently, we describe our usage of dynamic analysis methods, applied to a smaller, select subset of popular apps, which enables us to delve into the specifics of their behaviour more extensively.

6.1.1 Static Analysis

Here, we delve into the details of our dataset and the methodology we use to apply static analysis techniques to characterize the usage of WebViews and CTs in apps. Figure 6.2 helps illustrates each of the steps in our static analysis pipeline.

Dataset

We use Androzoo, a widely used repository of Android apps, which periodically fetches Android Package Kit (APK) files from multiple app stores including the Google Play Store [195]. Using the January 13, 2023 snapshot of Androzoo, we fetch the list of ∼6.5M apps that have appeared in the Google Play Store. Since our aim was to gain insights about popular apps that are currently being used by a large population of users, we collected app metadata from the Google Play Store, which included information such as the number of times an app was installed and the category of the app, among other information [196]. Next, after filtering for apps that had at least 100K downloads, and were being actively maintained, i.e., had been updated at least once after January 1, 2021, we narrowed down to a set of 146.8K apps. For each of these 146.8K apps, we downloaded their most recent APK from AndroZoo for further analysis.

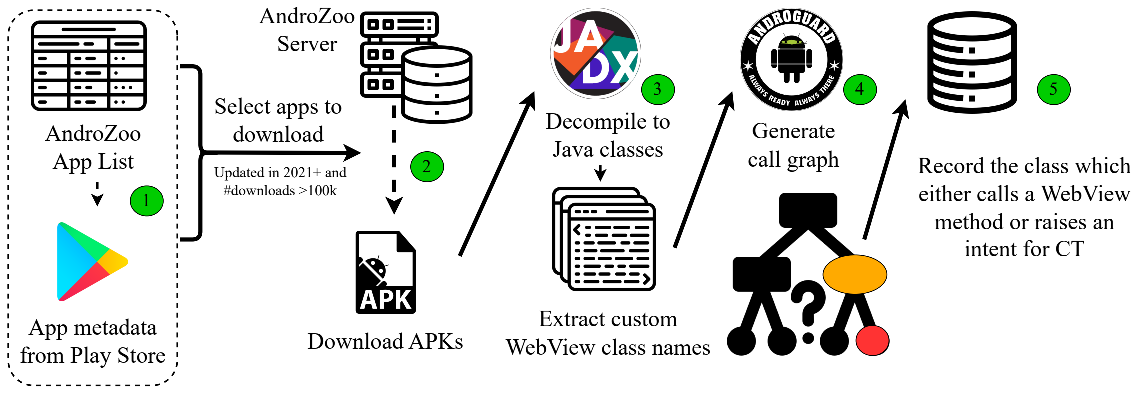

AndroZoo App List
1
App metadata from Play Store
Select apps to download
Updated in 2021+ and #downloads >100k
AndroZoo Server
2
APK
Download APKs
3
Decompile to Java classes
Extract custom WebView class names
4
Generate call graph
5
Record the class which either calls a WebView method or raises an intent for CT

Figure 6.2: Our static analysis pipeline.

Preprocessing APKs

For apps to display Web content using WebViews, they need to instantiate objects of the Android class `android.webkit.WebView` as part of their Activity layout [55]. The `WebView` class offers extensibility in terms of customizing the UI and other advanced configuration options, so it is only natural that developers extend the `WebView` class to build their own custom WebView experiences. To detect such custom WebView class implementations, our approach is two-fold. Firstly, we decompile the downloaded APK files into Java source code using JADX, a state-of-the-art tool with the lowest failure rate as compared to other tools [197, 198]. Secondly, for each class file in the source code that imports `android.webkit.WebView`, we use an open-source parser to parse the Java source code [199] and extract the names of classes which extend the WebView class. With this information, we can accurately pinpoint the entities in the source code which instantiate and interact with WebViews, as we will describe in detail shortly.

The behavior of a program, specifically its control flow relating to the sequence and dependencies of function calls, can be effectively comprehended through static analysis techniques like call graphs. As a crucial part of our preparatory process for analysis, we generated call graphs for all 146.8K APKs that we had procured. This was achieved using Androguard [200], an open-source reverse engineering tool specifically designed for the analysis of Android apps, and a component of well-established security and privacy toolkits like Cuckoo Sandbox and Exodus respectively. Our preference for Androguard, over other tools such as FlowDroid [201], was informed by several factors. FlowDroid, while widely cited in literature for smaller datasets, is primarily designed for taint-tracking, and leads to comparatively higher failure rates and increased resource overheads [202]. In contrast, Androguard enabled us to generate call graphs for an impressive 99.8% of the apps in our dataset. The 242 APKs that were not analyzed were discovered to be corrupt. A summary of our dataset is available in Table 6.1.

Table 6.1: Statistics for apps that we statically analyze.

Dataset	**No. of apps**
Play Store apps in Androzoo	6,507,222
Apps found on Play Store	2,454,488
Apps with 100k+ downloads	198,324
Apps with 100k+ downloads and updated after 2021	146,800
Apps successfully analyzed	**146,558**

Measuring WebView and CT Usage

An Android app is comprised of various components, specifically Activity, Service, Content Provider, and Broadcast Receiver. Any of these elements can serve as the initial point of interaction or "entry point" which leads to a WebView or CT [203]. Even within a single activity, multiple entry points exist, facilitated by lifecycle methods like `onCreate()` or callbacks tied to system or GUI events, each of which could follow a different control flow. Unlike other apps built with Java, an Android app lacks a 'main' function, which could be considered the primary entry point. Therefore, in order to exhaustively identify the usage of WebViews and CTs in an app, we traversed the app's entire call graph via all entry points, recording every call to WebViews and CTs. For WebView-related activity, we recorded which methods of `android.webkit.WebView` (and classes which extend it) were called, along with the names of the classes in which the respective call was made. For CT-related activity, we similarly recorded for calls made for the `CustomTabsIntent` intent class of `androidx.browser.customtabs` [204].

Next, to filter out app activities that are likely to host first-party Web content, we identified activities which can handle deep links to app content [205], and excluded them from further consideration. Specifically, we used the app manifest to check if an activity has the flag `exported` set to `true`, and contains an intent-filter of category `android.intent.category.BROWSABLE` which accepts data with a scheme of either `http` or `https`.

Identifying and Labelling SDKs

To understand the reasons an app might utilize a WebView or CT, we explored whether the Java package responsible for loading content into these components is part of a SDK with a defined function. In the case of WebViews, we searched for calls to one of the following methods: `loadUrl`, `loadData` or `loadDataWithBaseURL`. For CTs, we searched for method calls to `launchUrl`. These methods would need to be invoked to populate content. Since we had recorded which classes called these methods, we were able to extract the package names from those classes, assuming that package names adhere to the proper Java conventions [206]. In aggregate, we identified 141 packages, each of which was used by more than 100 apps each. We then used data from the Google Play SDK Index [207] and supplementary Google Search to label these packages into SDK categories if they were known to be part of certain SDKs. Excluding Google's Android SDK (`com.google.android`) – due to its multiple essential functions – we successfully categorized 126 out of 140 packages. The remaining 14 packages either had obfuscated labels (4), or they could not be associated with any known SDK (10).

We summarize statistics for the various types of SDKs found using WebViews and CTs in Table 6.2. We also illustrate examples of most popular SDKs we identified as using CTs and WebViews in Table 6.3 and Table 6.4 respectively.

Table 6.2: Statistics for use of WebViews and CTs in SDKs.

Type of SDK	**No. of SDKs**		
	Use WebViews	**Use CT**	**Use both**
Advertising	46	3	3
Payments	15	6	5
Development Tools	11	7	5
Engagement	12	0	0
Social	10	6	4
Authentication	7	10	6
Unknown	10	4	4
Hybrid Functionality	6	7	5
Utility	4	2	2
User Support	4	0	0
Total	**125**	**45**	**34**

Limitations

Our approach enables us to conduct a large-scale study of the usage of WebViews and CTs. However, it is worth noting some important limitations to our methodology.

- Static analysis methods can yield false positives for various reasons [208, 209, 210].

Table 6.3: Popular SDKs which use CTs.

Type of SDK	Total #apps	SDK Name	#apps
Social	23,807	Facebook	23,234
		NAVER	157
		Kakao	54
Authentication	7,802	Google Firebase	7,565
		NAVER	81
		AdobePass	55
Advertising	1,953	HyprMX	1,257
		Linkvertise	383
		Taboola	317
Payments	208	Juspay	77
		Ticketmaster Checkout	47
Development Tools	172	android-customtabs	53
		GoodBarber	48
		Mobiroller	27
Hybrid Functionality	87	Cube Storm	14
		Scripps News	13
Utility	71	Ticketmaster	55
		MyChart	16

For instance, certain logic within the app could decide not to initiate a segment of the app using WebViews or CTs, a scenario which could arise from user interactions.

- Our investigation is targeted and relies heavily on the identification of method calls based on their characterized behavior as detailed in Android's documentation. Consequently, our method may fall short in detecting obfuscated method calls exhibiting similar behaviors. It's worth noting, however, that obfuscation is relatively uncommon in apps available on the Google Play Store [211].
- Our exhaustive approach to measuring the usage of WebViews and CTs could inadvertently incorporate instances displaying first-party content. For a precise identification of WebView or CT instances used as IABs, static modeling of control or data flow instigated by user interactions with the app's GUI would be required. However, previous studies have found static analysis methods to be insufficient for such endeavors [210, 208].

Table 6.4: Popular SDKs which use WebViews.

Type of SDK	Total #apps	SDK Name	#apps
Advertising	39,163	AppLovin	27,397
		ironSource	16,326
		ByteDance	13,080
		InMobi	10,066
		Digital Turbine	8,654
Engagement	21,040	Open Measurement	11,333
		SafeDK	7,427
		Airship	652
		Branch	514
Development Tools	7,020	Flutter	5,568
		InAppWebView	1,868
		Corona	449
		AdvancedWebView	386
Payments	3,212	Stripe	1,171
		RazorPay	484
		PayTM	400
User Support	1,692	Zendesk	1,000
		Freshchat	438
		LicensesDialog	129
Social	1,686	VK	456
		NAVER	406
		Kakao	347
Utility	362	NAVER Maps	130
		Barcode Scanner	129
		Ticketmaster	64
Authentication	342	Gigya	120
		NAVER	90
		Amazon Identity	37
Hybrid Functionality	256	Baby Panda World	194
		SoftCraft	15
		Cube Storm	14

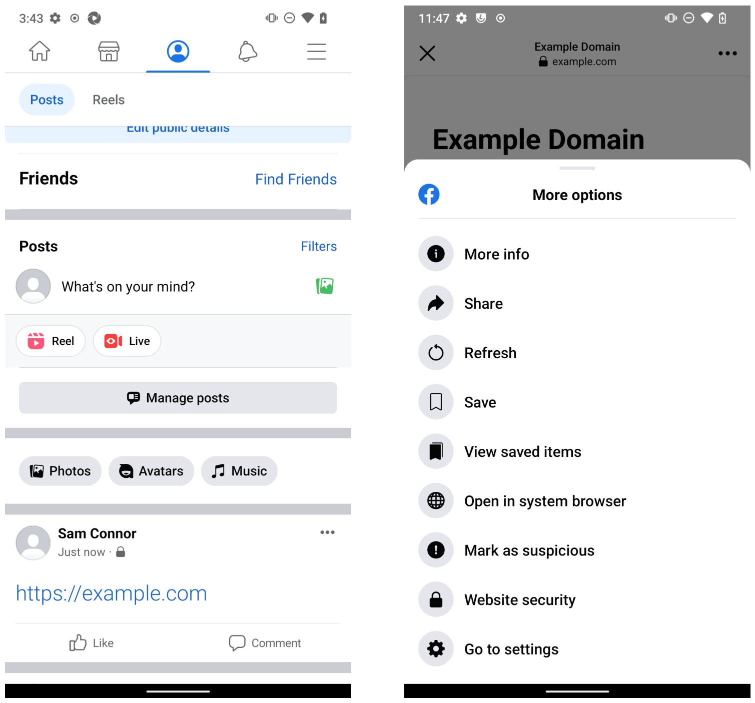
6.1.2 Semi-manual Dynamic Analysis

Building upon the insights garnered from our large-scale static analysis of WebView and CT utilization in apps, we meticulously carry out a semi-manual analysis of the top 1K apps with the goal of understanding how apps implement WebView-based IABs to handle third-party Web links.

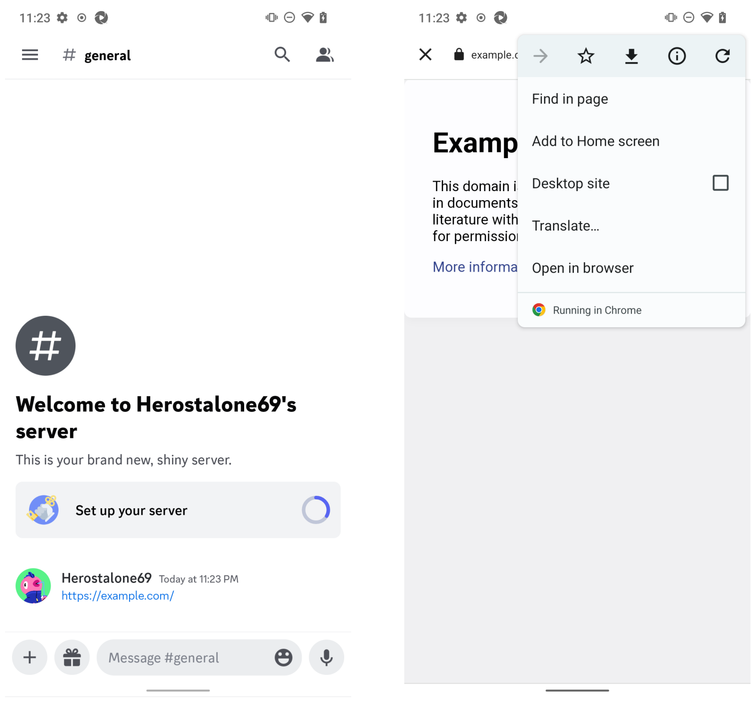
Dataset

From the ~146.8K apps we had selected in subsubsection 6.1.1, we further select 1K apps with the highest number of downloads. We programmatically download each app from the Google Play Store and install it on a Pixel device. Dummy accounts are manually created where necessary for the sake of accessing content within the app. Next, we manually investigate areas of the app which potentially contain user-generated Web links. Intuitively, we focus on sections that display user-generated content, for example, social media feeds with user posts and comments, direct message or chat interfaces, and profile biographies where users may link affiliated Web pages. This methodology echoes that of Zhang et al.'s approach to studying the security design of IABs [57]. Our analysis uncovers 38 apps, solely from the social and communication categories, where users can post links. We find that 905 of the 1K apps we analyze do not contain user-generated content. These are predominantly utility apps such as media players, entertainment apps, stock apps, and gaming apps. Notably, nine apps are browsers themselves, and the remaining 48 apps are unclassifiable due to various factors, as presented in Table 6.5.

For the 38 apps where users can post links, we manually submit a link to `https://example.com` and follow it. We observe that approximately 71% (27 of 38) of the apps open the links in a browser, which is the intended and default behavior for Web links in apps, thus these apps are not studied further. Interestingly, we discover around 26% (10 of 38) of the apps open this third-party link in a WebView-based IAB. For instance, as seen in Figure 6.3a, if a Facebook user clicks on a URL that they see on their news feed, it opens inside a WebView-based IAB. Discord is the only app that opens the link in a CT, as seen in Figure 6.3b. We compile these statistics in Table 6.5.



(a) WebView-based IAB in Facebook.



(b) CT-based IAB in Discord.

Figure 6.3: WebView and CT based IAB implementations.

Table 6.5: Our manual classification of the top 1000 Play Store apps.

Classification of apps	#apps
Users can post links.	38
Link opens in browser.	27
Link opens in a WebView.	10
Link opens in CCT.	1
Users can not post links.	905
Browser Apps.	9
Could not classify app.	48
Required a phone number.	24
App incompatibility error.	22
Required paid account.	2

Measurement Setup

We previously established that using CTs is the most efficient and secure approach to implementing IABs. To investigate the motives of apps using WebViews to implement IABs, we navigate each of the 10 WebView-based IABs to a controlled Web page hosted on our server, and record the following measurements:

- *App-WebView Interactions*: Using Frida [212], a widely-used dynamic instrumentation tool [213, 214, 215], we dynamically override all methods of `android.webkit.WebView` at run-time, in order to record the WebView APIs used by the app, along with the arguments passed. Specifically, when an app has interactions with the WebView beyond mere loading of the URL, this information serves to provide detailed insight for further analysis.
- *JS Code Injection*: An app can inject JS code into a WebView primarily via methods `evaluateJavascript` and `loadUrl`. `evaluateJavascript` allows executing JS code in the WebView and retrieve the result asynchronously, while `loadUrl` can be used to execute JS code once the page has finished loading, by prepending `javascript:` (as the scheme) to the code. In cases where such injection is detected,

we further record:

- *Web API usage*: Our controlled Web page is composed of common HTML elements [216]. The only JS script used on the page is meant to override all methods of all Web APIs as listed in MDN Web Docs [217] and submit the intercepted requests with parameters, back to our server [218]. Thus, when the injected JS code uses Web APIs (e.g., Document), our server records it.
- *Network Logs*: We use a rooted Pixel 3 mobile device running LineageOS 19 [219] for our measurements. LineageOS is a custom `userdebug` image of Android, which enables us to record the network logs directly from Chrome's network stack [159]. As opposed to capturing device-wide network traffic via `mitmproxy`, we are able to collect detailed network logs for each Web page visit via a specific WebView instance. Previous work has found such logs reliable for website-specific network measurements [220].

With these comprehensive set of measurements, we can gain a nuanced understanding of the user's privacy posture when employing WebView-based IABs. Finally, we use our measurement setup to investigate if the behavior of WebViews varies based on the website a user visits. To this end, we systematically crawl the landing pages of 100 randomly selected top sites, taken from Google Chrome's top 1K most visited origins in the snapshot of February 2023 (CrUX) [221], using the ten different WebViews previously identified. In addition to the WebViews found on apps, we also crawl each site using the Android's System WebView Shell App which gives us a baseline for the network requests expected to be made from a WebView without any injections [222]. These crawls were executed with a Pixel 3 device, which was connected to an academic ISP via WiFi. For each app, a distinct crawler was crafted to traverse the unique user interface of the app. During each website visit, the script utilizes Android Debug Bridge (ADB) commands to: (i) launch the app, (ii) navigate to the intended activity by simulating screen taps at predetermined coordinates, (iii) insert the desired crawl URL, (iv) tap on the URL to instigate a visit within a WebView, and (v) swipe upwards to scroll through to the end of the webpage. Following a 20-second wait to allow the page to fully load, we gather the device's network log. To ready the system for the next crawl, we also purge the logs on the device, terminate the app, and wait for 1 minute.

Ethics

In conducting our manual analysis of the top 1K apps, we ensured that our experimentation methodology did not induce a high load on online services, as we manually navigated the apps and tested how they implement IABs using only one benign URL. When carrying out our automated crawls, we recognized the possibility that some of the URLs could potentially contain sensitive or malicious content. As a proactive measure to safeguard other users, we ensured that none of the URLs are posted publicly, thereby eliminating any potential harm to other users on the platform.

Limitations

Our approach enables us to conduct comprehensive measurements for WebView-based IABs in real-world Android apps. However, it is worth noting some important limitations to our methodology.

- Our methodology inherently necessitates significant manual effort, thus constraining our scope to a limited number of apps. We focus on the top 1K apps due to their wide usage and significant influence on the user base. In all 10 apps where we discovered WebView-based IABs, creation of dummy accounts was a prerequisite to accessing the app content. Automating account creation is challenging to fully automate, given the broad diversity in app UI designs and authentication workflows. For the same reason, a random Monkey [223] - despite its efficacy in other studies - may also not be effective in our context.
- During our manual analysis, we observed only 1 WebView-based IAB per app. How-

ever, it is possible that there exist other WebView-based IABs, embedded deeper within the app or activated by specific user behavior we didn't replicate. While our observations might not fully encapsulate all potential privacy-invasive actions apps could be executing in the wild, they contribute an initial, meaningful characterization of why apps may still employ WebView-based IABs.

- Our crawl, although automated, was limited to the landing pages of the top 100 sites. The restriction was due to rate-limiting we experienced with the Facebook app, which restricted our account twice during the measurements, necessitating manual intervention and creation of new dummy accounts. WebViews might exhibit different behaviors on other Web pages that we did not crawl.
- During our crawls, we attempt to load the entire page by scrolling down through the end of the page and allowing a 20-second pause for resources to load. However, we did not emulate any additional user behaviors. WebViews could exhibit additional behavior when the user interacts with the page (e.g., form filling). Our measurements are a first step towards understanding the dynamics of real-world WebView-based IABs in Android apps.
- The `addJavascriptInterface` method of the `WebView` class facilitates an interface between the app native code and the JS Virtual Machine inside the WebView. Our measurements successfully record instances when such a bridge is exposed. However, our methodology lacks the ability to monitor the communication between the JS VM and the Java methods.

6.2 Findings

In this section, we examine the data from our measurements described in section 6.1, aiming to understand the diverse ways that Web content is embedded in Android apps. We mainly investigate the different use-cases for which apps use WebViews and CTs. Considering the security, privacy and performance issues related to WebViews, we hypothesize that WebViews should only be used in situations where CTs are insufficient – specifically, in cases where the app needs to interact with the Web content, such as in a hybrid app. Furthermore, we explore the use-cases where CTs could offer better security and privacy protection than WebViews. We also identify the apps that use WebViews to display IABs, and analyze the reasons and implications of this choice, particularly if it exposes users to any security or privacy risks.

6.2.1 Usage of WebViews vs. CTs in Android apps

In subsection 6.1.1, we describe our method for analyzing the usage of WebViews and CTs in the 146.5K most popular apps using static analysis techniques. Our analysis reveals that $\sim$55.7% of the apps use WebViews, $\sim$20% incorporate CTs, and $\sim$15% make use of both. To gain insights into the common use-cases, we've categorized SDKs — those implemented by over 100 apps — utilizing WebViews and CTs, as detailed in subsubsection 6.1.1. Cumulatively, we detected 125 SDKs using WebViews, 45 SDKs using CTs, and 34 SDKs using both, as depicted in Table 6.2. The popular SDKs we identified are used in approximately 67% of the apps integrating WebViews, 96% of the apps adopting CTs, and 76% of the apps utilizing both. These statistics are summarized in Table 6.6. It is worth noting that an app could be making use of more than one SDK, and it could also have multiple instances of WebViews or CTs, some of which might not be facilitated through the top SDKs. We take a holistic approach to examining the use-cases of WebViews and CTs through prevalent SDKs as it allows us to identify patterns and extract invaluable insights.

We analyzed the use-cases of WebViews and CTs in different app categories, as shown in Table 6.4 and Table 6.3, respectively. We found that WebViews were mainly used for advertising purposes, with 46 SDKs supporting this use-case in about 39K apps. In contrast, CTs were predominantly used for social media integration, with 6 SDKs enabling this use-case in nearly 23.8K apps. Figure 6.4 illustrates the distribution of use-cases (SDKs) per app category for the top-10 app categories that use WebViews and CTs respectively. We

Table 6.6: Statistics of the apps using WebViews and CTs.

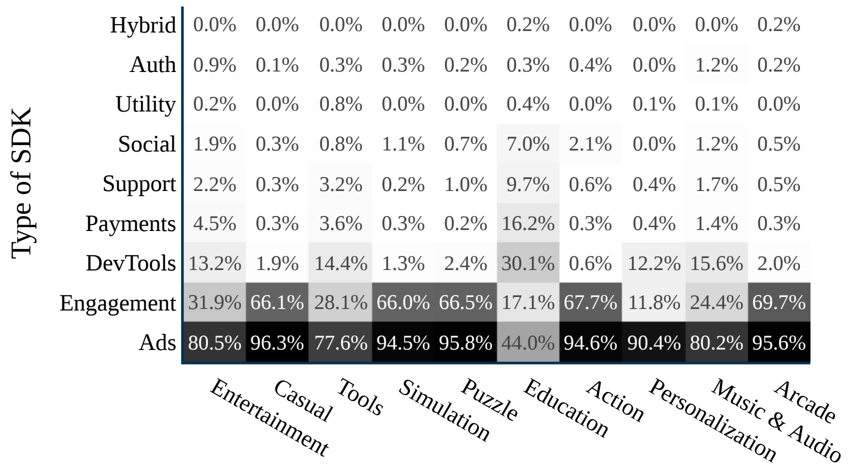
Dataset	**Total #apps**	**#apps using top SDKs**
Apps using WebViews	81,720	54,833
loadUrl	77,930	50,984
addJavascriptInterface	36,899	23,087
loadDataWithBaseURL	35,680	27,474
evaluateJavascript	26,891	18,716
removeJavascriptInterface	19,684	15,034
loadData	8,275	918
postUrl	5,028	2,678
Apps using CTs	29,130	27,891
Apps using both WebViews and CTs	21,938	16,810

observed that gaming apps (Puzzle, Simulation, Action, and Arcade) frequently used CT-based social media SDKs, while education apps used a lower proportion of WebView-based Ad SDKs (44%) and a higher proportion of WebView-based Payment SDKs ($\sim$16.2%). To acquire an exhaustive understanding of the various use-cases, we proceed with a comprehensive discussion of each individual use-case.

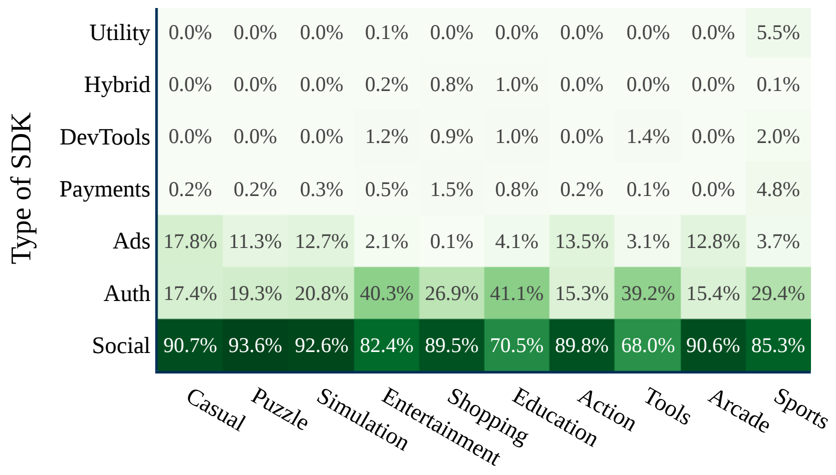
Advertising (Ads)

In-app Ads are a major source of revenue for free Android apps, and are essential to the mobile app economy. These ads are usually displayed in banner or interstitial formats, showcasing rich media, such as HTML5 content, or interactive ads, which are often streamed directly from the internet [224]. Google's Mobile Ads SDK empowers app developers to monetize their app content by showing Google's third-party ads adjacent to it. Although the SDK supports the use of Custom Tabs (CTs), it advocates for the use of WebViews, as they facilitate the synchronization of ads with app content [225]. Similar to Google Ads, there are several other ad networks implementing mobile ad SDKs, many of which have been studied by previous work [226, 227, 228, 229, 230].

Notwithstanding the benign nature of most ads served by these networks, numerous



(a) For apps which use **WebViews** via SDKs.



(b) For apps which use **CTs** via SDKs.

Figure 6.4: Distribution of SDKs per app category.

instances exist of malicious ads exploiting WebViews' capabilities. Liu et al. discovered instances where popular mobile ad networks manipulated WebViews to present deceptive click ads, execute malevolent JavaScript code on the device (such as cryptojacking), redirect users to harmful websites, and deceive users into downloading malicious apps [229]. Browsers, which CTs can utilize, are constantly evolving to incorporate robust defenses against such Web threats. For instance, features such as Google's Safe Browsing offer real-time threat intelligence [157], which could potentially decrease users' exposure to harmful content. Since WebViews are customizable, Ad SDKs can choose to disable SafeBrowsing, whereas Ad SDKs using CTs would be subject to SafeBrowsing unless the user has explicitly disabled it in their browser. Research conducted by Son et al. has highlighted instances where malicious mobile ads have exploited the access granted by WebViews to extract sensitive information from the device's external storage, and in certain cases, even read the user's local files [227]. Furthermore, there have been proposals to enforce stricter separation of privileges between the ad SDK environment and the host app [228, 231]. Android's CTs practically implements those goals by securely segregating the execution context of the mobile ad from the host app.

Nevertheless, in our analyses, we found that the usage of CTs by Ad SDKs is minimal. Only 3 Ad SDKs were observed using CTs in $\sim$2K apps, all of which also utilized WebViews. As seen in Figure 6.5, more than 45% of apps displaying Ads via WebViews expose a JS Bridge to the WebView using the `addJavascriptInterface` method, while more than 30% apps inject JS via the `evaluateJavascript` method. This exposes an attack surface which might not always be necessary to the intended functionality. As shown in Table 6.4, the prevalent practice among Ad SDKs is still to use WebViews, with 46 SDKs implemented in $\sim$40K apps.

Takeaway: Ad SDKs should consider implementing CTs where possible, and app developers should carefully weigh the functionality offered by WebView-based Ad SDKs against the attack surface they expose. Our measurements show that ad networks primarily rely on WebViews, which have previously been exploited by malicious ads, and CTs offer a promising replacement.

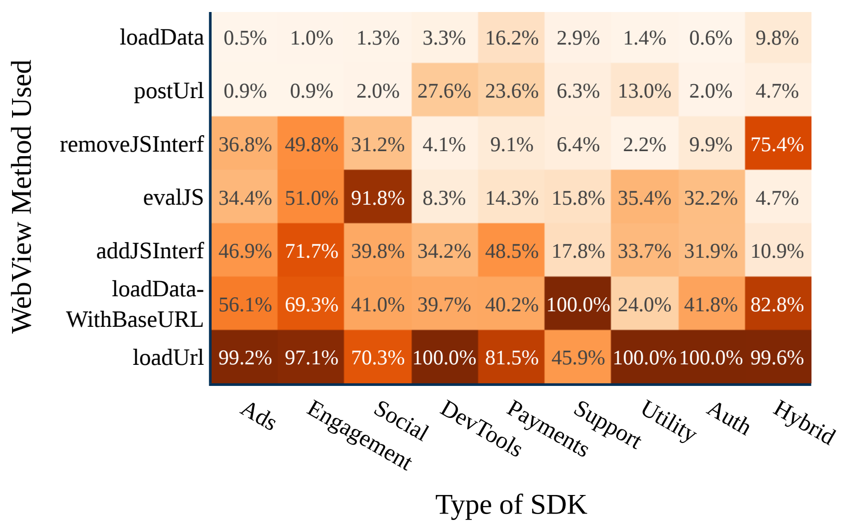


Figure 6.5: Heatmap of WebView API method calls made via SDKs.

Engagement

App developers frequently employ engagement measurement or analytics SDKs to determine which content is most engaging, intriguing, or valuable to the user. Our data indicate that ∼21K apps use a WebView via such an SDK. The Open Measurement (OM) SDK is the most prevalently used, found in an estimated ∼11.3K apps. The OM SDK facilitates third-party access to engagement measurement data. Its primary use is to measure and verify ad performance [232]. This involves checking whether an ad was visible, the duration of its visibility, its compliance with the set performance metrics, and whether it engaged in legitimate interactions as opposed to fraudulent views. Our manual analysis verifies that the OM SDK is predominantly used to support the ad networks AdColony and Ogury, which are found in ∼10.6K and ∼1.4K apps respectively, for performance measurement. We did not detect Engagement or Measurement SDKs using CTs. Even though CTs natively measure similar user engagement signals [233], we regard the use of SDKs to measure user engagement as a legitimate use case for WebViews. The functionality of the app, such as determining which ad to display and when, could benefit from more detailed and custom measurements than those currently provided by CTs.

Development Tools

We label SDKs which provide a set of software development tools and libraries to help streamline the process of creating apps as 'Development Tools' (Dev Tools). These SDKs enable developers to cost-efficiently build high-quality apps. For instance, Google's Flutter is a cross-platform framework that enables developers to build apps compatible with both Android and iOS using a single codebase [234].

In our measurements, $\sim$7K apps utilize WebView components provided by such Dev Tool SDKs. Specifically, we found $\sim$5.5K apps leveraging `url_launcher`, a Flutter plugin used by apps to open a URL in a WebView [235]. Interestingly, about $\sim$2K apps make use of 'InAppWebView', an actively maintained third-party Flutter plugin used to embed WebView-based IABs in apps [236]. This plugin, which has earned over 2.6K stars on GitHub, boasts functionality for visiting third-party Web content in an IAB, while also offering the ability to inject JS code into the WebView-based IAB. We found a limited number of 172 apps using CTs via such SDKs. Notably among them, 'android-customtabs' is an SDK developed to default apps to WebViews if no browser on the user's device supports CTs [237]. As adoption of CTs increases, we believe that support for Dev Tools SDKs using CTs will also grow. Our discussion of the security and privacy implications of Dev Tools SDKs utilizing either WebViews or CTs is limited, as their actual use-case depends on the app developers incorporating them – they alone specifically do not constitute a use-case.

Payments

Payment processing SDKs offer APIs, UI components, and other tools that make it easier to integrate payment processing into an app. They can be used by e-commerce apps, either directly (i.e., selling products on a store) or indirectly (i.e., displaying ads with buy buttons next to other content) – or by freemium apps that generate revenue from in-app purchases of premium features. Mahmud et al.'s security analysis of Android Payment SDKs uncovered

26 SDKs that used WebViews to enable payments functionality such as checkout [238]. They reported that none of the SDKs configured WebViews securely enough to comply with the OWASP Mobile Application Security Verification Standard [239]. In particular, 20 of those SDKs breached the PLAT4 platform requirement by exposing sensitive payment data (e.g., credit card numbers) from the WebView to the app. As seen in Figure 6.5, our measurements too indicate than 48.5% apps using WebViews for payments expose a JS Bridge to the WebView. Plaid Link, a financial services SDK that enables app users to connect their bank accounts, addressed these issues by switching to CTs [240]. They emphasized that besides the interception of sensitive credentials, the absence of security UI built into browsers such as Chrome (e.g., SSL verification lock icon) could allow a malicious app to imitate their UI flow and steal the user's credentials.

However, our own analysis showed that WebViews continue to be the prevailing choice for Payment SDKs. We identified their use in ∼3.2K apps, facilitated by 15 distinct Payment SDKs. The most prominent SDKs that utilize WebViews include Stripe (∼1.1K apps), RazorPay (484 apps), and PayTM (400 apps), all of which do so to facilitate checkouts. CTs, on the other hand, were found in 6 SDKs and were implemented by a mere 208 apps. Juspay (77 apps) and Ticketmaster Checkout (47 apps) - SDKs utilizing CTs, both also facilitate checkouts. Juspay's documentation outlines their use of CTs to enable payments via Amazon Pay [241]. Intriguingly, we observed that 5 out of the 6 SDKs supporting CTs also provide support for WebViews. We hypothesize that this is because these SDKs default to WebViews when a browser with CT support is unavailable.

Takeaway: Payment providers should use CTs instead of WebViews for integrating payment processing in their SDKs, as CTs provide more secure and user-friendly payment experiences. We find that most payment SDKs still use WebViews, which can leak sensitive payment data or expose users to phishing attacks.

User Support

User support SDKs allow developers to offer customer service options within their app, rather than directing users to call or email. We found that ∼1.7K apps used WebViews to enable in-app customer service. Of these, ∼85% relied on Zendesk and Freshcat SDKs to provide live chat support. This is a suitable use-case for WebViews, as the customer service Web component may need to access information from the app. For example, if a user requests support for an order they made, the customer service Web component may need to retrieve order status details from the app. This is supported by our analysis of WebView APIs by User Support SDKs. In particular, as seen in Figure 6.5, all app using WebViews for user support load local data into the WebView using the `loadDataWithBaseURL` method, while only 45.9% apps use `loadUrl`. Not surprisingly, we did not find any apps that used CTs for in-app customer service.

Social

Social Media SDKs enable developers to access social media platforms' functionalities such as authentication, sharing, and posting. We analyzed 6 SDKs that use CTs and 10 SDKs that use WebViews, primarily to enable authentication. We found that ∼23.8K apps use CTs, while only ∼1.7K apps use WebViews. Among the apps that use CTs, ∼98% of them rely on Facebook's SDK, which deprecated WebViews in October 2021 due to an increase in phishing attempts via WebViews [242]. With CTs, users can stay logged in to Facebook across sessions, which increases conversion rates for apps authenticating via Facebook. Similarly, NAVER, a South Korean online service, has also deprecated WebViews in favor of CTs for OAuth authorization [243]. We identified 406 apps that use WebViews and 156 apps that use CTs via NAVER's SDK. Using CTs for authorization requests is also in line with the best practices set out in the IETF RFC 8252 for 'OAuth 2.0 for Native Apps' [244].

However, not all social media SDKs have adopted CTs for authentication yet. For example, the SDK for VK, a Russian social media platform, uses WebViews, and is used by 456 apps to authenticate via VK. Kakao's SDK, another South Korean platform, also uses WebViews in 347 apps for OAuth. Additionally, 54 apps use CTs via Kakao's utilities SDK, but their purpose is unclear.

Takeaway: Social media SDKs should prefer CTs, especially for authentication. Widely-used SDKs such as Facebook and NAVER have already switched to CTs as it provides better usable security, however there still exist SDKs such as VK and Kakao which use WebViews.

Utility

We define 'Utility' SDKs as those that integrate useful features and utilities from other online services. We measured 4 Utility SDKs that use WebViews and 2 that use CTs in our study. For example, NAVER Maps SDK, which is used by 130 apps, provides map-related functionality to apps. Ticketmaster's SDK allows app developers to access Ticketmaster features such as ticketing and booking management. It uses both WebViews and CTs for authentication and checkout processes. We also detected 16 apps that use CTs through the MyChart SDK, an online service that enables patients to schedule appointments and access their medical records. We argue that WebViews are appropriate for utilities such as Maps, where they can leverage the capabilities of WebViews, but CTs are preferable for utilities such as MyChart, where sensitive information is involved.

Authentication

Authentication or identity SDKs help developers to add authentication and authorization features with identity providers (IDPs). They offer methods and classes to perform tasks such as signing in, signing up, requesting permissions, verifying credentials, and managing sessions. The threats discussed in subsubsection 6.2.1 for social media SDKs also apply to these SDKs. Authentication SDKs should avoid using WebViews because they handle sensitive information such as credentials, which can be compromised. User experience (UX) would also improve with CTs, as the users' active sessions will persist, and they can also use password managers without having to manually enter credentials. Moreover, WebAuthn, an upcoming Web standard for secure authentication via FIDO2-based security keys or passkeys, is supported in CTs, but not in WebViews [245].

We measured 7 identity SDKs that use WebViews in our study. They include Gigya (120 apps), a customer identity and access management platform used by SAP, NAVER corporate identity SDK (90 apps), and Amazon's IDP (37 apps). However, a vast majority of apps (∼7.8K) used CTs for authentication. Among them, ∼97% of the apps used Google Firebase's Authentication SDK, which allows apps to authenticate into custom IDPs and perform reCAPTCHA checks, among other functionality [246].

Takeaway: IDPs should prefer using CTs in their SDKs to protect sensitive information, and improve the user experience (via password managers and passkeys). Popular SDKs, such as Google Firebase, use CTs, while some others, such as Gigya and Amazon, still use WebViews.

Hybrid Functionality

Hybrid Functionality refers to the ability to combine Web technologies (HTML, CSS, and JS) with native features of the device in an app. They offer the advantages of both Web and native platforms – the reach and flexibility of the Web platform, and the power and full functionality of the native platform. WebViews, which have extensive capabilities, are designed for hybrid apps. As shown in Table 6.4 and Table 6.3, our measurements also find apps that use both WebViews and CTs to create hybrid gaming (Baby Panda World, Cube Storm) and news reading (Scripps News) apps.

6.2.2 WebView-based IABs

Android launches a Web URI Intent when a user clicks on an HTTP(S) URL within an app. As per Android's documentation, the default browser handles the Web URI intent on Android 12 and later versions, unless there is an app installed that can handle URLs from that specific domain [247]. For instance, a `maps.google.com` URL clicked from a social media app will launch the Google Maps app if it is present. Otherwise, it will open in the default browser. However, our semi-manual analysis of the top 1K apps, as detailed in subsubsection 6.1.2, revealed 11 popular social media apps that do not launch a Web URI intent, but instead open the URL using an In-App Browser (IAB). Among them, 10 apps used WebView-based IABs to open HTTP(S) URLs. This implementation is problematic for several reasons. First, WebViews are designed to display first-party content, not third-party content. Second, WebViews are insecure and vulnerable to various attacks, as discussed in section 2.4. Third, WebViews have extensive functionalities that can potentially be abused by apps. In this section, we will examine our comprehensive measurement datasets, as collected in subsubsection 6.1.2, to investigate whether such exploitation is occurring in practice.

We begin by examining App-WebView Interactions, which indicate whether and how apps use privileged WebView API methods. We observed that Snapchat, Twitter and Reddit did not inject any JS (via `evaluateJavascript`) or JS Bridge (via `addJavascriptInterface`) into the WebView. Pinterest injected a JS Bridge into the WebView-based IAB that was triggered when a URL in the DM activity was clicked, but the name of the exposed Java class was obfuscated, so we could not determine the purpose of the injection. For the remaining 6 out of 10 apps, we detected both JS and JS Bridge injections. Facebook and Instagram exhibited identical behavior, as did Moj and Chingari. LinkedIn and Kik showed their own unique behaviors. We explore these 4 behaviors in more depth by analyzing our measurements collected in subsubsection 6.1.2. To gain more insight into the reasons behind a certain measurement, we also manually investigated using Android logs collected by Logcat [248], and by using the remote GUI debugging tool for Android [249]. They facilitated our manual investigation with fine-grained and real-time information, that helped us comprehend activity on the device. We summarize our findings of the inferred intents for all 10 apps in Table 6.7.

Facebook and Instagram

Facebook and Instagram are both popular social media apps operated by Meta. Facebook's Android app has more than 8.4B downloads, and Instagram's Android app has more than 4.6B downloads. We manually analyzed the logcat logs when a user clicks on a URL in a Facebook news feed post or an Instagram direct message. We found that no intent was raised and instead, a WebView opened to load the URL. Using remote debugging, we discovered that the URL was implemented as a button that triggered the app logic to open the WebView. Our App-WebView interaction measurements further revealed that the app injected several JS scripts and JS Bridges into the Web content after loading. The code was well documented and we could infer the purpose of each injection. The following JS injections were performed:

- A JS script that inserted a Facebook Autofill script element into the page, as shown in Listing 6.1. We believe that the script was used to populate merchant checkouts with user information such as name, address and phone number from the user's Facebook profile.

```
(function(d, s, id){
  var sdkURL = "//connect.facebook.net/en_US/iab.autofill.enhanced.
    js";
  var js, fjs = d.getElementsByTagName(s)[0];
  if (d.getElementById(id)) {
      return;
  }
  js = d.createElement(s);
```

Table 6.7: Summary of WebView injection and its inferred intents.

No. of downloads	App Name	WebView Via	Inferred Intent for Injected Content	
			HTML/JS Injected	JS Bridge Injected
8.4B	**Facebook**	Post	Returns DOM Tag Counts. Returns simHash for page to detect cloaking[250]. Logs performance metrics. Insert FB Autofill SDK JS script.	Meta Checkout. AutoFillExtensions. Facebook Pay.
4.6B	**Instagram**	DM		
2.34B	**Snapchat**	Story	No injection.	No injection.
1.38B	**Twitter**	DM	No injection.	No injection.
1.2B	**LinkedIn**	Post	Calls to Cedexis traffic management API.	No injection.
840M	**Pinterest**	DM	No injection.	(Obfuscated)
289M	**Moj**	Profile Bio	Insert and manage a video Ad via Google Ads SDK.	Google Ads.
97.5M	**Chingari**			
124M	**Reddit**	DM	No injection.	No injection.
176.5M	**Kik**		Insert ads via Ad Networks: Google Ads, MoPub and InMobi.	Google Ads.

```
  js.id = id;
  js.src = sdkURL;
  fjs.parentNode.insertBefore(js, fjs);
}(document, 'script', 'instagram-autofill-sdk'));
```

Listing 6.1: JS executed by Facebook and Instagram's WebView-based IAB to embed a JS SDK that populates merchant checkouts with user information from their Facebook profile.

- A JS script that returned a frequency dictionary with the DOM tag counts.
- A JS script that returned locality sensitive hashes for (i) text and DOM elements, (ii) text elements, and (iii) DOM elements. The comments indicated that this information was collected to detect client-side cloaking based on Cloaker Catcher by Duan et al. [250].
- A JS script that logged performance metrics to the console. It recorded the time it took to load the DOM content and whether the page was an Accelerated Mobile Pages (AMP) page.

In the visit to our controlled Web page via Facebook and Instagram's WebView-based IABs, we also recorded the methods of the Document and Element Web APIs being called, as shown in Table 6.8. This confirmed that the injected JS code was not just injected, but also executed.

Besides JS code, the WebViews also had access to JS bridges, namely `fbpayIAWBridge`, `metaCheckoutIAWBridge`, and `_AutofillExtensions`. We believe that the first two enable payments via Facebook [251] and checkouts via Meta [252], if the user visits a website that supports these feature (e.g., by clicking on an ad). The third one provides the Java interface that the autofill JS code can use to retrieve user information from their Facebook profile. The network logs revealed that Facebook and Instagram's WebViews used redirectors hosted at `lm.facebook.com/l.php` and `l.instagram.com` respectively. They passed the intended URL and a random identifier to the redirector in the

Table 6.8: Web APIs accessed by apps, as recorded by our controlled webpage server.

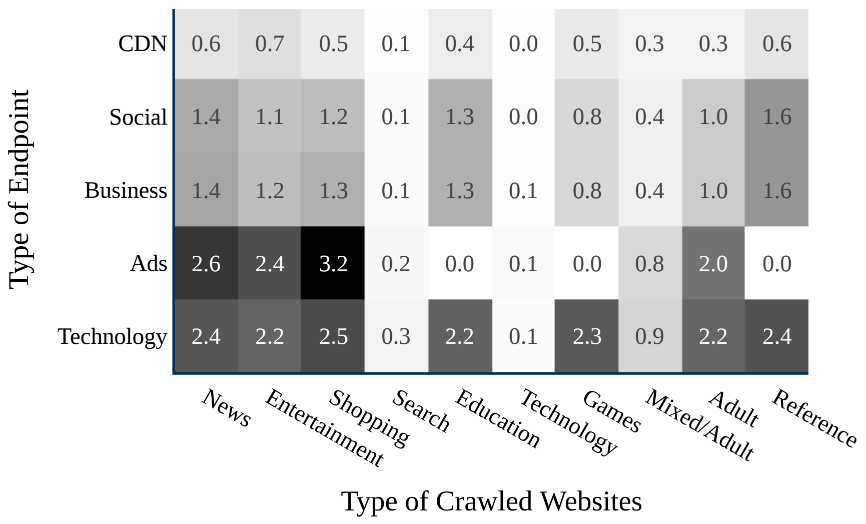
App Name	**Web API used**	
	Interface	**Method**
Facebook & Instagram	Document	getElementById
		createElement
		querySelectorAll
		getElementsByTagName
		addEventListener
		removeEventListener
	Element	insertBefore
		hasAttribute
		getElementsByTagName
	HTMLBodyElement	insertBefore
	HTMLCollection	item
	NodeList	item
	HTMLMetaElement	getAttribute
Kik	HTMLDocument	querySelectorAll
	HTMLMetaElement	getAttribute
	Document	querySelectorAll

GET request, which could be exploited for tracking the user [253] (a similar redirect via `t.co` was also observed for Twitter). In the crawl of top 100 websites, the WebViews did not make any network requests other than those to the intended website its resources.

LinkedIn

LinkedIn is a social network for professionals, and its Android app has more than 1.2 billion users. It uses a WebView-based IAB to display links found in their news feed posts. Other than visiting the intended webpage, we found that the IAB also loads resources from `radar.cedexis.com` and runs JS code to interact with the Cedexis Radar API at `cedexis-radar.net`. Cedexis Radar, currently part of Citrix's NetScaler Intelligent Traffic Management, aims to 'collect network performance data to enable smart routing decisions' [254]. It offers its customers, such as LinkedIn, access to a large set of network performance data collected from real users worldwide, in return for conducting and sharing network measurements from their own end-users. Radar's documentation states that it measures availability, response time, and throughput across different Cloud and CDN providers on an end-user's device. Radar's Android SDK called AndroidRadar, outlines how developers can integrate their network measurement library into their app WebViews [255].

Figure 6.6 illustrates the distribution of distinct endpoints contacted by LinkedIn's IAB, per different type of website crawled via the IAB. These endpoints were specific to LinkedIn's IAB and were not contacted by any other app's IAB. We classified the endpoint types using Symantec Sitereview [256]. We observed that for websites with rich content, such as News, Entertainment and Shopping, LinkedIn's IAB contacted more than 2 trackers on average (e.g., Cedexis trackers), as well as its own services, such as CDN (`licdn.com`), ad engagement APIs (`px.ads.linkedin.com`) and performance monitoring (`perf.linkedin.com`). We believe that the average number of endpoints contacted was smaller for Search or Technology websites because they contained less content to interact with. Our results broadly corroborate with the functionality of Radar as described in its documentation.



	News	Entertainment	Shopping	Search	Education	Technology	Games	Mixed/Adult	Adult	Reference
CDN	0.6	0.7	0.5	0.1	0.4	0.0	0.5	0.3	0.3	0.6
Social	1.4	1.1	1.2	0.1	1.3	0.0	0.8	0.4	1.0	1.6
Business	1.4	1.2	1.3	0.1	1.3	0.1	0.8	0.4	1.0	1.6
Ads	2.6	2.4	3.2	0.2	0.0	0.1	0.0	0.8	2.0	0.0
Technology	2.4	2.2	2.5	0.3	2.2	0.1	2.3	0.9	2.2	2.4

Figure 6.6: Distribution of no. of unique endpoints contacted by LinkedIn's IAB.

Moj and Chingari

Moj and Chingari are short video-based social media apps, developed by two different developers for the Indian audience. They have been downloaded more than 289M and 97M times, respectively. We discovered that both apps exhibit identical behavior in terms of WebView injections. The injected JS code was highly obfuscated, making it difficult to understand its functionality. However, we manually inspected the names of the JS Bridges exposed and the variable names used by the JS code, and inferred that the injections aimed to insert and manage video ads via the Google Ads SDK. Our inference was supported by the fact that one of the JS bridges injected was named `googleAdsJsInterface`, and the injections included JSON data fields with Ad specifications of ads from `doubleclick.net` – which is known to serve Google Ads [257]. Despite the numerous injections, we did not observe any ads on our test page, nor did our server record any Web API usage. Upon further inspection of the JSON objects containing the ad details for both Moj and Chingari, we observed that although they had populated parameters about which ad to display and where to fetch it from, the width and height were fixed to 0 – and a field 'notVisibleReason' was set to 'noAdView'. Based on this, we believe that the injected code did not execute to show any ad because there was no compatible ad view on the page. If there were an ad view on the page, the app's injection could potentially help Google Ads display an ad more relevant in the context of the app. However, to understand ad injection, we are limited to measurements from our controlled Web page, as measurements from our crawl dataset could be contaminated with activity from ad networks used by the crawled Web pages themselves.

Kik

Kik, an instant messaging application downloaded more than 176.5M times, utilizes a WebView-based IAB to open URLs that users open in private Direct Messaging windows. We discovered that Kik's IAB also injects the `googleAdsJsInterface` bridge that we found injected in Moj and Chingari. However, the JS code injected by Kik was different and markedly more obfuscated than Moj and Chingari. Due to the complex nature of the JS code, we were unable to parse the ad payload, even after manual inspection. Furthermore, as highlighted in Table 6.8, our Web server logged the IAB's use of only read-only Web

APIs, confirming that there was no actual modification to the DOM of our test page. However, our network logs did highlight that the IAB communicated with ad networks. For a more comprehensive understanding, we analyzed the distribution of endpoints contacted by Kik's IAB during our top site crawl, as displayed in Figure 6.7. The plot revealed that when visiting websites rich in content, Kik's IAB communicates with, on average, over 15 ad network endpoints. MoPub (ads.mopub.com) and InMobi (supply.inmobicdn.net) are two prominent ad networks contacted by the IAB. From these findings, we hypothesize that the JS code injected by Kik communicates with a multitude of ad networks, and could be selectively inserting ads, but the its logic framework remains unclear to us.

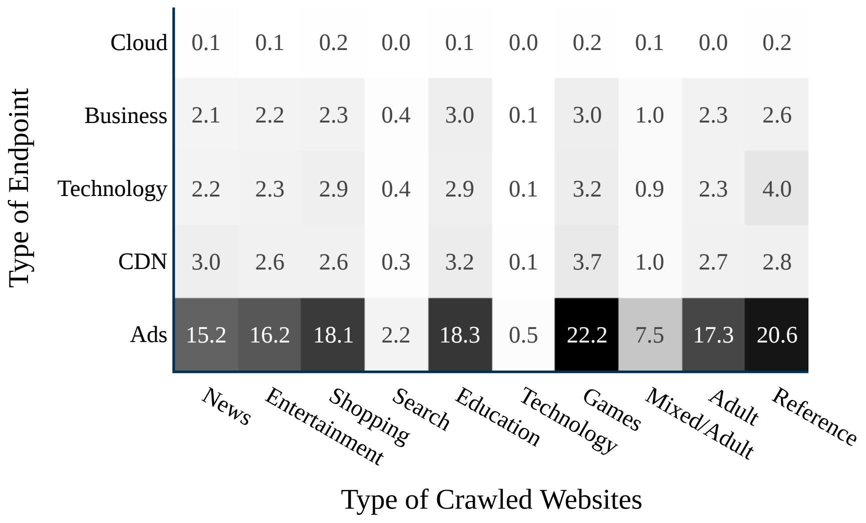


Figure 6.7: Distribution of no. of unique endpoints contacted by Kik's IAB.

6.3 Concluding Remarks

In this study, we conducted extensive measurements on popular Android apps with the goal of providing novel insights into the state of Web security in mobile apps. Our large-scale investigation illuminated the adoption rate of CTs, a secure and user-friendly alternative to WebViews. Furthermore, our detailed examination of WebView-based IABs in the top 1K apps revealed practices potentially harmful to the user's security and privacy.

We analyzed $\sim$146.5K apps, each boasting over 100K users, and highlighted the diverse use-cases of non-browser mobile apps presenting Web content via WebViews and CTs. We contend that for use-cases like Authentication and Authorization, using WebViews contravenes the least privilege principle. As a preferable alternative, CTs provided by browsers such as Chrome not only bolster security, but also enhance user experience through shared state with the user's default browser, a consistent and secure UI, and faster load times.

Our study is a first step in quantifying the adoption of CTs in widely used apps. Building upon previous work, we take an evidence-based approach in highlighting the risks for SDKs persisting in using WebViews. We note that many popular SDKs have already migrated to CTs; however, a considerable number of extensively used SDKs lag behind. We recognize SDKs that measure user engagement of first-party Web content, provide in-app support, and offer other utilities as legitimate use-cases of WebViews. Future research could focus on adapting Ad SDKs, the most common WebView application, to CTs, taking advantage of innovations like Partial CTs, which enable developers to launch resizable inline CTs in response to native ads, as showcased by Google in 2023 [258].

Typically, clicking on HTTP(S) URLs should invoke a Web URI Intent, which is then handled either by an appropriate app or the default browser. However, our study found 11 apps from the top 1K that disguised URLs as buttons with hyperlink-like text, which upon clicking launched an IAB, instead of raising an intent. A deeper probe revealed a variety of unique behaviors. Apps such as Facebook and Instagram used this method to facilitate payments and checkouts, despite the potential for misuse by malicious websites to pilfer personal user information. Other apps like Moj, Chingari, and Kik did so for potential ad injection, while LinkedIn incorporated a network measurement SDK to source performance metric data from user devices. Despite the seemingly benign behaviors found in WebView-based IABs, we suggest that the extensive attack surface they expose renders the trade-off inadequate. Although some apps offer an option to disable in-app browsers for privacy-focused users [259], we recommend apps make it an opt-in feature for improved user safety.

Our research forms the groundwork for improving transparency regarding how mobile apps display third-party Web content. Future research could consider including WebView usage for such content as a metric in the ‘privacy nutrition labels’. Educating users can both empower users with more agency over their online safety, while encouraging the ecosystem of apps and SDKs to adopt safer practices [260, 261]. Considering the multitude of people who use android devices for Web browsing every day, app and SDK developers can improve user safety and foster users’ trust in the ecosystem by making secure choices, such as implementing Custom Tabs wherever possible.

CHAPTER 7
CONCLUSION

This dissertation presents a series of empirical investigations addressing key issues in web security and privacy, ranging from authentication and anti-abuse measures on digital platforms to the privacy concerns of users using web browsers and non-browser mobile apps. A recurrent pattern discernible across our research exhibits that while the community innovates with new technologies to fulfill legitimate needs or use-cases – typically accompanied by proposed safeguards – the actual deployment often prioritizes the use-case at the expense of the prescribed precautions. This oversight creates a gaping vulnerability that malicious actors may exploit, thereby undermining trust in the constituent parts of the ecosystem. For instance, in our evaluation of user's web privacy, potent features like native app communication via WebSockets, and hybrid app development via WebViews, both run the risk of being manipulated for undue access. With the user's privacy hanging in the balance, we bear the responsibility to consistently create and enforce protective measures that facilitate legitimate use-cases without encroaching on user privacy. Since the publication of our findings, Chrome and Brave browsers have already begun limiting local network communication for their users.

In our evaluation of the tangible security merits of current passwordless authentication deployments, we noted that the theoretically proposed guardrails intended for protection against compromised clients prove ineffective. This failure isn't only a result of overlooked aspects during implementation but is also due to the compartmentalized nature of the ecosystem. While the protocol itself is flexible enough to allow the development of additional features through extensions, which can cater to specific scenarios such as payments – where the authentication is required not only for the service but also for the transaction data – it requires independent implementation by numerous entities (i.e., authenticator, operating system, client/browser, online service). This is a prerequisite for achieving widespread acceptance. In the case of the transaction confirmation extension, we noted that despite some parties implementing it, the proposal faded away due to other parties' failure to follow suit in a timely manner. Furthermore, each player in the ecosystem operates independently, often neglecting the cumulative effect their actions might have. In particular, the FIDO Alliance's Metadata Service (MDS), which is pivotal in providing trust information, including vulnerability alerts for authenticators that support passwordless authentication, is unfortunately isolated from various vendors. Our measurements indicate that even when Samsung publicly disclosed a major vulnerability in their widely used flagship mobile devices, the MDS did not update to reflect the vulnerabilities. Consequently, this undermines the overall trust in the ecosystem. As we wait for the ecosystem to evolve to a point where all components function in harmony, it's necessary to devise interim solutions, some of which we examine in this thesis.

Furthermore, trust in the ecosystem is also undermined when web platforms are considered easy to manipulate. My work on measuring view fraud abuse on YouTube shows that the impact of organic view fraud campaigns is rather short-lived. However, given that the campaigns persist over time, we conclude that the view fraud ecosystem exhibits snake oil properties.

As a community, it is incumbent upon us to measure such gaps between recommendations made in theory and reality at regular intervals. Even after identifying what to measure, conducting accurate measurements can be challenging as real-world implementations are often complex and opaque to an external researcher. In my thesis, I demonstrate measurement techniques which allow us to model reality in various contexts, and generate precise, actionable recommendations that can be adopted broadly and readily. As new features get added to the Web, our community needs to continue identifying and closing the gaps, in order to increase trust within the online ecosystem.

REFERENCES

[1] *Mobile vs. Desktop vs. Tablet Traffic Market Share*, https://www.similarweb.com/platforms.

[2] L. Lassak, A. Hildebrandt, M. Golla, and B. Ur, "“It's Stored, Hopefully, on an Encrypted Server”: Mitigating Users' Misconceptions About FIDO2 Biometric WebAuthn," in *USENIX Security Symposium*, 2021.

[3] S. G. Lyastani, M. Schilling, M. Neumayr, M. Backes, and S. Bugiel, "Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[4] K. Owens, O. Anise, A. Krauss, and B. Ur, "User Perceptions of the Usability and Security of Smartphones as FIDO2 Roaming Authenticators," in *Symposium on Usable Privacy and Security (SOUPS)*, 2021.

[5] F. M. Farke, L. Lorenz, T. Schnitzler, P. Markert, and M. Dürmuth, "“You still use the password after all” - Exploring FIDO2 Security Keys in a Small Company," in *Symposium on Usable Privacy and Security (SOUPS)*, 2020.

[6] M. Barbosa, A. Boldyreva, S. Chen, and B. Warinschi, "Provable Security Analysis of FIDO2," in *International Cryptology Conference (CRYPTO)*, 2021.

[7] N. Bindel, C. Cremers, and M. Zhao, "FIDO2, CTAP 2.1, and WebAuthn 2: Provable Security and Post-Quantum Instantiation," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2023.

[8] J. Guan, H. Li, H. Ye, and Z. Zhao, "A Formal Analysis of the FIDO2 Protocols," in *European Symposium on Research in Computer Security (ESORICS)*, 2022.

[9] C. Jacomme and S. Kremer, "An Extensive Formal Analysis of Multi-factor Authentication Protocols," *ACM Transactions on Privacy and Security (TOPS)*, 2021.

[10] M. Jubur, P. Shrestha, N. Saxena, and J. Prakash, "Bypassing Push-based Second Factor and Passwordless Authentication with Human-Indistinguishable Notifications," in *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2021.

[11] E. Ulqinaku, H. Assal, A. AbdelRahman, S. Chiasson, and S. Capkun, "Is Real-time Phishing Eliminated with FIDO? Social Engineering Downgrade Attacks against FIDO Protocols," in *USENIX Security Symposium*, 2021.

[12] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system," in *ACM Conference on Internet Measurement (IMC)*, 2007.

[13] U. Gargi, W. Lu, V. Mirrokni, and S. Yoon, "Large-Scale Community Detection on YouTube for Topic Discovery and Exploration," in *International AAAI Conference on Web and Social Media*, 2011.

[14] V. Bulakh, C. W. Dunn, and M. Gupta, "Identifying fraudulently promoted online videos," in *International Conference on World Wide Web (WWW)*, 2014.

[15] A. Tripathi, K. K. Bharti, and M. Ghosh, "A Study on Characterizing the Ecosystem of Monetizing Video Spams on YouTube Platform," in *International Conference on Information Integration and Web-Based Applications and Services*, 2019.

[16] H. S. Dutta, M. Jobanputra, H. Negi, and T. Chakraborty, "Detecting and Analyzing Collusive Entities on YouTube," in *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2021.

[17] Y. Li, O. Martinez, X. Chen, Y. Li, and J. E. Hopcroft, "In a World That Counts: Clustering and Detecting Fake Social Engagement at Scale," in *International Conference on World Wide Web (WWW)*, 2016.

[18] M. Marciel *et al.*, "Understanding the Detection of View Fraud in Video Content Portals," in *International Conference on World Wide Web*, 2016.

[19] N. Shah, "FLOCK: Combating Astroturfing on Livestreaming Platforms," in *The International Conference on World Wide Web (WWW)*, 2017.

[20] V. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis, "Puppetnets: misusing web browsers as a distributed attack infrastructure," in *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[21] J. Grossman and T. Niedzialkowski, "Hacking Intranet Websites from the Outside: JavaScript Malware Just Got a Lot More Dangerous," in *Blackhat USA*, 2006.

[22] S. Stamm, Z. Ramzan, and M. Jakobsson, "Drive-by pharming," in *International Conference on Information and Communications Security (ICICS)*, 2007.

[23] T. Gallagher, *Port Scanning and WebSockets*, https://datatracker.ietf.org/meeting/96/materials/slides-96-saag-1/.

[24] T. Gallagher, *Security enhancement for websockets to prevent private network mapping*, https://tools.ietf.org/html/draft-gallagher-hybiwebsocketenhancement-00, 2016.

[25] S. Lee, H. Kim, and J. Kim, "Identifying Cross-origin Resource Status Using Application Cache," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[26] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster, "Web-based Attacks to Discover and Control Local IoT Devices," in *Workshop on IoT Security and Privacy (IoT S&P)*, 2018.

[27] *XSS Rays*, https://github.com/beefproject/beef/wiki/Xss-Rays.

[28] *Port Scanning with HTML5 and JS-Recon*, http://blog.andlabs.org/2010/12/port-scanning-with-html5-and-js-recon.html.

[29] T. Hornby, *Port scanning local network from a web browser*, https://defuse.ca/in-browser-port-scanning.htm, 2015.

[30] Peppersoft, *Local Network Scanner with JavaScript*, http://peppersoft.net/local-network-scanner-javascript/.

[31] *JavaScript LAN Scanner*, https://www.myria.de/lan-scan/index.php.

[32] *lan-js: Probe LAN devices from a web browser*, https://github.com/joevennix/lan-js.

[33] *sonar.js: A Framework for Identifying and Launching Exploits against Internal Network Hosts*, https://github.com/mandatoryprogrammer/sonar.js.

[34] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring HTTPS Adoption on the Web," in *USENIX Security Symposium*, 2017.

[35] E. Stark *et al.*, "Does Certificate Transparency Break the Web? Measuring Adoption and Error Rate," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[36] S. Calzavara, A. Rabitti, and M. Bugliesi, "Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild," in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[37] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[38] S. Englehardt and A. Narayanan, "Online Tracking: A 1-million-site Measurement and Analysis," in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[39] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The Web Never Forgets: Persistent Tracking Mechanisms in the Wild," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[40] P. Snyder, L. Ansari, C. Taylor, and C. Kanich, "Browser Feature Usage on the Modern Web," in *ACM Internet Measurement Conference (IMC)*, 2016.

[41] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[42] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan, "BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications," *IEEE Transactions on Information Forensics and Security*, 2018.

[43] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[44] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and Scalably Vetting JavaScript Bridge in Android Hybrid Apps," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2017.

[45] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: static analysis framework for Android hybrid applications," in *ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[46] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, "A tale of two cities: how WebView induces bugs to Android applications," in *ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[47] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *Mobile Security Technologies Workshop (MoST)*, 2015.

[48] G. Yang and J. Huang, "Automated Generation of Event-Oriented Exploits in Android Hybrid Apps," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[49] G. Yang, J. Huang, and G. Gu, "Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities," in *USENIX Security Symposium*, 2019.

[50] A. Tiwari, J. Prakash, A. Rahimov, and C. Hammer, “Our fingerprints don’t fade from the Apps we touch: Fingerprinting the Android WebView,” *arXiv:2208.01968*, 2022.

[51] Z. Tang *et al.*, “Dual-force: understanding WebView malware via cross-language forced execution,” in *ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[52] L. Zhang *et al.*, “Identity Confusion in WebView-based Mobile App-in-app Ecosystems,” in *USENIX Security Symposium*, 2022.

[53] G. S. Tuncay, S. Demetriou, and C. A. Gunter, “Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android,” in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[54] X. Zhang *et al.*, “An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications,” in *USENIX Security Symposium*, 2018.

[55] *Android Developers: WebView*, https://developer.android.com/reference/android/webkit/WebView.

[56] P. Beer, L. Veronese, M. Squarcina, and M. Lindorfer, “The Bridge between Web Applications and Mobile Platforms is Still Broken,” in *Workshop of Designing Security for the Web (SecWeb)*, 2022.

[57] Z. Zhang, “On the Usability (In)Security of In-App Browsing Interfaces in Mobile Apps,” in *Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[58] K. Thomas *et al.*, “Data Breaches, Phishing, or Malware?: Understanding the Risks of Stolen Credentials,” in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[59] *FIDO Security Reference - FIDO Alliance Proposed Standard*, https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-ps-20220523.html.

[60] F. Putz, S. Schön, and M. Hollick, “Future-Proof Web Authentication: Bring Your Own FIDO2 Extensions,” in *Emerging Technologies for Authorization and Authentication (ETAA)*, 2021.

[61] *Universal 2nd Factor (U2F) Overview*, https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html.

[62] *FIDO UAF Protocol Specification - FIDO Alliance Proposed Standard*, https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-protocol-v1.2-ps-20201020.html.

[63] *Client to Authenticator Protocol (CTAP) - FIDO Alliance Proposed Standard*, https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-errata-20220621.html.

[64] *Web Authentication: An API for accessing Public Key Credentials, Level 3*, https://w3c.github.io/webauthn.

[65] *FIDO TechNotes: The Truth about Attestation*, https://fidoalliance.org/fido-technotes-the-truth-about-attestation/.

[66] *Certified Authenticator Levels - FIDO Alliance*, https://fidoalliance.org/certification/authenticator-certification-levels.

[67] *Authenticator Level 1 - FIDO Alliance*, https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-1.

[68] *Authenticator Level 1+ - FIDO Alliance*, https://fidoalliance.org/authenticator-level-1/.

[69] *Authenticator Level 2 - FIDO Alliance*, https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-2/.

[70] *Authenticator Level 3 - FIDO Alliance*, https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-3/.

[71] *Authenticator Level 3+ - FIDO Alliance*, https://fidoalliance.org/certification/authenticator-certification-levels/authenticator-level-3-plus/.

[72] *Apple Private PKI*, https://www.apple.com/certificateauthority/private/.

[73] *FIDO AppID and Facet Specification - FIDO Alliance Proposed Standard*, https://fidoalliance.org/specs/common-specs/fido-appid-and-facets-v2.1-ps-20220523.html.

[74] *FIDO UAF Authenticator-Specific Module API - FIDO Alliance*, https://fidoalliance.org/specs/fido-uaf-v1.2-ps-20201020/fido-uaf-asm-api-v1.2-ps-20201020.html.

[75] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, “Upgrading Your Android, Elevating My Malware: Privilege Escalation through Mobile OS Updating,” in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2014.

[76] S. Eskandarian *et al.*, “Fidelius: Protecting User Secrets from Compromised Browsers,” in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[77] *MDN: CredentialsContainer.create()*, https://developer.mozilla.org/en-US/docs/Web/API/CredentialsContainer/create.

[78] *Okta Sign-in Widget*, https://github.com/okta/okta-signin-widget/blob/master/docs/classic.md.

[79] *SAP Customer Data Cloud: accounts.auth.fido.register JS*, https://help.sap.com/docs/SAP_CUSTOMER_DATA_CLOUD/8b8d6fffe113457094a17701f63e3d6a/4594b321af26476ba6156c3dafd8428f.html.

[80] *Use a security key for 2-Step Verification*, https://support.google.com/accounts/answer/6103523.

[81] *PayPal: Two-Factor Authentication*, https://developer.paypal.com/braintree/articles/risk-and-security/control-panel-security/two-factor-authentication.

[82] A. Gavazzi *et al.*, "A Study of Multi-Factor and Risk-Based Authentication Availability," in *USENIX Security Symposium*, 2023.

[83] S. G. Lyastani, M. Backes, and S. Bugiel, "A Systematic Study of the Consistency of Two-Factor Authentication User Journeys on Top-Ranked Websites," in *Network & Distributed System Security Symposium (NDSS)*, 2023.

[84] *Google DevTools: Debug JavaScript*, https://developer.chrome.com/docs/devtools/javascript.

[85] *WebAuthn: Emulate authenticators - Chrome Developers*, https://developer.chrome.com/docs/devtools/webauthn.

[86] *google/openSK*, https://github.com/google/OpenSK.

[87] *FIDO Alliance Metadata Service*, https://fidoalliance.org/metadata/.

[88] *Passkeys*, https://developer.apple.com/passkeys/.

[89] A. D. Forums, *get webauthn attestation statement on Safari*, https://developer.apple.com/forums/thread/713195.

[90] *FIDO Alliance - Enterprise Adoption Best Practices*, https://media.fidoalliance.org/wp-content/uploads/Enterprise_Adoption_Best_Practices_Lifecycle_FIDO_Alliance.pdf.

[91] A. Sudhodanan and A. Paverd, "Pre-hijacked accounts: An Empirical Study of Security Failures in User Account Creation on the Web," in *USENIX Security Symposium*, 2022.

[92] P. Doerfler *et al.*, "Evaluating Login Challenges as a Defense Against Account Takeover," in *The World Wide Web Conference (WWW)*, 2019.

[93] *Android Developers - SafetyNet Attestation API*, https://developer.android.com/training/safetynet/attestation.

[94] H. Zhang, D. She, and Z. Qian, "Android Root and its Providers: A Double-Edged Sword," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[95] A. Kellner, M. Horlboge, K. Rieck, and C. Wressnegger, "False Sense of Security: A Study on the Effectivity of Jailbreak Detection in Banking Apps," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[96] R. Unuchek, *Kaspersky Daily: Rooting your Android*, https://usa.kaspersky.com/blog/android-root-faq/11581/.

[97] Y. Shen, N. Evans, and A. Benameur, "Insights into rooted and non-rooted android mobile devices with behavior analytics," in *ACM Symposium on Applied Computing*, 2016.

[98] *FIDO Biometric Requirements*, https://fidoalliance.org/specs/biometric/requirements.

[99] P. Markert, D. V. Bailey, M. Golla, M. Dürmuth, and A. J. Aviv, "This PIN Can Be Easily Guessed: Analyzing the Security of Smartphone Unlock PINs," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[100] *Registries for Web Authentication - Internet Assigned Numbers Authority*, https://www.iana.org/assignments/webauthn/webauthn.xhtml.

[101] *Intent to Ship: WebAuthn minPinLength extension*, https://groups.google.com/a/chromium.org/g/blink-dev/c/VnXR-U3jROc.

[102] *FIDO Metadata Service - FIDO Alliance Proposed Standard*, https://fidoalliance.org/specs/mds/fido-metadata-service-v3.0-ps-20210518.html.

[103] E. R. Alon Shakevsky and A. Wool, "Trust Dies in Darkness: Shedding Light on Samsung's Trustzone Keymaster Design," in *USENIX Security Symposium*, 2022.

[104] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "Rezone: Disarming trustzone with tee privilege reduction," in *USENIX Security Symposium*, 2022.

[105] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[106] *WebDevAuthn*, https://chrome.google.com/webstore/detail/webdevauthn/aofdjdfdpmfeohecddhgdjfnigggddpd.

[107] L. Abrams, *MFA Fatigue: Hackers' new favorite tactic in high-profile breaches*, https://www.bleepingcomputer.com/news/security/mfa-fatigue-hackers-new-favorite-tactic-in-high-profile-breaches/.

[108] *Guide to push phishing defense and best practices*, https://help.duo.com/s/article/7615.

[109] *FIDO Transaction Confirmation White Paper*, https://media.fidoalliance.org/wp-content/uploads/2020/08/FIDO-Alliance-Transaction-Confirmation-White-Paper-08-18-DM.pdf.

[110] S. Wiefling, L. Lo Iacono, and M. Dürmuth, "Is This Really You? An Empirical Study on Risk-Based Authentication Applied in the Wild," in *ICT Systems Security and Privacy Protection*, 2019.

[111] *Secured Transfer*, https://www.bankofamerica.com/security-center/faq/additional-security-features/.

[112] T. Koide, D. Chiba, and M. Akiyama, "To Get Lost is to Learn the Way: Automatically Collecting Multi-step Social Engineering Attacks on the Web," in *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2020.

[113] *Secure Payment Confirmation*, https://www.w3.org/TR/secure-payment-confirmation/.

[114] J. Prakash *et al.*, "Countering Concurrent Login Attacks in "Just Tap" Push-based Authentication: A Redesign and Usability Evaluations," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[115] *Chromium Issue 1341134*, https://bugs.chromium.org/p/chromium/issues/detail?id=1341134#c18.

[116] *Plan a passwordless authentication deployment in Azure Active Directory*, https://learn.microsoft.com/en-us/azure/active-directory/authentication/howto-authentication-passwordless-deployment.

[117] *FIDO Alliance - Passkeys*, https://fidoalliance.org/passkeys/.

[118] *Touch ID - Attestation*, https://developer.apple.com/forums/thread/708982.

[119] *FIDO Alliance - PSD2 Compliance*, https://fidoalliance.org/psd2-compliance/.

[120] *Web Authentication: An API for accessing Public Key Credentials, Level 1*, https://www.w3.org/TR/webauthn-1/.

[121] *Remove unimplemented extensions*, https://github.com/w3c/webauthn/issues/1386.

[122] D. Freeman, S. Jain, M. Dürmuth, B. Biggio, and G. Giacinto, "Who Are You? A Statistical Approach to Measuring User Authenticity," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[123] T. Whalen *et al.*, "Let The Right One In: Attestation as a Usable CAPTCHA Alternative," in *Symposium on Usable Privacy and Security (SOUPS)*, 2022.

[124] *How can I enable/disable Two-Factor Authentication?* https://www.namecheap.com/support/knowledgebase/article.aspx/9253/45/how-can-i-enabledisable-twofactor-authentication/.

[125] S. Wiefling, M. Dürmuth, and L. Lo Iacono, "What's in Score for Website Users: A Data-Driven Long-Term Study on Risk-Based Authentication Characteristics," in *Financial Cryptography and Data Security (FC)*, 2021.

[126] S. Wiefling, T. Patil, M. Dürmuth, and L. Lo Iacono, "Evaluation of Risk-Based Re-Authentication Methods," in *ICT Systems Security and Privacy Protection*, 2020.

[127] S. Li and Y. Cao, "Who Touched My Browser Fingerprint?: A Large-scale Measurement Study and Classification of Fingerprint Dynamics," in *ACM Internet Measurement Conference (IMC)*, 2020.

[128] M. E. Ahmed, H. Kim, S. Camtepe, and S. Nepal, "Peeler: Profiling Kernel-Level Events to Detect Ransomware," in *European Symposium on Research in Computer Security (ESORICS)*, 2021.

[129] F. Alaca and P. C. Van Oorschot, "Device fingerprinting for augmenting web authentication: classification and analysis of methods," in *Annual Conference on Computer Security Applications (ACSAC)*, 2016.

[130] A. Bates, R. Leonard, H. Pruse, D. Lowd, and K. R. Butler, "Leveraging USB to Establish Host Identity Using Commodity Devices," in *Network and Distributed Systems Security (NDSS)*, 2014.

[131] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[132] *Yup, YouTube Counts Video Ads As Regular Views*, https://techcrunch.com/2011/06/19/youtube-counts-video-ads-regular-views/.

[133] *World's most popular pirated movie site being run from Vietnam: US trade association*, https://e.vnexpress.net/news/news/world-s-most-popular-pirated-movie-site-being-run-from-vietnam-us-trade-association-3722948.html.

[134] *Wikipedia: 123Movies*, https://en.wikipedia.org/wiki/123Movies.

[135] M. Keller, *The Flourishing Business of Fake YouTube Views*, https://www.nytimes.com/interactive/2018/08/11/technology/youtube-fake-view-sellers.html, 2018.

[136] *How engagement metrics are counted*, https://support.google.com/youtube/answer/2991785.

[137] *How can I buy ads from Tube Corporate websites?* http://feedback.tubecorporate.com/en/articles/2864513-how-can-i-buy-ads-from-tube-corporate-websites.

[138] *How to set up In-Video (VAST) ads*, https://support.adspyglass.com/en/articles/4141834-how-to-set-up-in-video-vast-ads.

[139] *mitmproxy*, https://mitmproxy.org/.

[140] *Adspyglass*, https://www.adspyglass.com/.

[141] *Wayback Machine*, https://web.archive.org/.

[142] *YouTube Data API*, https://developers.google.com/youtube/v3.

[143] A. Mathur, A. Narayanan, and M. Chetty, "Endorsements on Social Media: An Empirical Study of Affiliate Marketing Disclosures on YouTube and Pinterest," in *ACM on Human-Computer Interaction*, 2018.

[144] *List of Most Viewed YouTube videos*, https://en.wikipedia.org/wiki/List_of_most-viewed_YouTube_videos.

[145] Wikipedia, *Rickrolling*, https://en.wikipedia.org/wiki/Rickrolling.

[146] YouTube, *Terms of Service*, https://www.youtube.com/t/terms.

[147] *YouTube Partner Program overview*, https://support.google.com/youtube/answer/72851.

[148] *RFC1918: Address Allocation for Private Internets*, https://datatracker.ietf.org/doc/html/rfc1918.

[149] D. Kumar *et al.*, "All Things Considered: An Analysis of IoT Devices on Home Networks," in *USENIX Security Symposium*, 2019.

[150] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[151] SURBL, *URI reputation data*, http://www.surbl.org/lists.

[152] Abuse.ch, *URLhaus*, https://urlhaus.abuse.ch/.

[153] *PhishTank*, http://phishtank.org/.

[154] GitHub, *Puppeteer*, https://github.com/puppeteer/puppeteer.

[155] GitHub, *Selenium*, https://github.com/SeleniumHQ/selenium.

[156] *Navigator.webdriver*, https://developer.mozilla.org/en-US/docs/Web/API/Navigator/webdriver.

[157] Google, *Safe Browsing*, https://safebrowsing.google.com/.

[158] J. Nielsen, *How Long Do Users Stay on Web Pages?* https://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/.

[159] *NetLog: Chrome's network logging system*, https://www.chromium.org/developers/design-documents/network-stack/netlog/.

[160] *ThreatMetrix*, https://risk.lexisnexis.com/products/threatmetrix.

[161] *Service Name and Transport Protocol Port Number Registry*, https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

[162] *Service Name and Transport Protocol Port Number Registry*, https://isc.sans.edu/port.html.

[163] L. Abrams, *List of well-known web sites that port scan their visitors*, https://www.bleepingcomputer.com/news/security/list-of-well-known-web-sites-that-port-scan-their-visitors/.

[164] *K42323285: Overview of the unified Bot Defense profile*, https://support.f5.com/csp/article/K42323285.

[165] J. Wagnon, *Proactive Bot Defense Using BIG-IP ASM*, https://devcentral.f5.com/s/articles/proactive-bot-defense-using-big-ip-asm-25685.

[166] *K33440533: The BIG-IP ASM Bot Defense may erroneously subject clients and web servers to Open Redirection attacks*, https://support.f5.com/csp/article/K33440533.

[167] C. Budd, *Urgent Call to Action: Uninstall QuickTime for Windows Today*, https://blog.trendmicro.com/urgent-call-action-uninstall-quicktime-windows-today/.

[168] F5, *K19556739: Overview of big-ip asm client fingerprinting*, https://support.f5.com/csp/article/K19556739, 2019.

[169] *F5 iApps: Moving Application Delivery Beyond the Network*, https://www.f5.com/services/resources/white-papers/f5-iapps-moving-application-delivery-beyond-the-network.

[170] *nProtect Online Security*, https://nos.nprotect.com/.

[171] *AnySign for PC*, https://hsecure.co.kr.

[172] *Discord*, https://discord.com/.

[173] *Zoom US*, https://zoom.us/.

[174] *BlueJeans*, https://bluejeans.com/.

[175] Wikipedia, *Xunlei*, https://en.wikipedia.org/wiki/Xunlei.

[176] *OWASP-Xenotix-XSS-Exploit-Framework*, https://github.com/ajinabraham/OWASP-Xenotix-XSS-Exploit-Framework.

[177] *LiveReload.js*, https://github.com/livereload/livereload-js.

[178] *SockJS-node*, https://github.com/sockjs/sockjs-node.

[179] R. S. Raman, A. Stoll, J. Dalek, R. Ramesh, W. Scott, and R. Ensafi, “Measuring the Deployment of Network Censorship Filters at Global Scale,” in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[180] *Risk & Business Analytics teach-in*, https://www.relx.com/~/media/Files/R/RELX-Group/documents/presentations/risk-teach-in-8Nov18.pdf.

[181] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupe, “Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting,” in *USENIX Security Symposium*, 2019.

[182] *The Privacy Sandbox*, https://www.chromium.org/Home/chromium-privacy/privacy-sandbox.

[183] C. Trueman, *Pandemic leads to surge in video conferencing app downloads*, https://www.computerworld.com/article/3535800/pandemic-leads-to-surge-in-video-conferencing-app-downloads.html, 2020.

[184] A. Epstein, *The pandemic has turned everyone into gamers*, https://qz.com/1904276/everyone-is-playing-video-games-during-the-pandemic/, 2020.

[185] *The Web Platform Incubator Community Group (WICG)*, https://www.w3.org/community/wicg/.

[186] *Fetch API*, https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[187] *Private Network Access - Draft Community Group Report*, https://wicg.github.io/private-network-access/.

[188] *Private Network Access update: Introducing a deprecation trial*, https://developer.chrome.com/blog/private-network-access-update.

[189] *Localhost Resource Permission*, https://brave.com/privacy-updates/27-localhost-permission.

[190] *Android developers: Web-based content*, https://developer.android.com/develop/ui/views/layout/webapps.

[191] A. Tiwari, J. Prakash, S. Groß, and C. Hammer, "LUDroid: A Large Scale Analysis of Android – Web Hybridization," in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019.

[192] A. Tiwari, J. Prakash, S. Groß, and C. Hammer, "A Large Scale Analysis of Android — Web Hybridization," *Journal of Systems and Software*, 2020.

[193] *Android Developers Blog: Chrome custom tabs smooth the transition between apps and the web*, https://android-developers.googleblog.com/2015/09/chrome-custom-tabs-smooth-transition.html.

[194] *Open a Custom Tab for links in a WebView*, https://developer.chrome.com/docs/android/custom-tabs/howto-custom-tab-from-webview.

[195] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: collecting millions of Android apps for the research community," in *International Conference on Mining Software Repositories*, 2016.

[196] *google-play-scraper*, https://pypi.org/project/google-play-scraper/.

[197] *skylot/jadx*, https://github.com/skylot/jadx.

[198] N. Mauthe, U. Kargén, and N. Shahmehri, "A Large-Scale Empirical Study of Android App Decompilation," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021.

[199] *Javalang*, https://pypi.org/project/javalang/.

[200] *androguard/androguard*, https://github.com/androguard/androguard.

[201] S. Arzt *et al.*, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android app," *ACM SIGPLAN Notices*, 2014.

[202] *Analysis does not finish*, https://github.com/secure-software-engineering/FlowDroid/issues/27.

[203] *App components*, https://developer.android.com/guide/components/fundamentals.html.

[204] *Custom Tabs: Getting started*, https://developer.chrome.com/docs/android/custom-tabs/guide-get-started.

[205] *Create Deep Links to App Content*, https://developer.android.com/training/app-links/deep-linking.

[206] *Naming a Package*, https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html.

[207] *Google Play SDK Index*, https://play.google.com/sdks.

[208] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation," in *USENIX Security Symposium*, 2018.

[209] L. Li *et al.*, "Static analysis of android apps: A systematic literature review," in *Information and Software Technology*, 2017.

[210] Y. Wang, H. Zhang, and A. Rountev, "On the unsoundness of static analysis for Android GUIs," in *ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2016.

[211] S. Dong *et al.*, "Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild," in *Security and Privacy in Communication Networks (SecureComm)*, 2018.

[212] *Frida*, https://frida.re.

[213] A. Aldoseri and D. Oswald, "insecure://Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2022.

[214] X. Zhang *et al.*, "Understanding the (In) Security of Cross-side Face Verification Systems in Mobile Apps: A System Perspective," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2023.

[215] S. Pourali, N. Samarasinghe, and M. Mannan, "Hidden in Plain Sight: Exploring Encrypted Channels in Android apps," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[216] C. Bracco, *Html5 test page*, https://github.com/cbracco/html5-test-page.

[217] *Web APIs: MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Web/API.

[218] *Trace.js*, https://gist.github.com/nicoandmee/62ecd1829d761fbed779dc3a3ba35c64.

[219] *LineageOS Android Distribution*, https://lineageos.org/.

[220] D. Kuchhal and F. Li, "Knock and talk: Investigating local network communications on websites," in *ACM Internet Measurement Conference (IMC)*, 2021.

[221] K. Ruth, D. Kumar, B. Wang, L. Valenta, and Z. Durumeric, "Toppling top lists: Evaluating the accuracy of popular website lists," in *ACM Internet Measurement Conference (IMC)*, 2022.

[222] *System WebView Shell*, https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/webview-shell.md.

[223] *UI/Application Exerciser Monkey*, https://developer.android.com/studio/test/other-testing-tools/monkey.

[224] *Ad units, ad formats, & ad types*, https://support.google.com/admob/answer/6128738.

[225] *Integrate the WebView API for Ads*, https://developers.google.com/admob/android/webview.

[226] M. Backes, S. Bugiel, and E. Derr, “Reliable Third-Party Library Detection in Android and its Security Applications,” in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[227] S. Son, D. Kim, and V. Shmatikov, “What Mobile Ads Know About Mobile Users,” in *Network and Distributed System Security (NDSS)*, 2016.

[228] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, “AdDroid: privilege separation for applications and advertisers in Android,” in *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2012.

[229] T. Liu *et al.*, “MadDroid: Characterizing and Detecting Devious Ad Contents for Android Apps,” in *The Web Conference (WWW)*, 2020.

[230] G. Chen, W. Meng, and J. Copeland, “Revisiting Mobile Advertising Threats with MAdLife,” in *The World Wide Web Conference (WWW)*, 2019.

[231] S. Shekhar, M. Dietz, and D. S. Wallach, “AdSplit: Separating Smartphone Advertising from Applications,” in *USENIX Security Symposium*, 2012.

[232] *Open Measurement SDK*, https://iabtechlab.com/standards/open-measurement-sdk/.

[233] *Measuring User Engagement*, https://developer.chrome.com/docs/android/custom-tabs/guide-engagement-signals.

[234] *Flutter*, https://flutter.dev.

[235] *url_launcher*, https://pub.dev/packages/url_launcher.

[236] *flutter_inappwebview*, https://pub.dev/packages/flutter_inappwebview.

[237] *android-customtabs*, https://github.com/saschpe/android-customtabs.

[238] S. Y. Mahmud, K. V. English, S. Thorn, W. Enck, A. Oest, and M. Saad, “Analysis of Payment Service Provider SDKs in Android,” in *Annual Computer Security Applications Conference (ACSAC)*, 2022.

[239] *owasp-masvs*, https://github.com/OWASP/owasp-masvs.

[240] *Securing WebViews with Chrome Custom Tabs*, https://plaid.com/blog/securing-webviews-with-chrome-custom-tabs.

[241] *juspay: AmazonPay S2S Tokenised Flow*, https://developer.juspay.in/v5.1/docs/amazonpay-s2s-tokenised-flow.

[242] *Deprecating support for FB Login authentication on Android embedded browsers*, https://developers.facebook.com/blog/post/2021/06/28/deprecating-support-fb-login-authentication-android-embedded-browsers/.

[243] *NidOAuthBehavior.kt*, https://github.com/naver/naveridlogin-sdk-android/blob/master/Nid-OAuth/src/main/java/com/navercorp/nid/oauth/NidOAuthBehavior.kt.

[244] *Oauth 2.0 for native apps*, https://datatracker.ietf.org/doc/html/rfc8252.

[245] *Google Groups: Clarifying WebView and Chrome Custom Tabs support for Android's implementation of Passkeys*, https://groups.google.com/a/fidoalliance.org/g/fido-dev/c/SWuq7ORnnLQ.

[246] *Firebase Android SDK Release Notes*, https://firebase.google.com/support/release-notes/android#2020-11-12.

[247] *Web links*, https://developer.android.com/training/app-links#web-links.

[248] *View logs with Logcat*, https://developer.android.com/studio/debug/logcat.

[249] *Remote debug Android devices*, https://developer.chrome.com/docs/devtools/remote-debugging/.

[250] R. Duan, W. Wang, and W. Lee, "Cloaker catcher: A Client-based Cloaking Detection System," in *arXiv preprint arXiv:1710.01387*, 2017.

[251] *Meta Pay*, https://pay.facebook.com.

[252] *About checkout on Facebook and Instagram Shops*, https://www.facebook.com/business/help/2509359009104717?id=533228987210412.

[253] M. Koop, E. Tews, and S. Katzenbeisser, "In-depth Evaluation of Redirect Tracking and Link Usage," in *Privacy Enhancing Technologies (PETs)*, 2020.

[254] *Radar*, https://docs.netscaler.com/en-us/citrix-intelligent-traffic-management/radar.html.

[255] *AndroidRadar*, https://github.com/cedexis/androidradar.

[256] P. Vallina *et al.*, "Mis-shapes, Mistakes, Misfits: An Analysis of Domain Classification Services," in *ACM Internet Measurement Conference (IMC)*, 2020.

[257] H. Qayyum *et al.*, "A First Look at Android Apps' Third-Party Resources Loading," in *Network and System Security (NSS)*, 2022.

[258] *Partial Custom Tabs*, https://developer.chrome.com/blog/whats-new-in-web-on-android-io2023/#partial-custom-tabs.

[259] *How to Disable In-App Browser for Android Apps*, https://gadgetstouse.com/blog/2019/12/21/how-to-disable-in-app-browser-for-android-apps/.

[260] K. Kollnig, A. Shuba, M. Van Kleek, R. Binns, and N. Shadbolt, "Goodbye Tracking? Impact of iOS App Tracking Transparency and Privacy Labels," in *ACM Conference on Fairness, Accountability, and Transparency*, 2022.

[261] H. J. Hutton and D. A. Ellis, "Exploring User Motivations Behind iOS App Tracking Transparency Decisions," in *Conference on Human Factors in Computing Systems (CHI)*, 2023.