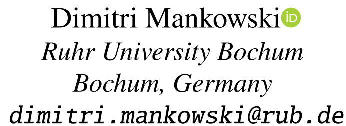
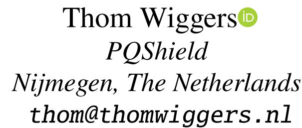
TLS → Post-Quantum TLS: Inspecting the TLS landscape for PQC adoption on Android


Dimitri Mankowski
Ruhr University Bochum
Bochum, Germany
dimitri.mankowski@rub.de


Thom Wiggers
PQShield
Nijmegen, The Netherlands
thom@thomwiggers.nl

Veelasha Moonsamy
Ruhr University Bochum
Bochum, Germany
email@veelasha.org

***Abstract*—The ubiquitous use of smartphones has contributed to more and more users conducting their online browsing activities through apps, rather than web browsers. In order to provide a seamless browsing experience to the users, apps rely on a variety of HTTP-based APIs and third-party libraries, and make use of the TLS protocol to secure the underlying communication. With NIST's recent announcement of the first standards for post-quantum algorithms, there is a need to better understand the constraints and requirements of TLS usage by Android apps in order to make an informed decision for migration to the post-quantum world. In this paper, we performed an analysis of TLS usage by highest-ranked apps from Google Play Store to assess the resulting overhead for adoption of post-quantum algorithms. Our results show that apps set up large numbers of TLS connections with a median of 94, often to the same hosts. At the same time, many apps make little use of resumption to reduce the overhead of the TLS handshake. This will greatly magnify the impact of the transition to post-quantum cryptography, and we make recommendations for developers, server operators and the mobile operating systems to invest in making more use of these mitigating features or improving their accessibility. Finally, we briefly discuss how alternative proposals for post-quantum TLS handshakes might reduce the overhead.**

1. Introduction

As the adoption rate of smartphones steadily grew over the years, users have developed an innate preference for conducting their online activities, such as emails and banking, via mobile apps rather than on desktop or mobile browsers [25, 28]. Much like web applications, Android apps also make increasing use of many services offered over the internet, usually through HTTP APIs. The amount of traffic sent from apps ranges from application content, such as posts in social networking apps, to metadata such as usage analytics, logs, and crash reports. Apps often allow signing in through providers such as Google, Facebook or Apple, which is generally implemented by including the vendor's SDK libraries in the app. Many mobile apps also include advertising frameworks, which share user profiles with advertisers and download and show appropriate ads. Moreover, libraries, SDKs, and frameworks often include their own analytics on top of what the app developer is using directly. Any software library may transitively pull in more or distinct versions of libraries through dependencies. As a result, apps have the possibility to start up many components on launch, which may connect to one or more services over the time that the app is running. To protect users' privacy and security, this data is sent over the network in encrypted form, using the Transport Layer Security (TLS) protocol. Naively, a secure connection needs to be set up for each of these connections, which results in overhead. The TLS handshake uses ephemeral key exchange and authentication data to set up the secure channel, and without optimizations, this involves transmitting the data each time.

Currently, the cryptography commonly used in TLS is fairly efficient, both in terms of bandwidth and in computational requirements, so the overhead is manageable. A typical elliptic-curve-based (ECC) key exchange with RSA (Rivest–Shamir–Adleman) authentication uses about 1000 bytes of data. However, to protect against attacks breaking ECC and RSA using quantum algorithms, we need to transition to post-quantum cryptography, as already demonstrated by the work of Sikeridis et al. [45]. The US National Institutes of Standards and Technology (NIST) is currently standardizing such quantum-secure algorithms for key exchange and digital signatures [30]. In July 2022, they selected the first standards for post-quantum key exchange (Kyber [41]) and digital signatures (Dilithium [27], Falcon [34] and SPHINCS$^+$ [23]). These algorithms have much larger bandwidth requirements than ECC or RSA: a Kyber-based key exchange requires at least 1568 bytes for its public key and ciphertext, while a Dilithium public key and signature together take 3732 bytes. As TLS' public key infrastructure relies on many signatures, this may result in over 17 kB of additional handshake data if Dilithium is used for all of them [50]. Knowing how the TLS protocol is used by Android apps can inform the discussion about the standardization of post-quantum TLS or alternative secure transport protocols by organizations such as the TLS working group at the Internet Engineering Task Force (IETF).

There exist several prior work that have already made an attempt at better understanding the implications for the TLS protocol when faced with migration to the post-quantum world. Of particular interest is the work by Sikeridis et al. [45] in which the authors assessed the additional overhead of post-quantum cryptography in TLS 1.3 and SSH. More specifically, the authors studied the integration of post-quantum key encapsulation mechanisms (KEMs) and post-quantum authentication, and presented an overview of the protocol performance when both are used concurrently. Another closely relevant work is that of Tasopoulos et

This work was completed while Thom Wiggers was at Radboud University, Nijmegen, The Netherlands.

al. [47], where the authors evaluated the performance of post-quantum TLS on resource-constrained embedded systems. In their empirical analysis, they focused on modifying the existing TLS 1.3 architecture to incorporate the post-quantum algorithms being considered by NIST, and evaluated the execution time, memory and bandwidth requirements. It should be noted that none of the existing work so far have studied the consequences of migration to post-quantum TLS for Android apps.

In this paper, our goal is to address the gap in the current literature by investigating the use of the TLS protocol and its associated features by Android apps in order to provide recommendations for migrating to post-quantum TLS. We do so by studying the most popular Android apps in the Google Play Store to establish a minimal baseline of the TLS connections for most top-ranked apps. We focus on the Games and general-purpose app categories, and collected the top 45 Android apps in each category, as of January 2023. Each app was executed on a real Android device and the resulting network traffic was collected. We then extracted relevant TLS features, such as client handshakes, session duration, and resumptions. These metrics show how app developers use TLS and if features that reduce its overhead are actually used. Finally, we estimate the impact of the upcoming migration to post-quantum algorithms would have on the overhead induced by how apps use TLS. The amount of data required for a TLS handshake will increase by a factor of more than 8; for specific examples of Apps and Games, this increases the data that is transmitted solely as TLS handshake overhead from approximately 400 kB to 3 MB. We make the following contributions:

- We implemented a customized scraper to extract applications from the Google Play Store and built a pipeline to download and run apps on a real device in order to capture network traffic.
- Based on our experimental evaluation, we presented insights into how the TLS protocol is used in practice and how TLS features are implemented by the top-ranked apps.
- We provide recommendations for how to improve TLS configurations on both the client and the server-side, and prepare for the impact of the migration to post-quantum TLS.
- To help reproducibility of our results, we make our dataset of network traffic records for all collected Android apps and the customized crawler publicly available[1].

The remainder of the paper is organized as follows: in Section 2, we provide background knowledge on TLS and how it is implemented in Android apps. We also present an overview of the related work. We explain our experimental setup and methodology in Section 3, followed by presentation of our results in Section 4. We discuss our results and present recommendations in Section 5, and conclude in Section 6.

1. https://zenodo.org/record/7950522

2. Background & Related Work

2.1. Transport Layer Security

The Transport Layer Security (TLS) protocol, invented as SSL by browser developer Netscape in 1995 [14], is perhaps the most-used cryptographic protocol. An ever-increasing number of websites and other services use TLS to protect traffic from eavesdroppers on the network. TLS is not just used for websites, which are using the HTTP protocol; it is used for many applications, ranging from VPNs [33], to secure file transfer [16] and email [21, 31]. Even offline, TLS can be used, for example, to secure networks inside of cars [51].

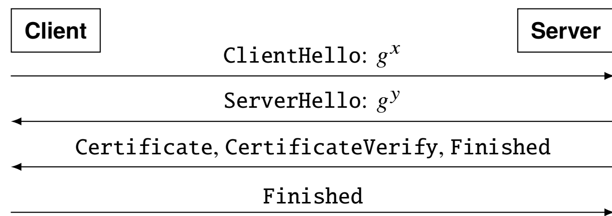


Figure 1. Overview of the TLS 1.3 handshake

TLS allows a client, such as a web browser or Android app, to connect to a server, such as the one hosting a website or API service. We give a sketch of the TLS handshake protocol in Fig. 1. The server is identified by a *certificate*, which is a statement of the server's identity, signed by a certificate authority. Optionally, TLS also supports identification of the client.

TLS has seen a lot of development since its initial version. Many extensions and additional features were developed, but the principle behind the most common mode has mainly stayed the same. The latest version of the protocol is TLS 1.3, which was released in 2018 by the standardization organization IETF as RFC 8446 [37], though TLS 1.2 [38] is also still widely used. As it is not uncommon to connect to the same server many times over time, *resumption* mechanisms have been added to the protocol. These allow a TLS client to resume a prior connection without exchanging the certificates, which saves a significant amount of computation and handshake traffic. We show a sketch of the TLS 1.3 handshake with resumption in Fig. 2. TLS 1.3 even allows the client to submit *early data* while a connection is being resumed, which results in a 0-round trip (0-RTT) handshake: the request is sent off before the server has completed the handshake. This further reduces the overhead. Another improvement in TLS 1.3 is the use of handshake encryption, which offers additional privacy to the client and server.

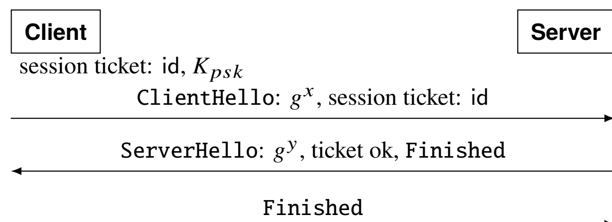


Figure 2. Overview of TLS 1.3 resumption

Traditionally, TLS is run on top of the TCP transport protocol. To get around the TCP SYN/ACK handshake, which adds head-of-line blocking to connection establishment, the QUIC protocol was developed [24]. QUIC is a reliable transport protocol, like TCP, but it runs over UDP and uses TLS 1.3's handshake protocol to encrypt all traffic. It also offers more advanced features such as connection multiplexing and quickly switching underlying connections between WiFi and mobile networks. Like TLS 1.3, QUIC offers 0-RTT connection resumption. HTTP/3, the latest version of the HTTP protocol, runs over QUIC [10].

2.2. TLS in Android

Just as how browser developers have pushed for increased use of TLS on the web (see, e.g., [40, 49]), both Google and Apple have increasingly been pushing for increased use of TLS to secure the traffic that apps send out. Similar to browsers, most connections are using the HTTP protocol, this is usually synonymous with using HTTPS (HTTP over TLS) [36]. Since version 9, Android's default HTTPS implementation does not allow unencrypted HTTP traffic by default (though it can be re-enabled by developers); on Apple's iOS version 9.0 and above, HTTP is similarly disabled by default, and re-enabling it requires justification in App Store reviews [6].

Performing HTTPS requests is natively supported by the Android standard library. The documentation shows that performing an HTTPS request can be done with a few lines of code [18]. We conjecture that most developers and libraries likely use this API, which is also confirmed by the observations made in [32]. There also exist third-party libraries, such as Block's OkHTTP [11], which offers additional functionality like HTTP/2. Google itself also offers Cronet, which provides Chromium's network stack to apps as a networking library and supports HTTP/2, and HTTP/3 over QUIC [17]. Using these libraries does require adding external dependencies to apps, and their implementation might require more work.

2.3. Related work

In this section, we provide a brief overview of the relevant literature related to the usage and applications of TLS in the Android ecosystem.

The investigation conducted by Razaghpanah et al. [35] was one of the firsts to provide insightful observations on how TLS is used in Android apps. In their holistic and large-scale study, the authors crowd-sourced network traffic by leveraging the Lumen Privacy Monitor, a free Android app which has the capability of collecting data from normal user-app interactions, map network flows to apps and collect data related to TLS handshake. Of particular interest to our work, the authors uncovered diverse ways in which apps use default and third-party TLS libraries, including ones that have security vulnerabilities and weaknesses, such as weak cipher suites, in their implementations.

A similar observation was also made by Kotzias et al. [26], whereby the authors conducted a large-scale, longitudinal study to examine the evolution of the TLS ecosystem during the period of 2012-2018. In one of their main findings, they reported that while browsers are quick to adopt new algorithms (in response to known TLS-based attacks), they are, however, slow to drop support for older, insecure ones. More recently, extending the work of Holz et al. [22], Birghan and van der Merwe [9] presented further insights into the deployment of TLS 1.3 in Firefox. Leveraging telemetry data provided by Mozilla, they reported on the TLS 1.3 adoption rate, cipher suite usage, 0-RTT usage and the uptake of post-handshake authentication mechanism. Their findings further consolidate the main motivation behind the goal of our work, which is that of, preparing the TLS 1.3 landscape for migration to post-quantum TLS.

While much can be learned from understanding how TLS is adopted in order to mitigate attacks, there are also concerns surrounding provision of quality of service by network administrators. Given the diverse app ecosystem of the Android landscape, it is challenging to properly administer adequate bandwidth since the traffic generated by the plethora of apps is largely diverse. The work of Sengupta et al. [44] highlight the hurdles faced by app developers when implementing TLS and configurating complicated end-to-end architecture with bare minimum domain expertise, thus resulting in bloated apps with a wide array of cipher suites, key-sizes, and certificate management styles. Such insights provide further crucial information to help decide how to migrate Android apps to post-quantum TLS.

Along the same vein, in 2012, Fahl et al. [15] proposed MalloDroid to help identify and mitigate vulnerable TLS code which could lead to man-in-the-middle attacks in Android apps. More importantly, their work also focused on the usability aspect related to users' ability to understand the meaning of warning messages and what are the safest behaviors to adopt. Incidentally, their online survey revealed that users rarely know what is the best and safest recourse to take when faced with security warning messages related to insecure connections. In their follow-up study published in 2021 [32], the authors performed an evaluation of the then-existing measures provided by Google for performing TLS certificate validation in Android apps. The results of the study confirmed that developers still struggle to perform certificate validation, despite the additional documentation provided by Google. This shows that there is no guarantee that official developer guidelines are being properly adhered to by app implementers in real-life.

3. Method

In this section, we first describe and justify the collected data. Then we present the experimental setup with technical details.

3.1. Dataset collection

The aim of our study is to investigate how TLS is used in a representative collection of applications and for this reason, we focus primarily on the most popular Android apps. We opted to use the top-ranked applications from Google Play, which is the largest app marketplace for the Android operating system. Google Play [19] offers a wide range of applications, divided into five primary categories, namely Games, Apps, Movies & TV, Books, and Kids. The categories Games and Apps provide the Top charts sub-categories, which rank 45 applications heavily influenced by popularity [5]. In the following, we refer to

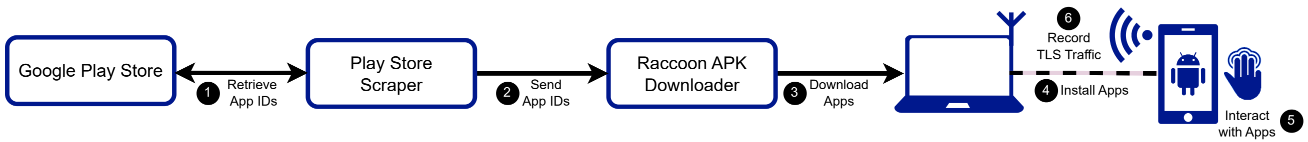


Figure 3. Experiment pipeline

applications from the App category as Apps and from the Game category as Games. When we use the term "applications" or "apps" without specifying a category, we are referring to applications from both categories. We collected free applications from both Top charts rankings on 23 January 2023, resulting in a total of 90 applications, with downloads ranging from 100K+ to 5B+, with a median of 10M+. Given that the rankings vary from one country to another and there is no global top ranking, we limited our selection to the German market. The selected applications are listed in Table 6 for Apps and Table 7 for Games (both in Appendix A), including the app IDs and names as listed in the Google Play Store. We executed the apps on an Android phone with Android 11 and recorded the TLS traffic between January 30th and February 1st, except for the Game `Stumble Guys` because it required an update and was re-downloaded on February 1st. We captured the traffic using Tshark [48] and stored it in the pcap [20] package capture file format. When parsing the pcaps, we extracted the following features:

- Number of TLS client handshakes
- Number of TLS resumptions
- Information about each TLS connection
  - Protocol (TLS/QUIC)
  - TLS version
  - Traffic in MB
  - Session time
  - Destination host IP address and server name
  - Resumption attempt
  - Resumption acceptance

3.2. Experiment setup

Our experiment pipeline involved several steps, which are shown in Figure 3.

In step 1, we retrieved the app IDs which are unique identifiers assigned by Google Play to manage and distribute the applications. When accessing the Google Play website, it does not show all the applications, but gradually loads them while scrolling. For this reason we implemented a Google Play Scraper using Selenium [12]. While the scraper is able to get to the app IDs of all main categories and subcategories by scrolling through the entire page and all subcategories, for our experiment, we only collected the app IDs of the Top charts subcategories, as described earlier. We sent the app IDs to the Raccoon APK Downloader [1] in step 2, in order to download the apps in step 3. Raccoon is able to download the applications directly from Google Play to a laptop. In step 4, we installed the apps on the Nokia G11 Android phone connected to a laptop via cable to set up an Android Debug Bridge [4] and via a hotspot to intercept the traffic (step 6). In step 5, we interacted with each application for five minutes, attempting to simulate normal human interaction. We repeated steps 4, 5 and 6 for each app and deleted the previous app before installing a new one. We started intercepting the traffic immediately after running the application and stopped before uninstalling it, thus creating a separate pcap file for each application. We selected the recording duration of 5 minutes based on a pre-experiment where we recorded the traffic of 10 apps, which showed no significant changes in their TLS traffic after 5 minutes.

One limitation of our experimental setup is that the traffic is not decrypted, which does not allow us to examine the exchanged data content. One problem that arises from this is that we cannot distinguish the traffic of the target app from the background traffic generated by the Android operating system. Although the background noise is not directly related to the tested applications, it may still be recorded in our measurements. To verify this, we conducted a 60-minute recording of traffic while interacting with the native Android calculator application, during which we observed a total of 44 handshakes. On average, we noted 4 handshakes in each 5-minute phase, with a median of 3. The 5-minute phase with the lowest number of handshakes recorded was 0, while the maximum number of TLS handshakes sent in a 5-minute phase was 12. Overall, we believe that the number of handshakes recorded is relatively low when compared to the amount typically generated by apps found on Google Play. Therefore, we are confident that the impact of background traffic on our results is negligible.

We have automated the processing of the captured traffic using a combination of Tshark and Python scripts. Tshark is used to filter out all packets containing TLS fragments, and to convert the pcap file to json. In Python, we loaded this json file and used the Tshark-provided identifiers of unique TCP (for TLS) and UDP (for QUIC) streams to reconstruct individual packets into sessions. Finally, we dissected these sessions, e.g. by extracting the server name from the ClientHello message or checking the ClientHello and ServerHello TLS messages for extensions that are used in session resumption.

4. Results

During our experiment, we recorded various statistics and obtained a number of interesting results. In this section we present our findings under multiple aspects.

The implementation of TLS can impact application performance and have implications for network traffic utilization. To assess the current state of TLS usage in popular Android apps and evaluate their adherence to best practices, we have gathered statistics affecting performance which are discussed in the following.

During a TLS handshake, the client and server exchange multiple messages, which adds additional Round Trip Time (RTT) overhead. The easiest way to reduce latency would be to reduce the **number of TLS connections**. However, this measure would require to decide which connections are unimportant enough to be removed and goes beyond the TLS

implementation. If the client sends multiple handshakes to the same servers, TLS session **resumptions** can reduce the overhead. These allow the client and server to re-establish a connection without performing a full TLS handshake. Compared to TLS 1.2, **TLS 1.3** reduces the number of round trips by combining the handshake and session resumption into a single round trip. In addition, TLS 1.3 introduces the zero round trip mode, which allows the client to send data immediately after the first handshake. The **QUIC** protocol which uses the TLS 1.3 protocol combines the connection setup and encryption handshake into a single step. This allows to reduce the number of round trips required to establish a connection even further. Opening TLS sessions for a longer period of time without terminating them could also reduce the number of TLS handshakes, which is why we measure the **duration of each session**. Finally, it is important to avoid redundant traffic to reduce the impact on network consumption. Although encryption prevents us from viewing the exact data being transmitted, the **volume of data** sent during each TLS session provides valuable insights into data consumption.

Table 1 highlights the differences between applications from the Apps category and those from the Game category. As evident from the table, Games exhibit higher average activity in almost all aspects. The lower activity of Apps, on the other hand, can be attributed to their commercial nature, such as shopping apps, companion apps, or banking apps. These Apps are designed to serve a specific purpose and do not require constant data transmission. In contrast, Games generate revenue through advertising and data collection, leading to a higher volume of data being sent to various servers. This could explain the observed difference in activity levels between the two categories. In the Tables 2 and 3,

Table 1. Comparison of Apps and Games

		Apps	Games	All
Handshakes	Mean	86	203	144
	Median	57	135	94
Resumptions	Mean	18	54	36
	Median	11	20	14
Servers	Mean	32	58	45
	Median	25	60	37
Traffic in MB	Mean	9.5	17.7	13.6
	Median	3.2	9	6.2
Session Time in secs	Mean	4.5	3.7	4.1
	Median	1.1	2.3	1.8
TLS 1.3 usage in %	Mean	73	58	66
	Median	77	67	69
QUIC handshakes	Mean	10	11	11
	Median	10	8	9

six noteworthy applications from each of the respective categories are listed together with their performance-related statistics. The Resumptions column shows a subset of all handshakes from the Handshakes column. The Servers column indicates the number of unique server names. Tables 6 and 7 in Appendix A provide a full description of all the applications and their respective statistics.

Handshakes & Resumptions. With 917 handshakes, the game `Candy Crush Saga` is by far the most active in this regard. However, it connects to the same server most of the time and uses resumptions more consistently than any

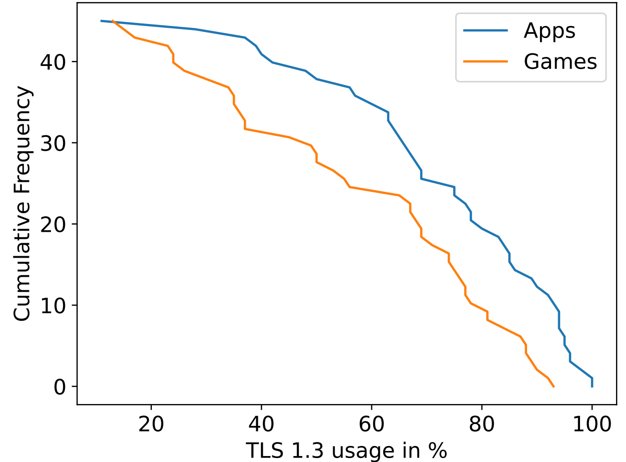


Figure 4. Cumulative distribution of TLS 1.3 usage

other application. A similar behavior regarding the ratio of resumptions to handshakes is only shown by the Game `Farm Heroes Saga`, published by the same company as Candy Crush (King.com). All other applications make less use of resumptions. Interestingly, the app `Haircut prank, air horn & fart`, which sends the most handshakes of all applications in the Apps category, could also be classified as Games in Google Play due to its functionality.

TLS 1.3. We observe that all applications use either TLS 1.2 or TLS 1.3. Figure 4 shows the cumulative distribution of TLS 1.3 usage for Apps and Games. As can also be seen from Table 1, Apps rely more on TLS 1.3 than Games. In our experiment we found that 13 Apps use TLS 1.3 in more than 90% of their handshakes, whereas only 2 applications from Games do so. However, there are also negative outliers in both categories. For example, the App `Klarna | Shop now. Pay later` only uses TLS 1.3 in 11% of 56 handshakes. Among the Games, `Woodoku - Block Puzzle Games` is the application that uses the least TLS 1.3 handshakes, with only 13%.

QUIC. The QUIC protocol is used by Apps and Games to an equal extent, with the most frequently accessed servers being provided by various Google and Facebook services. For the most part, however, QUIC is not used extensively. For instance, the App `Joyn | deine Streaming App`, which boasts the highest number of QUIC handshakes in its category, only sends 34 QUIC handshakes out of a total of 317 overall handshakes. Similarly, `Lighter Simulation`, the Game that uses QUIC the most, only sends 39 out of 290 handshakes using QUIC.

Data Volume. Most data is exchanged by `Stormshot: Isle of Adventure` and `Instagram` for the Apps category. Instagram's data usage of 124.7 MB can be largely explained by the high volume of video content being streamed through the App. The second and third places with the most data traffic in the Apps category are also occupied by the video streaming Apps `TikTok` and `Snapchat`. The reason behind the even greater amount of data exchanged by `Stormshot` remains unclear to us. We speculate that the application is making use of various third-party libraries and analytics services, which would be a probable cause of the high traffic load.

Table 2. Notable applications from Apps category with their statistics

App Title App ID	Handshakes	Resumptions	Servers	TLS 1.3	Time (s)	Traffic (MB)	QUIC handshakes
Haircut prank, air horn & fart com.pranksounds.appglobaltd	430	110	100	77%	2.2	27.0	18
Instagram com.instagram.android	51	7	23	94%	0.7	124.7	16
Klarna | Shop now. Pay later. com.myklarnamobile	56	5	26	11%	1.2	2.3	1
Dezor net.dezor.browser	21	4	14	95%	32.7	0.2	0
Joyn | deine Streaming App de.prosiebensat1digital.seventv	317	71	110	69%	1.1	20.0	34
S-pushTAN com.starfinanz.mobile.android.pushtan	4	3	3	100%	0.4	0.0	0

Table 3. Notable applications from Games category with their statistics

App Title App ID	Handshakes	Resumptions	Servers	TLS 1.3	Time (s)	Traffic (MB)	QUIC handshakes
Candy Crush Saga com.king.candycrushsaga	917	900	9	93%	0.3	5.3	0
Stormshot: Isle of Adventure com.sivona.stormshot.e	41	7	21	37%	9.1	182.1	0
Woodoku - Block Puzzle Games com.tripledot.woodoku	171	24	79	13%	3.1	3.3	1
Royal Match com.dreamgames.royalmatch	23	3	16	30%	29.3	7.0	1
Lighter Simulation com.smoke.lighter.simulator	290	33	83	90%	3.0	24.4	39
Lords Mobile: Kingdom Wars com.igg.android.lordsmobile	9	2	8	89%	0.9	47.3	2

Session Duration. In terms of session duration, there are no major differences between the two categories. While most applications keep connection times very short, some have significant outliers. In particular, in the Apps category, three applications maintain their connection for a median of more than 30 seconds, which is a lot considering that the overall median for the app category is only 1.1 seconds. Games show less variance in this regard.

Applications with Little Activity. Several apps have very limited activity in terms of handshakes. This is often due to special requirements that prevented us from fully interacting with them. For example, messaging Apps like `WhatsApp Messenger` and `Telegram` require users to provide a phone number, but we were unable to do so as our test phone did not have a SIM card. Similarly, `Samsung Smart Switch Mobile` is only compatible with Samsung phones, which limits its usage to a specific set of devices. `S-pushTAN`, which has the lowest amount of handshakes, is a banking App that requires users to have a banking account in order to use it. We also encountered technical difficulties with `Lords Mobile: Kingdom Wars`, as the Game failed to launch properly and got stuck on a black loading screen. Despite only sending 9 handshakes, the Game managed to transfer a surprisingly large amount of data, i.e. 47.3 MB.

Handshakes over Time. Figures 5 (a)–(f) show timing diagrams depicting the number of client hello messages, resumptions and repeatedly accessed servers over the course of the interaction with the apps. The majority of apps show their main activity at the beginning such as `Amazon` 5(c), `Disney+` 5(e) and `Farm Heroes Saga` 5(f) or after an initial user interaction such as a registration in `Roblox` 5(b). `Candy Crush Saga` 5(d) shows a very untypical behavior, as it sends a vast number of handshakes in the first approx. 100 seconds and then stops abruptly. The figures represent different behavioral patterns that lead to different overhead costs. The App `OTTO - Shopping & Möbel` 5(a) and the Game `Roblox` exhibit the most inefficient behaviour by sending a high number of handshakes frequently to previously accessed servers, almost without using resumptions. The App `Amazon` initiates fewer handshakes and uses resumptions for about half of the repeatedly accessed servers. The Game `Candy Crush Saga` resumes almost all connections, however the amount of handshakes seems excessive. `Farm Heroes Saga` also resumes most of the handshakes but makes significantly less connections than `Candy Crush Saga`. `Disney+` seems to be the most efficient application in this regard, as it generates the least number of handshakes and consistently resumes connections to servers that are repeatedly accessed.

5. Discussion

In this section, we will summarize and reflect on our results, before we show the large impact migrating to post-quantum secure algorithms would have on the overhead in TLS handshakes. We also suggest some general approaches that can help improve the performance and reduce the overhead of performing full TLS handshakes for every HTTPS request, which mitigate the impact of the post-quantum transition, but may also improve the performance and data usage of apps today. Finally, we will briefly discuss alternative handshake protocols that have been proposed for post-quantum TLS, and which may be especially applicable to mobile applications.

5.1. How apps use TLS

In our analysis, we have discovered that the adoption of new TLS standards among Android applications has been slow. Despite the introduction of TLS 1.3 as the new standard in 2018, and being supported since Android 10 (2019), it is only used in 66 % of connections. This is only slightly more than the amount of TLS 1.3 connections in web browsers in November 2021 [9]. We also see that, on median, only 31%

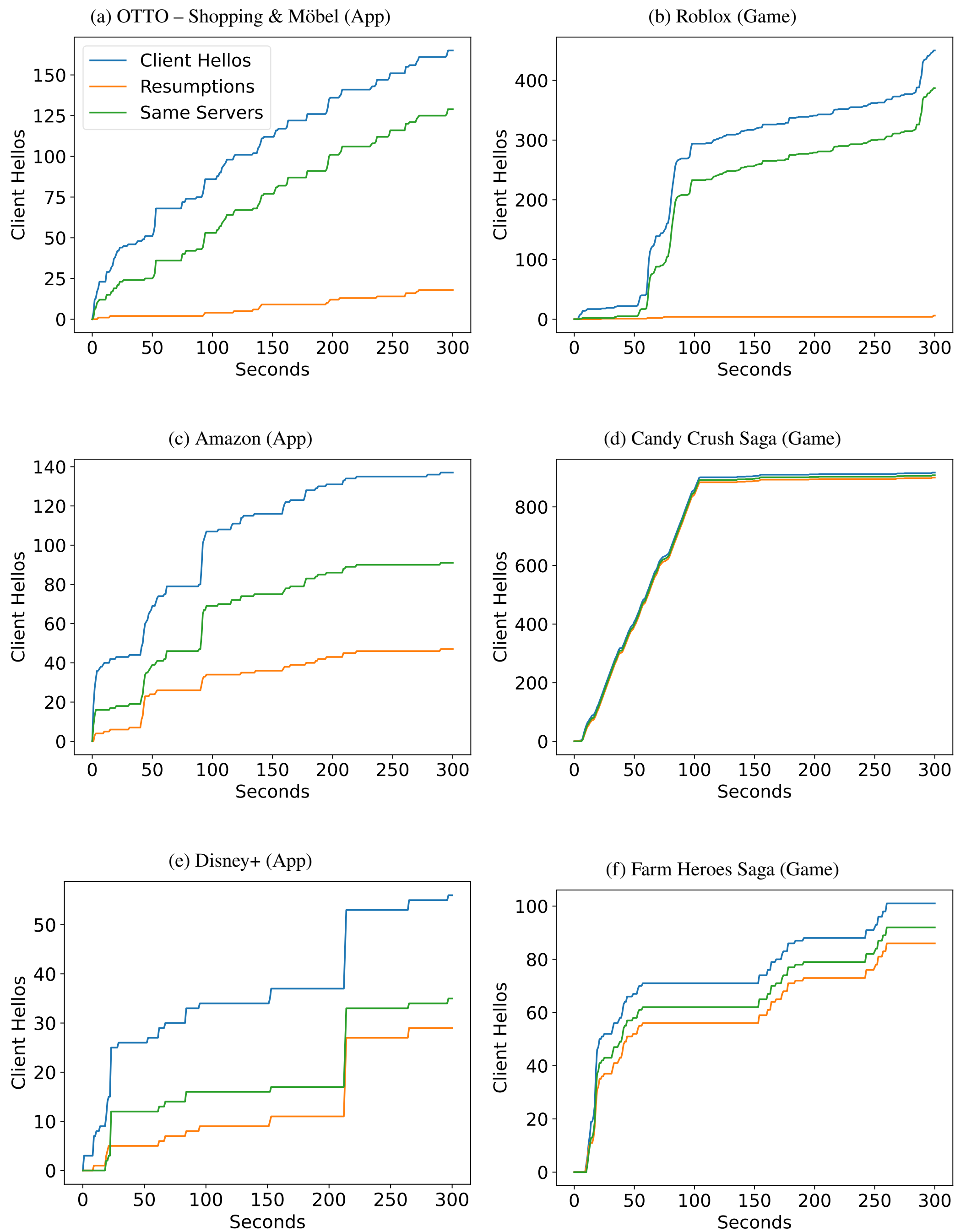


Figure 5. Timing diagrams for interesting apps showing handshakes, resumptions and repeatedly accessed servers

of connections to the same host use resumptions and that the median session length is only 1.1 seconds for Apps and 2.3 seconds for Games.

The use of the more advanced features such as TLS session resumption or use of the QUIC protocol remains low and is primarily limited to servers owned by major corporations like Google and Facebook. This may be in part due to popular web servers, such as NGINX, not yet natively supporting QUIC or HTTP/3. Nevertheless, Games such as `Candy Crush Saga` or Apps like `Joyn` do make use of these features. In the case of `Candy Crush Saga`, the 900 resumed handshakes would each have included two RSA-2048-based certificates and a handshake signature, adding up to 1.13 MiB, roughly 20 %, of additional data. These savings clearly illustrate that, even though these features may not be used often, their development and the effort spent on implementing them by the IETF and TLS library developers has been worthwhile.

Still, the main theme in our results has been the high number of handshakes, which appears to be excessive in some cases. Our findings show that some Games, in particular, generate a significant number of connections, with only a fraction of them being resumed. This suggests that the focus of developers is largely not on network optimization and that there is still room for improvement in optimizing connection handling in some applications. In Section 5.3, we will offer

some suggestions to both app developers and the Android ecosystem to improve this situation.

5.2. The impact of post-quantum cryptography

The large number of non-resumed connections that many apps set up result in many full key exchanges and transmissions of full certificate chains. If this is not mitigated (as we will discuss below), the transition to post-quantum cryptography and large increase in connection establishment overhead will greatly inflate the data used by mobile applications.

We can estimate how large this difference in size may be for a number of the apps that we sampled. We base our calculations on the assumption that every non-resumed handshake currently uses ephemeral key exchange, authenticates using a server certificate, and includes an intermediate certificate to authenticate that certificate. We assume that X25519 [8] is used for the key exchange (which takes 64 bytes) and RSA-2048 [39] is used as signature algorithm in all certificates, the public keys and signatures of which add up to 1376 bytes of data[2]. This means we have two public keys for ephemeral key exchange, three signatures (one handshake signature and two signatures in certificates), and one signature algorithm public key in each certificate. If we replace X25519 by the post-quantum key encapsulation mechanism (KEM) Kyber-512 [41] and RSA-2048 by post-quantum signature scheme Dilithium2 [27], the two primary schemes that have been selected for standardization by NIST in 2022 [2], we instead require 1586 bytes for key exchange and 9984 bytes for authentication; a total of over 11 kilobyte. We show the theoretical impact on handshake sizes of using Kyber-512 and Dilithium2 for some of the apps that we have sampled in Table 4.

As observed by Sikideris et al. [45, 46], when the size of the server certificate message exceeds the initial congestion window (`initcwnd`) size, it will also result in additional round-trips before the individual handshake can be completed. This would be the case if the recommended[3], more conservative Dilithium3 parameter set would be used: this would require 13 783 bytes just for authentication.

Table 4. Estimates of data in kB used for asymmetric cryptography in non-resumed handshakes by apps when using pre-quantum or post-quantum cryptographic primitives

App	# Full HS	Key exchange	Data	Auth.	Data	Total crypto overhead
Klarna	51	X25519	3.3	RSA-2048	66.9	70.2
		Kyber-512	80.0	Dilithium2	504.1	584.1
Lighter Simulation	257	X25519	16.4	RSA-2048	337.2	353.6
		Kyber-512	403.0	Dilithium2	2540.2	2943.2
Haircut prank, air horn & fart	320	X25519	20.5	RSA-2048	419.8	440.3
		Kyber-512	501.8	Dilithium2	3162.9	3664.6

5.3. Reducing the overhead of (post-quantum) TLS handshakes

We see that many apps perform a very large amount of TLS handshakes, even though they often connect to the same hostnames. This suggests that although TLS or protocols like QUIC offer resumption mechanisms, and application-layer protocols like HTTP allow connection multiplexing, these features are only used by very few apps. Evidently, these features are either not well-known to developers, or not very easy to use compared to setting up new connections. This could be improved by offering developers better documentation and making it easier to re-use connections or perform multiple requests over the same connection. Android's standard HTTPS library could for example be set up to use HTTP/3 over QUIC by default, and use resumption whenever possible. Alternatively, it could be made possible to intercept any calls to the standard library HTTPS stack so that dependencies in apps can be uplifted to HTTPS implementation with more features or better connection coalescing and caching. Finally, Android could offer metrics or profilers to developers on HTTP connection usage and encourage their use for improving application performance, so that developers have a better understanding of what their apps are doing and how they connect to the internet. Such utilities already are a core component of web browsers' development tools [7, 29].

The operators of the API services that apps call over HTTPS also need to ensure that their endpoints support modern transport protocols, such as TLS 1.3, HTTP/2, QUIC, or HTTP/3. TLS session resumption also requires special attention: if a hostname is served by more than one TLS server, the servers need to share a session database or a symmetric session cookie encryption key. These session databases or cookie keys are very sensitive: if obtained by an adversary they could be used to perform a machine-in-the-middle attack.

5.4. Further ways of mitigating the impact of post-quantum cryptography

The most important way to reduce the impact of post-quantum cryptography on the performance of mobile apps and the amount of data that they use will be to reduce the number of TLS handshakes and use session resumption as much as possible. However, session resumption does have considerable deployment concerns: it is not always possible to share a session database between servers, and setting up different servers with a shared symmetric session cookie encryption key may have security implications. Finally, although connection multiplexing or session resumption are arguably the best way to make an impact, they can only affect the *second* request that an app makes: the first request still needs to complete the full handshake to set up the encrypted channel and/or obtain a session ticket to be used in later resumptions.

The biggest contributor to the size of the post-quantum TLS handshakes are the post-quantum signature schemes: a Dilithium2 public key and signature are much larger than a Kyber-512 public key and ciphertext. This difference can be exploited by using KEMTLS, proposed by Schwabe et al. at ACM CCS 2020 [43]. Using KEMTLS, which authenticates using a KEM key exchange instead of a signature scheme, would allow to replace, for example, the Dilithium2 public key and signature (3732 bytes) used for handshake authentication in post-quantum TLS by a Kyber-512 public key and ciphertext (1568 bytes): this leads to a 2164 byte reduction in handshake size. But for mobile

2. X25519 is used in the vast majority of the recorded handshakes.

3. The Dilithium team writes on their website: "We recommend using the Dilithium3 parameter set, which—according to a very conservative analysis—achieves more than 128 bits of security against all known classical and quantum attacks" [13].

apps, which we have seen connect many times to only a handful of hosts that have often been hard-coded in the application, we can also consider another approach.

KEMTLS-PDK, a variant of KEMTLS proposed by Schwabe et al. at ESORICS 2021 [42], can be used if the client already has a long-term public key belonging to the server. This can, for example, be used in mobile applications, which also benefit from having reliable update mechanisms through the app stores, by bundling the server's public key in the application. Note that public keys are much easier to manage than the symmetric keys required for TLS' PSK mechanism, and KEMTLS-PDK servers do not need to maintain or synchronize a session database. KEMTLS-PDK can greatly reduce the handshake size: it allows to omit the certificates from the (KEM)TLS handshake altogether. We compare post-quantum TLS, KEMTLS and KEMTLS-PDK in Table 5.

At the cost of storing the server's long-term KEM public key in the application, we are able to authenticate the server using only one ciphertext of traffic. When using Kyber-512, this is 768 bytes. But in KEMTLS-PDK we can also use the very conservative Classic McEliece [3] scheme, which has very large public keys but very small ciphertexts. Classic McEliece's public keys are too large for transmission as a mceliece348864 public key is 261 120 bytes. However, when used in KEMTLS-PDK, only the ciphertext needs to be transmitted. This allows us to further reduce the handshake size, to almost the same size as TLS 1.3's `psk_dhe` pre-shared-key handshake. At the same time, KEMTLS-PDK avoids having to set up session databases or session cookie encryption keys on the server side.

Table 5. Sizes of alternative TLS handshake proposals when instantiated with post-quantum cryptography

Handshake	**Algorithms**	**Size of public key crypto (bytes)** KEX	Auth.	Sum
TLS	Kyber-512 & Dilithium2	1568	9884	11 452
KEMTLS	Kyber-512 & Dilithium2	1568	7720	9288
KEMTLS-PDK	Kyber-512	1568	768	2336
KEMTLS-PDK	Kyber-512 & McEliece348864	1568	96	1664

6. Conclusion

We presented a first investigation of the adoption of post-quantum TLS for Android apps. We studied the top 45 highest-ranked applications in the Games and Apps categories from the Google Play Store. Leveraging the network traffic that was collected after executing the applications on a real Android device, we extracted a variety of TLS features which we then used to assess the expected overhead of migrating to post-quantum TLS for Android apps. Our results showed that Apps and Games set up large numbers of TLS connections, often to the same hosts, and many applications make little use of resumption to reduce the overhead of the TLS handshake. We argue that this will greatly magnify the impact of the transition to post-quantum cryptography, and recommend that developers, server operators and the mobile operating systems invest in making more use of these mitigating features or improving their accessibility. Finally, we briefly discussed how the KEMTLS and KEMTLS-PDK proposals for post-quantum TLS handshakes might reduce the overhead.

Future work will focus on conducting a deep-dive into the applications themselves in order to better understand the origin of the TLS connections related to advertising libraries, analytics and log reporting. Finally, instead of probing apps in a laboratory setting, it would be interesting to replicate the large-scale study by Birghan and Van der Merwe [9] on real-world TLS usage by Firefox users, in the context of mobile devices in order to collect a larger set of real-world telemetry data.

Acknowledgments

Dimitri Mankowski and Veelasha Moonsamy were funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. Thom Wiggers was supported by the European Research Council through Starting Grant No. 805031 (EPOQUE).

References

[1] Patrick Ahlbrecht. *Raccoon: The APK Downloader*. https://raccoon.onyxbits.de. Accessed: February 28, 2023.

[2] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. NISTIR 8413. updated version. National Institute of Standards and Technology, Sept. 26, 2022. DOI: 10.6028/NIST.IR.8413-upd1.

[3] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. *Classic McEliece*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions. National Institute of Standards and Technology, 2022.

[4] *Android Debug Bridge (adb)*. Accessed: February 28, 2023. Android Open Source Project.

[5] *App Discovery and ranking - play console help*. Accessed: March 22, 2023. URL: https://support.google.com/googleplay/android-developer/answer/9958766?hl=en.

[6] Apple. *Preventing insecure network connections*. URL: https://developer.apple.com/documentation/security/preventing_insecure_network_connections (visited on 03/07/2023).

[7] Kayce Basques. *Inspect Network activity – Chrome Devtools*. Feb. 8, 2019. URL: https://developer.chrome.com/docs/devtools/network/ (visited on 03/07/2023).

[8] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records." In: *PKC 2006*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. LNCS. Springer, Heidelberg, Apr. 2006, pp. 207–228. DOI: 10.1007/11745853_14.

[9] Moritz Birghan and Thyla van der Merwe. "A Client-Side Seat to TLS Deployment." In: *IEEE Security and Privacy Workshops (SPW)*. 2022, pp. 13–19. DOI: 10.1109/SPW54247.2022.9833861.

[10] Mike Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114. URL: https://www.rfc-editor.org/info/rfc9114.

[11] Block, Inc. *OkHTTP*. 2022. URL: https://square.github.io/okhttp/ (visited on 03/07/2023).

[12] Selenium Contributors. *Selenium*. https://www.selenium.dev/. Accessed: February 28, 2023.

[13] Dilithium team. *Dilithium*. URL: https://pq-crystals.org/dilithium/ (visited on 03/09/2023).

[14] Taher Elgamal and Kipp E.B. Hickman. *The SSL Protocol*. Internet-Draft draft-hickman-netscape-ssl-00. Work in Progress. Internet Engineering Task Force, Apr. 1995. 31 pp. URL: https://datatracker.ietf.org/doc/draft-hickman-netscape-ssl/00/.

[15] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. "Why Eve and Mallory love Android an analysis of AndroidSSL (in)security." In: *ACM CCS 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM Press, Oct. 2012, pp. 50–61. DOI: 10.1145/2382196.2382205.

[16] Paul Ford-Hutchinson. *Securing FTP with TLS*. RFC 4217. Oct. 2005. DOI: 10.17487/RFC4217. URL: https://www.rfc-editor.org/info/rfc4217.

[17] Google. *Perform network operations using Cronet*. June 11, 2022. URL: https://developer.android.com/guide/topics/connectivity/cronet (visited on 03/07/2023).

[18] Google Developers. *Security with network protocols*. Dec. 13, 2022. URL: https://developer.android.com/training/articles/security-ssl (visited on 03/07/2023).

[19] *Google Play*. https://play.google.com/store. Accessed: February 28, 2023.

[20] Guy Harris and Michael Richardson. *PCAP Capture File Format*. Internet-Draft draft-ietf-opsawg-pcap-02. Work in Progress. Internet Engineering Task Force, Jan. 2023. 10 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-opsawg-pcap/02/.

[21] Paul E. Hoffman. *SMTP Service Extension for Secure SMTP over Transport Layer Security*. RFC 3207. Feb. 2002. DOI: 10.17487/RFC3207. URL: https://www.rfc-editor.org/info/rfc3207.

[22] Ralph Holz, Jens Hiller, Johanna Amann, Abbas Razaghpanah, Thomas Jost, Narseo Vallina-Rodriguez, and Oliver Hohlfeld. "Tracking the Deployment of TLS 1.3 on the Web: A Story of Experimentation and Centralization." In: 50.3 (2020). DOI: 10.1145/3411740.3411742.

[23] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. *SPHINCS+*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022.

[24] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[25] Imran Khan, Linda D Hollebeek, Mobin Fatma, Jamid Ul Islam, Raouf Ahmad Rather, Shadma Shahid, and Valdimar Sigurdsson. "Mobile app vs. desktop browser platforms: the relationships among customer engagement, experience, relationship quality and loyalty intention." In: *Journal of Marketing Management* (2022), pp. 1–23.

[26] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. "Coming of Age: A Longitudinal Study of TLS Deployment." In: *Proceedings of the Internet Measurement Conference 2018* (2018).

[27] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022.

[28] MobiLoud. *Mobile Apps vs Mobile Websites: Which is Best for 2023?* https://www.mobiloud.com/blog/mobile-apps-vs-mobile-websites. 2023.

[29] Mozilla. *Network Monitor*. URL: https://firefox-source-docs.mozilla.org/devtools-user/network_monitor/ (visited on 03/07/2023).

[30] National Institute for Standards and Technology. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. 2016.

[31] Chris Newman. *Using TLS with IMAP, POP3 and ACAP*. RFC 2595. June 1999. DOI: 10.17487/RFC2595. URL: https://www.rfc-editor.org/info/rfc2595.

[32] Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. "Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications." In: *USENIX Security 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 4347–4364.

[33] *OpenVPN Protocol*. URL: https://openvpn.net/community-resources/openvpn-protocol/ (visited on 07/20/2022).

[34] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *FALCON*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022.

[35] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. "Studying TLS Usage in Android Apps." In: *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. 2017, 350–362. DOI: 10.1145/3143361.3143400.

[36] Eric Rescorla. *HTTP Over TLS*. RFC 2818. May 2000. DOI: 10.17487/RFC2818. URL: https://www.rfc-editor.org/info/rfc2818.

[37] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[38] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[39] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Communications of the Association for Computing Machinery* 21.2 (1978), pp. 120–126.

[40] Emily Schechter. *Moving towards a more secure web*. Sept. 8, 2016. URL: https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html (visited on 03/07/2023).

[41] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. National Institute of Standards and Technology, 2022.

[42] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "More Efficient Post-quantum KEMTLS with Pre-distributed Public Keys." In: *ESORICS 2021, Part I*. Ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12972. LNCS. Springer, Heidelberg, Oct. 2021, pp. 3–22. DOI: 10.1007/978-3-030-88418-5_1.

[43] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures." In: *ACM CCS 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350.

[44] Satadal Sengupta, Niloy Ganguly, Pradipta De, and Sandip Chakraborty. "Exploiting Diversity in Android TLS Implementations for Mobile App Traffic Classification." In: *The World Wide Web Conference (WWW)*. 2019, 1657–1668. DOI: 10.1145/3308558.3313738.

[45] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. "Assessing the Overhead of Post-Quantum Cryptography in TLS 1.3 and SSH." In: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. 2020, 149–156. DOI: 10.1145/3386367.3431305.

[46] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. "Post-Quantum Authentication in TLS 1.3: A Performance Study." In: *NDSS 2020*. The Internet Society, Feb. 2020.

[47] George Tasopoulos, Jinhui Li, Apostolos P. Fournaris, Raymond K. Zhao, Amin Sakzad, and Ron Steinfeld. "Performance Evaluation of Post-Quantum TLS 1.3 on Resource-Constrained Embedded Systems." In: *Information Security Practice and Experience*. Ed. by Chunhua Su, Dimitris Gritzalis, and Vincenzo Piuri. 2022, pp. 432–451.

[48] The Wireshark team. *Tshark: The Network Protocol Analyzer*. https://www.wireshark.org/docs/man-pages/tshark.html. Accessed: September 23, 2021. 2021.

[49] Tanvi Vyas and Peter Dolanjski. *Communicating the dangers of non-secure HTTP*. Jan. 20, 2017. URL: https://blog.mozilla.org/security/2017/01/20/communicating-the-dangers-of-non-secure-http/ (visited on 03/07/2023).

[50] Bas Westerbaan. *Sizing up post-quantum signatures*. Nov. 2021. URL: https://blog.cloudflare.com/sizing-up-post-quantum-signatures/ (visited on 12/22/2022).

[51] Daniel Zelle, Christoph Krauß, Hubert Strauß, and Karsten Schmidt. "On Using TLS to Secure In-Vehicle Networks." In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ARES '17. Reggio Calabria, Italy: Association for Computing Machinery, 2017. ISBN: 9781450352574. DOI: 10.1145/3098954.3105824. URL: https://doi.org/10.1145/3098954.3105824.

A. Tables

In Tables 6 and 7, we show the complete list of games and apps that we measured, including the metrics that we have collected.

Table 6. All tested applications from Apps category with their statistics

App Title App ID	Handshakes	Resumptions	Servers	TLS 1.3	Time (s)	Traffic (MB)	QUIC handshakes
QRScanner - Super QR Code Tool app.android.qrscanner	97	13	27	93%	0.4	2.2	28
Amazon Prime Video com.amazon.avod.thirdpartyclient	124	11	72	50%	0.3	2.8	11
Amazon Shopping com.amazon.mShop.android.shopping	137	47	46	39%	0.8	6.4	5
BeReal. Your friends for real. com.bereal.ft	15	1	8	60%	24.3	0.5	2
kaufDA - Leaflets & Flyer com.bonial.kaufda	65	17	19	42%	19.1	12.4	2
Booking.com: Hotels and more com.booking	54	5	32	83%	0.2	1.1	9
Blood Pressure com.bpfit.bloodpressure.health	91	15	40	86%	0.3	1.8	27
Disney+ com.disney.disneyplus	56	29	23	48%	0.2	0.8	2
Duolingo: language lessons com.duolingo	77	12	26	57%	0.3	18.4	13
eBay Kleinanzeigen Marketplace com.ebay.kleinanzeigen	205	44	64	78%	4.8	7.7	17
Fasto VPN com.fretopvp.org	40	9	20	78%	3.5	1.2	9
QR & Barcode Scanner com.gamma.scan	27	2	19	96%	31.8	0.8	3
Google Translate com.google.android.apps.translate	47	14	23	66%	0.2	1.0	9
Hook VPN - Fast & Secure VPN com.hookvpn.vpn	57	11	28	89%	2.5	1.2	11
Instagram com.instagram.android	51	7	23	94%	0.7	124.7	16
Kaufland - Shopping & Offers com.kaufland.Kaufland	105	28	36	28%	0.4	6.8	11
CapCut - Video Editor com.lemon.lvoverseas	41	10	23	95%	11.8	21.1	0
Lidl Plus com.lidl.eci.lidlplus	57	14	25	63%	1.6	9.4	14
AppLock : Lock app & Pin lock com.lutech.applock	66	16	25	92%	6.6	3.6	17
Klarna | Shop now. Pay later. com.myklarnamobile	56	5	26	11%	1.2	2.3	1
Netflix com.netflix.mediaclient	59	28	25	63%	2.4	1.6	12
PayPal - Send, Shop, Manage com.paypal.android.p2pmobile	51	9	21	75%	0.3	1.1	10
Haircut prank, air horn & fart com.pranksounds.appglobaltd	430	110	100	77%	2.2	27.0	18
Samsung Smart Switch Mobile com.sec.android.easyMover	8	2	5	75%	0.5	0.1	0
Shenzo VPN - Private & Safe com.shenzo.vpn.free	78	10	20	94%	1.4	6.1	4
Snapchat com.snapchat.android	52	10	21	98%	30.0	41.2	12
Spotify: Music, Podcasts, Lit com.spotify.music	37	8	24	68%	1.8	3.6	7
S-pushTAN com.starfinanz.mobile.android.pushtan	4	3	3	100%	0.4	0.0	0
QR & Barcode Reader com.teacapps.barcodescanner	33	6	15	85%	3.7	0.8	10
Netto-App com.valuephone.vpnetto	108	12	41	69%	0.3	3.2	21
Voilà AI Artist Cartoon Avatar com.wemagineai.voila	85	21	38	94%	3.8	7.4	6
WhatsApp Messenger com.whatsapp	9	2	5	100%	0.4	0.2	0
TikTok com.zhiliaoapp.musically	177	30	81	80%	2.3	56.2	30
SHEIN-Fashion Shopping Online com.zzkko	57	6	32	84%	0.9	3.6	3
OTTO – Shopping & Möbel de.cellular.ottohybrid	163	18	37	64%	5.9	7.2	16
CHECK24 Vergleiche de.check24.check24	206	40	67	67%	1.5	5.1	12
Mein dm de.dm.meindm.android	104	7	42	90%	0.3	5.3	17
ElsterSecure de.elster.elstersecure.app	10	5	4	40%	0.4	0.3	1
McDonald's Deutschland de.mcdonalds.mcdonaldsinfoapp	185	39	36	65%	2.0	3.5	12
Joyn | deine Streaming App de.prosiebensat1digital.seventv	317	71	110	69%	1.1	20.0	34
REWE - Online Supermarkt de.rewe.app.mobile	57	14	29	37%	1.4	2.3	4
Zalando – online fashion store de.zalando.mobile	48	7	26	96%	0.2	4.1	4
Dezor net.dezor.browser	21	4	14	95%	32.7	0.2	0
File Miner net.fileminer.android	68	16	17	85%	1.7	1.6	20
Telegram org.telegram.messenger	16	2	10	56%	0.2	0.2	3

Table 7. All tested applications from Games category with their statistics

App Title App ID	Handshakes	Resumptions	Servers	TLS 1.3	Time (s)	Traffic (MB)	QUIC handshakes
Tall Man Run com.VectorUpGames.TallManRun	427	128	86	56%	1.6	7.7	21
Snake.io - Fun Snake .io Games com.amelosinteractive.snake	136	21	63	71%	1.4	11.8	10
Magic Princess: Dress Up Games com.anime.magic.princess.doll.avatar.dressup.games	223	47	60	50%	6.3	10.3	24
Block Blast Adventure Master com.block.juggle	201	59	66	35%	2.3	6.3	17
Darts Club: PvP Multiplayer com.boombitgames.Dartsy	83	16	46	37%	3.3	2.2	8
Ragdoll Playground com.cdtg.ragdoll.playground	120	8	59	92%	2.3	13.0	5
Chess - Play and Learn com.chess	118	17	63	34%	1.6	1.7	11
Block Crazy Robo World com.crazy.block.robo.monster.cliffs.craft	135	20	54	77%	3.0	3.3	18
Craft Skyland Loki Pro com.crazy.rainbow.building.pixelart.mini.pro.craft	75	3	37	87%	8.1	7.4	7
CubeCraft com.cww.cubecraft	134	9	67	84%	1.4	8.0	9
Royal Match com.dreamgames.royalmatch	23	3	16	30%	29.3	7.0	1
FIFA Soccer com.ea.gp.fifamobile	139	16	58	55%	2.9	22.5	2
Frozen City com.fct.global	103	14	56	17%	1.3	4.6	3
Mahjong Club - Solitaire Game com.gamovation.mahjongclub	108	21	47	78%	2.4	3.3	22
Lords Mobile: Kingdom Wars com.igg.android.lordsmobile	9	2	8	89%	0.9	47.3	2
Draw Action: freestyle fight com.kayac.DrawFight	395	102	80	88%	2.5	21.3	26
Gear Clicker com.kb.gearclicker	651	122	111	74%	2.3	12.1	33
Subway Surfers com.kiloo.subwaysurf	72	9	38	76%	1.6	1.2	5
Candy Crush Saga com.king.candycrushsaga	917	900	9	93%	0.3	5.3	0
Farm Heroes Saga com.king.farmheroessaga	101	86	9	24%	0.1	1.0	0
Stumble Guys com.kitkagames.fallbuddies	76	7	43	67%	1.8	4.3	4
Parking Jam 3D com.lszenlamzr.parkingjam	295	75	74	77%	5.5	13.8	23
Makeover Studio: Makeup Games com.makeovergame.studio	376	111	96	15%	2.2	24.0	12
UNO!™ com.matteljv.uno	90	9	49	81%	0.6	1.7	13
Save the Doge com.miracle.savethedoge.an	258	35	88	69%	1.9	6.8	20
Ludo Club - Fun Dice Game com.moonfrog.ludo.club	103	10	50	69%	1.2	4.2	7
Fishdom com.playrix.fishdomdd.gplay	77	26	30	35%	1.1	6.5	4
Gardenscapes com.playrix.gardenscapes	47	18	18	45%	13.1	21.5	2
Homescapes com.playrix.homescapes	39	13	20	49%	7.4	4.2	1
Township com.playrix.township	32	13	16	53%	8.3	40.8	1
Find the Difference - Spot it com.puzzle.find.differences	281	52	81	67%	1.0	14.1	21
Attack Hole - Black Hole Games com.redlinegames.attackhole	369	54	82	75%	3.3	21.3	19
Roblox com.roblox.client	447	6	63	74%	10.8	12.6	0
Stormshot: Isle of Adventure com.sivona.stormshot.e	41	7	21	37%	9.1	182.1	0
Lighter Simulation com.smoke.lighter.simulator	290	33	83	90%	3.0	24.4	39
Snake Run Race 3D Running Game com.snakeattack.game	475	103	111	36%	2.2	9.2	18
Makeover & Makeup ASMR com.storm.beauty.makeover	279	76	76	81%	3.7	23.0	21
Makeover salon: Makeup ASMR com.storm.beauty.makeover.nail	303	38	98	88%	3.9	22.0	38
Dice Dreams com.superplaystudios.dicedreams	123	11	60	68%	0.8	47.8	8
Primitive Era: 10000 BC com.tg.ysrgb.gp	28	7	16	50%	3.5	83.9	2
Triple Tile: Match Puzzle Game com.tripledot.triple.tile.match.pair.game.three.master.object	155	20	77	24%	1.7	6.4	5
Woodoku - Block Puzzle Games com.tripledot.woodoku	171	24	79	13%	3.1	3.3	1
Mob Control com.vincentb.MobControl	213	58	78	23%	2.4	9.5	10
Wörter Los! - Kreuzworträtsel com.wordgame.newcross.android.de	135	16	66	26%	1.4	4.0	4
Magic Tiles 3 com.youmusic.magictiles	269	26	94	65%	2.6	9.0	11