WireWatch: Measuring the security of proprietary network encryption in the global Android ecosystem

Mona Wang
Princeton University
Princeton, New Jersey, USA
monaw@princeton.edu

Jeffrey Knockel and Zoë Reichert
Citizen Lab, University of Toronto
Toronto, Ontario, Canada
{*jeff,zoe.reichert*}*@citizenlab.ca*

Prateek Mittal and Jonathan Mayer
Princeton University
Princeton, New Jersey, USA
{*pmittal,jonathan.mayer*}*@princeton.edu*

***Abstract*—We present WireWatch, a large-scale measurement pipeline to evaluate the network security of Android apps. WireWatch measures apps' usage of plaintext network traffic and non-standard, proprietary network cryptography. We found that 47.6% of top Mi Store applications used proprietary network cryptography without any additional encryption, compared to only 3.51% of top Google Play Store applications. We analyzed the 18 most popular protocols from WireWatch, which belonged to 9 protocol families, including cryptosystems designed by Alibaba, iQIYI, Kuaishou, and Tencent. We found that 8 of these protocol families sent requests that allowed network eavesdroppers to decrypt underlying data, including browsing data and device metadata, among various other issues, such as being downgradable, not validating TLS certificates, and the use of RSA without OAEP. These vulnerabilities affected 26.9% of our Mi Store dataset with a cumulative 130 billion downloads. Ultimately, WireWatch reveals that a large portion of massively popular applications are using insecure proprietary network protocols to encrypt sensitive user data.**

1. Introduction

Network encryption on the web is nearly universal. As of writing, over 80% of Firefox requests and over 90% of Chrome requests are using TLS [1], [2]. Despite the ongoing success of standardized, strong encryption on the web, the same story is not yet true on mobile. Prior work studying privacy and network security issues within the Google Play ecosystem [3], [4] often overlooks large parts of the global Android app market by ignoring Chinese app markets.

Researchers have previously studied non-TLS network encryption used by popular Chinese mobile apps, but at smaller scales. In particular, many massively popular apps (e.g., WeChat [5]) use home-rolled proprietary network cryptography over standard encryption schemes like TLS, HTTPS, or QUIC. Such proprietary cryptographic protocols often suffer from massive flaws such as being decryptable by network attackers [4], [6], [7], [8]. These kinds of vulnerabilities have historically been leveraged for mass surveillance [9].

In this work, we show that these cases were the tip of the iceberg, and the phenomenon of massively popular applications using flawed proprietary network cryptography protocols is much larger than the dozens of applications previously evaluated. To assess the scope of this phenomenon, we designed WireWatch, a measurement pipeline to evaluate the network security of mobile apps and to identify the usage of non-standard cryptography. WireWatch automatically scrapes application stores, installs and exercises their UI, analyzes the resulting network requests for proprietary encryption, and clusters requests containing proprietary encryption. We ran 1,699 of the most popular mobile apps, 882 from the Google Play Store and 817 from the Mi Store, through this pipeline.

WireWatch found that on the Xiaomi Mi Store, not only were a majority of applications sending unencrypted requests, but nearly half were using proprietary network cryptography to encrypt requests. Although only 12.9% of the applications from the Play Store sent unencrypted requests, 65.4% of the applications from the Mi Store did. Notably, **47.6%** of the applications from the Mi Store were utilizing proprietary network cryptography with no additional encryption, compared to 3.51% of the applications on the Google Play Store. In some cases, the application itself developed a custom protocol (such as WeChat [5]) for network transmissions. In other cases, various third-party SDKs were responsible for the network transmissions containing proprietary cryptography.

We analyzed the 18 most popular protocols from WireWatch, which we determined as belonging to 9 cryptosystems: Alibaba mPaaS, Beizi AdScope SDK, iQiyi, MobSDK, Kuaishou Ad SDK, Shumei, Tencent DNSPod, Tencent MMTLS, and Tencent WUP. Of these 9, 8 sent or received some network communications that were decryptable by network adversaries. Attacks included decrypting symmetrically-encrypted payloads with publically inferrable keys, MITMing TLS sessions, reading the contents of files on a user's phone, exploiting an AES-CBC padding oracle, and inferring an AES key using a novel chosen-ciphertext attack on a textbook RSA construction. The data sent or received by these SDKs using custom cryptography included user browsing data and network and device metadata. The scope of these vulnerabilities alone is massive, affecting 26.9% of our Mi Store dataset with a cumulative 130 billion downloads.

In summary, our contributions are as follows.

- We develop WireWatch, an open-source measurement pipeline to identify the usage of non-standard cryptography by mobile apps, and analyze 1.7k top apps from Google Play Store and Xiaomi Mi Store.
- WireWatch finds that 47.6% of top Mi Store apps use proprietary network cryptography, as compared to 3.51% of top Google Play Store apps.
- We manually reverse-engineered the 9 most popular protocol families identified by WireWatch, and found that 8 sent requests vulnerable to decryption.

Ultimately, we find that the continued usage of poorly designed custom cryptography, as well as plaintext network transmissions, is still a systemic issue across the most popular mobile apps in the world.

1.1. Ethical considerations and vulnerability disclosure timelines

In this work, we provide detailed descriptions of cryptosystems actively being used by hundreds of applications that are themselves used by hundreds of millions of people, possibly affecting over one billion users in total. For completeness and reproducibility, we provide code for retrieving the underlying data or conducting other attacks against these protocols. A summary of the issues we found, and the scope of these issues, is available in Table 3.

We are aware of the risk of premature publication and the potential ethical ramifications if these vulnerabilities are not addressed quickly. We undertook extensive co-ordinated vulnerability disclosures with the 7 vendors affected by these issues. We sent disclosures to Alibaba, Beizi, iQIYI, Kuaishou, and Shumei, MobTech, and Tencent between November 12-26, 2024. The disclosures request companies to fix the issues within 45 days. As of April 1st, 2025, iQIYI, Kuaishou, MobTech, and Tencent have replied with the intention to fix the above vulnerabilities, and to inform downstream apps to update their SDK versions, but we have not received any response from the other vendors. iQIYI, Kuaishou, and MobTech have deployed fixes for these issues which have reached downstream, updated versions of applications.

2. Motivation and background

Modern mobile applications are expected to transmit data securely with standard encryption like TLS. In 2018, researchers found that of the top 200 applications in the Google Play Store, only 3 transmitted sensitive data over HTTP or improperly validated HTTPS [10]. However, this work, and other studies of TLS usage in Android, have largely focused on the Google Play Store context [3], [11], [12], [13]. As the Google Play Store is not available in China, such studies overlook a significant portion of the global Android user base.

Recent global studies that do include apps from Chinese markets have demonstrated remarkable differences in the security of apps from the Google Play market and apps from Chinese stores [14]. Li et al. found that apps developed for the Chinese market tend to prefer proprietary protocols for DNS resolution, ignoring the system DNS [15]. Pourali and Yu et al. find that applications from Chinese app stores fail to validate TLS certificates at a rate much higher than applications on the Google Play Store [16].

Researchers have also started identifying the insecurity of proprietary network protocols in popular Chinese mobile applications. In a series of 2015–2016 reports, researchers found that mobile browser applications popular in China used broken proprietary network cryptography [6], [17], [18], [19], [20]. In 2016, researchers manually reviewed 60 apps from a Chinese app store; however, since they use port numbers to distinguish proprietary encryption, they only find 6 cases of insecure proprietary encryption [7]. In 2024, researchers discovered a similar set of issues affecting nearly all software keyboards (Input Method Editors, or IMEs) popular in China, allowing network eavesdroppers to decrypt and obtain users' keystrokes [8].

The trend of apps using proprietary cryptography is large enough that it has been observed in adjacent privacy studies. Pourali et al., while studying the usage of encryption to obfuscate privacy analysis, discovered network vulnerabilities in at least 24 applications using insecure proprietary cryptography [4]. This work relied on hooking standard JDK/Java cryptography libraries or non-obfuscated method names, which may have incidentally missed uses of proprietary cryptography. Cryptography is often implemented in NDKs for performance, and Chinese apps are more likely to use obfuscation [21]. Like related work in automated privacy measurement [22], this study also only reviewed Play Store apps.

Given the growing evidence, we hypothesize that there is a large class of massively popular, yet understudied mobile apps that encrypt sensitive network data with insecure proprietary cryptography. This work seeks to answer the following research questions for applications popular globally:

1) How common is proprietary network encryption?
2) How secure are these protocols?

Such use of insecure proprietary network encryption may be unnoticed by the computer security community due to the trend of excluding Chinese applications in global security measurements. We developed WireWatch to characterize the nature of proprietary encryption across popular applications, and to provide researchers with tooling for performing such security evaluations at scale.

2.1. Threat model

We are primarily concerned with the security of data in transit. For instance, when evaluating messaging applications such as WeChat, Facebook Messenger, or Telegram, we are not concerned with whether the underlying data is accessible to the platform provider (e.g., via end-to-end encryption). However, we are concerned whether third-party network attackers can decrypt communications between the

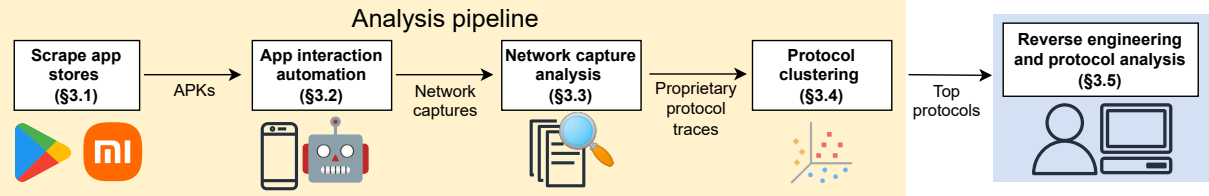

Figure 1. Outline of WireWatch design, our automated analysis pipeline.

client and server. For instance, we are concerned with network adversaries interested in leveraging vulnerabilities in transport security to scale mass surveillance. One historical example of such an attack is the Five Eyes' XKEYSCORE UCWeb plugin, which exploited weak network cryptography in a Chinese application to ingest user data into a mass surveillance database [9], [17].

Specifically, our threat model includes two types of adversaries: a passive network eavesdropper and an active network machine-in-the-middle (MITM). Most of the vulnerabilities we discovered simply require a passive network eavesdropper to decrypt network data, but a few of them also involve an active network MITM. We note that a passive network eavesdropper on the victim's local network can obtain an active MITM position via ARP spoofing. In the case of a passive network eavesdropper, the adversary's capability is simply observing network traffic going to or sent from the user's device. In the case of an active network MITM, the adversary is in a network position such that they can spoof, intercept, or alter network messages from either the user device or the server. For instance, such an attacker may be able to MITM a TLS session if the end device is not validating the server's certificate when establishing TLS connections. In all cases, the adversary can also obtain copies of the application to reverse engineer or study themselves, but they do not have elevated privileges or other access to the end device or servers.

3. Methodology and WireWatch design

In this section, we describe our methods and design of WireWatch, which is summarized in Figure 1. We ran WireWatch throughout the months of August and September 2024 and completed our protocol reverse engineering in October and November 2024.

3.1. Application crawler

Our first goal for WireWatch was to construct a large dataset of globally popular Android applications. In summary, we collected the most recent version of the most downloaded applications from the Google Play Store, via Androzoo [23]. Androzoo is a historical collection of almost 25 million Android applications that is popular among researchers for mobile research [23]. The vast majority of Androzoo applications (up to 22 million) are sourced from the Google Play Store. We also sought application datasets from a popular application store in China.

3.1.1. Scraping the Mi Store. Other researchers have designed scrapers specifically for Huawei AppGallery and Mi Store, the two most popular application stores in China. Unfortunately, existing crawlers for Huawei either were outdated and did not work with current APIs or did not work to download applications popular domestically, i.e., within China [14], [24], [25]. Existing Xiaomi crawlers exclusively used website listings on *mi.com*. However, the web-based application download links on the *mi.com* website have since been taken down. We developed our own scraper by reverse engineering an application store directly. As we had access to a rooted Xiaomi Mi 3 device, we reverse engineered the application discovery and application download APIs used by Mi Store, and leveraged these internal APIs to design a scraper for WireWatch. We hope to contribute this tooling upstream to other work [23], [25].

3.2. App instrumentation and automation

Next, WireWatch automates interaction with each app in order to generate network traffic. Our hardware setup for WireWatch uses an unlocked, rooted Pixel 6 device running Android 12, connected to the host device via USB. We note that root was necessary for our later reverse-engineering, but is actually not necessary for WireWatch. We also did not implement root evasion. The device is also connected to a Wi-Fi network that is bridged at the host such that the host can collect network traffic logs from the device. As our host is in a MITM position, it can also actively alter or inject phone traffic, as in our experiments determining whether clients were validating TLS certificates. The experiment setup is also demonstrated in Figure 2.

For the automation of user interface (UI) interactions, WireWatch's goal is to exercise as many UI paths as possible, thereby generating diverse streams of network traffic from testing differing code paths. WireWatch is driven by Android's UI Automator, a UI testing framework that provides an interface for receiving UI data from the device, as well as sending UI events to the device. We tested Android's Application Exerciser Monkey API as well, but found that we wanted to have more control over certain actions, such as backing out of login pages. Built atop UI Automator, WireWatch performs a breadth-first-search, exercising all clickable elements and text fields in each separate view, while preferring to accept consent dialogs and backing out of login or registration pages when possible. The agent also grants any permissions requested by the application. We run

our agent twice to capture differences when opened after install, closing and restarting the application between each run, with time capped at 5 minutes per run. Finally, we reinstall the application and run the agent once with a TLS certificate MITM (presenting a self-signed certificate) configured at the host using *mitmproxy* in transparent mode, routing all TCP traffic from the device to *mitmproxy*. WireWatch uninstalls each application after testing it.

3.3. Network capture analysis

From the network data collected in this process, we can make various observations about the nature of network requests made by mobile applications.

WireWatch first reconstructs TCP streams and filters empty UDP/TCP/HTTP requests, which are commonly used for network connectivity tests. For all plaintext HTTP and other non-standard TCP or UDP traffic, we additionally analyze the payload to flag whether it could contain encrypted payloads, e.g., proprietary encryption.

3.3.1. Identification of proprietary protocols. In order to identify potentially encrypted payloads, WireWatch first isolates non-standard TCP or UDP traffic, as well as plaintext HTTP traffic. Then, WireWatch tests if the packet payload contains encoded (e.g., base64 or hexadecimal) or structured (e.g., protobuf or JSON) data. If it is encoded, we decode the structured data and recursively perform the same check. If it is structured, we unpack the structure and then recursively perform the same check for all values in the structure (e.g., all values in a JSON/Protobuf list or dictionary). Our base case is non-structured, non-encoded data, for which we perform NIST's statistical test suites for evaluating the randomness of pseudorandom number generators [26]. Our analysis flags any protocol containing sufficiently random data and also reports whether the data is aligned to standard DES or AES block sizes (8 or 16 bytes).

3.3.2. Finding the boundaries of pseudorandom data. If the entire payload is sufficiently random, the boundaries of random data are easy to determine. However, in many cases, the proprietary wire formats we encountered had structured header and footer data surrounding a ciphertext. As such, even though it contained encrypted, pseudorandom data, performing a statistical test on the entire payload led to false negatives. WireWatch thus performs the following check for non-structured data larger than 128 bytes to identify boundaries of pseudorandom data within a payload.

We start searching from the end of the payload rather than the start since structured footers are less common than structured headers. First, WireWatch scans with a window of 64 bits from the end of the payload, calculating the randomness of each window. If we find 64 bits of data that are sufficiently random, we fix the end of the range and binary-search the start of the window range in blocks of 64 bits, such that the data in the range is above the statistical threshold we set for randomness.

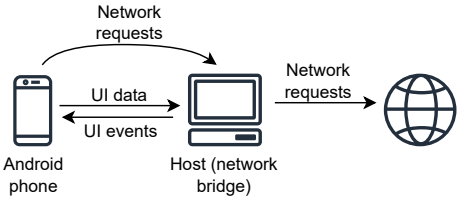

Figure 2. Our experiment setup. For our automation experiments, WireWatch used a rooted Pixel 6 device running Android 12.

3.3.3. Proprietary protocol identification validation. To validate WireWatch's identification of proprietary protocols, we constructed a ground-truth dataset from 100 randomly selected apps. We were unable to scrape 5 (the APK download consistently failed), and 3 did not successfully install on our test device. For the remaining 92 applications, we manually interacted with the application, exhausting as many UI paths as possible in order to generate network traffic. We then comprehensively analyzed the resulting network traffic using Wireshark, including identifying the usage of standard encryption and manually looking through packet traces of non-standard protocols to identify encrypted payloads. In the cases where we identified proprietary encryption, we also briefly reverse engineered the application to determine whether it was indeed encrypted. We then compared our results from the automated analysis.

3.4. Unsupervised clustering of protocols

WireWatch's next goal is to determine common protocols across our dataset, in order to prioritize them for reverse engineering and analysis. WireWatch performs unsupervised clustering of all network payloads flagged as containing proprietary encryption.

3.4.1. Feature, hyperparameter, and clustering algorithm selection. There is extensive prior work on feature selection, hyperparameter optimization, and selection of clustering algorithms for the purpose of network traffic classification [27], [28], [29].

For feature selection, we started with the 15 network features as discussed in [28] and validated in [29]. As we did not want to cluster based on the underlying transport protocol (e.g., TCP vs. HTTP vs. UDP) and with more focus on request contents and structure, we removed network layer features such as port and IP number. WireWatch additionally uses other structural features derived from the network capture analysis (as described in Section 3.3), such as the boundaries of the encrypted payload, the presence of various encoding types in the payload (i.e., raw data, hexadecimal, base64, JSON, or compression). If the encrypted payload was contained in a named key (e.g., a JSON key-value store or a URL query-parameter store), WireWatch includes this named key in the features, one-hot encoded.

WireWatch uses k-means clustering due to its prior performance on unsupervised clustering of network data [29]. WireWatch also performs a hyperparameter search in order

to identify the optimal number of clusters (between 50 and 300) to separate the data, by optimizing the clusters' silhouette score.

3.4.2. Clustering validation. To evaluate WireWatch's clusters, we hold out 100 network requests from the set and cluster them manually as a ground truth dataset. Then, we use standard entropy metrics for cluster evaluation: *homogeneity*, *completeness*, and the *V-measure* [30]. Homogeneity measures whether all points in each cluster are of the same type, and completeness measures whether all messages of a particular type are assigned to the same cluster. The V-measure is the harmonic mean of the two.

3.5. Reverse engineering and protocol analysis

For the protocols determined by WireWatch, we calculated the "impact" of each protocol as the cumulative number of downloads of all applications that we observed sending network requests using that protocol. We sorted the protocol clusters by this value, and started analyzing them from the top of the list. As we will describe in Section 3.5.1, in some cases, protocols that appeared differently on the wire actually belonged to the same family, e.g. using the same software libraries or cryptographic primitives to encrypt payloads that were serialized differently on the wire. We grouped such protocols together into the same *protocol families*. In this way, we continued reverse engineering down the list until we had fully analyzed 18 different protocols, which we determined belonged to 9 different protocol families.

In order to reverse engineer each protocol, we used jadx to analyze the decompiled Java code, and we used IDA Pro 9 and Ghidra to analyze disassembled native libraries, which were commonly used for encryption. We also used Frida to hook various function calls and analyze device memory while the applications were running, as well as Wireshark to capture and analyze network traffic.

Finally, we verified whether these vulnerabilities affected other applications using the same protocol. For passive attacks, we decrypted payloads from each network trace in the cluster with the appropriate key. For active attacks, we manually tested affected applications.

3.5.1. Defining a "network encryption protocol". In this work, we detect "proprietary protocols" based on structural and statistical analyses of network payloads. However, as these proprietary protocols are not standardized or well-defined, the boundary of what is a distinct "network protocol" is ambiguous. For instance, on one hand, applications can use similar wire formats to carry payloads that are encrypted using different cryptosystems. On the other hand, applications can also use different wire formats to carry payloads that are encrypted using the same cryptosystem. In all of these cases, the boundary defining a single "encryption protocol" breaks down, and can only be distinguished or clarified via static analysis and reverse engineering.

Without deeper visibility into the application, WireWatch must use the wire format and other network features as the distinguishing factor between "protocols". In our subsequent analysis of the top 18 protocols as determined by WireWatch, we corrected for such ambiguities by identifying cases when popular protocols were using the same cryptosystem. In these cases, we grouped them together into "protocol families". As an example, Alibaba mPaaS provides a low-level cryptographic library for use by applications. UC Browser, Taobao, and other applications leveraged this library to design custom network encryption protocols, though the underlying (insecure) key derivation and encryption algorithms used were the same. Although WireWatch reports these as being different protocols, since they were using the same underlying cryptosystem, we grouped these protocols together into the same Alibaba mPaaS "protocol family".

3.6. Limitations

Next, we discuss the limitations of WireWatch. First, there will always be code paths that WireWatch will not have traversed in the automation of these applications. This is especially true since we did not register accounts or create logins for these applications, due to the scale of our study. In addition, as our device was rooted and we did not implement root detection evasion, any application that implemented root detection may have altered our results. Thus, our results are biased towards false negatives and represent a lower bound: some applications could indeed be sending more requests containing proprietary encryption, or with no encryption at all, that we did not identify or capture in our analysis.

WireWatch separates app traffic by testing in sequence and only having apps installed when tested. We did not separate system traffic, as we did not identify proprietary encryption or plaintext data sent by Android. However, if it did, this could introduce false positives in our data.

To validate WireWatch, we built a "ground-truth" dataset by analyzing 92 applications at random for the use of proprietary encryption, as well as holding out 100 network request samples from clustering to create a "ground-truth" clustered dataset. We did not balance classes during validation. Due to the number of apps and requests to analyze, we were limited by the amount of time we could spend on manual analysis to create these datasets. As such, this "ground-truth" validation data may also suffer from human error. We note that this analysis is separate from the more complete reverse-engineering process (as described in Section 3.5) that we perform to evaluate the information security of top protocols determined by WireWatch.

In addition, the way we clustered requests and determined "popular protocols" is also subject to bias. For instance, since we are clustering by network requests, protocol families that send network requests more often and protocols used by SDKs included in many applications will be over-represented in the dataset.

App Store	# apps analyzed	Plaintext traffic	Proprietary crypto	TLS not validated
Google Play	882	12.9%	3.5%	2.2%
Mi Store	817	65.4%	47.6%	49.1%
Total	1,699	38.1%	24.2%	24.1%

TABLE 1. SUMMARY OF RESULTS FROM OUR MEASUREMENT PIPELINE.

# DLs	# Mi Store Apps	% using Proprietary Crypto
$\geq$ 1B	69	67.2%
$\geq$ 500M, $<$ 1B	44	59.1%
$\geq$ 100M, $<$ 500M	214	50.5%
$\geq$ 50M, $<$ 100M	218	44.5%
$<$ 50M	272	40.81%

TABLE 2. ON THE MI STORE, PROPRIETARY CRYPTOGRAPHY IS MORE COMMON IN MORE DOWNLOADED APPLICATIONS.

Finally, although we selected protocols based on impact, there remain over 150 protocol clusters that we did not analyze. We note that thoroughly reverse engineering these protocols often takes a large amount of time and effort, especially due to the obfuscation we encountered, and limits the number of protocols we could feasibly reverse engineer. In addition, our method of protocol selection (by cumulative application download) biases towards proprietary encryption that are used across many different applications, e.g., used by popular SDKs. As such, there remain many proprietary protocols that are utilized by applications with hundreds of millions of users.

We note that despite these limitations, we found a significant number of vulnerabilities affecting hundreds of popular apps. Our limitations are such that our findings represent a lower bound of the usage of insecure proprietary encryption in the global Android ecosystem.

4. Results

After selecting the top 1k downloaded applications from both the Google Play Store and Mi Store, our pipeline successfully ran on 1,699 of the applications, completing at a rate of 85.0%. The 1,699 applications consisted of 882 apps from the Google Play Store and 817 from the Mi Store. The applications which our pipeline failed to analyze either failed to download via our scraping method or were incompatible with our testing device. Table 1 presents a summary of our results.

4.1. Validation of WireWatch

We performed two validation experiments with WireWatch, as described in Section 3.3.3 and Section 3.4.2. These experiments demonstrated that WireWatch performed similarly compared to expert review.

4.1.1. Validating the identification of proprietary network encryption in apps. The authors manually reviewed 92 applications in our dataset for the use of proprietary encryption. WireWatch produced three false positives and one false negative when identifying proprietary encryption, as compared to our manual analysis. In the false negative case, manually browsing the application elicited a network request containing proprietary encryption, but our automated agent did not induce similar network requests. In two false positive cases, the application received pseudorandom data from the server in a plaintext HTTP response, which was flagged as proprietary encryption. However, this data was used as a random seed and itself was not an encrypted payload. In the remaining false positive, a long, randomly generated user ID of 256 bits, encoded in base64, was flagged as proprietary encryption. In total, our pipeline detected whether an application is using proprietary encryption, compared to expert review, with 95% accuracy.

4.1.2. Validating WireWatch's unsupervised clustering. The authors selected 100 network requests at random that contained proprietary encryption, and clustered them manually based on the wire format to produce a ground-truth dataset. Compared to our ground-truth clusters, WireWatch's unsupervised clusters obtained a homogeneity score of 91.0%, a completeness score of 98.7%, with a combined V-measure of 94.7%. Though WireWatch's clustering was slightly more *complete* than *homogenous*, it overall performed well compared to expert review.

4.2. Overall transport security

First, we provide an overview of the use of plaintext traffic and network security in general across the apps surveyed by WireWatch. **Plaintext traffic was significantly more popular in Mi Store apps (65.4%) than in Google Play Store apps (12.9%).** In total, 38.1% of applications sent plaintext traffic. This traffic included static resources, such as images and JavaScript payloads loaded from CDN endpoints, as well as sensitive device and network metadata.

In the Mi Store dataset, many applications sent plaintext HTTP requests carrying DNS payloads. The popular Tencent DNSPod protocol that we reverse engineered in Section 4.4 similarly carried DNS payloads, and thus is also a type of proprietary DNS protocol. HTTPDNS, e.g., DNS via HTTP as a transport, and other proprietary DNS protocols, are notably popular in the Chinese application ecosystem, possibly as an adaptation to widespread DNS hijacking. These results are in line with Li et al.'s 2023 study of proprietary DNS protocol usage in popular applications [15].

49.1% of Mi Store apps did not properly validate TLS certificates, compared to only 2.12% of Google Play Store applications. In other words, we were able to successfully MITM a TLS connection with a self-signed certificate, without modifying the device certificate chain, in 24.10% of all apps that we analyzed. This independently confirms concurrent work which found that 55.3% of apps

TABLE 3. SUMMARY OF THE TOP PROPRIETARY NETWORK PROTOCOL FAMILIES AFTER REVERSE-ENGINEERING EACH.

Protocol family	# apps	Cumulative downloads	Most downloaded	MAU	Method(s) of encryption	Decryptable request	Contents	Additional issues
Kuaishou	76	35.10B	Kuaishou	692M	AES-CBC+XOR mask	YES	Device metadata	TLS MITM
MobSDK	82	30.30B	Xiaohongshu	312M	AES-ECB / RSA+AES-CBC	YES	Device metadata	No OAEP padding
Alibaba	15	25.43B	Taobao	921M	AES-CBC	YES	Browsing data	
DNSPod	11	18.10B	Pinduoduo	695M	DES-ECB	YES	DNS requests	
WUP	7	17.62B	QQ Browser	571M	RSA+AES-CBC	YES	Browsing data	Downgradable to no OAEP
iQIYI	3	11.28B	iQIYI	429M	DES-CBC	YES	Network metadata	
Shumei	37	10.34B	Xiaohongshu	312M	DES-ECB / RSA+AES-CBC	YES	Configuration	Remote file access†
MMTLS	1	9.23B	WeChat	1.3B	DH+AES-GCM	NO	-	
Beizi	38	9.02B	Baidu Netdisk	107M	AES-CBC	YES	Device metadata	

† By injecting data into the downloaded configuration, a network MITM can read the contents of files on the client's device.

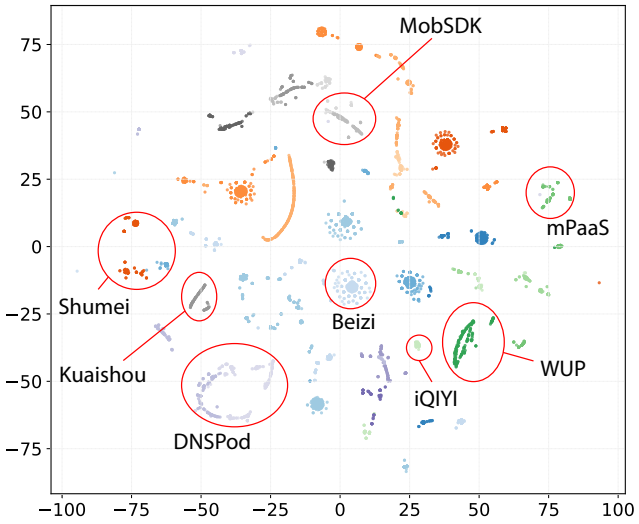

Figure 3. Unsupervised clustering of network requests, plotted against t-SNE feature embeddings and colored differently per cluster. We annotated the most popular clusters of protocol families, summarized in Table 3.

from the Qihoo 360 store and 6.4% of apps from the Google Play Store did not validate TLS certificates [16].

4.3. How common is proprietary network encryption?

After isolating all payloads containing "plaintext" traffic (e.g., non-SSL/TLS TCP data and non-QUIC UDP data), WireWatch analyzes them as described in Section 3.3 to determine whether the application was potentially using proprietary encryption.

47.6% of Mi Store apps sent requests containing proprietary encryption, whereas only 3.51% of the Google Play Store apps did. All except for 3 apps used HTTP requests as transports for proprietary encryption, with ciphertexts either in the HTTP POST request body, or base64-encoded and in the resource URL. In a handful of cases, we observed requests carried over raw UDP or TCP as transports. We suspect this is because middleboxes may be more likely to interfere with or drop TCP or UDP packets carrying non-standard payloads.

In the Mi Store, more popular apps tended to use proprietary cryptography more often. As demonstrated in Table 2, 67.2% of apps with over one billion downloads were using proprietary encryption, whereas only 40.8% of apps with under 50 million downloads were.

4.3.1. Protocol clusters. Finally WireWatch clustered the requests into 177 unique "protocols". 94 of these clusters represented protocols only used by one application; this indicated that the application developed their own, exclusive protocols. In the remaining 83 clusters, the protocols were being used by different applications. In many cases, the endpoints that the applications used were often similar, if not the same. Certain popular SDKs or libraries seem to be developing their own protocols, cryptosystems, and families of protocols in order to send and receive network requests. We visualize the WireWatch clusters in Figure 3.

4.4. How secure are these protocols?

Finally, we want to characterize the security of these protocols. It was infeasible for us to reverse engineer hundreds of applications, so we chose to work from the most popular protocols identified by WireWatch. We reverse engineered 18 protocols in total, which we identified as belonging to 9 different protocol families. These reverse engineering results are summarized in Table 3. This work presents thorough reverse-engineering results for each of these protocols and protocol family.

In the case of Kuaishou, MobSDK, Alibaba, and Shumei, in the process of reverse engineering, we found that they used multiple "protocols" that were driven by the same cryptosystem. We call these "protocol families" as per Section 3.5.1, and thus by fully evaluating one of the protocols in a protocol family we were effectively evaluating all apps, SDKs or cryptosystems using protocols from that protocol family. In the case of attack development, we tested our attacks against all apps sending requests from those protocol families. For MobSDK and Shumei, this is especially relevant because one of their protocols used RSA, and the other did not.

8 of the 9 protocol families sent requests that were decryptable by network adversaries. This is generally marked by the use of a symmetric encryption algorithm,

such as AES, DES, and/or XOR masks, without any asymmetric encryption for bootstrapping the keys. The keys are hard-coded or otherwise generated by a deterministic function with a fixed or known seed across devices, and can be discovered through reverse engineering. For Tencent WUP, we were able to successfully decrypt the underlying data by exploiting an AES-CBC padding oracle, and by exploiting RSA oracles to learn information about the encryption key.

The protocols contained additional flaws that revealed sensitive user data beyond being decryptable. Kuaishou, for instance, had started to tunnel some of their requests using proprietary encryption in HTTPS. However, they failed to verify the TLS server certificate, which means that the tunneled data (which was encrypted with a static symmetric key) was still decryptable by an active network MITM. In the case of Shumei, we also designed a proof-of-concept attack such that a remote network MITM could read the contents of files on the end user's device. We describe this attack in detail in Section 5.2.

3 of the 9 protocol families sent requests that were encrypted with RSA-bootstrapped keys, 2 of which used textbook RSA without OAEP padding. In this construction, the symmetric encryption key is randomly generated per-request, then encrypted using an RSA public key pinned in the application. Both the encrypted key and the ciphertext are delivered to the server. MobSDK used a custom serialization scheme without OAEP or other randomized padding. Although WUP uses RSA encryption by default, we were able to downgrade their encryption with an active attack, inducing WUP to use textbook RSA (i.e. without OAEP). RSA without OAEP is widely regarded as insecure, as the determinism of the construction can enable padding or decryption oracles [31], [32], as well as decreasing the security of the construction overall [33]. In WUP's case, we were able to design a chosen ciphertext attack, leveraging an RSA oracle thanks to the lack of padding, that retrieves the underlying encryption key. Prior versions of WUP have also been vulnerable to decryption, via yet another chosen ciphertext attack due to their use of textbook RSA [20].

None of the protocols except for MMTLS contained any checks for cryptographic authenticity or integrity. Shumei makes an attempt, using an MD5 of the plaintext and key for authentication; however, MD5 should not be considered cryptographically strong authentication. The remainder of the protocols except for MMTLS contained no cryptographic authentication or integrity checks whatsoever, meaning that ciphertexts were entirely malleable.

Even when using asymmetric encryption to bootstrap key exchange, these protocols contained other cryptographic flaws during key generation. Shumei's 128-bit AES key is generated with only around 76 bits of randomness, as they select 16 lowercase letters "a–z" at random. In the case of MobSDK, some applications containing an older version of the SDK used the system timestamp in milliseconds to seed their key generation; in this case, we were able to brute-force the correct AES key from the approximate timestamp of the network request, even if they used RSA-bootstrapped keys.

The contents of decrypted data included browsing data, device metadata, and network metadata. Alibaba mPaaS was used to encrypt browsing information (e.g., anything the user typed into a URL bar) as well as device and network metadata. Sensitive device metadata includes the device make and model, operating system, manufacturer, as well as memory size, storage size, screen size, and network carrier, network information. From Kuaishou SDK, we decrypted requests containing hundreds of settings extracted from the device, as well as lists of every single module loaded by the application, including the memory locations at which each of these modules were loaded. Generally, protocols encrypted similar data and metadata across apps. However, one exception was Alibaba mPaaS, as we found mPaaS was a low-level cryptography library used by different applications for varying purposes, including, for instance, by UC to encrypt browsing data.

5. Case studies

In this section, we describe three of the cryptosystems in depth as case studies that exemplify the nature of proprietary cryptography we analyzed, as well as the type of data encrypted within: Alibaba mPaaS, Shumei, and Tencent WUP. At the end of this section, we include a short summary of the remaining protocol families that we reverse engineered.

5.1. Alibaba mPaaS SDK

Alibaba's mPaaS SDK was used as a lower-level cryptographic library by many popular applications, many of them owned by Alibaba, to encrypt data. At a high level, this cryptosystem effectively encrypts payloads via AES-CBC using a static key. The keys are stored in a local image resource file: `res/drawable/yw_1222.jpg`. We provide code for extracting the key database from any application using this SDK, de-obfuscating `libsgmain*.so`, and decrypting payloads that are encrypted in this manner.

The native library utilized heavy obfuscation, and the static keys were additionally encrypted inside the image resource file using a fixed key derived from the DER encoding of the app's public APK RSA key. Further descriptions of the exact decryption and obfuscation process can be found in the Appendix.

5.1.1. Decrypted data payloads. In the case of `com.UCMobile`, the database entry `3` with value `C7fU4ct8b4lbEo9BomWvWA76` was used to generate keys for encrypting browsing data, or any information typed into the URL bar. The following was encrypted and sent to the domain `sugs.m.sm.cn` in the body of an HTTP POST request. The following network transmission was sent shortly after we had typed `hellocanyoureadthis` in the URL search bar of the browser:

```
{'1': 'hellocanyoureadthis',
     // [etc...]
 '4': 2,
 '5': 0,
```

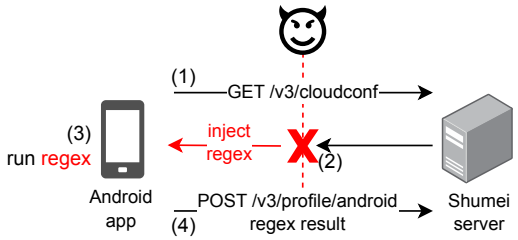

Figure 4. Shumei remote file access attack. (1) The application fetches the security configuration file. (2) As the response is insufficiently encrypted, the attacker can decrypt it, inject regex commands, then re-encrypt the file. (3) The app runs the regex. (4) The result of the regex is encrypted using a slightly more secure encryption algorithm, then sent to Shumei. By designing the injected regex carefully, we can use the size of the encrypted ciphertext in (4) as a side-channel to determine the contents of a particular file on the user's device.

```
5  '6': '搜索大全0216_https1',
6  '7': 158,
7  '8': 1}
```

We were additionally able to decrypt device metadata contained in the bodies of HTTP POST requests to `amdc.m.taobao.com` and `cdn.ynuf.aliapp.org` using the same methods. These included the device model, operating system, manufacturer, a unique device identifier, network information, as well as placeholders for GPS coordinates and the Wifi BSSID.

5.2. Shumei

Shumei uses broken cryptography to retrieve a certain configuration file containing a series of regex instructions. Network attackers can thus inject arbitrary regex commands into this configuration file downloaded by the application. The result of these regex commands is then transmitted via improved encryption. While the results of the regex commands is not directly readable, the number of matches can be inferred by the size of the encrypted payload.

We develop a PoC where a network attacker extracts content from a file readable by the application. Shumei runs regex-matching for arbitrary attacker-provided regex patterns on any readable filepath, and the number of matched patterns is leaked in the attacker-readable payload size. To leak file content, our PoC injects custom regex patterns such that file content is encoded into the number of matches.

5.2.1. Proprietary encryption. We identified two proprietary protocols used by Shumei's cryptosystem. We found that base64-encoded payloads contained in network requests received from `fp-it.fengkongcloud/v3/cloudconf` were encrypted using DES-ECB.

Second, we found that network requests sent to `fp-it.fengkongcloud/v3/profile/android` were encrypted using RSA-bootstrapped AES-CBC from the same cryptographic libraries.

5.2.2. Remote file access attack. We found that by injecting data into the DES-encrypted configuration file, an attacker can cause the client to perform various searches on their filesystem (see Figure 4 for an illustration). The results of these searches are then encrypted via RSA+AES and sent to `v3/profile/android`. Although this payload cannot be trivially decrypted, we can design the injection such that the size of the resulting payload acts as a side-channel. In this way, a network attacker can remotely identify the presence of any file that the application has permissions to read.

Specifically, the DES-encrypted configuration file fetched by Shumei contains a field "risk_files". The contents of this fieldis doubly encrypted via AES-CBC with a hard-coded key and IV.

The "risk_files" field contains a list of objects such as below:

```
1  {"path": "file:///system/lib/libriru_edxp.so",
2   "type": "file",      // or "dir"
3   "key": "mgsk7",
4   "option": "exists"  // or "match", "regex"}
```

This configuration is directing the client to perform various checks for files on the device; for instance, the check above is likely searching for the presence of Magisk, a common tool for rooting Android devices. The results of these checks are then encrypted and sent to `v3/profile/android`. One consequence of this, is that a network attacker can induce false positives or false negatives for various checks.

We found that we could infer the number of "regex" matches from the size of the resulting payload sent to `v3/profile/android` if there are matches, as the SDK includes all results from the regex search in the payload sent to `v3/profile/android`. Leveraging this vector, we can successfully DOS the client by injecting a resource-intensive regex. Of even more concern, is that an attacker can use this vector to identify the presence of files that the application has permission to read and also exfiltrate values inside of those files.

We designed a proof-of-concept attack to allow a network eavesdropper to read the BogoMIPS value from `/proc/cpuinfo` on the victim's device. Although we cannot read the client's encrypted transmissions to the endpoint `v3/profile/android`, the attack works by leaking the BogoMIPS value in the length of the payload. Effectively, the payload's length is proportional to the magnitude of the BogoMIPS value. We were able to exfiltrate the BogoMIPS value to the nearest whole BogoMIP, supporting any value from a range of 1 to 65536. Testing on two Android devices, the payload lengths varied between 3.3 and 3.4MB. This attack can be generalized for any content in file accessible by any application using Shumei.

5.3. Tencent WUP

Tencent WUP is a cryptosystem developed by Tencent, used in QQ Browser, Sogou Browser, and other applications. WUP uses two possible cryptosystems, mode 12 and mode 17, indicatedin the URL query parameters to WUP servers. Mode 12 uses RSA with no OAEP, and AES-ECB encryption with PKCS7 padding, whereas mode 17 uses RSA with

OAEP, and AES-CBC encryption with PKCS7 padding. By default, apps use mode 17. In both modes, WUP works as follows. The HTTP POST query parameter `qbkey` is the hex-encoded RSA encryption of some randomly generated 128-bit secret. This secret is then used to encrypt a payload via AES, which is serialized in the HTTP POST body.

We developed two chosen ciphertext attacks on Tencent WUP, where a network attacker can obtain the plaintext.

5.3.1. Decryption through a CBC padding oracle. For this attack, the network attacker does not need a MITM position against the victim, but does need to send queries to WUP servers. Using this technique, the attacker can decrypt any past payload that was encrypted with WUP mode 17.

When using mode 17, WUP servers return a unique error number, -89, when the padding of the decrypted plaintext is incorrect. The WUP servers return a different error number, -2, if the plaintext padding is correct but the remaining plaintext is not interpretable by their servers. Using this padding oracle, we were able to successfully and consistently decrypt payloads sent by WUP using the standard CBC attack construction with minimal modifications [34].

5.3.2. Retrieving the AES key from an RSA oracle. For this attack, the network attacker does need an active MITM position against the victim to downgrade WUP mode 17 to WUP mode 12. To query the RSA oracle, the attacker must also send queries to WUP servers. However, this vulnerability also enables attackers to decrypt any past payloads encrypted with WUP mode 12.

Ultimately, this attack leverages the malleability of RSA without padding to learn information about the encrypted AES key. Using this attack, we can recover 18.3% of AES keys.

1. Downgrading to RSA without OAEP. We found that we could downgrade clients using the WUP cryptosystem to remove OAEP during RSA encryption, from mode 17 to mode 12. An active MITM attacker can alter the response to WUP requests to report HTTP code 702. Receiving a 702 error causes the client to downgrade to mode 12, thus removing OAEP padding for future requests. After conducting the initial downgrade attack, the client will continue to use RSA without OAEP for the subsequent 24 hours. For the remainder of the attack, the attacker no longer needs to maintain their MITM network position, and can simply eavesdrop and collect client messages that are encrypted using this cryptosystem.

2. Developing RSA oracles. WUP servers return a unique error, -3, if the decrypted RSA payload is greater than 2^{128}, likely as a stop-gap measure to prevent a previous vulnerability [20]. Leveraging this, and leveraging the fact that RSA is homomorphic under modular multiplication, we can construct multiple oracles to learn information about the underlying AES key.

Multiplication oracle: For any integer k, by calculating $k^e c \pmod N$ and sending this to the WUP servers as `qbkey`, the response will tell us whether $km \geq 2^{128}$.

Division oracle: For any integer k, $k^{-e} c \pmod N$ and sending this to the WUP servers as `qbkey`, the response will tell us whether $m = 0 \pmod k$, i.e., whether m is evenly divisible by some integer k.

3. Factoring the AES key.

With the *division oracle*, we can determine all prime factors of the AES key m under some reasonable bound B. Let F be the product of all prime factors of m underneath B, and let R be the product of all prime factors of m above B. Thus, $m = FR$. As there are approximately $B \ln(B)$ primes under B, we can determine F in around $B \ln(B)$ *division oracle* queries.

After determining F, we can further narrow down the search space for the remaining R as follows. As our multiplication oracle can report whether some multiplication of $kR \geq 2^{128}$, we can binary-search k such that $(k+1)R \geq 2^{128}$ and $kR < 2^{128}$. This will take at most 128 oracle queries and bound our search space for R:

$$L_R = \frac{2^{128}}{k+1} \leq R < \frac{2^{128}}{k} = U_R$$

Finally, we brute-force the remaining possibilities for R. As a realistic benchmark for brute-forcing capabilities, our lab machine can perform approximately 2^{50} AES calculations in a day. Assuming an attacker would commit similar computational resources for a day, we model the probability of success in finding the key as $P(U_R - L_R < 2^{50}) = P(\lceil \frac{2^{128}}{R} \rceil - \lfloor \frac{2^{128}}{R} \rfloor < 2^{50})$.

We ran 100k simulations, each time choosing a new m in [0, 2^{128}) and factoring it, to estimate our probability of success for varying values of B. If we are willing to perform up to one million requests per-key ($B \approx 2^{24}$), similar to Bleichenbacher's million-message RSA attack [32], the probability of success is 18.3%. We implemented a proof-of-concept attack and successfully obtained an AES key using this method. After factoring the key and determining F, the space of the remaining bound for R was approximately 2^{43}, which we were able to brute-force using our machine within 30 minutes. We then successfully decrypted all network requests that were encrypted using this key.

5.3.3. Decrypted data. The decrypted data includes detailed device metadata and user browsing reports, including URLs of pages visited in QQ and Sogou Browser.

5.4. Remaining case studies

MobSDK. Some requests made by MobSDK used RSA-bootstrapped AES. Specifically, they use a pinned RSA public key to encrypt a randomly-generated AES key. That key is then used to encrypt the payload via AES-ECB with PKCS7 padding. The resultant ciphertext is transmitted alongside the RSA-encrypted key. However, MobSDK does not use OAEP padding in their encryption of the AES key, serializing it as the length of the key in four bytes, big-endian, followed by the key itself. This serialized data is then interpreted as a BigInteger for the RSA computation.

Within this protocol family, some POST requests were encrypted differently. This data was encrypted, also using AES-ECB, but this with the fixed key `b'sdk.commonap.sdk'`. These easily decryptable requests contained device metadata such as the device model, remaining storage on the device, device manufacturer, network carrier, screen size, as well as device memory size.

DNSPod. Payloads in the URL parameters of an HTTP GET request, as well as the response body, were encrypted using DES-ECB with PKCS7 padding. Keys were fixed per-app. In this case, payloads carried DNS requests and responses. We note that our device's operating system was configured to always use a DNS-over-HTTPS resolver.

Kuaishou SDK. HTTP POST requests to `gdfp.gifshow.com` contained encrypted data in the body. The data is encrypted first with a XOR-mask-like algorithm, and encrypted again using AES, both using the same fixed key. Pseudocode is provided in the Appendix. Although some of these POST requests used TLS, the client did not validate the server certificate, so we were able to decrypt the underlying data with a standard TLS MITM. The decrypted data contained hundreds of system settings extracted from the user's phone, as well as every `.so` module loaded by the application, and the memory addresses at which they were loaded. Payloads also contained device make, manufacturer, and network metadata.

iQIYI. iQIYI and related video apps sent HTTP POST requests to `qy.qchannel03.cn`, encrypting data via DES-CBC with PKCS7 padding. In this case, the key is derived from the first eight ASCII bytes of the MD5 lowercase hex digest of an ASCII-encoded timestamp. This timestamp, "s1", is transmitted alongside the payload in the clear. These requests contained the device manufacturer, network data, the device platform, as well as unique user IDs.

Beizi. HTTP POST requests to `sdk.beizi.biz` contain base64-encoded payloads of encrypted data in the POST request body. This data is first gzip-compressed, then encrypted using AES-CBC with PKCS5 padding, with the key `b'8iuaKct.PMN38!!1'` and IV `b'abcdefghijk1mnop'`. These requests contained metadata (such as the originating app, and its date of install, and version), device information (such as the OS, manufacturer, screen resolution), and additionally contained placeholders for advertising IDs and tracking information.

6. Discussion and recommendations

With the aid of WireWatch, we have demonstrated that (1) proprietary protocols are popular, especially in applications intended for the Chinese market, and that (2) these proprietary protocols often contain severe vulnerabilities.

On one hand, security research communities across the globe seem to understand the risk of "rolling your own cryptography". On the other hand, our analysis demonstrates that insecure proprietary network protocols continue to be massively popular. So how do we get to "TLS everywhere" in practice?

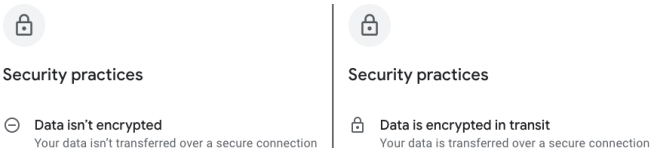

Figure 5. An example of the security disclosures on the Google Play Store.

Systemic issues require systemic solutions. We demonstrated that the use of plaintext traffic and insecure proprietary encryption is an ecosystem issue across a large portion of popular Android apps. Although we thoroughly analyzed and reverse engineered the most popular proprietary protocols, there remain over 150 protocols which we did not analyze. Thoroughly reverse engineering these protocols takes a large amount of time and effort, especially when obfuscated. As such, although WireWatch can give us a high-level overview of network security issues in the Android ecosystem, the widespread popularity of (often insecure) proprietary encryption protocols presents a significant challenge. Manually reverse engineering and reporting issues for each individual protocol limits the scale of network security vulnerabilities we can fix. Instead, the ecosystem needs systemic fixes.

In the remainder of this section, we present broader recommendations for stakeholders in the Android application ecosystem. The goal of such recommendations is to systemically address applications transmitting sensitive data that is not sufficiently encrypted.

6.1. Application stores

The Google Play Store provides "Security Practices" information for popular applications that declare whether or not data is encrypted in transit. This information is provided by the application developers, and is manually reviewed by the application store. Examples of these informational modals are shown in Figure 5. In Google Play's documentation, this is a disclosure that "all of the user data collected by your app is encrypted in transit".

64.5% of the Google Play applications that sent or received at least one plaintext request had incorrect informational modals claiming that "Data is encrypted in transit". The remainder had the "Data isn't encrypted" in their "Security Practices" information box, as expected. We note that, although applications are reviewed manually when initially submitting their Data Safety practices, application developers are expected to update their data safety information, as subsequent updates to the application are not manually reviewed [35]. App developers might accidentally introduce security regressions in future updates, such as by fetching a plaintext resource or including an SDK that makes plaintext requests.

We suggest that application stores periodically review particularly popular applications for continued compliance with their stated Data Safety practices. One could also leverage WireWatch for this purpose, to identify the use of

proprietary cryptography or plaintext network transmissions by applications at scale. Such reviews could be further-expedited or automated if the applications declare strict encryption settings and are not found to be using low-level network APIs. The existence of an UNSAFE_INTERNET permission, as discussed in the following Section 6.2, could further expedite such reviews.

We suggest that other application stores also implement network encryption disclosures in their data safety and privacy policy transparency programs. Manual review and application store disclosures are not foolproof, as a reviewer with limited time and resources cannot possibly fully guarantee the network security of an application. However, if a developer is found to be making false claims about aspects of their application, as with privacy policies, there may be modes of enforcement, via takedowns from the application stores, or via fines from local regulatory bodies. Popular Chinese app stores already claim to manually review uploaded applications for various security and privacy risks [14]. However, to our knowledge, no Chinese app store requires an "encrypted-in-transit" security disclosure by the developer similar to the Google Play Store's data safety transparency program [14].

6.2. Operating systems developers

Both Android and iOS have network security features designed to prevent the transmission of insecure network data. Specifically, Apple's App Transport Security and Android's `usesCleartextTraffic` manifest option prevent higher-level networking APIs from sending network requests that are not encrypted with TLS [36], [37].

However, there are limitations to these features. First, not all libraries respect and enforce Android's Network Security Configuration (NSC); in fact, developers often downgrade NSC protections [38]. Finally, neither feature prevents apps from using lower-level socket APIs in order to send insecure traffic. Many of the protocols we studied use these lower-level APIs, as were other vulnerable proprietary protocols studied by researchers in the past [8]. These same researchers proposed the addition of an UNSAFE_INTERNET attribute, which would be required for applications in order to use lower-level socket APIs that do not use TLS. They also note the potential drawbacks of such a permission. For instance, in the case where the operating system is compromised or no longer receiving updates, but the application is up-to-date, the application may prefer to use their own implementation of TLS. In this case, the application would still have to declare the UNSAFE_INTERNET permission, even if their network transmissions are being encrypted properly.

Given the scope of the issue at hand, we still advocate for such a permission and highlighting the usage of lower-level socket APIs. At the very least, we implore operating system developers to explore such an option that might surface whether an application is using lower-level, unencrypted socket APIs.

6.3. Developers using third-party SDKs or libraries

When including third-party SDKs or libraries to use in their applications, developers should verify or otherwise audit the security of such SDKs or libraries. Certain red flags can be apparent when setting up or configuring these SDKs. For instance, if the SDK setup requires the application to use a plaintext (HTTP) endpoint or requires the developer to generate a symmetric key in order to function, it may indicate that the SDK is transmitting data either in plaintext or using proprietary, non-standard encryption.

Similar red flags were present in the SDK documentation for the protocols where we discovered critical vulnerabilities. The Shumei SDK setup configuration asked developers to explicitly enable the `usesCleartextTraffic` attribute in their manifest file [39] and add exceptions to their Network Security Profile, which indicates the intent to either send data in plaintext or to use HTTP as a transport for proprietary encryption. Neither case is desirable and application developers should prefer to use standard encryption. In another example, both the Alibaba mPaaS SDK and Tencent DNSPod documentation mention needing to generate and configure a secret key for decryption and encryption, without any mention of TLS or CA certificates [40], [41].

If the third-party API endpoints are configurable, developers should always prefer to use API endpoints that use HTTPS, TLS, or QUIC over unencrypted endpoints or endpoints that use custom cryptography.

Android app developers can also strengthen their application network security by setting `usesCleartextTraffic` to false in their manifest file [36] and declaring a strict Network Security Policy [42]. Though this is not foolproof, as discussed in Section 6.2, this can prevent the transmission of cleartext using higher-level networking APIs. Apple's App Transport Security feature does the same for iOS apps [37].

6.4. SDK and application developers

SDK and app developers should use well-studied encryption protocols, such as HTTPS or QUIC, and avoid rolling their own cryptography. Although TLS has had its fair share of vulnerabilities over the past several decades, as it is now the de-facto standard for network encryption, it benefits from the most academic and public scrutiny compared to any other transport security protocol. Cryptography is notoriously difficult to design and implement correctly, and it is common wisdom that using more popular, higher-level APIs for encryption is less susceptible to critical vulnerabilities [43].

Developers should also validate TLS certificates. Any mobile application transmitting traffic over TLS/QUIC should pin the expected server certificate when possible, or otherwise validate the server certificate against a standard certificate trust store. In general, developers should follow best practices for transmitting network traffic securely.

6.5. Security researchers

This work and others have noted differences between security practices in the Google Play ecosystem vs. the Chinese application ecosystem [15], [16]. Although we addressed many possible avenues for remediation to close this gap, we cannot address the root of the problem if we do not know why security practices differ between the two contexts. These are important questions that could be answered by future human-centered research in the Chinese security ecosystem.

More scrutiny needs to be paid to the Chinese mobile app ecosystem in general. Though many more privacy and security researchers have been including datasets of Chinese apps in their studies, this should be the expectation, rather than the exception, for global app studies. One barrier to wider adoption is that there is not currently a stable set of methods for retrieving apps from Chinese stores. For all of the prior works we tried to build on, datasets were either not available or the techniques used to scrape the apps were outdated due to updates by the app store. Removing the barrier of having to reconstruct app datasets would increase the amount of work that studies Chinese apps in addition to Google Play Store apps.

Finally, the mobile application ecosystem at large would benefit from longitudinal telemetry. Most longitudinal measurement studies of TLS/HTTPS study encryption on the web, but to our knowledge there is no such ongoing study for mobile encryption. Prior works studying mobile privacy and security at a large scale have only been run as one-off studies. While we plan on continually running WireWatch in the future to collect longitudinal data, other studies could benefit from such measurements to identify how certain privacy and security properties trend over time and in response to interventions such as additional app store review.

7. Related work

In this section, we present related network security measurements, existing security measurements of the Android mobile application ecosystem, Android UI fuzzing, and finally, identifying and evaluating proprietary network cryptography. Our work built on methods from all these fields in the design of WireWatch.

Vulnerabilities in transport security are especially concerning since they enable third-party network attackers to compromise user privacy at scale, e.g., by enabling mass or targeted surveillance. There is a significant body of literature aimed towards measuring both the usage of standard security protocols, as well as evaluating the security of popular protocols and encryption algorithms. Researchers have historically studied vulnerabilities in SSL or TLS, the most widely used transport security protocol [44], [45]. Along this same line of research, in 2018 Böck et al. measured the practical mass-exploitability of Bleichenbacher oracles in popular SSL/TLS ciphersuites [32], [46].

Due to the universal popularity of Android devices, researchers have also devoted large efforts towards measuring the security of the Android ecosystem, either through application instrumentation or static analyses of APKs. Researchers have measured cryptographic flaws across the Google Play ecosystem [47] and studied the misuse of low-level cryptographic APIs [48]. In the case of TLS/SSL, researchers have also measured multiple pitfalls in how mobile application developers interact with its higher level APIs in practice [11], [12], [13].

WireWatch uses automated UI fuzzing methods, which is another field with extensive literature [49], [50], [51], [52]. These tools expansively explore all possible states of an application in order to discover bugs during development. The state-of-the-art in this field is leveraging LLM technology to direct GUI testing and mobile task automation [53], [54]. As such, although these systems are very complete, all prior work presents results from fuzzing under 100 applications, as the method often depends on some seeded manual interaction with the application [50], [52] or the method takes significant time and resources to fully automate each application [51], [53]. Since our goal is to scale this pipeline for testing upwards of a thousand applications, we used lighter-weight solutions that may not be as comprehensive, but succeed in generating large amounts of network traffic, similar to related pipelines designed to exercise apps in order to evaluate their privacy practices [4], [22].

Others have proposed automated reverse engineering and detection of proprietary network protocols in different contexts. In 2021, Ye et al. proposed probabilistic methods for reverse engineering proprietary network protocols used by IoT devices [55]. Researchers have also worked towards identifying encrypted network traces in the context of malware fingerprinting and identification, as malware command-and-control network requests often use proprietary encryption to avoid detection by automated intrusion detection systems [56].

8. Conclusion

Recent research has been discovering flaws in proprietary encryption used by massively popular applications, but with WireWatch, we were able to uncover the true scope of this ecosystem issue. After analyzing the most popular 18 protocols discovered in 1.7k top apps, we found that not only did most of them contain critical vulnerabilities which allowed network eavesdroppers to decrypt the underlying data, but that even the implementations of asymmetric cryptography contained basic flaws in their construction.

Despite the growing trend towards TLS everywhere, our analysis, aided by WireWatch, finds that a vast number of popular applications developed by large, well-resourced companies continue to rely on insecure cryptography to transmit sensitive user data. We hope the analysis in this work provides evidence and motivation for a stronger push to fully and securely encrypt the Internet.

Acknowledgments

The authors would like to thank Keegan Ryan and Seth Schoen for their review of the cryptographic attacks in this work. The authors would also like to thank Ron Deibert and Adam Senft for their review and guidance around coordinating vulnerability disclosures.

References

[1] Let's Encrypt, "Let's Encrypt stats," https://letsencrypt.org/stats, [Online; accessed 2024-11-10].

[2] Google Transparency Report, "HTTPS encryption on the web," https://transparencyreport.google.com/https/overview, [Online; accessed 2024-11-10].

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

[4] S. Pourali, N. Samarasinghe, and M. Mannan, "Hidden in plain sight: Exploring encrypted channels in Android apps," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2445–2458.

[5] M. Wang, P. Lin, and J. Knockel, "Should we chat, too? Security analysis of WeChat's MMTLS encryption protocol," The Citizen Lab, Tech. Rep., Oct. 2024.

[6] J. Knockel, A. Senft, and R. Deibert, "Privacy and security issues in BAT web browsers," in *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*. Austin, TX: USENIX Association, Aug. 2016.

[7] Y. Zhang, J. Shu, J. Li, Q. Wang, and D. Gu, "An empirical study of insecure communication in Android apps," in *International Conference on Wireless Communication and Network Engineering (WCNE 2016)*. DEStech Transactions on Computer Science and Engineering, 11 2016.

[8] J. Knockel, M. Wang, and Z. Reichert, "The not-so-silent type: Vulnerabilities across keyboard apps reveal keystrokes to network eavesdroppers," The Citizen Lab, Tech. Rep., Apr. 2024.

[9] Network Tradecraft Advancement Team (Five Eyes), "Synergising network analysis tradecraft," May 2012. [Online]. Available: https://www.eff.org/document/20150521-cbc-synergising-network-analysis-tradecraft

[10] D. Shin and J. Sun, "An empirical study of SSL usage in Android apps," in *2018 International Carnahan Conference on Security Technology (ICCST)*, Oct. 2018, pp. 1–5.

[11] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in)security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 50–61.

[12] L. Onwuzurike and E. De Cristofaro, "Danger is my middle name: Experimenting with SSL vulnerabilities in Android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '15. New York, NY, USA: Association for Computing Machinery, 2015.

[13] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl, "To pin or not to pin: helping app developers bullet proof their TLS connections," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 239–254.

[14] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu, "Beyond Google Play: A large-scale comparative study of Chinese Android app markets," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 293–307.

[15] B. Li, Y. Zhu, Q. Liu, Y. Sun, Y. Zhang, and L. Guo, "Wrapping DNS into HTTP(S): An empirical study on name resolution in mobile applications," in *2023 IFIP Networking Conference (IFIP Networking)*, 2023, pp. 1–9.

[16] S. Pourali, X. Yu, L. Zhao, M. Mannan, and A. Youssef, "Racing for TLS certificate validation: A hijacker's guide to the Android TLS galaxy," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 683–700.

[17] J. Dalek, K. Kleemola, A. Senft, C. Parsons, A. Hilts, S. McKune, J. Q. Ng, M. Crete-Nishihata, J. Scott-Railton, and R. Deibert, "A chatty squirrel: Privacy and security issues with UC browser," The Citizen Lab, Tech. Rep., May 2015.

[18] J. Knockel, A. Senft, and R. Deibert, "WUP! there it is: Privacy and security issues in QQ Browser," The Citizen Lab, Tech. Rep., Mar. 2016.

[19] ——, "A tough nut to crack: A further look at privacy and security issues in UC browser," The Citizen Lab, Tech. Rep., Aug. 2016.

[20] J. Knockel, T. Ristenpart, and J. R. Crandall, "When textbook RSA is used to protect the privacy of hundreds of millions of users," *arXiv*, 2018.

[21] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *Security and Privacy in Communication Networks*, R. Beyah, B. Chang, Y. Li, and S. Zhu, Eds. Cham: Springer International Publishing, 2018, pp. 172–192.

[22] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the Android permissions system," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 603–620.

[23] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471.

[24] M. Almansoori, A. Gallardo, J. Poveda, A. Ahmed, and R. Chatterjee, "A global survey of Android dual-use applications used in intimate partner surveillance," in *Proceedings on Privacy Enhancing Technologies Symposium*. Privacy Enhancing Technologies Symposium, 2022, pp. 120–139.

[25] Electronic Frontier Foundation, "apkeep - a command-line tool for downloading APK files from various sources." [Online]. Available: https://github.com/EFForg/apkeep

[26] National Institute of Standards and Technology, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," U.S. Department of Commerce, Washington, D.C., Tech. Rep. National Institute of Standards and Technology Special Publication (NIST SP) 800-22, Revision 1, 2010.

[27] J. Erman, M. Arlitt, and A. Mahanti, "Traffic classification using clustering algorithms," in *Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data*, ser. MineNet '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 281–286.

[28] O. Aouedi, K. Piamrat, S. Hamma, and J. K. M. Perera, "Network traffic analysis using machine learning: An unsupervised approach to understand and slice your network," *Annals of Telecommunications - annales des télécommunications*, vol. 77, pp. 297–309, 2022.

[29] H. Canever and X. Wang, "Network traffic classification using Unsupervised Learning: A comparative analysis of clustering algorithms," Jul. 2023, working paper or preprint.

[30] A. Rosenberg and J. Hirschberg, "V-Measure: A conditional entropy-based external cluster evaluation measure," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, J. Eisner, Ed. Prague, Czech Republic: Association for Computational Linguistics, Jun. 2007, pp. 410–420.

[31] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," in *Notices of the AMS*, vol. 46, Feb. 1999, pp. 203–212.

[32] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *Advances in Cryptology — CRYPTO '98*, H. Krawczyk, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–12.

[33] D. Boneh, A. Joux, and P. Q. Nguyen, "Why textbook ElGamal and RSA encryption are insecure," in *Advances in Cryptology — ASIACRYPT 2000*, T. Okamoto, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 30–43.

[34] S. Vaudenay, "Security flaws induced by cbc padding — applications to ssl, ipsec, wtls..." in *Advances in Cryptology — EUROCRYPT 2002*, L. R. Knudsen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 534–545.

[35] Google, "Provide information for Google Play's data safety section," https://support.google.com/googleplay/android-developer/answer/10787469?hl=en, [Online; accessed 2024-11-10].

[36] A. Klyubin, "Protecting against unintentional regressions to cleartext traffic in your Android apps," *Android Developers Blog*, Apr. 2016.

[37] Apple Inc., "Preventing insecure network connections," https://developer.apple.com/documentation/security/preventing-insecure-network-connections, [Online; accessed 2024-11-10].

[38] M. Oltrogge, N. Huaman, S. Klivan, Y. Acar, M. Backes, and S. Fahl, "Why eve and mallory still love android: Revisiting TLS (In)Security in android applications," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 4347–4364. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/oltrogge

[39] Shumei NEXTDATA, "SMSDK android documentation," https://help.ishumei.com/docs/tw/sdk/android/developDoc/, [Online; accessed 2024-11-10].

[40] Tencent Cloud, "HTTPDNS configuration information description," https://www.tencentcloud.com/document/product/1130/44467, [Online; accessed 2024-11-10].

[41] Alibaba Cloud, "Use mPaaS plug-in," https://www.alibabacloud.com/help/en/mobile-platform-as-a-service/latest/use-mpaas-plug-in, [Online; accessed 2024-11-10].

[42] A. Possemato and Y. Fratantonio, "Towards HTTPS everywhere on Android: We are not there yet," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 343–360.

[43] B. Schneier, "Schneier's Law," Apr. 2011. [Online]. Available: https://www.schneier.com/blog/archives/2011/04/schneiers_law.html

[44] C. Meyer and J. Schwenk, "SoK: Lessons learned from SSL/TLS attacks," in *Information Security Applications*, Y. Kim, H. Lee, and A. Perrig, Eds. Cham: Springer International Publishing, 2014, pp. 189–209.

[45] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the tls protocol: A systematic analysis," in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–448.

[46] H. Böck, J. Somorovsky, and C. Young, "Return of Bleichenbacher's oracle threat (ROBOT)," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 817–849.

[47] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 73–84.

[48] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic APIs," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 154–171.

[49] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 77–83.

[50] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 623–640.

[51] Y.-M. Baek and D.-H. Bae, "Automated model-based Android GUI testing using multi-level GUI comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 238–249.

[52] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "NetworkProfiler: Towards automatic fingerprinting of Android apps," in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 809–817.

[53] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024.

[54] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "AutoDroid: LLM-powered task automation in Android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, ser. ACM MobiCom '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 543–557.

[55] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "NetPlier: Probabilistic network protocol reverse engineering from message traces," in *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, 2021.

[56] C. Rossow and C. J. Dietrich, "ProVeX: Detecting botnets with encrypted command and control channels," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, K. Rieck, P. Stewin, and J.-P. Seifert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–40.

Appendix A.
Additional data from protocol clustering

In Table 4, we provide the top 50 network protocol clusters, sorted by the cumulative downloads of applications that sent requests in the cluster. We analyzed the first 18. We additionally annotated the "Unique indicator" and "Protocol family" for the data in each cluster.

In Table 5, we provide the most downloaded 80 applications affected by the vulnerabilities we discovered, with the version and names of each application.

Appendix B.
Alibaba mPaaS SDK Case Study Details

Our clustering showed that several applications owned by or affiliated with Alibaba were sending similar network requests containing base64-encoded payloads of random

	# apps	Cumul. DLs	Protocol family	Transport	Unique indicator	Identified ciphertext	Block size	Encoding	Vulnerable?
1	76	35.10B	Kuaishou	HTTP(S)*	secretkey=	“data”	128	Base64	Decryptable, TLS MITM
2	82	29.33B	MobSDK	HTTP	/dinfo	“m”	128	Base64	Decryptable
3	80	29.26B	MobSDK	HTTP	/openid	“m”	128	Base64	No OAEP
4	82	28.70B	MobSDK	HTTP	/gcl	Body	128	Raw	No OAEP
5	61	27.47B	MobSDK	HTTP	/getDuidBlacklist	“data”	128	Base64	Decryptable
6	39	26.00B	Tencent DNSPod	HTTP	dn=	“dn”	64	Hex	Decryptable
7	36	25.38B	MobSDK	HTTP	d/dgen	Body	128	Base64	Decryptable
8	68	22.31B	MobSDK	HTTP	/gcf	“sc”	128	Base64	Decryptable
9	2	19.83B	Alibaba mPaaS	HTTP	/g9m6	Body	128	Base64	Decryptable
10	7	17.66B	Tencent WUP [18]	HTTP	qbkey=	“qbkey”, body	128	Raw	Decryptable, No OAEP
11	7	17.66B	Tencent WUP	HTTP	tk=	Body	128	Raw	Decryptable (padding oracle)
12	3	17.58B	Kuaishou	HTTP	apissl.ksapisrv	“device_info”	128	Hex	Decryptable
13	13	17.55B	Alibaba mPaaS	HTTP	mobileDispatch	“secData”	128	Base64	Decryptable
14	4	12.65B	iQIYI	HTTP	qchannel03.cn	“data” param	64	Base64	Decryptable
15	37	10.52B	Shumei SDK	HTTP	v3/cloudconf	“data”	128	Base64	Decryptable
16	22	9.23B	Shumei SDK	HTTP	v3/profile/android	“fingerprint”	128	Base64	Side-channel for file access
17	1	9.23B	Tencent MMTLS [5]	HTTP,TCP	f1 04 00	Body	128	Raw	
18	40	7.73B	Beizi SDK	HTTP	sdk.beizi.biz	Body	128	Base64	Decryptable
19	2	7.58B	Weibo	HTTP	encry_params	“encry_params”	64	Base64	
20	3	7.42B	UC Browser	HTTP	puds	Body	Varies	Raw	
21	3	7.42B	UC Browser	HTTP	ucid=	Body	Varies	Raw	
22	1	7.30B	Weibo	HTTP	ad/preload	“sdk_ad_params”	128	Hex	
23	18	7.20B	MobSDK	HTTP	/api/log	Body	128	Base64	
24	25	6.30B	Hubcloud	HTTP	api/sdk/task/list	Body	128	Base64	
25	3	6.29B	iQIYI	HTTP	/drm/register	“cert”	Varies	Base64	
26	1	6.26B	MobTech	HTTP	/duc/conf	Body	128	Base64	
27	1	6.26B	iQIYI	HTTP	/mixer	“ed”	128	Base64	
28	3	6.04B	sm.cn	HTTP	/sdk_log	Body	128	Raw	
29	9	5.98B	data.lianjia.com	HTTP	/report?	Body	Varies	Raw	
30	2	5.70B	UC Browser	HTTP	usquery.php	Body	Varies	Raw	
31	8	5.43B	MobTech	HTTP	/api/pv	Body	128	Base64	
32	7	5.24B	XYCloud	HTTP	/psdk_param	Body	128	Raw	
33	7	5.24B	XYCloud	HTTP	live_p2p_mobilesdk	Body	128	Raw	
34	1	5.01B	Baidu Map	HTTP	/cloudmodule	Body	128	Base64	
35	2	4.69B	WifiLocating	HTTP	/trackData/collect	Body	128	Raw	
36	2	4.63B	QQ	HTTP,TCP	13 00 00	Body	Varies	Raw	
37	1	4.45B	WifiLocating	HTTP	/fcompb.pgs	Body	Varies	Raw	
38	22	4.32B	MobSDK	HTTP	/v3/bind	Body	128	Base64	
39	1	4.25B	Ximalaya Music	HTTP	/xrc/rt/v1	Body	Varies	Raw	
40	1	3.30B	QQ Music	HTTP	vkey=	“vkey”	64	Hex	
41	29	3.18B	Tencent Bugly	HTTP	bugly.qq.com	Body	Varies	Raw	
42	1	3.15B	QQ News	HTTP	diffvmind	“vkey”	64	Base64	
43	24	2.58B	Shumei	HTTP	deviceprofile/v4	“data”	128	Base64	
44	1	2.55B	Netease Music	HTTP	ad/loading/current	“key”	128	Hex	
45	15	2.29B	Tencent CDN	HTTP	license/v1	“encryptedLicense”	128	Base64	
46	1	2.11B	Dianping	HTTP	getlivepushdata	Body	128	Raw	
47	4	2.01B	shuzilm.cn	HTTP	/valid?	Body	64	Raw	
48	2	1.95B	mantis.bayescom	HTTP	sdkevent	JSON “device”	128	Base64	
49	9	1.73B	Tencent	HTTP	log.tbs.qq	Body, “key”	64	Raw	
50	1	1.72B	Quark Browser	HTTP	pdds-cdn	Body	64	Raw	

TABLE 4. TOP 50 PROTOCOL CLUSTERS, BY CUMULATIVE APPLICATION DOWNLOAD COUNT, ANNOTATED WITH UNIQUE *INDICATORS* THAT ARE PRESENT IN ALL REQUESTS IN THE CLUSTER AND NOT IN OTHERS. NOTE THAT BELOW CLUSTER 18, THE “PROTOCOL FAMILY” COLUMN IS UNCONFIRMED, AND WERE ANNOTATED BASED ON EITHER THE ORIGINATING APPLICATION(S) OR THE COMMON DESTINATION IP ADDRESSES/DOMAIN NAMES OF EACH PROTOCOL.

data, aligned to AES block sizes. We reverse engineered three applications that we observed making these requests:

APK package name	App name	Version
`com.taobao.taobao`	Taobao	10.35.10
`com.tmall.wireless`	Tmall	15.20.0
`com.UCMobile`	UC Browser	17.0.0.1331

These payloads are protected by a common cryptosystem provided by Alibaba’s mPaaS SDK, implemented in `libsgmain*.so`. At a high level, this cryptosystem effectively encrypts payloads via AES-CBC using a static key. The keys are stored in a local image resource file: `res/drawable/yw_1222.jpg`. We provide code for extracting the key database from any application using this SDK, de-obfuscating `libsgmain*.so`, and decrypting payloads that are encrypted in this manner.

B.1. De-obfuscating mPaaS

The native code obfuscation methods used by `libsgmain*.so` required us to develop custom methods to de-obfuscate and analyze these libraries, including IDA plugins.

De-obfuscating cryptographic constants and string literals. To obfuscate their usage of cryptographic constants and string literals, the `libsgmain*.so` stored them, encrypted, in its data section. We dumped the entire `.bss` section of the binary at runtime after observing the behavior we were interested in further analyzing, allowing us to load the data from a dump for a selected string literal, array, or other value to see the decrypted values of these encrypted constants during analysis. This had the additional benefit of, when encountering constants that had not been decrypted in our dump, allowing us to quickly conclude that they and the code referencing them was unrelated to whatever behavior we were trying to analyze.

De-obfuscating jumps and function pointers. `libsgmain*.so` contained indirect branches whose destination addresses were the result of complex arithmetic computations to confuse disassemblers and decompilers. To address this, we patched all artificially introduced indirect branches with corresponding direct, PC-relative branches. Though IDA's decompiler and disassembler were often sophisticated enough to fold the complex arithmetic computations calculating the indirect jumps' destination addresses into constants, both would mistakenly interpret such jumps as function boundaries. To fix IDA's function analysis, we used the destination address computed by IDA's decompiler or disassembler to replace these indirect jumps with corresponding direct, PC-relative jumps.

B.2. Deriving encryption keys

A database of encryption keys is stored, encrypted, in the static resource file `res/drawable/yw_1222.jpg`. This database can also be decrypted with publicly available key material, as follows. First, a 32-byte key k_m is derived from the app's APK RSA key: a null byte followed by the first 31 bytes of the public bytes of the APK RSA key when DER encoded and serialized in PKCS#1 format. The header of `res/drawable/yw_1222.jpg` contains three IVs, v_1, v_2, and v_3, followed by the encrypted database ciphertext c.

Let $\mathrm{D}(\mathit{ciphertext}, \mathit{key}, \mathit{iv})$ be AES-CBC decryption. The plaintext p is derived as

$$p = \mathrm{D}(\mathrm{D}(\mathrm{D}(\mathrm{D}(c, k_1, v_1), k_2, v_2), k_3, v_3), k_1, v_1), \text{ where}$$
$$k_1 = k_m[0{:}24],\ k_2 = k_m[0{:}16], \text{ and } k_3 = k_m[0{:}32].$$

After being zlib-inflated, p can be parsed as a serialized dictionary of keys and values. The dictionary values are then used to derive keys for encryption.

Let $\mathrm{E}(\mathit{ciphertext}, \mathit{key}, \mathit{iv})$ be AES-CBC encryption. Alibaba mPaaS encrypts plaintexts p to ciphertexts c with keys and IVs derived from the MD5 hex digest of a dictionary value e such that

$$c = \mathrm{E}(p, k, k), \text{ where } k = \mathrm{hex}(\mathrm{md5}(e))[0{:}16].$$

As these keys are all derived from public information, we can automatically extract the key database from `res/drawable/yw_1222.jpg`, derive all possible keys, and decrypt any requests encrypted using this cryptosystem.

Appendix C. Kuaishou SDK Case Study Details

HTTP POST requests sent to `gdfp.gifshow.com` contain JSON payloads in the POST request body. These JSON blobs contain base64-encoded encrypted data aligned to AES block sizes.

Requests to `/kuaishou/temp/data/s` and `/r/t/h` were over unencrypted HTTP, and requests to the endpoint `/f/a/p` were additionally tunneled via TLS (i.e., were HTTPS POST requests). However, the client SDK never validates the server's TLS certificate. We developed code to decrypt Kuaishou SDK requests, and additionally developed a PoC to MITM any HTTPS connection to this domain and decrypt the data within.

Kuaishou's encryption was reverse engineered from Kuaishou's native library `libweapon*.so`, which is loaded dynamically by the application.

Decrypting the payload. The network request payload is first gzip-compressed, then encrypted twice. It is first encrypted with a custom XOR-mask-like algorithm, and encrypted again using AES-CBC with PKCS7 padding, both using the same key. In the case of AES-CBC, the key is also re-used as the IV. We note that the XOR-mask-like algorithm is symmetric, e.g., $\mathrm{D}_{\mathrm{xor}}(\mathrm{E}_{\mathrm{xor}}(p, k), k) = p$. Below is Pythonic pseudocode for this algorithm:

```
def expand_key(key):
  xorkey = list(range(0, 256))
  keyi = 0, j = 0
  for i in range(0, 256):
    j = key[keyi] + xorkey[i] + j & 0xff
    xorkey.swap(i, j)
    keyi = (keyi + 1) % len(key)
  return xorkey

def xor_decrypt(ciphertext, key):
  xorkey = expand_key(key)
  o = [], keyi = 0, j = 0
  for i in range(0, len(ciphertext)):
    keyi = (keyi + 1) & 0xff
    j = (xorkey[keyi] + j) & 0xff
    xorkey.swap(keyi, j)
    o += xorkey[(xorkey[keyi] + xorkey[j])
                & 0xff] ^ ciphertext[i]
  return bytes(o)
```

HTTP POST requests to the endpoints `/r/t/h` and `/kuaishou/temp/data/s` use the key `b'tV3779net2y0VOEs'`, and HTTPS POST requests to `/f/a/p` use the key `b'ksriskctlbusinss'`.

Although the requests to `/f/a/p` are additionally encrypted using TLS, the client does not validate the TLS server certificate for these requests. These can be intercepted and decrypted by network adversaries, e.g., by presenting a certificate owned by the attacker to conduct a TLS MITM.

Decrypted data payloads. Data decrypted in this way included hundreds of system settings extracted from the user's phone, as well as every `*.so` module loaded by the application, and the memory addresses at which they were loaded. The payloads also contained device make, manufacturer, and network information.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

The paper introduces WireWatch, an automated framework to detect proprietary network encryption in Android apps. The approach relies on scraping apps from marketplaces, executing the apps via automated UI interaction, capturing network traffic, and applying unsupervised k-means clustering to identify non-standard cryptographic protocols. The paper provides manual validation of clustering and custom protocol detection. Furthermore, by extensive manual reverse engineering efforts, the paper identifies impactful and widespread vulnerabilities, especially in the Mi Store ecosystem.

D.1.1. Scientific Contributions.

- Independent Confirmation of Important Results with Limited Prior Research
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

D.2. Reasons for Acceptance

1) While previous studies have reported the usage of custom cryptographic protocols, their scope has been limited and lacked comprehensive comparison between the Google Play Store and Chinese app markets. This paper presents a large-scale analysis of custom encryption protocols focusing on the Google Play and Mi stores, significantly expanding on prior work.
2) The paper identifies cryptographic vulnerabilities in apps with a large user base. Vulnerable apps are prominent in the Mi Store (47% use plaintext traffic or insecure proprietary encryption).

D.3. Noteworthy Concerns

1) **Limited dataset.** Although the analysis focuses on the most downloaded apps from the Google Play and Mi stores, the dataset only includes the top 1K for each store. Out of these 2K apps, the framework successfully executes on 1,699 apps. These represent only a fraction of the available apps available on the stores, which could limit the generalizability of the results.
2) **Scalability.** While the proposed framework is automated and scalable at detecting custom encryption protocols, the paper heavily relies on manual reverse engineering efforts to identify vulnerabilities in the encryption schemes. WireWatch can help security analysts prioritize which protocols to evaluate, but it does not provide insights into the actual vulnerabilities of the custom protocols used by the processed apps.

APK name	Version	Name	Translated name	DL count	Affected by vulnerabilities
com.xunmeng.pinduoduo	7.22.0	拼多多	Pinduoduo	17B	DNSPod
com.smile.gifmaker	12.7.20.38014	快手	Kuaishou	11B	Kuaishou
com.taobao.taobao	10.39.10	淘宝	Taobao	11B	MPaaS
com.eg.android.AlipayGphone	10.6.28.8000	支付宝	Alipay	9B	MPaaS
com.kuaishou.nebula	12.7.20.8502	快手极速版	Kuaishou Lite	7B	Kuaishou
com.qiyi.video	15.8.0	爱奇艺-《九部的检察官》独播	iQIYI	6B	iQIYI
com.UCMobile	17.0.0.1331	UC浏览器-好搜好看好好用	UC Browser	6B	Kuaishou
com.xingin.xhs	8.49.0	小红书	Xiaohongshu	5B	MobSDK, Shumei
com.snda.wifilocating	5.0.55	WiFi万能钥匙-安全省流	Wifi MasterKey	4B	MobSDK
com.ximalaya.ting.android	9.2.78.3	喜马拉雅-818宝藏会员节	Himalaya Music	4B	MobSDK, iQIYI
com.wuba	13.11.5	58同城-招聘找工作租房家政买车	58.com	3B	MobSDK
com.baidu.netdisk	12.13.10	百度网盘	Baidu Netdisk	2B	BeiziSDK
com.taobao.idlefish	7.17.10	闲鱼	Xianyu	2B	MPaaS
com.lemon.lv	14.5.0	剪映	CapCut	2B	BeiziSDK
com.taobao.litetao	10.32.49	淘宝特价版	Taobao Special Edition	1B	MPaaS
cn.kuwo.player	10.9.1.1	酷我音乐	Kuwo Music	1B	iQIYI
com.tencent.karaoke	8.24.38.278	全民K歌	National karaoke	1B	DNSPod
com.duowan.kiwi	12.2.45	虎牙直播	Huya Live	1B	MobSDK
com.tmall.wireless	15.30.0	天猫	Tmall	1B	MPaaS
com.shoujiduoduo.ringtone	8.9.76.0	铃声多多-铃声彩铃壁纸来电秀	Lots of Ringtones	1B	Kuaishou
com.kmxs.reader	7.54	七猫免费小说	Seven Cats Free Novel	1B	MobSDK, Shumei
com.shizhuang.duapp	5.48.1	得物-得到运动x潮流x好物	Dewu Shopping	1B	MobSDK
com.baidu.tieba	12.67.1.0	百度贴吧	Baidu Tieba	978M	Kuaishou
com.cubic.autohome	11.65.3	汽车之家-5亿人都在用的汽车App	Autohome	951M	DNSPod, Shumei
com.xunlei.downloadprovider	8.20.0.9405	迅雷	Xunlei Downloads	920M	Kuaishou, MobSDK
com.moji.mjweather	9.0878.02	墨迹天气	Moji Weather	913M	MobSDK
com.sinovatech.unicom.ui	11.7.2	中国联通	China Unicom	910M	MobSDK
com.tencent.wemeet.app	3.28.11.475	腾讯会议	Tencent Meetings	888M	DNSPod
com.tencent.map	10.10.5	腾讯地图	Tencent Map	840M	DNSPod
com.jifen.qukan	3.20.60.000.0815.0658	趣头条	Qutoutiao	833M	Kuaishou, MobSDK
com.alibaba.wireless	11.31.2.0	阿里巴巴	Alibaba	807M	MPaaS
air.tv.douyu.android	7.7.9.1	斗鱼-娱乐游戏直播平台	Douyu	765M	MobSDK, Shumei, iQIYI
com.sohu.sohuvideo	10.0.55	搜狐视频-长梦留痕全网独播	Sohu Video	763M	DNSPod
com.handsgo.jiakao.android	8.62.0	驾考宝典-驾校学车考驾照优选app	Driving test guide	755M	Kuaishou
com.tencent.wework	4.1.28	企业微信	Enterprise WeChat	683M	DNSPod
com.hpbr.bosszhipin	12.14	BOSS直聘	BOSS direct recruitment	576M	DNSPod
com.wuba.zhuanzhuan	10.43.0	转转-二手官方验	Zhuanzhuan-Second-hand	551M	MobSDK
com.xiachufang	8.8.40	下厨房	Go to the kitchen	532M	Kuaishou
com.le123.ysdq	5.9.9	影视大全	Film and television collection	511M	Kuaishou, BeiziSDK
com.duowan.mobile	8.44.3	YY	YY	423M	MobSDK, BeiziSDK
com.qiyi.video.lite	4.8.30	爱奇艺极速版-九部的检察官独播	iQIYI Express	413M	Kuaishou
com.fenbi.android.solar	11.56.0	小猿搜题	Ask Xiaoyuan	381M	DNSPod
com.taobao.live	3.63.18	点淘-淘宝直播官方APP	Diantao - Taobao Live	378M	MPaaS
com.duoduo.child.story	6.5.1.3	儿歌多多-18亿父母推荐	Lots of children's songs	370M	Kuaishou
com.kuaiduizuoye.scan	6.31.0	快对-拍照翻译	Kuaidui	365M	Kuaishou
cn.xiaochuankeji.tieba	6.2.1	最右	Rightmost	362M	Shumei
com.p1.mobile.putong	6.3.8.1	探探	Tantan	355M	MobSDK
cmccwm.mobilemusic	7.41.15	咪咕音乐-让音乐更有趣	Migu Music	353M	BeiziSDK
com.shuqi.controller	12.2.1.219	书旗小说-海量图书	Shuqi Novel	345M	DNSPod
com.kuaikan.comic	7.71.0	快看漫画-快看，你的漫画乐园	Kuaikan Comics	340M	Kuaishou
com.sup.android.superb	5.0.7	皮皮虾	Pipi Shrimp	330M	Kuaishou
com.ifeng.news2	7.78.1	凤凰新闻	Phoenix News	321M	Shumei
com.didapinche.booking	9.64.1	嘀嗒出行-顺风车出租车用嘀嗒	Dida Travel	303M	Shumei
com.hupu.shihuo	7.98.0	识货	Know the goods	286M	MobSDK
com.lalamove.huolala.client	6.9.81	货拉拉-拉货搬家跑腿	Lalamove	259M	DNSPod
com.chaoxing.mobile	6.3.3	学习通	learning pass	249M	MobSDK
com.bokecc.dance	8.3.6	糖豆	jelly beans	248M	Kuaishou, MobSDK
com.snda.lantern.wifilocating	6.8.28	WiFi万能钥匙极速版	WiFi master key speed version	243M	MobSDK
com.tongcheng.android	10.9.1.1	同程旅行-单单返现金	Travel on the same journey	240M	MobSDK, DNSPod
com.mygolbs.mybus	6.6.8	掌上公交-精准实时公交查询	Palm Bus	222M	Kuaishou, MobSDK, BeiziSDK
com.wudaokou.hippo	6.9.0	盒马	Hema	218M	MPaaS
cn.etouch.ecalendar	9.2.6	微鲤万年历-原中华万年历	Micro Carp Perpetual Calendar	216M	Kuaishou
com.ganji.android.haoche_c	10.10.0.6	瓜子二手车-先试3天再买车	Guazi used cars	208M	DNSPod
com.douban.frodo	7.82.0	豆瓣	Douban	206M	DNSPod
com.jiongji.andriod.card	7.6.8	百词斩-学英语、背单词必备	Baicizhan	191M	MobSDK
com.mymoney	13.1.97.0	随手记-记账就用随手记	SuiShouJi	186M	MobSDK
com.kwai.m2u	4.36.0.43602	一甜相机	A sweet camera	183M	Kuaishou, BeiziSDK
com.mampod.ergedd	4.1.2	儿歌点点-宝宝儿歌故事动画大全	Children's Songs Diandian	177M	Kuaishou
com.ushaqi.zhuishushenqi	4.85.61	追书神器	Book chasing artifact	175M	Kuaishou, Shumei
com.max.xiaoheihe	1.3.331	小黑盒	little black box	173M	Shumei
com.fenbi.android.leo	3.89.1	小猿口算-1秒检查作业	Little Yuan's oral arithmetic	165M	DNSPod
com.huajiao	9.3.0.1038	花椒直播	Huajiao Live	152M	MobSDK, Shumei
com.fenbi.android.servant	6.17.34.1	粉笔	chalk	147M	DNSPod
com.alicloud.databox	6.1.0	阿里云盘	Alibaba cloud disk	144M	MPaaS
com.baidu.baidutranslate	11.5.1	百度翻译	Baidu Translate	144M	BeiziSDK
com.babytree.apps.pregnancy	9.63.0	宝宝树孕育-备孕怀孕育儿软件	BabyTree Pregnancy	143M	BeiziSDK
com.mfw.roadbook	11.1.9	马蜂窝	hornet's nest	136M	MobSDK
com.cctv.yangshipin.app.androidp	3.0.1.24816	央视频	CCTV	130M	DNSPod

TABLE 5. MOST DOWNLOADED 80 APPS AFFECTED BY THE VULNERABILITIES WE DISCOVERED, WITH THE VERSION AND NAMES OF EACH APPLICATION. THE NAMES ARE MACHINE-TRANSLATED WITH SOME CORRECTIONS BY THE AUTHORS.