A Closer Look at QUIC Traffic: Characterizing QUIC Usage and Privacy Issues in the Wild

Shaoqi Jiang
Concordia University
shaoqi.jiang@concordia.ca

Mohammad Mannan
Concordia University
m.mannan@concordia.ca

***Abstract*—The availability of modern proxy tools has enabled more detailed analysis of application-layer traffic. Existing research has shown that open-source tools like mitmproxy are effective in observing the inner content of on-the-fly traffic, especially in HTTP and HTTPS requests. However, with HTTP/3 being increasingly adopted in both apps and web services, new challenges are posed because QUIC, the foundational protocol for HTTP/3, lacks full support in widely-used open-source mitmproxy versions, which significantly hinders comprehensive research on HTTP/3 traffic within mobile applications. To address this limitation, we develop QuicMitm, a specialized QUIC man-in-the-middle proxy. Our proxy can observe plaintext HTTP/3 based on QUIC and handle HTTP requests from Android mobile apps. Using QuicMitm, we tested 3,452 popular apps to observe their HTTP/3 traffic. Also, we compared the privacy-related information leakage of HTTP/3 and HTTP/2 in these apps. Our observations provide a glance of the real-world prevalence of QUIC usage across mobile applications. We hope that our tool can assist researchers in conducting large-scale, dedicated measurements and analysis of QUIC-transmitted content.**

I. Introduction

QUIC [19] has been rapidly adopted by major platforms such as Google, YouTube, Facebook, Cloudflare, and Akamai. It integrates reliable delivery, congestion control, and TLS encryption directly into its UDP-based transport layer, unifying what TCP and TLS provide separately into a single protocol stack. By multiplexing independent streams over one connection to eliminate TCP's head-of-line blocking, and folding the TLS handshake into the transport handshake (achieving 1-RTT setup), QUIC delivers lower latency and greater performance. However, these innovations also mean existing interception tools—built for TLS over TCP—cannot simply "plug into" QUIC; they require bespoke implementations of QUIC's congestion control, stream framing, and in-protocol TLS, along with custom configuration to parse and re-encrypt QUIC packets.

Standard man-in-the-middle (MITM) tools like mitmproxy excel at intercepting TCP-TLS traffic but only offer limited QUIC support, nor does Burp Suite provide any official support for QUIC interception. When competing with modern applications that contact multiple or dynamic domains, existing tools cannot flexibly decrypt QUIC content or inspect HTTP/3 traffic, preventing researchers from effectively analyzing traffic behavior on mobile platforms. Since HTTP/3 is built on top of QUIC, in this study, we treat HTTP/3 and QUIC traffic as equivalent for simplicity, unless otherwise noted.

Although extensive research has examined mobile app privacy leaks–mostly focusing on network traffic or application behavior [34], [35], [47], [23], [10], [48], [25]–the content of QUIC traffic is often overlooked. This issue becomes increasingly critical as modern mobile apps rely more heavily on QUIC for transport, much of which is served by major providers (e.g., Google) [27]. As a result, potential privacy leaks in mobile apps remain unexplored in this channel. Other studies related to QUIC security focus more on the protocol itself (e.g., [13], [26], [41]) and its fingerprint feature in transport layer (e.g., [40], [39]).

Traditional interception proxies for analyzing mobile app traffic, such as mitmproxy, typically require prior knowledge of the endpoints requested by the apps to decrypt or inspect QUIC connections. This requirement limits scalability and generality, motivating the need for flexible proxies capable of handling QUIC's protocol intricacies without prior knowledge of app-requested endpoints. To address this challenge, we propose *QuicMitm*, a QUIC-specific MITM proxy implementation, based on existing QUIC protocol stacks. QuicMitm uses a popular QUIC library to intercept and process encrypted communications through three key mechanisms: (i) dynamic generation of trusted certificates for QUIC-based TLS in client authentication, (ii) establishment of dual QUIC tunnels (client-proxy and proxy-server) and (iii) frame-ordered stream management for each QUIC connection. We validated the proxy through comprehensive testing including mobile device traffic interception, demonstrating practical feasibility under real-world network conditions.

Our key contributions include:

1) We design and implement QuicMitm, a novel MITM proxy tailored to handle HTTP/3, HTTP/2, and HTTP/1.1 traffic.
2) We conducted measurements across 3,452 mobile apps of varying popularity to analyze their QUIC usage, which provide a characterization of the endpoint ecosystem in real-world deployments.

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-06-0
https://dx.doi.org/10.14722/madweb.2026.23024
https://www.ndss-symposium.org

3) Using QuicMitm, we conducted traffic inspection and analysis in HTTP-level on these popular mobile apps and identified potential privacy leaks in their HTTP/3 traffic. These potential privacy leaks in HTTP/3 traffic are comparable to those we observed in their HTTPS traffic. Our code is available at: https://github.com/Madiba-Research/QuicMitm.

II. BACKGROUND

A. QUIC overview

QUIC leverages UDP to integrate transport reliability, congestion control, and TLS encryption into a single modular stack [22]. Connections begin with Initial packets trading cryptographic material, then proceed to the Handshake phase. Handshake packets negotiate TLS parameters, derive encryption keys, and transition the connection into secure application data transmission, often carrying data immediately after handshake completion. By multiplexing independent streams–each with its own flow control and loss recovery–QUIC avoids TCP's head-of-line blocking, where a lost packet on one stream can block the delivery of subsequent data, and reduces setup latency. Throughout the transmission, the headers and payloads of different types of packet are encrypted using either fixed values or values exchanged between the client and the server.

B. Mitmproxy on HTTP/3

MITM proxy acts as an interactive man-in-the-middle by generating dynamic SSL/TLS certificates, decrypting HTTPS traffic, and re-encrypting it for onward delivery [8]. Currently, the widely used MITM proxy, the mitmproxy project [8] supports HTTP/1.1, HTTP/2, other protocols such as WebSocket and DNS, and HTTP/3 partially. Leveraging the aioquic library [9], [1], mitmproxy enables HTTP/3 interception across three of its proxy modes: reverse-proxy, local capture, and WireGuard [9].

The reverse-proxy mode is bound to a single, preconfigured domain [17]. All client requests and server responses are funneled through that bound domain, limiting the monitoring of real-world mobile and web applications, which typically issue HTTP requests to multiple dynamic domains during runtime. Moreover, in the presence of CDNs, such dynamic domain requests cannot be known in advance.

In the local mode, only QUIC flows from applications on the same host can be intercepted. In WireGuard mode, mitmproxy spins up a user-space WireGuard VPN server and provides a matching client profile; by installing this configuration on their WireGuard clients, devices can route their QUIC traffic through mitmproxy. TCP and UDP traffic through WireGuard is forwarded and parsed by mitmproxy, where QUIC traffic can only be forwarded without internal inspection. Consequently, HTTP/3 interception in these modes is recommended only for simple cURL-based tests, and remains prone to bugs in diverse real-world scenarios [9]. These limitations leave most QUIC traffic beyond reach of TLS-decrypted inspection.

C. Threat model

We investigate whether some apps or APIs deliberately use QUIC as a dedicated/stealth channel to carry specific information (e.g., sending sensitive payloads "only over QUIC" to bypass traditional traffic-auditing workflows that mainly rely on visibility into HTTP/1.1 or HTTP/2). We do not consider attacks that require compromising application servers, tampering with cloud infrastructure, or breaking the cryptographic primitives of TLS 1.3/QUIC; instead, we focus on data exposure and evasion behaviors that can occur under existing trust configurations and within normal system capabilities over standard network connectivity.

Malicious profiling app. The attacker develops and distributes apps that collect privacy-sensitive information on the client device. To evade existing dynamic-analysis and traffic-auditing tools (e.g., mitmproxy that primarily target HTTP/1.1 or HTTP/2 interception and parsing), the app uploads its sensitive or high-value payloads (e.g., user identifiers, profiling feature vectors, contacts/location, or tracking/attribution event logs) exclusively over a QUIC/HTTP3 channel that is harder to observe and decrypt, to adversary-controlled backends or third-party data-collection endpoints. This can cause privacy leakage to be underestimated or missed by conventional traffic observation, making malicious collection harder to detect and attribute, and increasing the risk of sensitive data exfiltration through opaque channels.

III. RELATED WORK

Early QUIC measurement studies primarily focused on controlled testbeds and transport-layer performance characteristics, such as throughput, latency, congestion control behavior, and protocol feature deployment [31], [38], [12], [33], [28], [37], [21]. More recent work has gradually shifted toward examining QUIC traffic content and its privacy implications:

Jonas et al. [32] inferred deployment strategies of large CDNs through passive observation. By inspecting packet-level features (i.e., UDP to port 443), Madariaga et al. [27] measured QUIC usage across mobile apps, highlighting its growing prevalence in transport protocol level. Elmonhorst et al. [11] performed a censorship analysis on HTTP/3, revealing dedicated UDP blocking and major IP blocklisting affecting QUIC in some regions. Zohaib et al. [49] recently showed that the GFW has developed the capability to censor QUIC traffic. Through detailed experiments, they indicated the GFW's ability of inspecting the SNI in QUIC ClientHello messages and the rules adopted to block the associated UDP packets.

For privacy/application analysis, Li et al. [23] used mitmproxy to intercept and inspect network traffic. They analyzed intercepted network traffic to identify user data collection behaviors, which were compared with the runtime privacy notices (RPNs) provided by the apps to check for compliance with GDPR requirements. Pourali et al. [35] used mitmproxy to intercept and discover custom encrypted data in the TLS channel, and then identified custom encrypted plaintext by comparing the input and output of standard encryption functions called when the app was running. These studies excluded

QUIC due to their dependence on mitmproxy. Similarly, FaceTime's semantic data traffic (over QUIC) as analyzed by Cheng et al. [6], was affected by mitmproxy's inability to decrypt QUIC; HTTP/3 requests were removed from the mobile browser traffic inspected by Pegioudis et al. [34].

IV. DESIGN METHODOLOGY

In this section, we first present the design of our QuicMitm. Then, using this proxy, we describe the design of our Android app testing framework.

A. Proxy Design with QUIC Adaption

QuicMitm operates between the client apps and the target server, capturing packets on both TCP port 443 and UDP port 443 to handle TLS and QUIC flows, respectively.

Initialization of TLS/QUIC Interception. In the initialization stage, the proxy verifies the presence of a valid local Certificate Authority (CA) certificate, which should also be installed in the trusted root store of testing device. If the certificate does not exist, or cannot be successfully parsed by the X.509 parser in our certificate tool,[1] the proxy will generate a valid self-signed CA certificate and terminate itself, giving a chance to install the new CA certificate into the trusted root certificate store of the client device.

Hosted by QuicMitm, the CA certificate underpins subsequent TLS and QUIC traffic interceptions. Upon receiving a TLS handshake (or TLS handshake encapsulated in QUIC) from the client, the proxy extracts the Server Name Indication (SNI) from the `ClientHello` message. It then generates a server certificate with that SNI as the Common Name (CN), signs it with the CA's private key, and presents it to the client for subsequent TLS verification. As we installed the CA certificate into the device's system root store, the device trusts it. When the device receives our generated server certificate signed by that CA, it accepts it and establishes a TLS connection.

Protocol Adaption. QuicMitm intercepts both TCP and UDP traffic sent from the mobile device. Upon receiving a TCP connection from the client, the proxy accepts it as a TLS stream and performs the TLS handshake.

In addition, as the Distinguished Name (DN) of generated server certificate, the proxy also uses the SNI as the actual destination when establishing a valid connection with the server. To establish a TLS connection with the server, the proxy utilizes *webpki-roots* to obtain the trusted root certificates for server authentication. The *webpki-roots* library [36] contains Mozilla's trusted root certificates, ensuring a valid TLS handshake between the proxy and the server. We also extract the ALPN (Application-Layer Protocol Negotiation) extension from the `ClientHello` message, which a client and a server to negotiate the application-layer protocol to be used over the secure connection. For a TCP connection, we check if the value of ALPN is HTTP/1.1 or HTTP/2, and pass the HTTP request to its version-specified handlers.

When the proxy receives UDP on port 443, the proxy accepts it as QUIC connection. Since QUIC has a built-in TLS module, the intercepting process is similar to the TLS handshake mentioned above, including extracting the SNI from the client, completing the TLS authentication with the client, and establishing a QUIC connection with the server. The proxy then checks if the value of ALPN is HTTP/3. Once confirmed, it passes the connection request to a dedicated handler specifically designed for HTTP/3 traffic.

QUIC Stream Multiplexing. QUIC's multiplexing feature enables multiple streams to operate concurrently within a single connection. In this case, multiple HTTP/3 requests share one QUIC connection. Our HTTP/3 handler is designed to manage this stream multiplexing. When a new stream is received, the handler establishes a corresponding stream on the server-side connection. This ensures that data are transmitted through the correct stream, maintaining a one-to-one mapping between client and server streams.

In QUIC, each stream is divided into frames for transmission, and each frame is assigned a unique sequence number to guarantee correct reassembly at the receiver. To ensure the sequence of stream frames, the proxy buffers the complete HTTP/3 payload for each client stream. The proxy then forwards the payload to the server via the corresponding stream. This mechanism prevents frame reordering and data loss, allowing for reliable transport of HTTP/3 traffic over QUIC. It also benefits us from logging the complete HTTP/3 plaintext in the correct order.

Proxying Environment. We focus on the traffic proxying of Android device. After installing the CA certificate generated in the initial stage on the device, we connected the host running the proxy and the test phone to the same WiFi and ensured that they could discover each other in this network context. Then, we set up the phone's `iptables`, so that TLS and QUIC traffic could be forwarded to the proxy.

B. Measurement Framework for Apps

We use QuicMitm to design a framework specifically to observe and detect privacy leakage in Android apps; see Fig. 1.

Our framework consists of four core components. (1) *Organizer:* It loads a list of popular APK package names, verifies device connectivity via Android Debug Bridge (ADB), and controls the device's network connections, such as the access to QuicMitm. (2) *Automated UI interactor:* Driven by ADB, the interactor opens the Google Play Store, downloads, installs, and traverses app UIs to trigger diverse HTTP requests. Based on ThirdEye [35], our interactor leverages ADB to parse Android layout files, identifies interactable elements (buttons, input fields), and systematically explores these UI elements. (3) *MITM proxy:* QuicMitm intercepts TLS over TCP and QUIC traffic, generates dynamic certificates, and demultiplexes streams. For HTTP/3, we decrypt the QUIC TLS layer. TLS connections over TCP are also decrypted to extract HTTP/1.1 and HTTP/2 traffic. (4) *Database &*

[1]That is: set "X509v3 Basic Constraints" to empty, rather than "critical"; set "CA" as "true" but under the attribute "X509v3 extensions", rather than the attribute "X509v3 Basic Constraints"

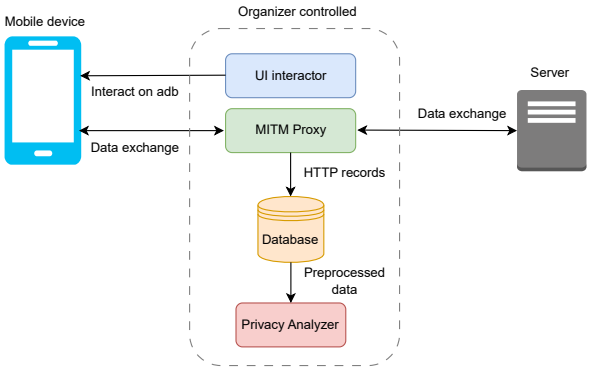

Fig. 1. The design of our framework to measure privacy leakage inside QUIC traffic by mobile apps.

Analyzer: All intercepted HTTP flows are stored in database. The analyzer retrieves records per app run, parses common encoded content, and matches payload fields against a privacy dataset (e.g., serial number of our tested device) in both plaintext and Base64.

V. Implementation

In this section, we present the implementation of QuicMitm proxy. We then introduce the implementation of our privacy measurement framework for Android mobile apps.

Our work included over 3,000 lines of code. The core of QuicMitm is composed of 1,380 LoC in Rust. We developed 483 LoC in Rust and 597 LoC in Python for extracting and analyzing privacy-related information. Beside original work from ThirdEye [35], we added additional 443 LoC in Python for organizer and UI automation on Android.

A. Proxy Implementation

As one of the mature and widely used QUIC libraries in the Rust ecosystem, Quinn [44] provides us the implementation of the QUIC transport protocol. We did not select quiche [7] as its lower-level API would increase development and maintenance complexity. We use Rustls [45], a TLS library in Rust, to manage certificate-related operations, including certificate generation and certificate signing. To parse and inspect the content of different versions of HTTP, we use libraries Hyper (HTTP/1.1 and HTTP/2 support) [18] and H3 (HTTP/3 support) [43].

Certificate Configuration. For certificate generation and signing, we use Rustls [45], a library designed for secure and efficient certificate management and TLS operations. The proxy automatically generates valid CA certificates with the following configurations: the validity period is set to 3 months (to avoid errors from Android's BoringSSL TLS library [14]); the certificate type is set to `CA` with type of `Unconstrained`; `KeyUsage` is set with `DigitalSignature`, `KeyCertSign`, and `CrlSign`. These values ensure that the generated CA certificate can be correctly parsed by the x509 parser in Rustls library, and then sign the dynamically generated server entity certificate.

Dynamic Certificate Generation. We implemented the `ResolvesServerCert` "trait"[2], which determines how to choose a certificate chain and signing key for server authentication (the client views the proxy as server in our case), to resolve the `ClientHello` from the application client TLS. During connection establishment, the trait extracts the server name from `ClientHello`, generates a corresponding certificate, and signs it with our CA certificate. The newly generated certificate is then returned to the client to complete the subsequent handshake (for more details, see Appendix A).

Transport Layer Handling. Quinn [44], compatible with Rustls, is employed for establishing and managing QUIC connections and streams mechanism. We instantiate one server endpoint for handling TLS connections and another for handling QUIC connections, and configure our `ResolvesServerCert` implementation for both (for more details, see Appendix B). To ensure forwarding efficiency and prevent interference between different network connections, our connection establishment is handled asynchronously (using Tokio [46]). We use `spawn` to generate two tasks (asynchronously executable units in tokio) to run separately. One task is used to wait for and receive the TCP-based TLS connection we mentioned earlier; the other task is for QUIC connection. For either task, when a new connection is received, `spawn` will be called to generate a new task, handling this new connection, while the original task remains in the listening loop, accepting other upcoming connections without blocking.

HTTP Requests Handling. After decryption from the security channel of QUIC, HTTP/3 requests are parsed with H3 [43], a library for HTTP/3 processing. For incoming HTTP requests over TLS, once the data has been decrypted from the secure channel, we parse both HTTP/1.1 and HTTP/2 requests with Hyper [18].

Because of the implementation of these libraries, QuicMitm has to deal with different versions of HTTP requests in different manners. For HTTP/3, once we receive the client connection from the proxy's `Endpoint`, we call `lookup_host` to perform DNS resolution on server's host name. Then we create a client `Endpoint` to connect to the remote server based on the address we looked up. Both connections are established and ready for future data transmission and HTTP processing. After that, the proxy loops the function `accept` on the connection with the client to receive one or more `RequestStreams` (from the H3 library). Again, we `spawn` a new task for each accepted stream to ensure concurrent data processing. It's worth noting that before forwarding the stream's data, we first call `send_request` to send the

[2]In Rust, a trait represents a collection of methods defined for the functionality of a particular data type.

request headers to the server separately, as required by the H3 library.

For TLS connection over TCP, once QuicMitm accepts `TlsStream` from the client, it calls `alpn_protocol` to check if the application protocol is HTTP/1.1 or HTTP/2. To forward both HTTP/1.1 and HTTP/2 requests, we implement different version-specified HTTP services to proxy the I/O stream to the target server (for more details, see Appendix C).

B. Configuration of Tested Mobile Device

We utilize a Pixel 6 running Android 13 as our test device. Operational communication between the device and the host is facilitated through the use of Android Debug Bridge (ADB [3]), a command-line tool for the host to control and interact with an Android device. In our framework, ADB is extensively used by the organizer, to configure CA certificates root store, modify the device's network settings, receive the layout files of app's pages, control the UI, and terminate app processes. The device was rooted using Magisk, enabling us to modify system settings and configurations. With root privileges, we employed `iptables` to redirect all network traffic from the device to QuicMitm. This setup ensured that both TCP and UDP traffic could be processed in a transparent manner. Additionally, our experimental setup includes a WiFi router connected to an external network. Both the host running the proxy and the mobile device are under the local network of this router. This configuration simplifies the process for the devices to locate the proxy's address.

To make Android system trust the proxy's CA certificate, we used the Magisk module "custom-certificate-authorities" to mount the certificate into the system's trusted certificate directory. Additionally, during our experiment, we discovered that updates of Android's APEX modules, which are used to install and update lower-level system modules independently of full system OTA updates, could modify the system's trust anchor configuration. In our case, the new APEX module "com.google.android.conscrypt" included its own trusted certificates, which were also used as trust anchors. To prevent this update mechanism from interfering with the proxy's certificate, we manually set the folder where APEX modules are downloaded as immutable (e.g., `chattr +i /data/apex/active`). This setting effectively blocks APEX updates and ensures the normal operation of the Android system.

C. Other Components in Mobile Apps Test

Application Runtime Hooking. We used LSPosed [24] to implement runtime hooking on our Android device. We implemented our LSPosed module to hook the output of `getLatitude` and `getLongitude` into fixed values. This allows Android's location APIs return our preset coordinates every time, easier to be identified in HTTP traffic.

Data Processing. We choose MongoDB [30] as our database, which provides API in Rust. The privacy analyzer is also implemented in Rust, enabling efficient extraction and processing of data from the database. In our experiment, we test each mobile app twice. In the first test, QuicMitm forwards both TCP and QUIC traffic; in the second test, we forward TCP traffic while blocking QUIC traffic. After each test, the analyzer retrieves all the HTTP records generated in this test from the database and analyzes them.

Before starting all experiments, we enumerated the PII records of the device under test that we intended to monitor, such as its serial number, the account email it logged in (a test email account), contact phone numbers stored (no real contact information), and other information. After collecting HTTP requests, we checked whether any of the recorded PII appeared in the requests. We also examined whether request payloads could be decompressed using publicly known formats (e.g., gzip, zip, and zstd) and inspected the contents for PII in form of plaintext or Base64-encoded.

UI Interactor. Python is used for the automated UI interactor of mobile device, which facilitates app usage to collect data. This component is based on the implementation of ThirdEye's interactor [35], focusing on UI recognition and interaction, which traversed and interacted with various UI elements (e.g., input fields and buttons) to explore different pages and trigger as many HTTP requests as possible. The interactor follows predefined inputs and interaction priorities, detecting pages with registration/login-related keywords, filling input fields with predefined credentials, and subsequently clicking login-related buttons where predefined keyword is detected. We carefully updated the layout checking logic and the operating rules for apps' download from Google Play Store, to accommodate changes in the newer version of Google Play Store user interface, i.e., the order of operations to download an app after opening its page in the Play Store.

VI. Results

In this section, we (i) briefly discuss our experimental setup; (ii) quantify how widespread HTTP/3 usage is in real-world Android apps, along with the distribution of their domain names; and (iii) compare the privacy-related leaks in HTTP/3 vs. HTTPS.

A. Experimental Setup

We use a Linux host to run all framework components. This host is network-connected (via WiFi) to a Pixel 6 test device. We root the Pixel 6 with Magisk to install our CA certificate and redirect traffic (using `iptables`) transparently through the proxy. Our test proceeds on each app twice as follows. *Test 1 (forwarding QUIC):* the proxy forwards both TCP 443 (HTTP/1.1 and HTTP/2) and UDP 443 (HTTP/3) traffic; and *Test 2 (blocking QUIC):* the proxy forwards only TCP 443, intercepts UDP 443 without forwarding, ensuring only HTTP/2 and HTTP/1.1 traffic. Comparing the traffic from the two tests, we anticipate to figure out whether privacy-related information in HTTP/3 is only transmitted via QUIC or falls back to HTTPS (i.e., HTTP/2 and HTTP/1.1) when QUIC is unavailable. Before each test, the interactor closes background apps and resets the device state to avoid cross-round contamination. According to testing methodology of

Pourali [35], the interactor also limits that each run should be executed in 5 minutes, or it would terminate the running app automatically.

B. HTTP/3 usage in real world

We retrieved app package names from AndroidRank [5] for our test. For each category listed in AndroidRank, we collected the package names of the top 160 most downloaded apps. Following Pourali's approach for app selection [35], we exclude games out of experiment, as their layouts and interaction patterns may not be compatible with the interactor. Note that some apps appear in multiple categories; we excluded these duplications. From February 17, 2025, to October 16, 2025, 3,452 (out of 5120) apps were successfully analyzed. Other apps failed for various reasons, such as being removed from the store, region restrictions preventing download, or crashing instantly without traffic generated. Out of the 3,452 apps in our valid test, we observed that 1,696 apps (49.1% of the total) generated HTTP/3 traffic during usage. In total, the HTTP/3 requests were sent to 298 unique addresses; see Table I. We noticed that some HTTP/3 requests, even when generated from different apps, were directed to the same destination addresses. Specifically, 119 domains (40% of all detected HTTP/3 addresses) received these requests across at least 2 apps, which we refer as common HTTP/3 destinations. In contrast, the remaining 179 addresses only received HTTP/3 requests when a specific app was running, referred as unique HTTP/3 destination. From the app perspective, a significant number of apps (1,674 apps, 48% of the total) send HTTP/3 requests to common destinations. In contrast, only 121 apps send HTTP/3 requests to unique addresses that are not observed in the HTTP/3 traffic of any other apps.

For apps with HTTP/3 traffic detected, we tracked their download counts in Figure 2 (excluding seven apps that had been removed from the Play Store by the time of our data collection). The $\geq$100K bucket represents the number of apps with downloads between 100K and 1M; $\geq$1M represents the number of apps with downloads between 1M and 100M; and $\geq$100M represents the number of apps with downloads between 100M and 1B. We found that apps sending HTTP/3 requests to addresses that also receive traffic from other apps generally have high download counts, spread across the range from under 100K to 1B+. Since these shared domains are predominantly Google APIs–which are triggered by the OS or common SDK dependencies at runtime rather than by app-specific logic–and because our apps' test set is sampled from download rankings, this distribution could more reflect the download-count distribution of the apps we tested, rather than an app-related HTTP/3 behavior. However, apps that send HTTP/3 requests to addresses not shown in other apps tend to have high download counts—eight of these exceed 1B downloads, such as Google Meet and Google News. Moreover, relatively low-download apps (under 100K) are less likely to send HTTP/3 requests to exclusive addresses. Compared to apps communicating with common domains, high-download apps exhibit a noticeably higher proportion of unique HTTP/3 destinations (as reflected by the red/blue bar ratios across download bucket in Figure 2). This reflects that high-download apps–and thus the developers of these apps–are active adopters of HTTP/3, and suggests that focusing on high-download apps is a practical scope for studying real-world QUIC traffic.

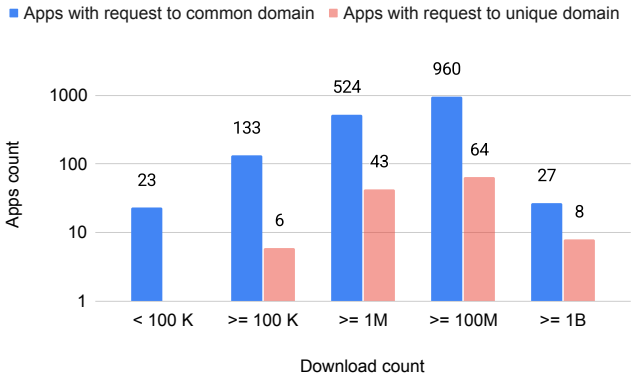

Fig. 2. Distribution of apps by download volume that issue HTTP/3 requests to the either app-unique domains or apps-common domains. Each bar shows the number of apps (in log scale) within a given download-count bucket that send HTTP/3 requests to those domains.

Table II highlights the top 10 organizations that registered domains accessed by mobile apps using HTTP/3, ranked by the number of apps. It reveals that HTTP/3 is predominantly used by major service providers, including large companies offering content delivery networks (CDNs), analytics services, advertising networks, and cloud storage platforms. Large companies such as Google, Meta and Amazon are at the forefront of HTTP/3 adoption, playing a key role in shaping the usage of this protocol across mobile apps. In the ecosystem of Android, Google LLC dominates the HTTP/3 traffic, with its registered domains shared across multiple apps.

Querying domain's organization on Whois, We also identified Moloco, Inc., a machine learning company focused on performance advertising [29]. Beside these companies, we noticed organizations such as DNStination Inc., Identity Protection Service, and REDACTED FOR PRIVACY. These organizations indicate that the actual domain owner is using privacy protection services to shield their personal information.

C. Static Analysis of QUIC Adoption

We therefore aim to perform static analysis to investigate how apps implement the capability to initiate HTTP/3 requests. After excluding common HTTP/3 requests sent to Google (such as those related to Google Ads and Google's app monitoring services, since our goal was to investigate how the apps themselves employ QUIC in development), we selected 108 apps that issued HTTP/3 requests. We then applied `grep` to the decompiled files of these apps to search for the keywords "quic" and "QUIC", while excluding matches containing such as "quick" or "Quick". Finally, we examine the listed results.

We started from how QUIC was utilized at the library implementation level in these results. Among the 108 apps, we

TABLE I
HTTP/3 AND HTTP/2 TRAFFIC ANALYSIS ACROSS APPS AND DESTINATION ADDRESSES.

App Metrics	Count	Percentage
Total apps tested	3,452	100%
Apps generating HTTP/3 traffic	1,696	49.1%
Apps generating HTTP/2 traffic	3,408	98.7%
Apps sending HTTP/3 requests to common destinations	1,674	48%
Apps sending HTTP/2 requests to common destinations	3,408	98.7%
Apps sending HTTP/3 requests to unique destinations	121	4.0%
Apps sending HTTP/2 requests to unique destinations	2,059	59.6%
HTTP/3 Address Metrics		
Total HTTP/3 destinations	298	100%
HTTP/3 destinations shown in at least 2 apps	119	40%
HTTP/3 destinations shown only in 1 app	179	60%
HTTP/2 Address Metrics		
Total HTTP/2 destinations	11,373	100%
HTTP/2 destinations shown in at least 2 apps	2,187	19.2%
HTTP/2 destinations shown only in 1 app	9,186	80.8%

TABLE II
WHOIS LOOKUPS ON HTTP/3 REQUESTS OF THEIR TARGET DOMAINS AND TOP TEN ORGANIZATIONS RANKED BY APP COUNT.

Rank	Organization	# of Apps
1	Google LLC	1,632
3	Meta Platforms, Inc.	50
5	Amazon Technologies, Inc.	18
7	Moloco, Inc.	15
10	Instagram LLC	5
10	Akamai Technologies, Inc.	5
12	Uber Technologies, Inc.	3
13	Expedia, Inc.	2
13	MercadoLibre Inc.	2
13	DPG Media Services NV	2
	Privacy or Proxy Services	
2	DNStination Inc.	96
4	Identity Protection Service	30
5	Domains By Proxy, LLC	18
8	REDACTED FOR PRIVACY	8
9	DATA REDACTED	7

found that 91 included definitions of QUIC within `Okhttp` or `Okhttp3` (as an enumeration of library defined class `Protocol`). In addition, 9 out of 108 apps contained definitions of QUIC as a protocol version within `Ktor`. QUIC appears as the protocol enumeration [42], but `Okhttp` does not provide actual support on it. The enumeration is included so that a developer could add their own custom interceptor to enable QUIC. Similarly, in clients created with `Ktor`, QUIC is merely listed as a protocol option [20], intended to maintain compatibility with the underlying network request engine on which it depends (i.e., `Okhttp3` on the Android platform).

The actual implementation of QUIC in these cases usually relies on `Cronet` to issue requests. `Cronet` is the Chromium network stack provided as a library for use in applications, and it offers an option to enable QUIC. In our analysis, we identified 33 apps whose decompiled APKs explicitly invoked `Cronet` with `enableQuic` option as true. We also found that nine apps issued QUIC requests to specific domains even though no relevant keyword matches were detected. Further manual analysis revealed that these apps transmitted requests through `WebView`. Similar to `Cronet`, Android `WebView` relies on the Chromium network stack to establish connections. However, unlike `Cronet`, `WebView` does not provide developers with an option to enable or disable QUIC. This implies that apps could implicitly establish QUIC connections with servers through `WebView`.

Not all instances of QUIC are directly introduced by the apps themselves. By examining the paths where relevant keywords appeared, we found that some apps enabled QUIC establishment through the integration of third-party SDKs. Most of these were advertising-related: 16 apps integrated the Unity3D advertising SDK, 17 integrated MBridge, 1 Ysocorp and 1 AdsByNimbus. Beyond advertising, 1 cloud service SDK also enabled QUIC by incorporating customized builds of `Cronet`: tunein.player, a news application, integrated Mapbox's mapping service, which had QUIC enabled. 2 other apps established QUIC connections through SDKs provided by Amazon Web Services and Tencent Cloud.

D. Privacy Leaks in HTTP/3 vs. HTTP/2

Our comparison is based on two key considerations: (i) what privacy-sensitive information is shared over HTTP/3 when the app is allowed to use both HTTP/2 and HTTP/3, and (ii) how blocking HTTP/3 (forcing the app to use only HTTP/2) impacts the sharing of privacy-sensitive information?

The comparison should focus on the differences between HTTP/3 and earlier versions of HTTP. Therefore, for simplicity of expression, we combined and analyzed the collected HTTP/2 and HTTP/1.1 requests. In the following analysis, any reference to "HTTPS" denotes both HTTP/2 and HTTP/1.1 requests.

For case (i), we filter the HTTP requests for each app as follows: we first record all requests from the initial run of an app, where both UDP (representing HTTP/3) and TCP (representing HTTPS) traffic are forwarded to their destinations. Then, we group the records based on the HTTP version: one group for HTTP/3 requests, and the other for HTTPS requests. Finally, we compare the privacy leaks between these two groups; see Table III. The PII in the table are categorized into protection level of *normal* and *dangerous*, based on their

degree of privacy sensitivity as defined in Android documentation [2]; we group apps based on the PII classification from Pourali et al. [35]. The values in Table III denote the app counts against various leakage categories. For example, the first row can be interpreted as follows: for the PII item Display ID (the current build ID displayed on the device, corresponding to different Android release versions [4]), when both HTTP/3 and HTTPS are available, one app transmits the Display ID exclusively over HTTP/3 during execution; 1,712 apps transmit it only over HTTPS; and another 1,687 apps transmit it over both HTTPS and HTTP/3.

We observe that several types of privacy are leaked consistently across both HTTPS and HTTP/3. This consistent privacy leakage indicates that version upgrades do not impact the functionality of services, as the same types of data are transmitted across both HTTPS and HTTP/3. However, the presence of certain data leaks exclusive to HTTP/3, such as 48 presences of CPU and 44 presences of device ABI information only in HTTP/3, raises privacy concerns. Such leakage has not been measured by any past studies. We further inspected these apps. When HTTP/3 is available, they transmitted device CPU and ABI information to googleapis.com and app-measurement.com over HTTP/3 instead of HTTPS. These requests could be triggered automatically by the system during app execution, as googleapis.com is used for Google Play services and core Google APIs, while app-measurement.com is associated with Google Analytics and Firebase measurement services.

For case (ii), we group collected HTTP requests as follows: for a given app, the first group consists of all HTTP/3 requests generated during its initial run. The second group comprises all HTTP requests produced during the second run of the app, when QUIC is disabled, indicating only HTTPS requests could be collected in the second run. We then compare the privacy leaks between these two groups; see Table IV. In this table, the column "Only HTTP/3" indicates private information unique to HTTP/3, which is not transmitted when traffic falls back to HTTPS after QUIC is disabled. Certain private information, such as device ABI, CPU, and device email, are uniquely leaked when HTTP/3 is used, while other information is leaked in both HTTP/3 and HTTPS traffic. In the first row of Table IV, as an example, 4 apps sent PII of device model to servers exclusively over HTTP/3 (i.e., they do not fall back to HTTPS to transmit device model when QUIC is unavailable), while 1,684 apps transmit device model information over both HTTP/3 and HTTPS.

The majority of apps transmit the same PII over HTTPS when HTTP/3 is unavailable, sending PII to the same domains. This downgrade mechanism is reflected in the increased app counts for the "Both HTTPS & 3" column in Tables III and IV for PII types such as device email and device ABI. On the other hand, we observed a few exceptions. For example, the four apps that appeared to send the device model only over HTTP/3 were, upon inspection, unable to run properly after launch, yet some HTTP/3 requests were still emitted before the apps exited. One of the cases is jp.naver.linecamera.android. After launching, the app reported that it could not run and exited. When HTTP/3 was enabled, we observed that an HTTP/3 request was sent to safebrowsing.googleapis.com, which contained device model as well as nearby WiFi BSSIDs. This API is used by applications to perform security risk assessment, such as detecting potentially malicious environments, network threats, or abnormal device states [16]. When HTTP/3 is disabled, the app still terminated, but no corresponding HTTPS request was triggered.

In addition, some apps transmit specific PII exclusively over HTTP/3. For example, com.rcplatform.livechat, when functioning with HTTP/3 enabled, sent the device Advertising ID (Ad ID) only over HTTP/3, which is an identifier for advertising purposes [15]. We found that when HTTP/3 was available, in addition to Google-owned domains, com.rcplatform.livechat also sent the Ad ID to rclog.rcplatformhk.com and 7fudun-conversions.appsflyersdk.com. However, when HTTP/3 was unavailable, the app still transmitted other PII to these two domains over HTTPS, but the payload no longer included the Ad ID.

Our findings show that apps are capable of sending specific data using a particular protocol (QUIC in our case). This discrepancy could also stem from app implementation (including its dependencies). An app enables QUIC by default but with an inconsistent fallback implementation—when QUIC is unavailable, requests cannot be sent over the legacy TLS channel, or requests can be downgraded but with inconsistent content. On the other hand, most apps still provide an effective fallback mechanism such that, when HTTP/3 is unavailable, the same content continues to be transmitted over HTTPS.

E. Custom Encryption in HTTP/3

Many mobile apps deploy additional encryption on top of HTTPS to further obfuscate their network traffic: Pourali et al. [35] proposed a dynamic analysis framework that instruments apps to intercept and analyze encryption by hooking standard Android cryptographic APIs and correlating plaintext with observed network flows. Building upon ThirdEye [35], we transplant this design into our study for scalability, but extend it to HTTP/3 traffic: while QuicMitm captures encrypted network flows, we simultaneously record Android framework-layer cryptographic hooks to match potential custom-encrypted data in the traffic with the corresponding plain text input and encrypted output of encryption functions.

Among all the apps we tested, 13 apps were found to transmit privacy-sensitive information using custom encryption over HTTP/3 requests during runtime. These apps sent requests to 12 distinct domains. After examining these 12 domains, we observed that 11 of them belong to Google (e.g., google, googleads, and gstatic), while one app, Step Tracker, sent the ESSIDs of surrounding WiFi access points to appshare.leap.app in the form of custom encryption during runtime.

In addition, we found that the encrypted privacy information mainly consisted of the ESSIDs of nearby WiFi networks. Besides other WiFi's ESSID, for app-measurement.com, the

TABLE III
PRIVACY LEAKS OBSERVED IN HTTPS AND HTTP/3 TRAFFIC WHEN BOTH PROTOCOLS ARE ENABLED DURING APP RUNTIME.

Privacy Data Type	Protection Level	Purpose	Only HTTP/3	Both HTTPS & 3	Only HTTPS
Display ID	Normal	Profiling	1	1,687	1,712
Device model	Normal	Profiling	1	1,688	1,711
Device name	Normal	Profiling	10	218	2,837
Device email	Dangerous	Persistent ID	54	35	463
Device ABI	Normal	Profiling	44	31	739
GPS (7m accuracy)	Dangerous	Location data	—	5	221
GPS (78m accuracy)	Dangerous	Location data	1	9	267
GPS (787m accuracy)	Dangerous	Location data	—	7	262
Device WiFi IP	Normal	Short-term ID	—	4	260
Build fingerprint	Normal	Profiling	16	189	2,768
Bootloader	Normal	Profiling	22	9	89
CPU	Normal	Profiling	48	5	192
Contacts	Dangerous	User asset	1	—	1
Ad ID	Normal	Persistent ID	—	499	428
Device WiFi ESSID	Dangerous	Location data	—	—	35
Nearby WiFi BSSID	Dangerous	Location data	—	—	23
Nearby WiFi ESSID	Dangerous	Location data	—	—	19
Device WiFi BSSID	Dangerous	Location data	—	—	35
Phone contact	Dangerous	User asset	—	—	34
Device resolution	Normal	Profiling	—	—	123
WiFi IPv6	Normal	Short-term ID	—	—	16
Timezone	Normal	Profiling	—	—	6
Device gateway IP	Normal	Short-term ID	—	—	2
Device proxy IP	Normal	Short-term ID	—	—	1
Device Serial	Dangerous	Persistent ID	—	—	1

encrypted data also included the device's advertising ID (AdID). We further compared whether the same apps also transmitted the personally identifiable information (PII) contained in custom-encrypted requests to the same domains via HTTPS. The results show that when HTTP/3 was available, certain requests containing custom-encrypted information were no longer sent over HTTPS during the same execution, and the same custom-encrypted information was still transmitted to these domains via HTTPS when QUIC was unavailable. This indicates that requests employing custom encryption preferentially use HTTP/3 when available, while effectively falling back to HTTPS when HTTP/3 is unavailable, suggesting that HTTP/3 is preferred as a transport rather than strictly required. These requests are summarized in Table V.

F. Incoherent QUIC Implementations

We noticed that AliExpress (by Alibaba) rejects our generated certificates–likely due to QUIC-specific pinning–and responds with QUIC Initial packets containing a CONNECTION_CLOSE frame with application error code 0x1d. Since implementation of Quinn ignores Initial packets carrying error 0x1d, QuicMitm fails to parse these responses correctly.

After reviewing the relevant specifications, we found the following statement (RFC 9000 [19], Sec. 10.2.3): *Sending a CONNECTION_CLOSE of type 0x1d in an Initial or Handshake packet could expose application state or be used to alter application state. A CONNECTION_CLOSE of type 0x1d MUST be replaced by a CONNECTION_CLOSE of type 0x1c when sending the frame in Initial or Handshake packets.* As mentioned in the document, when detecting errors in unverified packets, an endpoint may choose to discard such packets instead of immediately sending a `CONNECTION_CLOSE` frame to terminate the connection. Furthermore, for endpoints that have not yet established a connection state (e.g., a server receiving an invalid Initial packet), they should not transition into a closed or draining state. These measures help reduce the risk of denial-of-service (DoS) attacks against legitimate connections.

We also examined Alibaba's QUIC implementation, xquic, and identified the cause of this non-compliant behavior. The function `xqc_write_conn_close_to_packet` defaults to setting the packet type as Initial and sends the packet without verifying whether the error code is 0x1d. We reported this issue to the xquic maintainers via email, but no reply has been received so far.

VII. DISCUSSION AND LIMITATION

This study presents a novel MITM proxy capable of exposing plaintext traffic within QUIC. QuicMitm can parse the data transmitted in the HTTP/3 protocol generated by Android applications. To verify the effect, we check the PII in the HTTP/3 traffic. We compare the PII in the HTTP/3 traffic with the corresponding PII collected from the HTTPS traffic of the same application and to the same URL. Our result shows that the tested applications generated a significant amount of HTTP/3 traffic; 1,696 out of 3,452 sent HTTP/3 requests to 298 different domains.

In the analysis of the PII leak, we found that HTTP/3 was also used to transmit PII data. However, in comparison, when both HTTPS and HTTP/3 were available, most PII information was still transmitted via HTTPS, meaning more PII data was discovered via HTTPS, indicating that HTTPS remains the mainstream protocol. In another experiment, we disabled HTTP/3 by intercepting QUIC. In this case, we compared

TABLE IV
PRIVACY LEAKS OBSERVED IN HTTPS AND HTTP/3 TRAFFIC, WHERE HTTPS ARE COLLECTED FROM APPS RUNTIME WHEN QUIC IS DISABLED.

Privacy Data Type	Protection Level	Purpose	Only HTTP/3	Both HTTPS & 3	Only HTTPS
Display ID	Normal	Profiling	4	1,684	1,724
Device model	Normal	Profiling	3	1,686	1,723
Device name	Normal	Profiling	9	219	2,854
Device email	Dangerous	Persistent ID	27	62	566
Device ABI	Normal	Profiling	27	48	834
GPS (7m accuracy)	Dangerous	Location data	—	5	218
GPS (78m accuracy)	Dangerous	Location data	2	8	267
GPS (787m accuracy)	Dangerous	Location data	—	7	263
Device WiFi IP	Normal	Short-term ID	—	4	263
Build fingerprint	Normal	Profiling	10	195	2,796
Bootloader	Normal	Profiling	18	13	176
CPU	Normal	Profiling	17	36	230
Contacts	Dangerous	User asset	1	—	1
Ad ID	Normal	Persistent ID	1	499	427
Device WiFi ESSID	Dangerous	Location data	—	—	33
Nearby WiFi BSSID	Dangerous	Location data	—	—	23
Nearby WiFi ESSID	Dangerous	Location data	—	—	19
Device WiFi BSSID	Dangerous	Location data	—	—	33
Phone contact	Dangerous	User asset	—	—	47
Device resolution	Normal	Profiling	—	—	125
WiFi IPv6	Normal	Short-term ID	—	—	16
Timezone	Normal	Profiling	—	—	6
Device gateway IP	Normal	Short-term ID	—	—	2
Device proxy IP	Normal	Short-term ID	—	—	1
Device Serial	Dangerous	Persistent ID	—	—	1

TABLE V
CUSTOM-ENCRYPTED PII TRANSMITTED EXCLUSIVELY OVER HTTP/3 WHEN QUIC IS AVAILABLE.

App	Download	Req Destination	PII Type
com.myairtelapp	500M+	app-measurement.com	Nearby WiFi ESSID & Advertising ID
me.mycake	100M+	app-measurement.com	Nearby WiFi ESSID
com.rfi.sams.android	10M+	app-measurement.com	Nearby WiFi ESSID
com.inovel.app.yemeksepeti	10M+	geomobileservices-pa.googleapis.com	Nearby WiFi ESSID
oms.mmc.fortunetelling.gmpay.eightcharacters.zwz	100K+	infinitedata-pa.googleapis.com	Nearby WiFi ESSID

all PII records collected under HTTPS with those previously collected under HTTP/3 enabled. While limited inconsistent HTTP/3 and HTTPS requests were observed, we found that the number of PII records under HTTPS increased when HTTP/3 was disabled. This change indicates that applications could degrade to HTTPS and ensure data transmission when HTTP/3 is unavailable. In addition, our supplementary experiment shows that certain transmission characteristics previously observed in HTTPS–such as the use of custom encryption layers–also appear in QUIC deployments.

However, several limitations naturally arise. The increasingly stringent security mechanisms on Android restrict our analysis to Android applications. Newer Android versions and browser-specific certificate stores for QUIC introduce dedicated forms of certificate pinning, which pose challenges to QuicMitm-based interception approach. Addressing these limitations will require future work, particularly a deeper investigation into how modern browsers implement QUIC's evolving TLS authentication mechanisms. Regarding privacy leakage detection, our PII coverage is limited to ThirdEye [35], which primarily focuses on device-level information (e.g., serial numbers) and physical-context information (e.g., location and nearby WiFi), as these items were predefined in advance. Broader categories of privacy data, such as information generated during app usage, are not included in our detection scope, since capturing such data would require contextual understanding of UI interactions to identify relevant fields, as well as more dynamic mechanisms to track them within network traffic.

VIII. CONCLUSION

We built a custom MITM proxy for QUIC that enables direct inspection of HTTP/3 flows in mobile apps, revealing how privacy-related data is transmitted under TLS-encryption. Our measurements confirm that QUIC adoption is rising sharply–already a major portion of Google's traffic uses QUIC–and it underpins much of today's mobile services. We also show that HTTP/3 over QUIC leaks privacy information akin to HTTPS. We hope our tool will assist other researchers interested in examining the content transmitted over QUIC.

IX. ETHICAL CONSIDERATION

Our study only intercepts traffic from test devices using disposable Google accounts with no personal or user-identifiable data. All experiments were conducted in a controlled environment without involving real users. Hence, this work raises no ethical concerns.

ACKNOWLEDGMENT

We are grateful to the anonymous MADWeb 2026 reviewers for their insightful suggestions and comments. We also appreciate the help we received from the members of Concordia's Madiba Security Research Group. The second author is supported in part by an NSERC Discovery Grant.

REFERENCES

[1] Aioquic Developers, "Aioquic: QUIC and HTTP/3 implementation in Python," https://github.com/aiortc/aioquic, 2025.

[2] Android Developers, "Permission — App architecture," https://developer.android.com/guide/topics/manifest/permission-element.

[3] ——, "Android debug bridge (adb)," https://developer.android.com/tools/adb, 2023.

[4] Android Open Source Project, "Codenames, tags, and build numbers," https://source.android.com/docs/setup/reference/build-numbers.

[5] AndroidRank.org, "AndroidRank: Android app ranking and statistics," https://www.androidrank.org/, 2025.

[6] R. Cheng, N. Wu, M. Varvello, E. Chai, S. Chen, and B. Han, "A first look at immersive telepresence on Apple Vision Pro," in *Proceedings of the 2024 ACM on Internet Measurement Conference (IMC'24)*, Madrid, Spain, 2024.

[7] Cloudflare, "Quiche: Savoury implementation of the QUIC transport protocol and HTTP/3," https://github.com/cloudflare/quiche, 2025.

[8] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A free and open source interactive HTTPS proxy," https://mitmproxy.org/, 2025.

[9] ——, "mitmproxy: A free and open source interactive HTTPS proxy – concepts / protocols," https://docs.mitmproxy.org/stable/concepts/protocols/, 2025.

[10] Z. Dong, T. Liu, J. Deng, H. Wang, L. Li, M. Yang, M. Wang, G. Xu, and G. Xu, "Exploring covert third-party identifiers through external storage in the Android new era," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, 2024.

[11] K. Elmenhorst, B. Schütz, N. Aschenbruck, and S. Basso, "Web censorship measurements of HTTP/3 over QUIC," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021.

[12] A. Ganji and M. Shahzad, "Characterizing the performance of QUIC on Android and wear OS devices," in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021.

[13] Y. Gbur and F. Tschorsch, "Quicforge: Client-side request forgery in QUIC," in *Network and Distributed System Security (NDSS) Symposium*, 2023.

[14] Google LLC, "BoringSSL," https://boringssl.googlesource.com/boringssl.

[15] ——, "Get a user-resettable advertising ID," https://developer.android.com/identity/ad-id.

[16] ——, "Google safe browsing API," https://developers.google.com/safe-browsing.

[17] M. Hils, "mitmproxy 10: First bits of HTTP/3!" https://mitmproxy.org/posts/releases/mitmproxy10/.

[18] Hyperium. Hyper: Fast and safe HTTP for the Rust language. https://github.com/hyperium/hyper.

[19] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," https://www.rfc-editor.org/info/rfc9000, May 2021.

[20] JetBrains, "Client engines – Ktor documentation," https://ktor.io/docs/client-engines.html, 2025.

[21] I. Kunze, C. Sander, and K. Wehrle, "Does it spin? On the adoption and use of QUIC's spin bit," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023.

[22] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[23] S. Li, Z. Yang, Y. Nan, S. Yu, Q. Zhu, and M. Yang, "Are we getting well-informed? An in-depth study of runtime privacy notice practice in mobile apps," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[24] LSPosed Team. (2023) LSPosed: A modern implementation of the Xposed framework. https://github.com/LSPosed/LSPosed.

[25] H. Lu, Y. Liu, X. Liao, and L. Xing, "Towards Privacy-Preserving Social-Media SDKs on android," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, Aug. 2024.

[26] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How secure and quick is QUIC? Provable security and performance analyses," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[27] D. Madariaga, L. Torrealba, J. Madariaga, J. Bermúdez, and J. Bustos-Jiménez, "Analyzing the adoption of QUIC from a mobile development perspective," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, 2020.

[28] A. Mishra and B. Leong, "Containing the Cambrian explosion in QUIC congestion control," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023.

[29] Moloco, Inc., "Moloco — the future of advertising is powered by AI," https://www.moloco.com/.

[30] MongoDB, Inc., "Mongodb: The developer data platform," https://www.mongodb.com/, 2024.

[31] N. Muthuraj, N. Eghbal, and P. Lu, "Replication: Taking a long look at QUIC," in *Proceedings of the 2024 ACM on Internet Measurement Conference*, 2024.

[32] J. Mücke, M. Nawrocki, R. Hiesgen, P. Sattler, J. Zirngibl, G. Carle, J. Luxemburk, T. C. Schmidt, and M. Wählisch, "Waiting for QUIC: Passive measurements to understand QUIC deployments," *Proceedings of the ACM on Networking*, 2025.

[33] J. Mücke, M. Nawrocki, R. Hiesgen, T. C. Schmidt, and M. Wählisch, "ReACKed QUICer: Measuring the performance of instant acknowledgments in QUIC handshakes," in *Proceedings of the 2024 ACM on Internet Measurement Conference*, 2024.

[34] J. Pegioudis, E. Papadogiannakis, N. Kourtellis, E. P. Markatos, and P. Papadopoulos, "Not only E.T. phones home: Analysing the native user tracking of mobile browsers," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023.

[35] 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*, 2022.

[36] Rustls Developers, "webpki-roots," https://github.com/rustls/webpki-roots, 2023.

[37] C. Sander, I. Kunze, L. Blöcher, M. Kosek, and K. Wehrle, "ECN with QUIC: Challenges in the wild," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023.

[38] S. J. Sheela, T. R. S. Kumar, and S. Sonu, "Quick multimedia data transfer with QUIC," in *2023 International Conference on Network, Multimedia and Information Technology (NMITCON)*, 2023.

[39] J.-P. Smith, L. Dolfi, P. Mittal, and A. Perrig, "QCSD: A QUIC Client-Side Website-Fingerprinting defence framework," in *Proceedings of the 31st USENIX Conference on Security Symposium*, 2022.

[40] J.-P. Smith, P. Mittal, and A. Perrig, "Website fingerprinting in the age of QUIC," in *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2021.

[41] M. Soni and B. S. Rajput, "Security and Performance Evaluations of QUIC Protocol," in *Data Science and Intelligent Applications*, 2021.

[42] Square, Inc., "QUIC – OkHttp Documentation," https://square.github.io/okhttp/5.x/okhttp/okhttp3/-protocol/-q-u-i-c/, 2025.

[43] The Hyper Contributors, "H3: An async HTTP/3 implementation," https://github.com/hyperium/h3, 2024.

[44] The Quinn Contributors, "Quinn: Async-friendly QUIC implementation in rust," https://github.com/quinn-rs/quinn, 2024.

[45] The Rustls Contributors, "Rustls: A Rust-native TLS library," https://github.com/rustls/rustls, 2024.

[46] Tokio Developers, "Tokio - an asynchronous Rust runtime," https://tokio.rs/, 2025.

[47] J. Wu, Y. Nan, L. Xing, J. Cheng, Z. Lin, Z. Zheng, and M. Yang, "Leaking the privacy of groups and more: Understanding privacy risks of cross-app content sharing in mobile ecosystem," in *Network and Distributed System Security (NDSS) Symposium 2024*, 2024.

[48] Y. Zhang, Z. Hu, X. Wang, Y. Hong, Y. Nan, X. Wang, J. Cheng, and L. Xing, "Navigating the privacy compliance maze: Understanding risks with Privacy-Configurable mobile SDKs," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, 2024.

[49] A. Zohaib, Q. Zao, J. Sippe, A. Alaraj, A. Houmansadr, Z. Durumeric, and E. Wustrow, "Exposing and circumventing SNI-based QUIC censorship of the great firewall of China," in *Proceedings of the 34th USENIX Conference on Security Symposium*, 2025.

APPENDIX

A. Dynamic Certificate Generation

Specifically, we implement the `resolve` function defined in the trait of `ResolvesServerCert`, which would be triggered whenever a `ClientHello` is received. Function `resolve` takes `ClientHello` as an input, from which we extract the server name and generate the corresponding certificate representing the server. In this process, we explicitly set `ServerAuth` for extended key usage, to inform client that the certificate's public key is authorized as a server. We then generate a key pair for the certificate and sign it with the CA certificate using `signed_by`. Both signed certificate and its private key are returned by `resolve`, so that the proxy can complete the handshake with the client.

B. Security Layer Handling

When the proxy is started, we prepare two `ServerConfigs` structures. `ServerConfigs` defines the TLS configuration used for incoming connections. We bind our implementation of `ResolvesServerCert` to these two settings. One of the `ServerConfigs` is wrapped in `TlsAcceptor`, which is a structure in Rustls for handling TLS connection in TCP. The other `ServerConfig` is wrapped in `Endpoint`, which is the structure of receiving QUIC connections in Quinn. This enables the proxy to handle the customized TLS handshake, while also offloading our work on protocol interactions unrelated to authentication (i.e., TLS state transition, encryption/decryption process).

C. HTTP/1.1 & HTTP/2 Forwarding

If HTTP/2 is detected, a corresponding `TlsStream` will be connected to the server. In Hyper, the `serve_connection` method enables a server (the proxy acts as a server for the client in our case) to process HTTP requests as a service.

When HTTP/1.1 is observed in the ALPN field, the proxy will delay the communication with the remote server: the proxy will first parse the complete HTTP/1.1 requests from the TLS channel, then establish corresponding connection to the server layer by layer (from TCP to TLS, then to HTTP). This implementation on HTTP/1.1 is forced by the inner implementation of the Hyper library. Hyper's service for processing HTTP/1.1 requests does not support asynchronous input, which we use for handling the proxy's forwarding I/O stream in HTTP/2 handling. Therefore, we first collect the complete HTTP/1.1 request and pass the entire request to a defined service. Within this service, we establish a connection to the server, send the request, and then forward the response.