Protocol Compliance in Popular RTC Applications

Peiqing Chen*
University of Maryland
College Park, Maryland, USA
pqchen99@umd.edu

Peng Qiu*
University of Pennsylvania
Philadelphia, Pennsylvania, USA
qiupeng@seas.upenn.edu

Lambda†
Lambda
Lambda, USA
Lambda@gmail.com

Zaoxing Liu
University of Maryland
College Park, Maryland, USA
zaoxing@umd.edu

Abstract

Real-time communication (RTC) has been prevalent since COVID-19, supporting billions of video calls and voice chat interactions. Protocols such as STUN, TURN, RTP, RTCP, and QUIC play a critical role in transmitting RTC media in various applications. Based on standardized protocol specifications, in this paper, we investigate the extent of protocol compliance by analyzing the network traffic in real-world one-on-one calls. We capture and filter RTC traffic, design a Deep Packet Inspection framework to identify all messages for RTC media transmission, and systematically evaluate each message's compliance against protocol specifications. Our analysis of six popular RTC applications—Zoom, FaceTime, WhatsApp, Facebook Messenger (i.e., Messenger), Discord and Google Meet—reveals that: 1) None of the studied applications strictly follow all RTC protocol specifications, and existing protocol implementations, except for QUIC, have some level of non-compliance; 2) Existing applications either implement proprietary protocols or modify existing message types to achieve the desired protocol functionality.

CCS Concepts

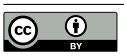
• **Networks** → **Network protocol**; **Network measurement**; **Peer-to-peer protocols**.

Keywords

Real-time Communication (RTC), Network Protocols, Network Measurement

ACM Reference Format:
Peiqing Chen*, Peng Qiu*, Lambda†, and Zaoxing Liu. 2025. Protocol Compliance in Popular RTC Applications. In *Proceedings of the 2025 ACM Internet Measurement Conference (IMC '25), October 28–31, 2025, Madison, WI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3730567.3764438

*The two authors contributed equally in this work.
† We anonymize the author to protect the author's identity.



IMC '25, Madison, WI, USA

ACM ISBN 979-8-4007-1860-1/2025/10
https://doi.org/10.1145/3730567.3764438

1 Introduction

Real-Time Communication (RTC) has been a cornerstone technology, shaping the way we connect and interact on the internet. RTC applications, such as Zoom, FaceTime, WhatsApp, Facebook Messenger (i.e., Messenger), Discord, and Google Meet, build upon a stack of media transmission protocols to ensure low-latency, secure, and reliable transmission of audio and video between clients. Key components include RTP [35] for media delivery, RTCP [15] for performance feedback, STUN [34] and TURN [31] protocols for NAT traversal. These protocols are integrated into widely adopted RTC frameworks such as WebRTC [13, 16, 22], which provide unified APIs and reference implementations to simplify and standardize RTC application development.

In this paper, we focus on an important but underexplored aspect of RTC protocol behavior: *protocol compliance*. We define protocol compliance as if observed protocol messages strictly follows their corresponding protocol specifications (e.g., RFC 3550 for RTP [35], RFC 5389 for STUN [23], RFC 5766 for TURN [31], and RFC 9000 for QUIC [14]), including correct use of message types, attributes, header fields, and semantic structure. Today, many RTC applications diverge from protocol specifications in both structure and semantics. Some introduce proprietary protocols that encapsulate or obfuscate standard messages; others redefine message types, overload standard attributes, or use RTC protocols for non-standard purposes such as bandwidth probing or session signaling [4, 25].

Studying protocol compliance is not merely a theoretical concern; it has direct practical implications. At its core, compliance is a prerequisite for RTC interoperability—the ability of different applications to understand and exchange real-time media without bespoke adaptation layers. This principle has been formally recognized by regulatory efforts such as the European Union's Digital Markets Act (DMA) [1, 24], which mandates that by 2028, major RTC applications must support cross-platform voice and video calls between individual users. While the current RTC ecosystem lacks interoperable applications, we argue that adherence to common protocol specifications lowers this barrier. When applications implement RTC protocols consistently, they inherently improve their ability to parse and process packets from other implementations. Conversely, by measuring the degree of non-compliance, we can also estimate the technical challenges involved in achieving such interoperability. Beyond regulatory requirements, protocol compliance is fundamental to the RTC ecosystem itself. For developers, it fosters code modularity, reusability, and compatibility

across applications and vendors. For network operators, it enables effective traffic classification, monitoring, and policy enforcement through protocol-aware network devices. For researchers and protocol designers, compliance measurements reveal how well protocol specifications are followed in practice, highlighting discrepancies between specification and real-world deployment, and informing future protocol evolution.

Despite its significance, protocol compliance in RTC applications has not been quantified. Prior work has examined traffic volume, quality metrics (e.g., latency, jitter), and codec performance [6, 21], but avoids message-level structural analysis due to several key challenges: 1) RTC applications are closed-sourced, preventing direct inspection of protocol logic; 2) messages are often encapsulated in application-specific proprietary formats, which confound standard DPI tools; and 3) protocol usage varies across platforms, complicating unified analysis.

This paper presents the first cross-application, protocol compliance study of real-world RTC traffic. To address the above challenges, we design a measurement framework to capture, extract, classify, and evaluate messages for RTC media transmission against their protocol specifications, and use it to evaluate six prevalent applications under 1-on-1 call experiments with varying network conditions. We develop a two-stage filtering pipeline to isolate RTC traffic from unrelated traffic generated by background activities and define a five-criterion compliance model to check the compliance of each message according to message types, fields, attributes, and semantics.

Summary of Findings. Using this framework, we reveal several protocol implementation strategies and widespread deviations from protocol specifications:

(1) **RTC applications use different subsets of protocols.** Zoom uses STUN, RTP, and RTCP. Messenger, WhatsApp, and Google Meet use STUN, TURN, RTP, and RTCP. FaceTime uses STUN, TURN, RTP, and QUIC. Discord uses only RTP and RTCP.
(2) **No application fully follows all the protocol specifications**. Zoom has a non-compliant implementation in STUN while compliant in RTP and RTCP. WhatsApp and Messenger both implement STUN and RTCP without following protocol specifications. FaceTime has non-compliance with STUN, TURN, and RTP. Discord has non-compliance in RTP and RTCP. Google Meet implements RTP compliantly but does not completely follow STUN and RTCP.
(3) **Two common modification strategies**: Zoom and FaceTime prepend proprietary headers to RTC protocol messages; WhatsApp, Messenger, and Discord introduce undefined message or attribute types, particularly in STUN and RTCP.
(4) **STUN and RTCP have the highest level of non-compliance in implementation,** with around half of their observed message types exhibiting compliance violations across multiple applications.
(5) **Among all the RTC applications, Google Meet, Messenger, and WhatsApp are more compliant, while Zoom, Discord, and FaceTime are less compliant comparatively.** Above 95% of messages in Google Meet, Messenger, and WhatsApp are compliant. Over 99.9% datagrams in Zoom traffic and over 72% datagrams in FaceTime traffic contain non-standard protocol headers. All 11 message types used in Discord are non-compliant.
(6) **We uncover several application-specific network behaviors** which have not been reported before. For example, Zoom always uses filler datagrams in RTC traffic; Discord applies SSRC=0 in RTCP messages; FaceTime transmits a high volume of proprietary datagrams in cellular-based calls only.

Together, our findings demonstrate that protocol non-compliance is both widespread and application-specific. To support future research, we publicly release our dataset and compliance analysis framework.[1] These resources enable reproducible evaluation and provide a foundation for new studies on RTC protocol behavior, fuzz testing, and deployment diagnostics.

2 Background and Motivation

A typical RTC application supports two types of calls: one-on-one (1-on-1) and group calls (with more than two participants). In this paper, we focus on 1-on-1 calls for two main reasons. First, 1-on-1 calls account for the majority of RTC traffic. It has been reported that over 80% of calls are 1-on-1 since the COVID-19 pandemic [5, 17]. Second, most well-known RTC frameworks and protocols, including ICE [18] and TURN [30], are primarily designed for 1-on-1 calls. Due to space limitations, we plan the study of group calls as future work. Next, we will discuss the 1-on-1 call procedure and related media transmission protocols.

2.1 1-on-1 RTC Call

1-on-1 RTC call involves two processes, *signaling* and *media transmission*, working as the control plane and the data plane, respectively. Signaling is essential in control tasks such as connection setup, session management, and call termination. It helps the devices discover each other's network presence and exchange session metadata necessary for media transmission. These changes are negotiated and propagated through the signaling server to ensure both peers remain synchronized. For example, when a client device toggles its microphone or camera, the signaling server would add or remove a media session [26]. Overall, the signaling phase does not transmit any media data; it only coordinates connection setup/termination. While there is no standardized signaling protocol, applications need to implement their own signaling mechanisms, commonly using HTTPS or WebSocket to relay session descriptions and connectivity candidates.

In the media transmission phase, the two devices exchange audio and video in a real-time manner. In this process, many network protocols will be used, including STUN, TURN, RTP, RTCP, and QUIC.

The first step in media transmission is to determine whether a direct connection can be established between the two clients. To do this, each client uses **STUN** (Session Traversal Utilities for NAT) [28] to discover its own public-facing IP address and port by sending *Binding Requests* to external STUN servers. Once both endpoints obtain their public addresses, they exchange this information and then use STUN messages again to probe each other's reachability to find a direct path. Besides, STUN also serves as the connectivity check between the peers during the call.

[1] https://anonymous.4open.science/r/rtc_code-631F/README.md

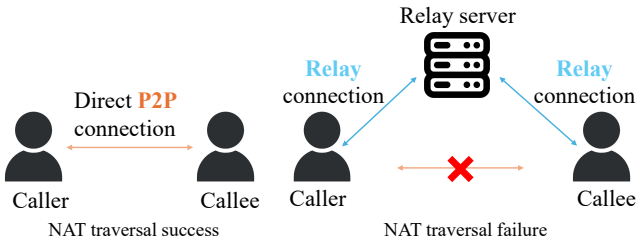


Figure 1: RTC transmission modes: P2P vs Relays. Whether a call uses P2P or Relay is determined by NAT traversal success.

If no direct path is found—due to symmetric NATs or firewalls—the call falls back to relay-based communication using **TURN** (Traversal Using Relays around NAT) [30]. In this case, each client sends media through a TURN server (i.e., relay server), which forwards packets to the other endpoint. TURN uses the same message format as STUN; thus, ***we discuss STUN/TURN jointly throughout the paper***. This process results in two possible transmission modes (Figure 1):

- **Peer-to-peer (P2P) mode:** If a direct path between the two devices is available, the application can establish a P2P connection to transmit media directly.
- **Server relay mode:** If P2P connectivity is not feasible, due to NATs or firewalls, the application relies on relay servers to traverse NAT. In this case, the media is forwarded through relay servers.

Once a session is established, the clients begin exchanging media using **RTP** (Real-time Transport Protocol) [35]. RTP carries the actual audio and video frames, enriched with sequence numbers and timestamps for proper playback synchronization. Each RTP stream is tagged with a unique SSRC identifier, allowing the receiver to distinguish concurrent streams (e.g., audio vs. video).

Alongside RTP, the clients periodically exchange **RTCP** (Real-Time Transport Control Protocol) [35] packets. These reports provide end-to-end performance metrics, including jitter, packet loss, and round-trip time. Applications use this feedback to perform congestion control, detect degradation, and adapt the bitrate or resolution dynamically. When security is required, the same control messages are protected with SRTCP (Secure RTCP) [40], which adds encryption and message-authentication tags to preserve confidentiality and integrity in transit.

In addition to these four protocols, **QUIC** (Quick UDP Internet Connections) [14] is increasingly adopted in RTC applications as an alternative transport protocol to UDP, encapsulating RTP media streams over its datagram extension to provide encrypted, multiplexed, and congestion-controlled delivery, while preserving real-time performance [3, 20, 39].

2.2 Protocol Compliance

In practice, these protocols are often implemented with deviations from protocol specifications in the RFCs, motivating a systematic study of protocol compliance: *If the implementations of these media transmission protocols strictly follow the RFC protocol specifications.* Specifically, a compliant protocol implementation instance is a message[2] that strictly follows the RFC protocol specifications in both structure and intended usage. For example, RTP compliance is defined according to RFC 3550 [35], while STUN compliance refers to RFC 3489 [33], RFC 5389 [23], and RFC 8489 [28].[3] A compliant message uses defined message types and attributes, preserves valid field values, and conforms to expected protocol behavior. Any violation will result in a non-compliant implementation.

Although these protocols are formally specified in RFCs, real-world implementations frequently diverge from those standards. We observe applications introducing undefined STUN message or attribute types, repurposing RTP header fields, or prepending proprietary headers before RTC messages. Such deviations—whether motivated by legacy constraints, optimization attempts, or engineering oversight—carry three critical consequences. First, they erode interoperability: when two endpoints interpret the same protocol message differently, or when one endpoint transmits a non-compliant message, end-to-end media exchange between otherwise compatible RTC systems can break, undermining recent regulatory pushes (e.g., the EU Digital Markets Act [1, 42]) for cross-vendor interoperability. Second, they can degrade performance: oversized attributes, redundant headers, and extra encapsulation increase bandwidth and processing overhead, harming the quality of experience on constrained networks. Third, they blind measurement and security tools: DPI engines, traffic classifiers, and network debuggers depend on standard header layouts; proprietary modifications obscure message semantics and make automated analysis error-prone.

These challenges call for a systematic study of protocol compliance in RTC applications. In this work, we develop a message-level compliance framework and apply it to six widely-used RTC platforms—Zoom, FaceTime, WhatsApp, Messenger, Discord, and Google Meet—evaluating whether their protocol messages adhere to the structural and semantic requirements defined in STUN, TURN, RTP, RTCP, and QUIC. Our goal is to quantify the extent and nature of protocol compliance.

2.3 Overview of Measurement Framework

Studying message-level compliance in production RTC applications is challenging: these systems are closed-source, lack debugging interfaces, and do not expose internal protocol logs. We treat these applications as black boxes and infer compliance solely from observed network traffic and behaviors. We focus on media transmission protocols rather than signaling protocols, because the former are specified by public RFCs, while signaling protocols are application-specific and lack universal specifications. Our study has four key steps:

Application selection. To ensure that our analysis captures the most representative instances of RTC protocol implementations, we aim to perform measurements over applications with both the largest user bases and the most diverse user populations. Studying such applications both allows us to learn about 1) what the current market needs and how they implement features to satisfy

[2]A message is one complete application protocol data unit, consisting of a header and payload. For protocols with multiple specification versions (e.g., STUN), we consider an implementation compliant if it adheres to any of the officially published RFCs.
[3]STUN has multiple versions.

them, and 2) the versatility and adaptability of RTC designs to different scenarios, covering both consumer solutions and business communications. Based on this motivation, we select six popular RTC applications: Zoom, FaceTime, WhatsApp, Messenger, Discord, and Google Meet. These applications are representative in today's RTC market [29, 41]. In this paper, we focus on the mobile applications only. It's worth noting that most of these mobile apps are WebRTC-based [22]. For example, there are explicit documentations on WhatsApp, Messenger, and Google Meet built using WebRTC components [8–12]. Comparing the compliance differences between mobile apps and their web browser versions is also an important research topic and will be left as future work.

Traffic collection and filtering (Section 3). To evaluate protocol compliance, we require RTC traffic that is diverse, realistic, and clean. The traffic must span different network conditions to capture representative protocol behaviors, last long enough to expose full message patterns, and be free from unrelated traffic that could distort compliance measurement. We meet these requirements by conducting 1-on-1 call experiments via both cellular and Wi-Fi connections, and extracting clean RTC traffic using timespan filters and stream-specific heuristics.

Evaluate compliance for each message (Section 4). To determine whether each message in the RTC call traffic is compliant, we need 2 steps: 1) identify all the RTC messages from the RTC call traffic using our own DPI, 2) determine the compliance of each RTC message according to a set of rules by checking their message type, header and attribute validity, and syntax integrity.

Compliance summary across protocols and applications (Section 5). Once the compliance results for each message are obtained, two natural questions arise: 1) *Across various protocols*, whose implementation complies most closely with its protocol specification? 2) *Across six applications*, which one implements their protocols most closely with their respective protocol specification? We answer these two questions by the message compliance result.

3 Traffic Collection and Filtering

To analyze the compliance of protocol implementations in various RTC applications, we collect their traffic that is representative of real-world use. To achieve this goal, we conduct 1-on-1 RTC calls of 5 applications across both cellular and Wi-Fi network conditions, and in both relay and P2P transmission modes. We use Wireshark [43] to record traffic from both client devices. Then, to extract RTC media traffic, we group packets into transport streams and apply a two-stage filtering process that removes background activities based on stream timespans and protocol-specific heuristics. The resulting dataset is over 17GB, containing 20 million UDP datagrams from 75 call sessions. As summarized in Table 1, these traffic are noise-filtered and grouped into streams. We will apply our deep packet inspection on it for protocol compliance analysis.

3.1 Call Experiment Setup

3.1.1 Device and Network Setup. We conduct 1-on-1 call experiments using two iPhone 11 devices. [4] Our setup covers two network conditions: Wi-Fi and 4G cellular. For Wi-Fi, we use a TP-LINK Archer A7 router running OpenWRT, configured with 400 Mbps download and 100 Mbps upload bandwidth to emulate a stable home network. For cellular, we use Verizon as our ISP.

To study P2P and relay mode communication, we control the NAT traversal behavior under Wi-Fi. By modifying local firewall rules on the router, we selectively enable or disable UDP hole punching. When P2P is allowed, media flows directly between devices; when blocked, applications are forced to route media through their designated relay infrastructure (e.g., TURN or SFU servers).

However, under cellular networks, we cannot manipulate NAT or firewall behavior. Whether media is transmitted via P2P or relay is entirely determined by the application's logic and its compatibility with the mobile carrier's network infrastructure. In our measurements, we observed application-dependent behaviors: Zoom and Discord consistently used relay mode; FaceTime consistently used P2P; while WhatsApp, Messenger, and Google Meet initially used relay mode and switched to P2P after 30 seconds. These findings reflect the application's design choices under the tested network and carrier conditions. It is important to note that our observations may not generalize to all cellular environments. P2P feasibility on mobile networks can vary significantly across regions, carriers, and infrastructure.

As a result, our experiments cover three network configurations: Wi-Fi with P2P enabled, Wi-Fi with P2P disabled (relay mode), and cellular connections with application-determined transmission modes.

3.1.2 Experiment Procedure and Trace Capture. Our experiment matrix spans six RTC applications and three network configurations, resulting in 15 unique experiments. Each experiment is repeated six times with 5-minute calls, totaling 90 calls and over 8 hours of captured traffic.

To capture network traffic from the iPhone devices, we use Apple's Remote Virtual Interface (RVI) technology. We connect both iPhones to a Mac via USB and use the *rvictl* command to create virtual network interfaces that expose the iPhone's network traffic, then run Wireshark [43] on the Mac to capture packets through these interfaces. The traffic is divided into the following three annotated phases:

- **Pre-call phase (60 seconds):** We begin capturing traffic 60 seconds before initiating the call. During this period, we perform standard app startup actions, such as opening the RTC application, logging in (if not already authenticated), and ensuring the app is brought to the foreground from background state. This phase is used to capture startup-related and other unrelated traffic, which later serves as a reference for filtering out non-RTC activities from the intra-call phase (period between call initiation and termination).
- **Call phase (5 minutes):** Starting from call initiation to termination, we capture all bidirectional traffic while the two devices exchange real-time audio and video. This phase aligns with the active RTC session between the caller and the callee.
- **Post-call phase (60 seconds):** After ending the call, we continue capturing traffic for 60 seconds. This phase helps us identify flows that persist beyond the call window.

[4] We select iPhone devices for our study because FaceTime is only available on iOS and macOS platforms. Although FaceTime recently introduced web-based participation for non-Apple devices, the feature lacks native protocol support.

Application	Raw Traffic			Unrelated Traffic (Filtered)				RTC Traffic	
	Volume (MB)	UDP Strms | Dgrams	TCP Strms | Segs	Stage 1 UDP Strms | Dgrams	Stage 2 UDP Strms | Dgrams	Stage 1 TCP Strms | Segs	Stage 2 TCP Strms | Segs	UDP Strms | Dgrams	TCP Strms | Segs
Zoom	2975.9	2.2k | 3.2m	2.3k | 469k	323 | 4.6k	1371 | 7.3k	919 | 252k	583 | 43.8k	**476 | 3.2m**	333 | 72.4k
FaceTime	4179.6	1.1k | 4.4m	1.6k | 284k	259 | 3.6k	664 | 2.7k	534 | 251k	434 | 14.9k	**204 | 4.4m**	124 | 1.2k
WhatsApp	2625.1	1.1k | 2.9m	1.6k | 273k	256 | 3.7k	575 | 9.5k	559 | 242k	500 | 18.6k	**310 | 2.9m**	79 | 601
Messenger	3884.1	2.6k | 4.1m	1.6k | 338k	518 | 21.1k	1182 | 9.3k	546 | 278k	397 | 41.8k	**896 | 4.1m**	239 | 3.3k
Discord	3546.2	1.1k | 4.0m	2.1k | 286k	502 | 4.3k	647 | 3.0k	501 | 253k	324 | 12.4k	**292 | 4.0m**	147 | 3.0k
Google Meet	5362.8	2.8k | 5.824m	2.6k | 720k	1287 | 20.0k	1154 | 13.6k	1390 | 663k	518 | 23.5k	**329 | 5.79m**	716 | 33.4k

Table 1: Summary of traffic traces and filtering progress across all applications. The highlighted UDP streams (strms) and datagrams (dgrms) are RTC traffic for protocol-compliance analysis.

During the experiment, we manually log timestamps for key events, including call initiation and callee join, to facilitate semantic alignment in post-analysis. We also record each device's private and public IP addresses before and after the call, which assists in determining whether the session used P2P or relay-based transport paths. After each experiment configuration, we collected 30 minutes of background activities to build a dataset for filtering unrelated traffic.

3.2 Unrelated Traffic Filtering

To study the protocol compliance in RTC traffic, we first filter out unrelated traffic generated by background activities from our captured traces. These background activities include OS updates, push notifications, online ad trackers, and LAN network management services. Otherwise, they interfere with our protocol analysis and are misclassified as non-compliant protocols.

One effective way to filter unrelated traffic is to group IP packets into *streams* based on their 5-tuple identifiers on the transport layer, i.e., source IP, source port, destination IP, destination port, and transport protocol. This grouping serves two purposes. First, many protocol behaviors span multiple packets in a single stream, such as periodic keepalives or multi-packet media delivery. Second, unrelated traffic also manifests as independent streams that can be distinguished by their timing, destination, or protocol features. Organizing packets into streams enables us to apply filtering heuristics at the stream level, making the removal of non-RTC traffic both more accurate and more scalable.

Once all packets are grouped, we apply a two-stage filtering (illustrated in Figure 2). The first stage filters streams whose active timespan does not fully align with the call window. The second stage targets intra-call background activities using protocol-aware heuristics, including TLS SNI matching, local IP exclusion, and port-based filtering. This process yields a high-fidelity RTC-only traffic dataset suitable for downstream compliance analysis.

3.2.1 Filter Streams by Timespans. We first classify unrelated traffic based on temporal alignment with the call session. We consider three types of unrelated traffic: 1) streams that begin before the call starts, 2) streams that end after the call ends, and 3) streams that span both. These flows are unlikely to be RTC-related, as legitimate sessions typically start and end in synchrony with user-initiated calls.

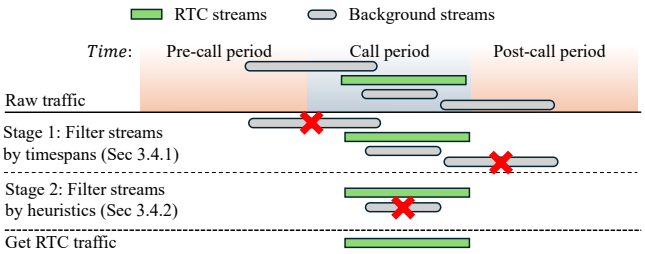


Figure 2: Two-stage filtering to extract RTC traffic.

To detect such flows, we compare each stream's active timespan to the call window. We leverage the pre-call and post-call capture periods described in Section 3.1.2. Specifically, we expand the call window slightly, by 2 seconds before and after the call, allowing for minor timing offsets or delayed packet delivery. Any stream not fully enclosed within this extended window is removed as unrelated traffic.

3.2.2 Filter Intra-call Unrelated Traffic by Heuristics. Despite stream-level filtering by timespan, we still observe some unrelated traffic within the call window. These include push notifications, app store polling, local service discovery, etc. Such activities may initiate short-lived connections that overlap with the call window, making them harder to distinguish. To address this challenge, we design a second-stage filtering process based on four protocol-aware heuristics. These heuristics aim to detect and remove intra-call unrelated traffic that evades the first-stage timing-based filter by exploiting additional side signals: IP/port reuse patterns, TLS metadata, IP address scope, and known non-RTC services.

3-tuple timing filter. Some background services, such as Apple Push Notification Service (APNS) on iOS, maintain persistent connections to remote servers throughout the experiment. These services use a fixed 3-tuple (destination IP, port, and protocol) while frequently varying the source IP or port due to NAT rebinding or OS-level socket reuse. These packets, although fulfilling the same background activity functionality, are grouped into different streams and thus evade the first-stage timespan filtering. To address this, we analyze the temporal behavior of each destination-side 3-tuple (i.e., destination IP, destination port, and transport protocol). If a 3-tuple appears outside the call window, we remove all packets in the call window that match it. This excludes consistent 3-tuple connections unrelated to the call.

TLS SNI-based filtering. For the remaining TCP traffic, 51.4% is encrypted using TLS. The encryption makes it difficult to infer their purpose based on payload content alone. To determine whether these streams are related to the RTC call, we inspect the *Server Name Indication* (SNI) field in the TLS Client Hello message, which reveals the intended destination domain before encryption begins. We construct a blocklist of known non-RTC domains (e.g., *oauth2.googleapis.com*, *web.facebook.com*) by analyzing 7.5 hours of idle-phone traffic. Any stream whose SNI matches a domain in this list is excluded as unrelated traffic.

Local IP filtering. Local network management services, such as LAN discovery, can introduce unrelated traffic during RTC calls. This is common when multiple devices are connected to the same LAN. To address this, we remove any stream where either endpoint falls within IPv6 link-local (*fe80::/10*), unique-local (*fd00::/8*), or IPv4 private address ranges, *and* whose IP pair also appears in the pre-call background capture. This ensures that we filter persistent local activity while preserving legitimate P2P traffic between the two call participants.

Port-based exclusion. Some background services communicate using well-known UDP ports unrelated to RTC, such as DNS (53), DHCP (67/547), and SSDP (1900). These streams can introduce unrelated traffic into the call window and confuse protocol analysis. To address this, we exclude any stream that uses a transport-layer port number reserved for non-RTC services, based on the IANA Service Name and Port Number Registry [2].

Together, these intra-call filters further extract RTC traffic from a variety of unrelated traffic, complementing the earlier stream-level filtering and enhancing the fidelity of our protocol compliance analysis.

3.3 Traffic Summary

We summarize the final dataset obtained after filtering in Table 1. In total, we capture 20.4 GB of raw traffic, consisting of over 26 million IP packets from 90 1-on-1 calls across six applications. After applying our two-stage filtering, the remaining dataset contains 2.9–5.8 million UDP datagrams per application, grouped into 200–900 active UDP streams.

In this paper, we focus our protocol compliance analysis solely on *UDP traffic*, because the TCP traffic volume is negligible. For instance, the proportion of TCP segments in WhatsApp and FaceTime is below 0.4% of total packets. Manual inspection confirms that these TCP sessions carry only signaling or periodic heartbeat messages, without media content. While this design choice may exclude a small portion of RTC messages in TCP segment payloads, we acknowledge this as a limitation of our study.

To evaluate the quality of our filtering process, we analyze the protocol composition of the remaining datagrams. We find that over 99% of these datagrams either contain recognizable messages or include identifiable application-specific headers. For any other datagrams, they appear in transport streams that also contain valid RTC messages, suggesting they are indeed part of the call session rather than background activities. These observations validate the effectiveness of our filtering process in isolating RTC-relevant traffic from background activities.

4 Evaluating Compliance for Protocol Messages

In this section, we evaluate whether the messages observed in RTC traffic conform to the RFC protocol specifications. These messages refer to protocol messages used in RTC communication, such as STUN, TURN, RTP, RTCP, and QUIC. We begin by using a custom DPI built upon Peaflow [37] to identify all such messages with their byte segments in the datagrams (Section 4.1). Then we assess their compliance using a structured methodology based on whether the message strictly follows the RFC protocol specifications in both structure and intended usage. We systematically check the message and attribute types, field values, and syntax (Section 4.2).

4.1 Extract RTC Messages

Deep Packet Inspection (DPI) tools are widely used to extract messages from traffic traces, such as nDPI [27], L7-filter [19], and PACE [32]. These tools typically identify protocols by matching a fixed protocol header pattern at the beginning of transport-layer payloads. They assume that standard message headers appear at offset zero and rely on deterministic parsers built strictly on protocol specifications.

However, such DPI systems are insufficient in our study due to two key limitations. First, they fail to handle proprietary protocol headers present in the datagrams. If a standard RTC protocol message is encapsulated in a proprietary protocol header, these DPIs will fail to recognize it. Second, existing tools are designed to detect only messages that strictly comply with protocol definitions. For example, if an RTC protocol message uses an unknown message type but fits all other protocol patterns, these DPIs will still not parse it as such a protocol. As a result, non-compliant messages may be missed. To overcome these limitations, we design a custom two-stage DPI, which is capable of identifying both standard protocol messages regardless of the presence of proprietary headers.

4.1.1 Deep Packet Inspection. As illustrated in Algorithm 1, our custom DPI consists of two steps: candidate extraction and protocol-specific validation. First, it attempts to find all possible messages in the datagrams: any contiguous byte sequence in the UDP payload that matches the structural pattern of an RTC protocol is marked as a candidate message. Second, it leverages protocol-specific heuristics to eliminate false positives in these candidates. After that, only structurally meaningful messages will be kept. This helps us extract all the protocol messages and also reveal the proprietary protocol headers implemented in the datagrams.

The first step is **candidate extraction**. Here we iteratively examine each UDP datagram's payload by shifting the starting position from byte offset i. At each offset, we apply the header patterns of the target protocol and check whether any of them match the byte string starting at that position. We use the header pattern as described in an existing framework Peafowl [37], but removed the header field restriction in Peafowl. For example, Peafowl only allows 30 valid payload type values for RTP [36], while we removed this restriction and allowed all values. If a match is found, we treat the matched segment as a candidate for the corresponding protocol. This offset-shifting pattern matching helps uncover protocol messages that may be embedded behind proprietary protocol headers.

Algorithm 1 DPI-based Detection of RTC Protocol Messages

Input: UDP datagram set $D = \{d | d \in$ one RTC call$\}$
Proto pattern $p \in$ {STUN, TURN, RTP, RTCP, QUIC}
Init: Candidate message set $\mathbb{C} \leftarrow \emptyset$
Validated message set $\mathbb{O} \leftarrow \emptyset$
Step 1: Candidate Extraction
for each UDP datagram $\in D$ **do**
  for $i = 0$ to k **do**
    if match(p, payload[i:]) **then**
      Extract matched message m from payload[i:]
      $\mathbb{C} \leftarrow \mathbb{C} \cup m$
    end if
  end for
end for
Step 2: protocol-specific Validation
for each $m \in \mathbb{C}$ **do**
  if protocol_validate(m) **then**
    $\mathbb{O} \leftarrow \mathbb{O} \cup m$
  end if
end for
Output: $\mathbb{O}$, message set belonging to target protocol p

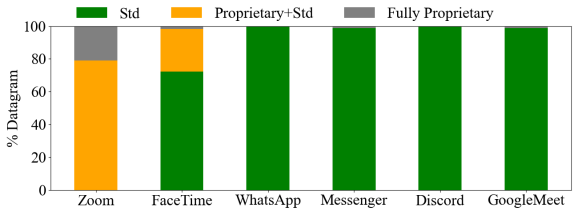


Figure 3: Breakdown of datagrams: standard vs proprietary.

Application/ Protocols	STUN/TURN	RTP	RTCP	QUIC	Fully Proprietary
Zoom	0.0%	78.9%	1.1%	N/A	20.0%
FaceTime	0.3%	97.6%	N/A	0.1%	2.0%
WhatsApp	1.2%	97.4%	1.0%	N/A	0.4%
Messenger	1.4%	87.4%	9.9%	N/A	1.3%
Discord	N/A	91.4%	7.9%	N/A	0.7%
Google Meet	19.8%	71.1%	7.8%	N/A	1.3%

Table 2: Message distribution by protocols and applications.

The candidate extraction step may yield false positives: two extracted candidate messages may claim to own several bytes simultaneously in the UDP datagram payload. For example, suppose two candidate messages are extracted at byte offsets 10 and 12 of the same UDP payload. If the first message declares a length of 40 bytes, it would span bytes 10 to 50; the second candidate message, starting at byte 12, would overlap with this range. Since a byte in the payload can belong to at most one well-formed protocol message, such an overlap indicates that at least one of the candidates must be invalid. To resolve such ambiguities, we apply **protocol-specific validation** as a second step, which checks for structural consistency and filters out semantically invalid candidates.

To address this, we design some heuristics for each target protocol, checking for field integrity, structural constraints, and internal consistency:

- **STUN/TURN:** valid *magic cookie*; valid *length* field; consistent *transaction ID* between request and response pair.
- **RTP:** valid *SSRC*, *CSRC*, and *payload type*; continuous *sequence number* within the same stream.
- **RTCP:** valid *version* field in the header, valid *SSRC identifier*; cross validated *sender SSRC* with known RTP streams.
- **QUIC:** valid *version* field in the header; consistent *DCID* and *SCID* across messages.

For example, to extract a STUN message candidate at UDP payload offset position i, our DPI will unpack the 20-byte header to extract the *message type*, *length*, *magic cookie*, and *transaction ID*. It verifies that the magic cookie matches the expected constant (0x2112A442), and then parses each STUN attribute by walking through the remaining payload as a sequence of TLV-encoded fields (2-byte type, 2-byte length, followed by a value). Parsing stops either when the payload ends or when the parsing logic exceeds the declared message length. In the validation step, we check whether the actual number of bytes following the header is sufficient to satisfy the declared message length. If not, the message is discarded as a false positive. Otherwise, we truncate the payload to exactly match the message length field.

To reduce processing overhead, we impose a maximum offset limit of k bytes (to iterator i) when extracting candidate messages. A larger k increases the recall rate in message detection, as messages that appear deeper within the payload can still be identified. However, higher values also incur substantial computational cost, especially for the streams carrying high-bitrate media streams.

Empirically, we found that $k = 200$ is sufficient. Across one representative trace from each application, extraction with $k = 200$ yielded the same set of validated messages as full-payload extraction. This confirms that our offset limit achieves a practical balance between accuracy and runtime efficiency. That said, a higher value of k may improve coverage in applications with RTC messages more deeply nested behind proprietary headers. Although our dataset did not contain any validated messages beyond offset 200, future work could explore adaptive offset bounds or application-specific heuristics to capture such edge cases when necessary.

4.1.2 Proprietary Header Detection. Our DPI framework enables us to detect datagram payloads that deviate from standard RTC protocol (i.e., STUN, TURN, RTP, RTCP, QUIC) structures. In particular, we identify two types of proprietary protocol implementations based on the presence and position of standard RTC protocol messages.

- **Proprietary headers:** If a valid STUN, TURN, RTP, RTCP, or QUIC message is detected starting at a non-zero offset within the UDP payload, we treat the preceding bytes as a proprietary header. These headers are not part of any known protocol. This case is common in Zoom and FaceTime.
- **Fully proprietary messages:** If no recognizable standard protocol message can be extracted from the entire payload, we classify the entire datagram as a fully proprietary message. This

suggests that the application is using a custom, non-standard protocol format for that datagram.

4.1.3 Message Extraction Results. We now present the results of our message identification, including the composition of these datagrams and the types of protocol messages extracted from them.

Figure 3 categorizes the datagrams into three types: (1) datagrams that consist entirely of standard protocol messages, (2) datagrams that contain a proprietary header and at least one standard protocol message, and (3) datagrams that do not match any known protocol pattern. We observe that WhatsApp, Messenger, Discord, and Google Meet rely almost entirely on standard protocol messages, indicating minimal proprietary protocol customization in their protocol implementations. In contrast, Zoom and FaceTime heavily rely on proprietary protocols. For Zoom, all of its datagrams contain a proprietary header, and 21% of them are fully proprietary. We provide a detailed analysis of Zoom's proprietary protocol implementations in Section 5.3. FaceTime follows a similar pattern, with 72.3% of datagrams containing a proprietary protocol header, which we also analyze in Section 5.3.

Table 2 shows that **these RTC applications use different combinations of network protocols**. Among the 5 applications, Messenger, WhatsApp, and Google Meet have the same set of protocols. FaceTime does not use RTCP but uses QUIC, whereas the other applications follow the conventional RTP/RTCP pairing. Discord does not use STUN at all. This may be because it always transmits its media through relay servers and never relies on P2P under any network condition. As a result, the NAT traversal function provided by STUN may be irrelevant to Discord. For Zoom, we observed it send out STUN messages while launching the app, which happens before any call is started. Additionally, STUN messages in the middle of calls only occur in P2P mode with Wi-Fi networks (Section 5.2). All the differences highlight that RTC application developers employ different strategies to solve the same set of challenges in RTC.

How well are false positives being eliminated: Because the baseline traffic composition is unknown for these closed-source applications, we cannot determine the exact number of false positive RTC-related messages. However, the extracted messages exhibit strong internal consistency after our protocol-specific validation (e.g., consecutive sequence numbers within an RTP stream, multiple RTP packets sharing the same SSRC, valid STUN/TURN transaction identifiers). We are therefore highly confident that the identified messages are genuine rather than false positives.

4.2 Compliance Assessment Methodology

For each identified message, we evaluate its compliance against the corresponding protocol specification. Protocol specifications considered include public WebRTC documentations and RFCs published by the IETF [14, 18, 28, 30, 35]. More specifically, we define five criteria and check the message in question against them in a sequential manner. A message must satisfy all five criteria to be deemed compliant; failure at any criterion will result in classification as non-compliant.

(1) **Message Type Definition.** For each message, we first identify the header field that categorizes the message, noted as *message type*, and we verify whether the type is explicitly defined in the target protocol specifications. For example, a STUN message with message type 0x0001 is compliant as it matches the *Binding Request* defined in the STUN RFC. In contrast, a STUN message with type *0x0800* is considered non-compliant, as it does not match any defined type.
(2) **Header Field Validity.** We then check whether all the rest of the header fields conform to the syntax and semantics defined in the corresponding protocol specification. Any deviation, such as a STUN message containing a *Transaction ID* that appears sequential rather than randomly generated, constitutes a compliance violation.
(3) **Attribute Type Validity.** Next, we examine fields in the message payload, where each set of TLV-encoded fields[5] is considered as an *attribute*. For each attribute, we check if its type is publicly defined in specifications. Any proprietary or undefined attribute leads to classification as non-compliant. For instance, a STUN attribute with type *0x4007*, absent from any protocol specifications, would fail this criterion.
(4) **Attribute Value Validity.** For each defined attribute, we verify that its value and structure adhere to protocol rules. As examples, a TURN *Allocate Request* carrying a *RESERVATION-TOKEN* attribute of incorrect length, or a STUN *Binding Success Response* incorrectly including a *PRIORITY* attribute, both constitute violations.
(5) **Syntax and Semantic Integrity.** Finally, we check the overall message consistency and its contextual behavior. This includes verifying structural dependencies between fields, attributes, and message types. Additionally, we consider behavioral context: for example, while a TURN *Allocate Request* may individually appear compliant, a repeated sequence of such requests forming a periodic ping-pong pattern would be flagged as non-compliant, since Allocate Requests are intended for session setup, not continuous connectivity checking.

The assessment process is strictly sequential: once a message fails any criterion, it is immediately classified as non-compliant without further checks. This ensures reliability by avoiding cascading evaluation errors based on already-invalid messages.

5 Compliance across Protocols and Applications

Having extracted all messages and evaluated them from a protocol compliance perspective, we now aim to answer the two key questions:

Q1: Across various protocols, which implementation complies most closely with its standard? **QUIC is most (100%) compliant, while the compliance of other protocols follows RTP > RTCP > STUN.** This pattern is consistent across traffic volume and protocol message types.

Q2: Across five applications, which one implements their protocols most closely with their respective protocol specification? **FaceTime is the least compliant by traffic volume.** Malformed RTP headers and undefined STUN/TURN attributes render more than 99% of FaceTime's observed bytes non-conformant, giving it the highest share of non-compliant traffic. **Discord is the least compliant by message type.** Every distinct message type we captured in Discord

[5]TLV (Type-Length-Value) is a flexible data encoding format where each data item is represented by its type identifier, the length of its value, and the actual value itself.

contains at least one structural or semantic violation in one of its messages.

5.1 Compliance Results

With all messages extracted and labeled as either compliant or non-compliant, a natural way to quantify protocol compliance is to compute the proportion of compliant messages over the total messages. For example, for each application, we aggregate all its messages and count the number of compliant messages within. Or, for each protocol, we aggregate all its messages across the 5 applications and count the number of compliant messages. We refer to this as **volume-based compliance metric**. Since RTP dominates traffic (e.g., >97% of messages in FaceTime and WhatsApp are RTP), this metric mainly reflects RTP compliance.

However, this volume-based compliance metric tends to statistically dilute protocols like STUN or RTCP. For example, even if all QUIC implementations in FaceTime are fully compliant, the overall compliance under this metric would remain low due to RTP's overwhelming volume. To complement this view, we introduce a second metric: **message-type-based compliance metric**. Rather than weighting by traffic volume, this metric treats each unique message type as a unit of analysis and marks it compliant only if *all* observed instances conform to the relevant protocol specification. This approach highlights systematic deviations in protocol usage, such as the introduction of undefined types or the overloaded use of standard attributes, regardless of traffic frequency. It is particularly useful for reasoning about engineering complexity and implementation design, as even rarely used message types often reflect deliberate customizations, development effort, or feature-specific optimizations.

5.1.1 Compliance by traffic volume. Figure 4 presents the compliance results using the volume-based metric. We found that Zoom and WhatsApp have near-perfect compliance across all transmitted messages, followed by Messenger, Google Meet, and Discord, which still maintain over 90% compliance. FaceTime exhibits the lowest compliance rate at around 1.4%, due to all its RTP messages being non-compliant.

From the protocol perspective, QUIC shows full compliance, followed by STUN at around 92%. RTP and RTCP have lower compliance ratios, at approximately 79% and 61% respectively, driven by widespread use of invalid headers and undefined attributes in several applications.

5.1.2 Compliance by Message Type. Table 3 summarizes the message-type-based compliance results across all applications and protocols. For each application, we report the number of compliant message types over the total number of observed types for each protocol. The final column aggregates these values across all supported protocols within the application. The bottom row aggregates results from a protocol-centric perspective. If multiple applications use the same message type, it is counted multiple times. This is because the same protocol element may be interpreted or modified differently by different vendors. To provide additional details, we list the specific message types observed for each application and protocol in three separate tables: Table 4 for STUN/TURN, Table 5 for RTP, and Table 6 for RTCP. QUIC is only observed in FaceTime

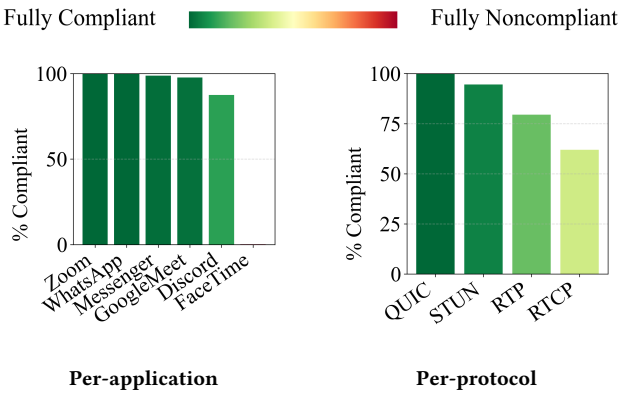


Figure 4: Compliance ratio by traffic volume.

Applications/ Protocols	STUN/ TURN	RTP	RTCP	QUIC	All Protocols
Zoom	0/2	50/50	2/2	N/A	**52/54**
FaceTime	0/4	0/5	N/A	4/4	**4/13**
WhatsApp	1/10	5/5	4/4	N/A	**10/19**
Messenger	11/18	5/5	4/4	N/A	**20/27**
Discord	N/A	0/4	0/5	N/A	**0/9**
Google Meet	15/16	11/11	0/7	N/A	**26/34**
All Apps	**27/50**	**71/80**	**10/22**	**4/4**	

Table 3: Protocol compliance ratio by message type.

traffic, where we witness long-header packets (types *0*, *1*, and *2*) and short-header packets. All the observed QUIC messages are compliant.

From a protocol-centric view (Figures 5), STUN/TURN and RTCP exhibit the highest rates of non-compliant message types, while QUIC is fully compliant, and RTP shows strong consistency with 71 out of 80 message types compliant.

From the application-centric perspective (Figures 5), Zoom is the most compliant, with 52 out of 54 message types passing validation. In contrast, Discord is the least compliant, with none of its 9 observed message types conforming to the standard. FaceTime, Messenger, WhatsApp, and Google Meet show mixed results, with varying levels of non-compliance across different protocols.

5.2 Non-compliance Case Studies

Throughout our study, we conducted a thorough examination of most non-compliance cases. Here, we present a selection of the most representative cases, each carefully chosen to illustrate a unique pattern of non-compliance that deviates from the corresponding protocol specification. These cases provide valuable insights into the specific ways in which non-compliance occurs.

5.2.1 Non-compliant cases in STUN/TURN. **WhatsApp introduces several undefined STUN message pairs.** WhatsApp uses a pair of STUN message types, *0x0801* and *0x0802*, which are not defined in the STUN protocol specification RFC 8489 [28]. This pattern is observed in both network conditions and transmission

Application	Compliant Types	Non-compliant Types
Google Meet	*0x0001, 0x0004, 0x0008, 0x0009, 0x0016, 0x0017, 0x0101, 0x0103, 0x0104, 0x0108, 0x0109, 0x0113, 0x0200, 0x0300, ChannelData*	*0x0003*
WhatsApp	*0x0001*	*0x0800–0x0805, 0x0003, 0x0101, 0x0103*
Messenger	*0x0004, 0x0008, 0x0009, 0x0016, 0x0017, 0x0104, 0x0108, 0x0109, 0x0113, 0x0118, ChannelData*	*0x0800–0x0802, 0x0001, 0x0003, 0x0101, 0x0103*
Zoom	–	*0x0002*
FaceTime	–	*0x0001, 0x0017, 0x0101, ChannelData*

Table 4: Observed STUN/ TURN message types across all RTC applications.

Application	Compliant Types	Non-compliant Types
Google Meet	*100, 103, 104, 109, 111, 114, 35, 36, 63, 96, 97*	–
WhatsApp	*97, 103, 105, 106, 120*	–
Zoom	*0, 3, 4, 5, 10, 12, 13, 19, 20, 25, 33, 35, 38, 41, 45, 46, 49, 59, 68, 69, 74, 75, 82, 83, 89, 92, 93, 95, 98, 99, 102–121, 123, 126, 127*	–
Messenger	*97, 98, 101, 126, 127*	–
FaceTime	–	*100, 104, 108, 13, 20*
Discord	–	*101, 102, 120, 96*

Table 5: Observed RTP message types across all RTC applications.

Application	Compliant Types	Non-compliant Types
Google Meet	–	*200, 201, 202, 204, 205, 206, 207*
WhatsApp	*200, 202, 205, 206*	–
Zoom	*200, 202*	–
Messenger	*200, 201, 205, 206*	–
Discord	–	*200, 201, 204, 205, 206*

Table 6: Observed RTCP message types across all RTC applications.

modes. Before the callee joins a call, the client sends/receives 16 consecutive *0x0801*/*0x0802* pairs within a burst lasting approximately 2.2 milliseconds. The *0x0801* messages are notably large, each 500 bytes in size, and include a custom attribute *0x4004* whose value consists of long zero-filled sequences. Conversely, the *0x0802*

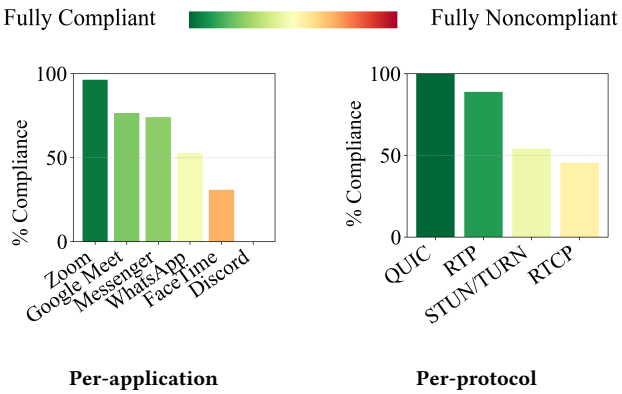


Figure 5: Compliance ratio by message type.

replies are compact 40-byte messages. Both message types carry an attribute *0x4003* with a fixed value of *0xFF*. Neither *0x4003* nor *0x4004* attribute type is defined in the STUN specification. Each *0x0801* and *0x0802* pair shares the same transaction ID. In the meantime, no standard *Binding Request* and *Binding Response* messages are observed. Therefore, these *0x0801* and *0x0802* may serve as request-response pairs.

Zoom applies undefined attributes in STUN *Binding Requests*. In P2P calls under Wi-Fi, Zoom employs the legacy version of STUN defined in RFC 3489 [33], which predates the introduction of the *magic cookie* field defined in RFC 8489 [28]. Despite the age of this protocol version, Zoom introduces additional non-compliant behaviors. Specifically, in *Binding Request* (*0x0001*) messages sent from client to server, Zoom consistently includes an undefined attribute (*0x0101*). Its attribute value is always a 20-byte ASCII string, which is composed of two identical numeric sequences: *1234567890*. Conversely, in *Shared Secret Request* (*0x0002*) STUN message sent from Zoom's server to the client, a different undefined attribute (*0x0103*) is present. The length of this attribute value is always 8 bytes. Neither attribute *0x0101* nor *0x0103* is documented in STUN protocol specifications (RFC 3489 [33], RFC 5389 [23], or RFC 8489 [28]).

FaceTime repeatedly transmits modified STUN *Binding Requests* without receiving any *Binding Responses*. FaceTime repeatedly sends out STUN *Binding Request* messages (*0x0001*) augmented with an unknown attribute (*0x8007*). Three distinct 4-byte values are observed for this attribute: *0x00000009* (present across all network configurations), *0x00000005*, and *0x00000000*. The latter two are related to the P2P mode only. Specifically, *0x00000000* is used on Wi-Fi networks, and *0x00000005* on Cellular networks. The client keeps transmitting *Binding Requests*, with the same *Transaction ID*, exactly once per second over one minute, yet no corresponding *Binding Response* with the same *Transaction ID* is ever observed. This interaction diverges from normal STUN semantics, where success responses or exponential-backoff retransmissions are expected. This suggests that FaceTime may repurpose these *Binding Requests* for other purposes.

FaceTime includes unexpected *CHANNEL-NUMBER* attribute in TURN *Data Indication* messages. RFC 8656 [30] specifies that a *Data Indication* (0x0017) must contain exactly two attributes—*XOR-PEER-ADDRESS* and *DATA*—and nothing else. The *CHANNEL-NUMBER* attribute (0x000C) is reserved for *Channel-Bind Request* (0x0009), where it maps a peer address to a 2-byte channel ID in the range 0x4000–0x4FFF. In FaceTime traffic, however, every observed *Data Indication* message carries an unexpected *CHANNEL-NUMBER* attribute whose value is a constant four-byte word 0x00000000. This deviates both the message's allowed attribute set and the attribute's prescribed length and value range, indicating a proprietary extension or overloaded use of TURN semantics.

FaceTime uses non-standard *address family* in STUN *ALTERNATE-SERVER* attribute. In FaceTime, approximately 29.4% of STUN *Binding Success Response* messages (*0x0101*) carry an invalid value in the *address family* field of the *ALTERNATE-SERVER* attribute (*0x8023*): specifically, the field is set to *0x00*, whereas the RFC mandates *0x01* (IPv4) or *0x02* (IPv6) as the only valid values. Moreover, these messages all include an undefined attribute *0x8008*, which carries a 16-byte random value. This *0x8008* attribute is also found in *Binding Error Response* messages (*0x0111*).

WhatsApp and Messenger transmit undefined STUN message type *0x0800* at call termination. We observe WhatsApp clients send four STUN messages with undefined message type *0x0800* near the end of calls. These messages are sent to the same set of servers that had been previously contacted during the call setup phase via TURN *Allocate Request* (*0x0003*). Each *0x0800* message carries both an undefined attribute *0x4000* and the standard *XOR-RELAYED-ADDRESS* attribute (*0x0016*). Messenger exhibits the same behavior, sending out six such messages to servers at call termination. Given the consistent occurrence of these undefined messages at the end of each call, they may be used for signaling the end of calls to relay servers.

5.2.2 Non-compliant cases in RTP.

FaceTime introduces undefined RTP header extensions. We observe that all RTP messages in FaceTime traffic attach one or more header extensions with undefined *profile identifiers* (e.g., *0x8001*, *0x8500*, *0x8D00*), none of which are documented in RFC 8285 [7] or related protocol specifications. These extensions appear across multiple *payload types* (*100*, *104*, *108*, etc.) and are consistently present in 100% of observed RTP messages. Such pervasive use of proprietary header extension profiles indicates that FaceTime may customize its RTP header semantics for application-specific purposes.

Discord deviates RTP header extension semantics by using reserved identifiers. Discord frequently uses RTP header extensions with the one-byte header form (profile *0xBEDE*), where the extension *local identifier* field has a value of zero in 4.91% of RTP messages. According to RFC 8285 [7]: "An ID of 0 is reserved as padding and has special semantics. Elements with an ID of 0 MUST be ignored by receivers." and "An extension element with an ID value equal to 0 MUST NOT have an associated length field greater than 0." However, in Discord's traffic, these ID=0 extension elements all have non-zero length fields and contain non-empty payloads, violating both the padding semantics and the length constraints defined in the protocol specifications.

Discord uses undefined header extension profiles in RTP messages with *payload type 120*. We also observe that 2.58% of Discord's RTP messages use undefined header extension profiles, with profile values ranging from *0x0084* to *0xFBD2*, which fall outside the standard profiles *0xBEDE* or range *0x1000~0x100F* defined in RFC 8285 [38]. These undefined profiles are exclusively observed in RTP messages of *payload type 120*.

Zoom does not randomize SSRC values across multiple calls. According to RFC 3550 [35], the SSRC identifier in RTP should be chosen randomly to minimize the probability of collision when multiple RTP streams are active. Reusing the same SSRC across calls violates this expectation and may reduce robustness in group calls. We observe Zoom assigning SSRC values from a fixed and very limited set, rather than generating random identifiers per call. For example, in the cellular setting we consistently observed the SSRC values *0x1001401*, *0x1001402*, *0x1000401*, and *0x1000402*; in the P2P Wi-Fi setting we observed *0x1000801*, *0x1000802*, *0x1000401*, and *0x1000402*; and in the relay mode Wi-Fi setting we observed *0x1000C01*, *0x1000C02*, *0x1000401*, and *0x1000402*. Within each network setting, exactly four distinct SSRC values were consistently present, and repeated experiments under the same setting showed that these SSRC values never changed across calls. This behavior indicates that Zoom deterministically assigns SSRCs based on network configuration, rather than generating them randomly.

5.2.3 Non-compliant cases in RTCP.

Google Meet does not include the authentication tag in some SRTCP messages. Google Meet uses the SRTCP protocol (RFC 3711 [40]) to encrypt its RTCP messages. In P2P Wi-Fi and cellular network settings, every message ends with a 14-byte authentication portion: a consistent 1-bit *E-flag*, a 31-bit *SRTCP index* increasing over time from 1, and a 10-byte *authentication tag*. However, when the call is being relayed and under Wi-Fi connection, we noticed that most of the RTCP messages only contain a 4-byte authentication portion, including the E-flag and the SRTCP index, without any additional bytes for the authentication tag. According to RFC 3711 [40], an authentication tag is required to be included at the end of an SRTCP message. Thus, Google Meet behavior presents a non-compliance with this specification.

Discord adds one extra byte marking transmission direction in RTCP messages. Each Discord RTCP message of message type *200, 204, 205* and *206* adds one extra byte at the end of the RTCP message. The value of this alone byte always fits the packet direction: it is always *0x00* when the packet is sent from Discord's server to the client device and is always *0x80* in the packet sent in the reverse path. This byte is undefined in any RTCP specification; nevertheless, its perfect correlation with packet direction indicates that Discord likely employs it as a proprietary direction flag.

5.3 Other Findings

Here we list all other application-specific findings other than non-compliance issues, but still worth discussing.

Zoom has datagrams containing multiple RTP messages. In our measurements, 0.21% of Zoom's RTP-carrying datagrams contain two RTP messages, a phenomenon that is consistently observed across all three tested network settings. Both RTP messages always appear in RTP payload *110*. All datagrams containing two RTP messages belong to the same stream in one call, and these two messages share the same SSRC and timestamp but carry different sequence numbers. The first RTP message always has a fixed short payload of 7 bytes, a behavior observed in every experiment, while the second RTP message carries a regular large payload (around 1000 bytes). According to RFC 3550, "typically one packet of the underlying protocol contains a single RTP packet, but several RTP packets may be contained if permitted by the encapsulation" [35]. Thus, while carrying two RTP messages in one datagram is not strictly disallowed, the expected behavior in practice is to place one RTP message per datagram. The consistent appearance of a fixed-size 7-byte-payload RTP message preceding a regular one, together with its non-consecutive sequence numbers, strongly suggests that this short RTP message is not used for carrying media data, but instead serves a different, Zoom-specific purpose.

Discord encrypts RTCP messages using a proprietary format other than SRTCP. While all Discord RTCP messages expose valid headers and SSRC fields, their payloads appear encrypted in a non-standard manner, as fields like *NTP timestamp* in *Sender Report* cannot be correctly decoded. They do not match the Secure RTCP (SRTCP) format: expected fields like the *E-flag*, *SRTCP index*, and *authentication tag* are missing. Instead, each message ends with a 3-byte trailer: the first two bytes form a monotonic counter, and the last byte signals direction (0x80 for client-to-relay server, 0x00 for relay server-to-client). These suggest a proprietary encryption scheme for RTCP payloads. This behavior is not a compliance violation, but indicates customized RTCP handling in Discord.

Discord uses zero as sender SSRC in RTCP feedback messages. In approximately 25% of Discord's RTCP Transport Layer Feedback (*205*) messages, the sender SSRC field is set to 0. This value does not match any SSRC in concurrent RTP streams, making Discord the only application in our dataset to adopt this usage. While RFC 3550 defines the SSRC as a unique identifier for each RTP participant, it does not explicitly forbid the value 0. All other header fields in these messages conform to the RTCP protocol specifications, suggesting that SSRC=0 is used intentionally. We hypothesize that they may use this for a special purpose.

FaceTime prepend proprietary headers before standard protocol messages only on relay mode. We observe that 89.2% of FaceTime's datagrams contain proprietary headers preceding RTP messages when the call is transmitted using relay mode. This proprietary header is witnessed in fewer than 50 appearances throughout any P2P mode call, no matter whether under a cellular or Wi-Fi connection. The length of these headers ranges from 8 to 19 bytes. All observed headers begin with a fixed 2-byte value *0x6000*, followed by a 2-byte field that indicates the total length of the remaining header fields plus the embedded standard protocol message.

FaceTime transmits a higher amount of fully proprietary messages under cellular than under Wi-Fi. We observe around 10% of FaceTime's RTC traffic belonging to fully proprietary datagrams under cellular network experiments. On the other hand, under Wi-Fi networks, the proportion of fully proprietary datagrams is always below 1%. These fully proprietary messages come in the same form of 36 bytes in length, starting with a fixed 6-byte attribute *0xDEADBEEFCAFE*. The last 8 bytes serve as 2 4-byte counters, whose value constantly increases during the call. These fully proprietary datagrams have a fixed packet rate of 20 packets per second with even intervals. These facts indicate that FaceTime may implement its own proprietary protocol for connectivity checks during cellular calls.

Zoom prepend media messages with a proprietary protocol header. Across all Zoom calls we examined, each RTP and RTCP packet is preceded by a 24 – 39 byte proprietary header. Prior work [25] splits this header into two sections: an SFU section and a media section. In the SFU section, a one-byte field indicates packet direction—0x00 for packets sent to Zoom's server and 0x04 for packets received from it. In the media section, another *type* field specifies payload function: *type 15* for audio RTP, *type 16* for video RTP, and *type 33 - 35* for RTCP. Using DPI-based RTC message detection (Section 4.1.1) on calls under cellular network and P2P-disabled Wi-Fi settings, we found that 6.9% of RTP/RTCP packets instead use *type 7*. In the header, type 7 acts as an additional wrapper around the original media types (e.g., 15, 16, 33, etc.), and the payload content follows the original type definition. When type 7 is present, the packet-direction byte becomes 0x01 (client → server) or 0x05 (server → client). Within the SFU section of every proprietary header, we identified a 4-byte field that remains constant for each RTP transport stream (defined by 5-tuple) within a call. We infer that Zoom uses this field as a media ID to identify each media stream session.

Zoom transmits extra filler messages not carrying media during the call. Zoom transmits a special class of fully proprietary messages, each consisting of 1000 identical bytes (e.g., all *0x01*, or all *0x02*). These messages account for 53% of all fully proprietary messages in Zoom RTC traffic. These messages do not contain any recognizable standard protocol message and are present in all observed Zoom calls. We refer to them as *filler messages*. These messages all share the same 5-tuple with one of the RTP or RTCP streams, and appear in bursts lasting 10–20 seconds at the start of each stream. During this burst, the packet rate increases from 0 to 500 packets/sec in relay mode, and from 0 to 180 packets/sec in P2P mode. Similar bursts occasionally appear intra-call without user-triggered actions. Furthermore, when the callee exits and rejoins the call, a new filler burst is immediately triggered. These behaviors suggest that Zoom may use these filler messages to probe available bandwidth by intentionally increasing the traffic load. While this mechanism may improve media adaptation, the use of undetectable, fully proprietary traffic deviates from standard-compliant RTC behavior.

6 Discussions

Observations from non-compliant implementations: Based on our study findings, we can speculate several possible motivations behind these non-compliant implementations: 1) **Potential performance optimization:** Examples include Zoom's proprietary filler messages likely serving as bandwidth probes, and FaceTime's custom connectivity checks under cellular networks. They

are likely used to support a higher resolution and a more stable call. 2) **Backward compatibility considerations:** For instance, Zoom continues to use an earlier RFC 3489 version of STUN, which may help preserve compatibility with legacy infrastructure. 3) **Different implementations of standard features:** For example, Google Meet does not include the SRTCP authentication tag in some RTCP messages, and Discord encrypts RTCP messages with a non-standard format. These applications are likely to implement their own method for the standardized functions. 4) **Possible extensions of existing protocols:** In some cases, applications add proprietary extensions to achieve functionality not directly supported by current specifications. Examples include Zoom's proprietary protocol headers and FaceTime's undefined RTP header extensions. We stress that these are informed hypotheses: as a passive measurement study without access to the application source code, we cannot determine the exact intent behind these design choices.

The prevalence of non-compliant implementation in RTC applications highlights the challenges for interoperability between them. The interoperability between RTC applications refers to the ability to establish real-time communication sessions and exchange media streams directly, without requiring extra adaptations. The consistent interpretation and processing of protocol messages across applications is the most critical to achieve this goal. The European Union's Digital Markets Act (DMA) mandates that by 2028, major platforms must support cross-application voice and video calls between end-users. Although the DMA does not specify how such interoperability should be achieved from protocol designs, protocol compliance is arguably one of the most practical solutions. However, our findings indicate that this assumption is far from reality. Across five popular applications—Zoom, FaceTime, WhatsApp, Messenger, and Discord—we observe pervasive non-compliance: proprietary headers, undefined attributes, and repurposed message types are widespread. In an interoperable future, each application would need to implement bespoke parsers to handle the protocol quirks of every other application, increasing engineering complexity and maintenance overhead. These deviations, therefore, present substantial barriers to interoperability, and overcoming them will require concerted efforts from service providers, standard bodies, and the broader RTC ecosystem.

The pervasive protocol modifications observed in today's RTC applications indicate that existing RTC protocol specifications are insufficient, and we call for community attention to the improvement and design of next-generation RTC protocols. Our findings reveal two key reasons that motivate us to call for this action: 1) **There is no widely adopted common set of RTC protocols.** Among the studied applications, each employs a distinct combination of protocols: Zoom uses STUN, RTP, RTCP, and proprietary protocols; FaceTime adopts STUN, TURN, RTP, QUIC, and proprietary protocols; WhatsApp and Messenger use STUN, TURN, RTP, and RTCP; while Discord relies solely on RTP and RTCP without using STUN. This diversity implies that the current RTC protocol ecosystem lacks a unified framework to guide developers in composing this set of protocols into a coherent application stack. As a result, developers are forced to make ad-hoc decisions on which protocols to adopt, extend, or modify to fulfill their specific requirements. 2) **Current protocols lack sufficient functionality, forcing developers to introduce modifications and proprietary protocols as extensions.** Our compliance analysis shows that the majority of non-compliance cases stem from newly designed behaviors, including implementing proprietary protocols (Zoom, FaceTime), new attributes (extensively used in STUN, TURN, RTP, and RTCP), and undefined message types (particularly in STUN). These extensive modifications suggest that existing protocols fail to meet the functional needs of modern RTC applications, or that achieving desired functionalities using existing standards would incur unacceptable complexity. Together, these observations call for a systematic effort to evolve or redesign RTC protocols to better serve the needs of current and future applications, reducing fragmentation and facilitating interoperability.

7 Conclusions

Our study provides the first message-level measurement of protocol compliance in RTC applications. We conducted analysis across five popular RTC applications—Zoom, FaceTime, Messenger, WhatsApp, and Discord, covering both cellular and Wi-Fi networks. Our findings reveal that while QUIC implementations remain fully compliant with their protocol specifications, all other RTC protocols—including STUN, TURN, RTP, and RTCP—are implemented with varying degrees of non-compliance across all applications. Also, every RTC application introduces modifications to the protocols it employs, such as proprietary headers, undefined message or attribute types, and altered message semantics, making their implementations deviate from the corresponding protocol specifications in the RFCs. We further discussed the practical implications of these widespread non-compliances, which not only introduce significant barriers to achieving RTC interoperability but also undermine the health and sustainability of the broader RTC ecosystem. These insights motivate our call for community attention towards the design of next-generation RTC protocols to achieve RTC interoperability.

8 Acknowledgments

We thank the anonymous IMC reviewers for their thorough comments and feedback. Liu and Chen were supported in part by NSF grants CNS-2431093 and SaTC-2415754. We also thank Jiachen Lu, Patrick Gough, Wei Wang, Yibo Zhao, Nengneng Yu, David Dong, and Ankit Penmatcha for their kind help.

A Ethics

This work does not raise any ethical issues.

References

[1] 2022. Digital Markets Act (DMA): Article 7 - Obligations for Gatekeepers Regarding Interoperability. https://www.eu-digital-markets-act.com/Digital_Markets_Act_Article_7.html. Accessed: May 2025.
[2] Internet Assigned Numbers Authority. 2024. Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
[3] David Baldassin, Ludovic Roux, Guillaume Urvoy-Keller, and Dino Martin Lopez Pacheco. 2024. Assessing the Interplay between WebRTC and QUIC congestion control algorithms. In *2024 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE, 1–6.
[4] Salman A Baset and Henning Schulzrinne. 2004. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017* (2004).
[5] Anastasia Belyh. 2023. Video conferencing statistics for 2024. https://www.founderjar.com/video-conferencing-statistics/

[6] Reed G Coda, Sana G Cheema, Christina A Hermanns, Armin Tarakemeh, Matthew L Vopat, Meghan Kramer, John Paul Schroeppel, Scott Mullen, and Bryan G Vopat. 2020. A review of online rehabilitation protocols designated for rotator cuff repairs. *Arthroscopy, sports medicine, and rehabilitation* 2, 3 (2020), e277–e288.

[7] R. Cordeiro, V. Singh, M. Westerlund, and C. Perkins. 2017. A General Mechanism for RTP Header Extensions. RFC 8285. https://datatracker.ietf.org/doc/html/rfc8285

[8] Google for Developers. 2023. Get started with Meet Media API. https://developers.google.com/workspace/meet/media-api/guides/get-started. Google recommends using libwebrtc for Meet Media API in native/mobile clients. Accessed: 2025-09-22..

[9] Google for Developers. 2023. Meet Media API concepts. https://developers.google.com/workspace/meet/media-api/guides/concepts. Google Meet documentation: clients use WebRTC to communicate with Meet servers. Accessed: 2025-09-22..

[10] Philipp Hancke. 2015. MESSENGER EXPOSED (report). https://webrtchacks.com/wp-content/uploads/2015/05/messenger-report.pdf. Reverse-engineering report: Facebook Messenger mobile apps used the WebRTC.org library. Accessed: 2025-09-22..

[11] Philipp Hancke. 2015. What's up with WhatsApp and WebRTC? https://webrtchacks.com/whats-up-with-whatsapp-and-webrtc/. Analysis showing WhatsApp includes pieces of WebRTC code in its mobile app. Accessed: 2025-09-22..

[12] Philipp Hancke. 2015. WHATSAPP EXPOSED (investigative report). https://webrtchacks.com/wp-content/uploads/2015/04/WhatsappReport.pdf. Investigative report examining WhatsApp internals and its use of WebRTC-related components. Accessed: 2025-09-22..

[13] Christer Holmberg, Stefan Hakansson, and Goran Eriksson. 2015. *Web real-time communication use cases and requirements.* Technical Report.

[14] J. Iyengar and M. Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://datatracker.ietf.org/doc/html/rfc9000

[15] J. Chesterfield J. Ott. 2010. RTP Control Protocol (RTCP) Extensions for Single-Source Multicast Sessions with Unicast Feedback. RFC 5760. https://datatracker.ietf.org/doc/html/rfc5760

[16] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman. 2018. Performance evaluation of WebRTC-based video conferencing. *ACM SIGMETRICS performance evaluation review* 45, 3 (2018), 56–68.

[17] Sagar Joshi. 2024. 50 VoIP Statistics to Reveal the Future of Phone Systems. https://learn.g2.com/voip-statistics

[18] A. Keränen, C. Holmberg, and J. Rosenberg. 2020. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445. https://datatracker.ietf.org/doc/html/rfc8445

[19] L7-filter Project. 2009. L7-filter: Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/. Accessed: 2025-05-04.

[20] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Los Angeles, CA, USA, 183–196. https://doi.org/10.1145/3098822.3098842

[21] Insoo Lee, Jinsung Lee, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. 2021. Demystifying Commercial Video Conferencing Applications. In *Proceedings of the 29th ACM International Conference on Multimedia* (Virtual Event, China) *(MM '21)*. Association for Computing Machinery, New York, NY, USA, 3583–3591. https://doi.org/10.1145/3474085.3475523

[22] Google LLC. 2011. WebRTC Source Code. https://webrtc.googlesource.com/src

[23] Philip Matthews, Jonathan Rosenberg, Dan Wing, and Rohan Mahy. 2008. Session Traversal Utilities for NAT (STUN). RFC 5389. https://doi.org/10.17487/RFC5389

[24] Medianama. 2023. EU designates six tech firms as 'gatekeepers' under the Digital Markets Act. https://www.medianama.com/2023/09/223-eu-dma-designated-gatekeepers/ Accessed: 2025-05-09.

[25] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. 2022. Enabling passive measurement of zoom performance in production networks. In *Proceedings of the 22nd ACM Internet Measurement Conference* (Nice, France) *(IMC '22)*. Association for Computing Machinery, New York, NY, USA, 244–260. https://doi.org/10.1145/3517745.3561414

[26] Mozilla Developer Network. 2024. Signaling and video calling — WebRTC API. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling. Accessed: 2025-05-16.

[27] ntop. 2023. nDPI: Open Source Deep Packet Inspection Toolkit. https://www.ntop.org/products/deep-packet-inspection/ndpi/. Accessed: 2025-05-04.

[28] R. Peterson, C. Jennings, and R. Mahy. 2020. Session Traversal Utilities for NAT (STUN). RFC 8489. https://datatracker.ietf.org/doc/html/rfc8489

[29] Ani Petrosyan. 2022. U.S. video call service usage during COVID-19 2020. https://www.statista.com/statistics/1119981/videoconferencing-services-us-coronavirus-pandemic/

[30] T. Reddy, A. Johnston, R. Mahy, and M. Petit-Huguenin. 2020. Traversal Using Relays around NAT (TURN): Relay Extensions to STUN. RFC 8656. https://datatracker.ietf.org/doc/html/rfc8656

[31] Tirumaleswar Reddy.K, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. 2020. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 8656. https://doi.org/10.17487/RFC8656

[32] Rohde & Schwarz Cybersecurity GmbH. 2024. PACE: Protocol and Application Classification Engine. https://www.ipoque.com/news-media/resources/brochures/product-brochure-dpi-engine-benefits-amp-key-features. Accessed: 2025-05-04.

[33] Jonathan Rosenberg, Christian Huitema, Rohan Mahy, and Joel Weinberger. 2003. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489. https://doi.org/10.17487/RFC3489

[34] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. 2008. RFC 5389: Session Traversal Utilities for NAT (STUN). RFC 5389. https://datatracker.ietf.org/doc/html/rfc5389.

[35] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. 2003. RTP: A Transport Protocol for Real-Time Applications. RFC 3550. https://datatracker.ietf.org/doc/html/rfc3550

[36] Daniele De Sensi. 2020. Peafowl: Deep Packet Inspection Library – RTP Inspector. http://github.com/DanieleDeSensi/peafowl/blob/89fdf35a18df6d1f13f4449067744999ff37ad85/src/inspectors/rtp.c. Accessed: 2025-09-22.

[37] Daniele De Sensi. 2021. Peafowl: High-Performance Deep Packet Inspection Library. https://github.com/DanieleDeSensi/peafowl. Accessed: 2025-05-04.

[38] David Singer, HariKishan Desineni, and Roni Even. 2017. A General Mechanism for RTP Header Extensions. RFC 8285. https://doi.org/10.17487/RFC8285

[39] Varun Singh, Jonathan Lennox, Christian Huitema, Bernard Aboba, and Youenn Fablet. 2024. RTP over QUIC. Internet-Draft draft-ietf-avtcore-rtp-over-quic-20, IETF. https://datatracker.ietf.org/doc/draft-ietf-avtcore-rtp-over-quic/ Work in Progress.

[40] Andreas B. Stokking and Mats Naslund. 2004. The Secure Real-time Transport Protocol (SRTP). RFC 3711. https://datatracker.ietf.org/doc/html/rfc3711

[41] Kate Sukhanova. 2023. The Latest Discord Statistics & Trends for 2023. https://techreport.com/statistics/discord-statistics/

[42] European Union. 2022. EU Digitial Markets Act. *European Union* (Sept 2022).

[43] Wireshark. 2024. The Wireshark Network Protocol Analyzer. https://www.wireshark.org