Analyzing the WebRTC Ecosystem and Breaking Authentication in DTLS-SRTP

Martin Bach
Technology Innovation Institute

Vukašin Karadžić
Technical University of Darmstadt

Lukas Knittel
Ruhr University Bochum

Robert Merget
Technology Innovation Institute

Jean Paul Degabriele
Technology Innovation Institute

Abstract

DTLS-SRTP was designed to secure real-time media communication and is found in prominent audio and video call platforms, including Zoom, Teams, and Google Meet. Notably, it is part of Web Real-Time Communication (WebRTC), a web standard enabling real-time communication in the browser. To this end, WebRTC uses multiple technologies, including HTTP, TLS, SDP, ICE, STUN, TURN, UDP, TCP, DTLS, (S)RTP, (S)RTCP, and SCTP. This amalgamation of technologies results in an overly complex system that is very challenging to audit systematically and automatically. As a result, the security of deployments of this core modern communication technology remains largely unexplored.

In this work, we aim to close this gap by developing an automated MitM testing framework (DTLS-MitM-Scanner (DMS)) to test the DTLS channel of a DTLS-SRTP connection. We use our framework to study the current state of the ecosystem in a case study spanning *24* service providers across their browser and mobile applications. Our analysis puts special emphasis on the authentication mechanism in DTLS-SRTP, where we test for *19* potential vulnerabilities that could lead to authentication bypasses for both the client and server. We find that among the *33* tested media server implementations, *19* contained vulnerabilities allowing an attacker to break authentication at the DTLS layer. For *9* of the affected systems, which serve hundreds of millions of users, we could also demonstrate that they could be exploited by an attacker to retrieve media data, assuming only Man-in-the-Middle capabilities. We highlight the impact of these vulnerabilities by building a Proof-of-Concept exploit to listen to Webex video conference calls.

1 Introduction

After the onset of the COVID-19 pandemic, Real-Time Communication (RTC) technologies experienced rapid growth in adoption and market share [60], driven by the shift to online teaching and the widespread adoption of remote work. As the proliferation of RTC technologies continues to increase, ensuring their privacy and resilience to attacks is a forefront priority for the IT security community.

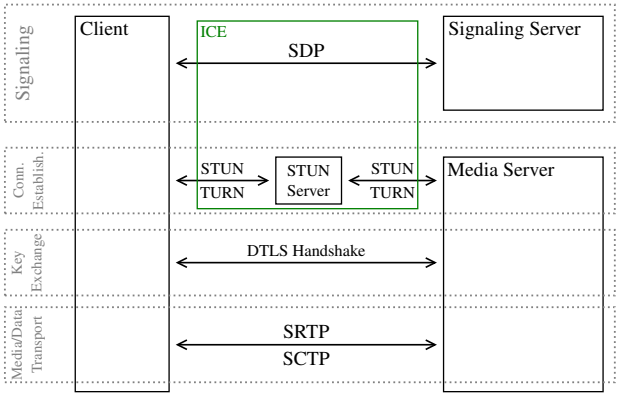

Figure 1: Phases of establishing a WebRTC connection.

At the heart of most RTC technologies is the DTLS-SRTP protocol [35], which specifies a mechanism for securely exchanging cryptographic key material between two communicating parties and then using this key material to establish secure channels for transmitting media and data. In particular, DTLS-SRTP is used in the Web Real-Time Communication (WebRTC) standard, which defines an API for browsers to establish RTC connections using DTLS-SRTP. WebRTC was initially proposed by Google in 2010 and later released as an open-source project in 2011 [4], after which it was quickly implemented in nearly all browsers. However, it was not until January 2021 that its first complete version of the standard was officially released [26]. A notable feature of WebRTC is that it was designed to provide a high degree of flexibility and be backwards-compatible with several existing technologies. Most strikingly, it does not specify a signaling protocol and attempts to be backwards-compatible with all possible options. As we shall see, this added complexity has a toll on security.

A textbook WebRTC call between two parties, *A* and *B*, would proceed as follows. Both parties must typically authenticate to the service through some web application, for example, by opening the service's webpage and logging into their account. This establishes a secure connection (TLS) with the *signaling server*. The signaling server then acts as an intermediary to assist *A* and *B* in establishing a direct connection. Now *A* can initiate a call with *B* through the signaling server, which relays the appropriate signaling messages. Both parties then generate self-signed certificates and exchange their fingerprints, along with several other call parameters, in a standardized text-formatted Session Description Protocol (SDP) message [10]. Next, they perform connectivity checks via the Interactive Connectivity Establishment (ICE) protocol [39] (STUN/TURN) in order to bypass potential NATs and communicate directly without further assistance from the signaling server. Once a connection has been established, a Datagram Transport Layer Security (DTLS) [46, 47, 48] handshake is initiated between the two peers, where one of the peers will act as a server and the other as a client. During the DTLS handshake, both parties present their self-signed certificates, thereby binding their identities to the ones established during signaling. If both can match the peer's certificate fingerprint against the one provided in the SDP, the handshake proceeds.

Once the DTLS handshake concludes, symmetric keys are exported from the DTLS [42] to establish Secure Real-time Transport Protocol (SRTP) and Secure RTCP (SRTCP) channels or used directly to secure Stream Control Transmission Protocol (SCTP) communication, as needed. Namely, media packets are protected via SRTP, whereas control traffic is carried over RTP Control Protocol (RTCP) and protected using SRTCP, both of which are specified in [8]. In addition to media, applications may negotiate data channels transported as SCTP over DTLS [57, 61]. The symmetric cryptographic algorithms, called protection profiles, are negotiated during the DTLS handshake via a DTLS extension [35].

While WebRTC was originally envisioned to be peer-to-peer (P2P), in reality, many applications relay or remix the communication through an intermediate media server. In such cases, both *A* and *B* establish separate DTLS-SRTP connections with the media server instead. This provides the application with more flexibility, allowing for compression and advanced features such as cloud recording, where encryption is not end-to-end. In SRTP, media streams are identified and multiplexed using an Synchronization Source (SSRC) identifier that is included in the Real-time Transport Protocol (RTP) header. The SSRC is exchanged in the SDP and helps the other participants or a forwarding party to map media streams to the respective entity.

The DTLS-SRTP/WebRTC ecosystem presents significant challenges for automated security analysis due to its complex architecture and the interplay of numerous protocols and standards.

Multitude of Components. A typical WebRTC session involves several protocols: HTTP, TLS, SDP, ICE, STUN, TURN, UDP, TCP, DTLS, (S)RTP, (S)RTCP, and SCTP. Each system may employ a distinct subset with different settings. This complicates the attack surface, and building automated tools that account for all these technologies and their possible configurations becomes particularly challenging.

Interleaved Protocol Progression. The DTLS connection cannot be tested in isolation, as it requires progression through both the signaling phase and the connectivity establishment phase. As discussed, signaling is not specified beyond SDP, with many details left open to the application layer. Typically, users must sign into a web application to trigger signaling and connection establishment, as otherwise, no server will listen for incoming DTLS connections. Thus, each test subject requires specific customization of the testing tools in order to trigger a DTLS-SRTP connection, creating per-application overhead that testers ideally want to avoid.

Hidden Signaling. A major challenge for system testing is that signaling is conducted through encrypted channels. Accordingly, external testers cannot observe or influence the negotiated parameters. This includes crucial information such as ICE candidates, SSRC values, X.509 certificate fingerprints, IP addresses, and ports of media servers. Consequently, testers are restricted to the same capabilities as a true Man-in-the-Middle attacker, limiting the scope of executable tests. While applications can be modified to exfiltrate private keys, this creates additional overhead per application or browser.

Parallel Connections. Real-world applications often establish multiple simultaneous DTLS-SRTP connections, typically using separate connections for different media types (audio and video). Additionally, distinct endpoints may not behave consistently due to load balancing and random port allocations, making it challenging to consistently test the same logical endpoint.

These challenges have rendered WebRTC particularly unattractive to test, resulting in limited tooling for analyzing the security of deployed implementations, especially regarding DTLS. As a result, this huge ecosystem used by hundreds of millions of users remains largely unexplored, which leads us to our first research questions:

> **RQ1:** What is the status of the DTLS-SRTP ecosystem? Which cryptographic algorithms and DTLS features are used to secure RTC communication?

To answer this question, we developed an automated DTLS-SRTP testing platform, called DMS, to probe the DTLS component of any party willing to establish DTLS-SRTP/WebRTC connections. Through it, we can gather information about the DTLS implementation and configuration of a target system, such as supported cipher suites and extensions, version support, assumed DTLS role, and RFC compliance, among other details. Our testing platform is built on TLS-Attacker [56], pcap4j [66], Selenium [51], and jitsi-srtp [27], and is entirely

black box—requiring only MitM access and no alterations to the system under test. Equipped with this testing platform and having noted several differences in the way that DTLS is employed in DTLS-SRTP, we were then faced with the next important question:

> **RQ2:** Is DTLS deployed securely in DTLS-SRTP, and are connections securely authenticated?

To address this question, we extended our testing platform with a suite of tests that specifically target the authentication mechanism in DTLS. We use it to analyze the behavior of many popular DTLS-SRTP implementations, with a special focus on WebRTC. Our study includes various browsers, mobile platforms, and apps, as well as numerous popular web applications, including Zoom, Discord, Teams, Google Meet, Webex, and more, through an extensive case study.

Results. Our case study reveals a diverse ecosystem comprised of numerous algorithms and features across the tested applications. Among the *24* tested applications, *19* implementations contained vulnerabilities that allowed an attacker to complete the DTLS handshake with their peer without owning the private key to the certificate. While this may not be enough for a complete exploit, we confirmed for *9* of those vulnerabilities that the vulnerability allows the attacker to receive media data from the peer. This would allow an active MitM attack to effectively join a media call instead of the intended client, thereby compromising the confidentiality of the media connections. Last but not least, we propose additional hardening measures that can be implemented in DTLS-SRTP implementations (like browsers) to reduce the potential for dangerous misconfigurations.

Contributions. We make the following contributions:

- We create the first testing platform, DMS, for analyzing DTLS-SRTP implementations in a black-box manner, requiring only MitM capabilities (Section 3). Our test framework is modular and open source.
- We present a suite of tests to perform a thorough evaluation of the DTLS-SRTP ecosystem, focusing on the DTLS component, gathering data on algorithm support and deployment practices, and, specifically targeting authentication (Section 4).
- We perform the first case study of the DTLS-SRTP ecosystem, spanning *33* media server implementations, with a particular focus on DTLS authentication. Among the tested implementations, we identify *19* cases where authentication is broken at the DTLS layer. In *9* of these cases, we find that the vulnerability is exploitable and can be used to decrypt media data from a pure Man-in-the-Middle position. Among the vulnerable applications are popular services with hundreds of millions of users, including Webex, Discord, Zoom, and Steam (Section 5).
- We demonstrate how an attacker can fully exploit the identified vulnerabilities and eavesdrop on teleconferencing calls, by implementing a Proof-of-Concept exploit for one of the identified applications (Webex) to highlight the severity of the discovered issues (Section 6).
- We analyze browser implementations of the WebRTC API, focusing on certificate generation and validation. Our findings reveal that all tested browsers accept weak 512-bit RSA certificates from media servers, and some browsers allow generating certificates with potentially insecure parameters (Section 7).

2 Background

Datagram Transport Layer Security (DTLS) [48] is a variant of the TLS protocol [43] aiming at providing equivalent security guarantees over datagram-based transport protocols like UDP. This requires DTLS to implement additional features, such as explicit sequence numbers, in order to reliably retransmit handshake messages.

To establish a DTLS connection, the client sends a *ClientHello* message, which includes a nonce, the highest supported protocol version, supported cipher suites and compression algorithms, an optional session ID of a previous connection, as well as a list of supported extensions. The server responds with a *ServerHello* message, which specifies the selected protocol version, cipher suite, and compression algorithm, a session ID, server nonce, and a list of extensions. The server then sends a *Certificate* message to the client containing its X.509 certificate chain. When the server selects an ephemeral cipher suite, it additionally sends a *ServerKeyExchange* message containing its ephemeral public key signed with the certificate's private key. If the server is configured to request client authentication, it will additionally send a *CertificateRequest* message to indicate this. The server then finishes its flight by sending a *ServerHelloDone* message, which indicates to the client that the server is awaiting a response from the client. If the server requested client authentication, the client sends its client certificate (or certificate chain) in a *Certificate* message. If the client does not have a suitable certificate, this message can be left empty. The client then proceeds to send a *ClientKeyExchange* message, which contains the client's ephemeral public key. If the client was able to send a certificate, it also sends a *CertificateVerify* message containing a signature over the hash of the current transcript of the handshake. The signature is computed with the private key of the client's certificate, thereby proving to the server that it is in possession of the certificate's private key. At this point, the client and the server have all the necessary information to compute the shared secret for the DTLS connection. Using this shared secret and a PRF, both parties evaluate a master secret, which is in turn used to derive the symmetric keys for securing data. The client then sends a *ChangeCipherSpec* message, indicating to the server that all subsequent messages will be encrypted using the negotiated keys, followed by a *Finished* message containing a crypto-

graphic checksum over the handshake transcript. Upon receiving this message, the server will recompute the checksum to verify that both parties saw the same handshake messages. If successful, the server will send its own *ChangeCipherSpec* and *Finished* messages as confirmation to the client. The handshake is complete, and application data can now be exchanged.

To prevent DoS attacks, a DTLS server can request the client to prove its ability to receive server messages and is thus not spoofing its IP address. To this end, the server can respond to the *ClientHello* message with a *HelloVerifyRequest* message containing a stateless 'cookie'. In return, the client will retransmit its *ClientHello* message with this cookie, to prove that it received it.

2.1 DTLS-SRTP

Real-time Transport Protocol (RTP) [50] is a protocol for delivering audio and video over IP networks, offering timing information and sequence numbering. Secure Real-time Transport Protocol (SRTP) [8] extends RTP by additionally providing confidentiality through encryption, message authentication, and replay protection for media streams. However, SRTP still requires an external key management mechanism to exchange the necessary symmetric keys. Accordingly, the DTLS-SRTP protocol augments SRTP with the DTLS handshake. During the DTLS handshake, both endpoints authenticate using certificates, which can be self-signed and generated on the fly. Self-signed certificates are authenticated by exchanging their fingerprints in Session Description Protocol (SDP) during signaling. Once complete, the shared secret is used to generate the master keys and salts required by SRTP to protect media packets.

2.2 WebRTC

Web Real-Time Communication (WebRTC) is a protocol suite enabling secure real-time communication in browser applications. It is specified jointly by the IETFs' *rtcweb* working group and W3Cs' *Web Real-Time Communications* working group. The IETF is responsible for defining and standardizing the protocols used in WebRTC, while W3C is tasked with standardizing the browser API WebRTC. Today, virtually all teleconferencing services (e.g., Zoom, Discord, Google Meet, Webex) use WebRTC in their web applications. Additionally, it is used in browser applications for streaming and voice interaction with AI agents.

2.2.1 WebRTC Connection Establishment

A WebRTC connection proceeds in four phases, as illustrated in Figure 1.

Signaling Phase. A WebRTC connection starts with the signaling phase, where two parties exchange SDP messages via a signaling server to negotiate parameters for the media connection and make proposals on how to establish a connection by exchanging Interactive Connectivity Establishment (ICE) candidates. Both parties exchange fingerprints of the certificates that they will use in the DTLS handshake, and they also negotiate which peer will assume the role of DTLS server and that of DTLS client.

Connection Establishment Phase. Once signaling is complete, they proceed to establish a network connection in order to communicate with each other. Albeit straightforward in a centralized setting, where clients (i.e., browsers) simply connect to a central media server, connection establishment is more challenging when both peers of the connection are clients. Clients are often behind NAT gateways and are thus not aware of their public IP address, and may also be restricted by firewalls. The ICE protocol [28] is used to overcome these limitations. In turn, ICE can use Session Traversal Utilities for NAT (STUN) [38], Traversal Using Relays around NAT (TURN) [37], or try to establish a direct connection. The STUN protocol allows a peer to learn its public-facing IP address and port. TURN is an extension of STUN, used when direct communication between two peers is not possible, for example, due to a firewall. In such cases, the peers can communicate via a TURN server that relays their communication. ICE will try different approaches until a connection is established.

Key Exchange Phase. Once a connection is established, both peers need to exchange keys to secure media and data traffic. As previously negotiated during signaling, one party will act as a DTLS client and the other as a DTLS server. Keys are exchanged using the DTLS handshake with client authentication, using the certificates that the parties have committed to in the signaling phase. Once the handshake is complete, the master secret of the connection is used to export keys for the media channel.

Media Phase. The peers are now ready to start exchanging media through the SRTP protocol. Video and audio are encapsulated in SRTP packets, whereas connection metadata and other control information are transported using SRTCP [8, 50]. The usage of SRTP and the concrete cryptographic algorithms are negotiated in the key exchange phase through the `use_srtp` DTLS extension. Additionally, WebRTC applications can also use SCTP [57, 61], typically for chat messages and meta information, which will be protected by DTLS directly.

2.3 TLS-Attacker

TLS-Attacker [56] is an open-source framework for analyzing TLS and DTLS implementations. With TLS-Attacker, users can generate arbitrary protocol flows and modify the structure of the included protocol messages at runtime.

3 DMS: Testing DTLS-SRTP

While previous work on WebRTC focused on the signaling channel and Man-in-the-Middle attacks with a malicious signaling server, other potential flows, like MitM-based attacks on the authentication on the DTLS layer that exploit implementation and configuration flaws, have not received the same attention. In this work, we therefore want to investigate the potential for Man-in-the-Middle attacks *without* a malicious signaling server.

For this work, we focus on the security of the DTLS connection. The security of DTLS is very closely related to the security of the TLS protocol, which has been heavily analyzed in the past and is considered, when implemented and configured correctly, as secure. However, no public study has analyzed whether DTLS-SRTP is implemented and configured securely in real-world systems. The DTLS protocol offers various features, some of which are no longer considered secure or state-of-the-art, which can lead to severe vulnerabilities. At the same time, incorrectly implementing the protocol can completely break all security properties of the protocol. To assess whether an implementation is correctly implemented and configured, we will use system tests that allow for the dynamic testing of implementations without significant changes to the system under test.

3.1 System Tests for DTLS-SRTP

To answer our research questions, we built a framework to analyze DTLS-SRTP applications on a system test level, without hooking deeply into the tested application, such that we can support a plethora of platforms and applications without custom code for each, beyond automating the startup of the application. We achieve this by leaving the tested application as is, not interfering with the signaling of the application, and only interacting with the tested application like a normal user (potentially by emulating mouse clicks and button presses) and then performing our tests from a Man-in-the-Middle position. Crucially, our approach does not require installing any certificates or private keys on the system under test nor do we modify the application's trust store to decrypt traffic. This makes our testing framework entirely black-box and portable across different platforms and applications. The overall architecture of our testing platform is visualized in Figure 2.

1. Booting. To start the analysis, DTLS-MitM-Scanner (DMS) can request the startup of a local application with a *Booter*. The Booter abstracts away the concrete steps to start or stop a DTLS-SRTP connection, for example, logging into the web application and starting a video conference, or hanging up a call on the web application. Booters can then either automatically start or stop the application using scripts or Selenium, or can be implemented by manually starting/stopping the application on the request of DMS.

2. Traffic Filtering. To enable testing, our testing tool is brought into a Man-in-the-Middle position, where it is able to intercept the network traffic between the media server and the local application. Using `iptables` and pcap4j [66], we route all UDP traffic from both communication directions to our analysis tool, where we make a decision of whether the UDP packet is interesting for further analysis (i.e., it belongs to a DTLS-SRTP connection). Packets that are not interesting get forwarded, while packets that are will be sorted into processing queues associated with a logical DTLS-SRTP connection.

3. Performing the Test. Once we are able to receive UDP packets and associate them with logical connections, test execution can start. We group the tests we want to execute into semantically similar test groups called a *Probe*. Each probe can request connections to be started via the Booter interface to then execute multi-context TLS-Attacker `WorkflowTraces`. The Probe then uses these results of the execution to draw conclusions about the configuration and implementation of the system under test. To implement our tests, we extended the TLS-Attacker framework with support for STUN and TURN, as well as additional *Actions* to better support our test cases. For each `WorkflowTrace`, we always try to find `evidence` of the behavior. For example, when testing for supported cipher suites, it may happen that we do not get a response from the system under test. From this, we could conclude that the cipher suites we offered were not supported. However, it may also be that for some application-specific reason, the media server closed the DTLS endpoint entirely, and that the server is, in fact, supporting a cipher suite we offered. We therefore execute tests where we did not get a final answer at the point of test, up to 5 times or until we get a definite answer (like an *Alert* message). We then treat the final answer as the answer of the target to the test. After all tests have been performed, we use the booter interface to bring the application back into its starting state. For the decryption of media traffic, we use the jitsi-SRTP library [27]. To collect evidence for multiple different behaving DTLS-SRTP implementations, we fingerprint all seen *ClientHello* messages, *ServerHello* in response to an unmodified *ClientHello*, and *Certificate* messages. For *ClientHello* messages, we use JA3 [49] fingerprints, for *ServerHello* we use JA3S [2] fingerprints, and for certificate messages, we built a custom similar solution based on hashing the number of certificates, the length of the issuer, the length of the subject, the public key OID and the signature algorithm OID.

4. Reporting. After all probes have been executed, a report with the results is returned. If this report contained more than one fingerprint for each analyzed message type, we conclude that there are multiple different behaving endpoints. From that point on, we attempt to manually identify a pattern that allows us to distinguish between the two endpoints. We discard the report and restart the scanning with an additional filter in place that only triggers on connections that have the selected

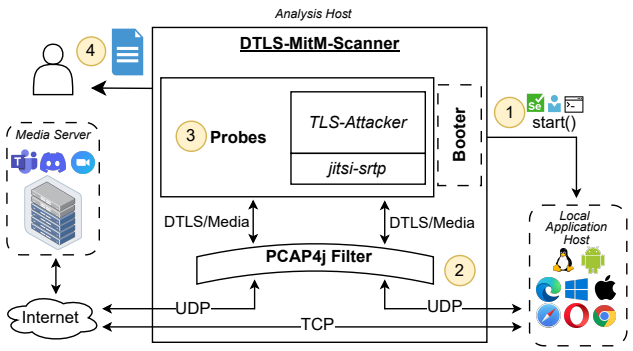

Figure 2: Sketch of the DMS architecture.

characteristic (i.e., a certain source port, JA3 fingerprint, or source IP).

3.2 Limitations

Missing TCP support. To minimize overhead, we deliberately do not intercept TCP traffic. Under this constraint, if a peer fails to receive responses on its UDP-based ICE candidate pair within the retransmission window, it will be marked failed, and the application switches to an alternative pair. In our observations, applications typically first migrate to a TURN-based UDP path and, if that also fails, may select a TCP-based ICE candidate. Because our tool monitors only UDP, any DTLS-SRTP association that moves to a TCP path becomes invisible to us. It is therefore essential that we process all UDP packets promptly to avoid triggering such fallbacks. Moreover, sessions that select a TCP-based candidate right at the start, such as Viber and Cloudflare Agents, are outside the scope of our evaluation.

TURN Channels. Our implementation supports TURN but does not support TURN channels [33]. To account for this, we drop all TURN channel-establishment messages, preventing the establishment of a TURN channel within the TURN connection. This forces peers to use STUN *SendIndication* and *DataIndication* to exchange payloads, if they support them. However, we are unable to analyze applications that rely exclusively on data exchange through TURN channel messages.

4 Implemented Tests

We added various tests to our framework to analyze the implementation and configuration of the target systems.

4.1 Property Tests

To answer **RQ1**, we built tests using TLS-Attacker to assess specific properties of the involved implementations. We organize the tests we perform into *probes*, each of which is responsible for a set of properties.

Selftest. To explain how these probes operate, we will first introduce the most basic one, the `SelfTestProbe`. The `SelfTestProbe` first starts the connection with the booter interface. Then the probe just forwards the messages between the peers until the client sends its certificate. After that, we use the booter to reset the application. This test allows us to verify that the booter is working properly, i.e., able to start the DTLS-SRTP connection, that we are able to intercept the traffic correctly, and also that we are able to forward messages correctly. Additionally, the probe allows us to read many properties of the connection straight away. The *ClientHello* message contains the client-supported cipher suites, the highest protocol version, supported compression algorithms, and supported extensions. The *ServerHello* tells us which parameters would be naturally negotiated between the client and the server (version, cipher suite, compression algorithm, and extensions). Additionally, the test shows us the structure of the certificates that are being used by both peers. Last but not least, the test allows us to see if the server requires the client to authenticate.

Basic DTLS Properties. We implemented a group of probes tasked with retrieving parameters from the server's configuration that are not visible through passive observation. In these tests, we intercept the original *ClientHello* message from the client and replace it with a crafted *ClientHello* message designed to force the server into negotiating different parameters to probe it for support for different parameters. For example, to test the different server-supported cipher suites, we first send a *ClientHello* message with all TLS-Attacker supported cipher suites that are allowed in DTLS (317). The cipher suite the server chooses is then considered supported. In the next connection, we then propose the same list of cipher suites, excluding those the server has already selected in a previous connection. We repeat this process until the server no longer chooses a cipher suite. We perform this test for the supported cipher suites, protocol versions, signature algorithms, named groups, and SRTP protection profiles.

4.2 Authentication Bypasses

To answer **RQ2**, we developed a total of *19* tests for the authentication mechanism of DTLS-SRTP. Some of the tests we designed are motivated by the existing literature. The rest were constructed partially in an ad-hoc manner, based on our experience and knowledge of common pitfalls that occur in (D)TLS implementations, while taking into account the specific nature of the WebRTC ecosystem that is based on self-signed certificates. We group our tests into the following categories.

Certificate Requested. If the DTLS server is not configured to request a certificate from the client, the client does not authenticate at all, meaning that the server later has no chance to detect illegitimate clients on the DTLS layer, leading to

a trivial 'authentication bypass'. We therefore test if DTLS servers request client authentication.

Authentication Required. DTLS libraries usually support *optional* and *required* authentication. *Required* authentication means that the server will not accept connections in which the client did not send a valid certificate during the handshake. With *optional* authentication, the DTLS server will also finish DTLS connections if the client did not present a certificate at all. In those cases, the DTLS library 'marks' the connection internally as not authenticated but still hands the connection to the application for consideration. In the context of WebRTC, optional authentication should not be used, as both parties *must* authenticate to establish a secure connection [20, Chapter 5]. To test if optional authentication is supported, we try to connect to the server with an empty certificate message.

Performs Identity Check. Another potential flaw that either peer can make is related to the identity check. Peers have to not only check that they receive *a certificate* from their peer, but also need to check that the certificate that they receive is the correct one (i.e., with the same fingerprint as exchanged in the SDP). To test if peers verify the identity, we differentiate between two cases: we either try to authenticate with a completely unrelated certificate (i.e., the TLS-Attacker default certificate) or try to authenticate with a *mimicry* certificate. This certificate mimics the expected certificate in all regards (same key type, same subject, same issuer, etc.), but the expected public key (and signature), which we replaced with our own. We perform this test to rule out identity checks on other (insecure) metrics, such as the common name. Variants of this test involve presenting a mimicked certificate alongside the original peer certificate in the hope that the system under test will authenticate us based on the original peer certificate fingerprint, while completing the handshake with keys from our own certificate.

Incorrect Trust Store. While DTLS-SRTP implementations are supposed to only accept certificates that match the exchanged certificate fingerprint, incorrectly configured DTLS implementations might also accept certificates that are accepted in general by the operating system's trust store. Therefore, it might be possible to confuse a peer into accepting a certificate with an incorrect fingerprint, which is generally trusted by the browser/Internet PKI. To perform this test, we send a certificate we received from Lets Encrypt [30] using RSA-2048 with SHA256, and ECDSA P384R1 with SHA384 for a domain under our control.

Signature Verified. When signatures are used in key exchange protocols, it is important that peers also verify the correctness of the transmitted signatures. As mentioned by Maehren et al. [32], the (D)TLS RFCs[1] never explicitly mention that peers are supposed to verify signatures. We therefore also perform tests for both peers where we invalidate the signature (by flipping bits in the middle) in the *ServerKeyExchange* and *CertificateVerify* messages to test if peers are correctly implementing this implicit requirement.

No Flow Bypass. Since DTLS is usually used on top of UDP, it is possible that messages naturally arrive out of order. However, implementations should not process messages out of order; instead, DTLS implementations *can* buffer messages that arrive out of order and process them at a later point. If an implementation can be tricked into processing out-of-order messages (maybe with an incorrect *message sequence number*, the implementation's internal state might get confused into accepting connections it should not accept. A prominent example of this was shown by Fiterau-Brostean et al. [21], who were able to present multiple variations of authentication bypasses in JSSE [36]. Inspired by Fiterau-Brostean et al. [21], we considered three different tests, a handshake where we omit the *Certificate* message and are therefore not presenting an identity, a handshake where we omit the *CertificateVerify* message and are therefore not proving that we are not in possession of the private key, and last but not least a handshake where we are neither presenting a *Certificate* nor *CertificateVerify* message, ignoring authentication completely. If any of these handshakes are completed, we have a potential authentication bypass.

Public Key Protected. The public key of each peer in mutual DTLS is protected by a signature, which, in the case of the client, is computed over the session transcript, while in the case of the server, it signs the public key with the nonces from the hello messages. Since DTLS implementations have to be flexible regarding their received message order, we try to see if it is possible to inject a second public key into the connection that is *not protected* by the signature. Our hope is that the peer verifies the signature with the original key, while it computes the shared secret using our maliciously injected key. For the client's public key, we do this by sending a *ClientKeyExchange* message after the *CertificateVerify* message. This out-of-order message should be discarded by the server. However, if it does not implement the state machine correctly and processes the message, the *ClientKeyExchange* message can potentially overwrite the client's public key in the server's internal state, allowing the attacker to bypass client authentication. We perform the same test for the server, sending a second *ServerKeyExchange* message after the first initial *ServerKeyExchange* message. For this test case, we are less optimistic about the results, as the *ServerKeyExchange* message contains a signature that we, as an attacker, cannot forge, meaning the client has to process the out-of-order message and ignore or not act on the invalid signature. In both cases, we do the test twice, once with a *correct* message sequence number and once with the same sequence number that the original key exchange message had.

[1] RFC 5246 (TLS 1.2), RFC 8446 (TLS 1.3), and RFC 6347 (DTLS 1.2)

4.3 Exploitability Tests

An issue that arises from our testing approach is that a completed DTLS handshake, which reveals an authentication flaw on the DTLS layer, may not necessarily result in real exploitable behavior on the application layer. We identified three main reasons for unexploitable issues:

- **Delayed Client Authentication.** Implementations could verify the state and properties of the established DTLS connection *after* the handshake was completed. Implementations can then abandon the connection before using it to send any sensitive data, making the perceived vulnerability unexploitable.
- **Application State Signaling.** Another mitigation could be implemented on the application layer. If only one of the peers is vulnerable to an authentication bypass, it may be that the peers wait for a signal on the application layer before they transmit data. If one of the peers never finishes the DTLS connection, the signal to start media data transmission may never be sent, preventing the leakage of confidential data to the attacker.
- **Application Layer Authentication.** Applications are free to not rely on the security of the DTLS-SRTP channel at all and can implement their own cryptography with their own authentication mechanisms on top of it. This results in a custom security architecture that no longer follows any public standards and is therefore also very difficult to test automatically.

At the same time, analyzing if a detected vulnerability is actually exploitable is challenging, as we do not have access to the details of the remote implementation. Applications use a diverse mix of protocols and technologies in the media connection, which may require target-specific messages on the media channel from the attacker to trigger the flow of media traffic, hindering an exploitability analysis. To better understand the impact of our identified vulnerabilities, we conduct additional tests to explore whether the observed behavior is actually exploitable and to rule out any limiting factors. Furthermore, we consider vulnerabilities as exploitable for which the vendor has applied a patch after our disclosure (unless otherwise communicated). We want to emphasize that the fact that we cannot show exploitability does not necessarily mean that the issue is not exploitable, as we are working in a black-box scenario.

Unprovoked Media. Some applications are willing to send media data immediately after finishing the handshake. We, therefore, added a probe that analyzes the behavior of the peer after a successful DTLS authentication bypass with our analysis tool. If a peer sends media data that we can decrypt, we conclude that the vulnerability is exploitable.

Ruling out Delayed Client Authentication. To rule out that the authentication test is simply performed after the DTLS handshake is completed, using the information presented in the DTLS handshake, we need to send the 'correct' application data and check if the peer responds with media data. For example, Discord's media server requires a client to send a valid SRTP message with a correct SSRC before it will start sending media to the client. Another example is Cisco Webex, which needs a data channel setup and a completion of the Webex Multistream protocol [15] before the attacker can receive media from the server. To avoid reverse engineering of applications and ensure the correct message is sent, we use a specialized browser to test the exploitability of web applications. This browser has been modified to accept any certificate fingerprint presented by our analysis tool, enabling us to use it to interact with the web application and generate application data. DMS will then use this application data within a connection in which it performed the authentication bypass. If we do not receive media data, we conclude that the application is merely performing the authentication check after the DTLS connection is established. In contrast, when we receive media data, we conclude that authentication on the DTLS layer is truly broken. For non-web-based applications, we omit this evaluation. Other exploit-hindering measures on the application layer may still be in place, but we consider their analysis as out of scope for this work.

5 Server Evaluation

We analyzed the state of the ecosystem and DTLS-SRTP implementations using our framework. As applications, we chose the web applications (WebRTC) of multiple different audio and video conferencing and chat systems. We performed all WebRTC tests on web applications using Chrome. To test DTLS-SRTP implementations on other platforms, we selected a similar list of Desktop and Android applications. The selected applications were chosen based on perceived popularity to explore prominent use cases across diverse platforms. More details on our test methodology can be found in Appendix A.

5.1 General Properties

DTLS Versions. There exist three distinct DTLS version, DTLS 1.0 [46] (2006), DTLS 1.2 [47] (2012), and DTLS 1.3 [48] (2022). Across all tested platforms, only Adobe Connect supported DTLS 1.0. In contrast, DTLS 1.2 was supported by *every* tested implementation. Support for DTLS 1.3 was effectively nonexistent at the time of our experiments. Firefox officially added DTLS 1.3 support with version 127. We used version 137 for our tests and found that it does not come with DTLS 1.3 enabled. At the time of writing, we observed that DTLS 1.3 was re-enabled in Firefox in later versions, suggesting that the absence of DTLS 1.3 in Firefox 137 was a bug. The lack of support for DTLS 1.0 indicates that DTLS-SRTP is atypical compared to recent studies on the

general DTLS ecosystem by Erinola et al. [18], where most servers supported version 1.0 and 1.2 simultaneously. The full list of DTLS versions supported across tested platforms is present in Table 1.

Cipher Suites. In Table 1, we also list the cipher suites that the tested applications supported. Cipher suites with weak parameters were generally not supported by any tested application. All implementations supported forward secure key exchange algorithms and AEAD cipher suites. In general, we did not observe any support for exotic TLS cipher suites or known broken cipher suites, such as `EXPORT` or `NULL`. Cipher suites using 64-bit block ciphers (vulnerable to the Sweet32 attack) were observed only in the Instagram web application. However, since servers do not negotiate this cipher with browsers (due to missing support), there is no real impact.

For WebRTC specifically, all implementations must support `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` [45]. During our evaluations, this cipher suite was often supported, but not universally available in the deployed configuration. Additionally, WebRTC implementations must favor cipher suites with forward secrecy over non-forward secure ones and must favor AEAD cipher suites over CBC cipher suites (RFC 8827). However, among the tested applications, many do not even enforce a server-preferred order, but instead rely on the ordering of the client-proposed list, giving browser developers more agency about algorithm choices.

The most concerning issue we found is that cipher suites are advertised as supported, but fail in practice when we tried to use them. Concretely, we observed servers willing to negotiate a specific cipher in their *ServerHello* message, but would then send an *Alert* message right after when trying to send a certificate message. We attribute this, for the most part, to server-side misconfigurations. The server is trying to use cipher suites that require a certificate with a public key type not present in the certificate it has committed to using in the SDP.

SRTP Profiles. Regarding supported SRTP protection profiles, we see little diversity among browsers in WebRTC. All tested browsers supported AES GCM 128/256, and all browsers supported the mandatory SRTP profile `SRTP_AES128_CM_HMAC_SHA1_80`. The only difference we found between browsers is that Firefox also supported counter mode with 32-bit HMACs. HMACs with 32-bit lengths are rather weak, as they allow an attacker to forge a MAC by guessing with non-negligible probability. Across other applications, we see no support for other protection profiles, with individual support for 32-bit HMACs. The full results of our analysis are given in Table 1.

Supported Groups. The results of our analysis of supported groups are presented in Table 5 in `Artifacts.pdf` file in our artifact. Some applications support a wide range of groups, including some that are also considered weak. However, to exploit their presence, both endpoints have to support the weak choice, which we did not observe in any application.

Signature+Hash Algorithms. Support for signature and hash algorithms is presented in Table 6 in `Artifacts.pdf` file in our artifact. The results are mostly unsurprising, with support focusing on RSA, ECDSA, or EDDSA. Signature algorithms supporting MD5 were generally not observed. Additionally, server implementations typically allow the client more leeway in their support than what they are willing to use themselves.

Certificate Analysis. The results of our certificate analysis are presented in Table 7 in `Artifacts.pdf` file in our artifact. In contrast to our expectations, many applications were not using fresh self-signed certificates for every connection. Many remote applications used the same certificate for multiple connections, while local applications always generated a fresh certificate. Additionally, some applications did not use self-signed certificates at all but used normal Internet PKI.

Our analysis of the certificates revealed that all tested applications use either ECDSA or RSA certificates. For ECDSA, we exclusively saw SECP256R1 certificates, likely motivated by browser support (see Table 3). For RSA, most applications use a 2048-bit RSA modulus, which is considered secure. Two exceptions to this were Vonage and Zoho. Vonage used a 1024-bit modulus, which, while not catastrophic, is on the border of what is considered crackable by motivated attackers and has been deprecated by NIST. More severely, Zoho was using a 512-bit modulus, which is unarguably too short for modern applications.

Curiously, many certificate subject names contained the names of WebRTC/DTLS-SRTP libraries, such as *mediasoup*, *FreeSWITCH*, and *LiveSwitch*, leading us to believe that these applications are using these libraries. Some application servers use certificates with long expiration dates (10 years or more). In one case, however, we even encountered an expired certificate (Discord).

5.2 Authentication Bypasses

Our study uncovered multiple server authentication bypasses on the DTLS layer across the tested applications. We have highlighted applications from which we successfully obtained media or encrypted metadata in Table 2.

Webex. When testing the Webex application, our tool reported that it is possible to authenticate to the Webex media server by presenting an empty certificate message. Cisco fixed the issue and assigned CVE-2025-20215 [16].

Discord. In the Discord web application, we discovered two authentication bypasses. In October 2022, we discovered that it was possible to finish the DTLS handshake with the Discord server using *any* X.509 certificate, indicating that Discord was not verifying the identity of peers. While we investigated the issue, we noticed that shortly after our discovery, Discord independently found and fixed the issue. We contacted them, and they confirmed that they independently fixed the issue. In February 2024, we discovered that the Dis-

		Platforms	DTLS Version v1.0	DTLS Version v1.2	DTLS Version v1.3	DTLS Version Order enforced	DTLS KEX ECDHE	DTLS KEX DHE	DTLS KEX RSA	DTLS Symmetric Cipher AES	DTLS Symmetric Cipher CHACHA20	DTLS Symmetric Cipher ARIA	DTLS Symmetric Cipher CAMELLIA	DTLS Symmetric Cipher SEED	DTLS Symmetric Cipher 3DES	DTLS Symmetric Cipher EXPORT	DTLS Symmetric Cipher NULL	DTLS Mode GCM	DTLS Mode CBC	DTLS Mode CCM	DTLS Mode CCM_8	SRTP Profile Order enforced	SRTP Profile SRTP_AES128_CM_HMAC_SHA1_80	SRTP Profile SRTP_AES128_CM_HMAC_SHA1_32	SRTP Profile SRTP_AEAD_AES_128_GCM	SRTP Profile SRTP_AEAD_AES_256_GCM
clients	Chromium		PV	y	n	-	✓		✓	✓	✓							✓	✓			-	✓		✓	✓
	Firefox		PV	y	n	-	✓			✓	✓							✓	✓			-	✓	✓	✓	✓
	Safari		PV	y	n	-	✓		✓	✓	✓							✓	✓			-	✓		✓	✓
	BBB 2.4		IP	y	n	-	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓	✓	✓	-	✓			
	BBB Demo v3.0.12 con 2		PV	y	n	-	✓			✓								✓	✓			-	✓		✓	✓
	ChatGPT con 1		PV	y	n	-	✓		✓	✓	✓							✓	✓			-	✓		✓	✓
	ChatGPT con 2		PV	y	n	-	✓			✓								✓	✓			-	✓		✓	✓
	Chime		PV	y	n	-	✓		✓	✓	✓							✓	✓			-	✓		✓	✓
	Clickmeeting con 2		IP	y	n	-	✓	✓	✓	✓	✓							✓	✓			-	✓	✓	✓	✓
	Goto Meet		PV	y	n	-	✓	✓	✓	✓	✓							✓	✓			-	✓			
	LiveKit con 2		PV	y	n	-	✓			✓								✓	✓			-	✓		✓	✓
	MatterMost		PV	y	n	-	✓			✓								✓	✓			-	✓		✓	✓
	Slack		PV	y	n	-	✓		✓	✓	✓							✓	✓			-	✓		✓	✓
	Slack con 1		PV	y	n	-	✓	✓	✓	✓	✓							✓	✓			-	✓		✓	✓
	Slack con 2		UM	y	n	-	✓	✓	✓	✓	✓	✓	✓					✓	✓	✓	✓	-	-	-	-	-
	Steam		PV	y	n	-	✓		✓	✓	✓							✓	✓			-	✓		✓	✓
	Vonage con 1		PV	y	n	-	✓	✓	✓	✓	✓							✓	✓			-	✓		✓	
	Wickr		n	y	n	-	✓			✓								✓				-	-	-	-	-
servers	Chromium		PV	y	n		✓			✓	✓							✓	✓			✓	✓		✓	✓
	Firefox		PV	y	n	✓	✓			✓	✓							✓	✓			✓	✓	✓	✓	✓
	Safari		PV	y	n		✓			✓	✓							✓	✓			✓	✓		✓	✓
	Adobe Connect		y	y	n		✓			✓								✓	✓			✓	✓			
	BBB Demo v3.0.12 con 1		PV	y	n	-	🔒			🔒								🔒				-	🔒			
	BBB Docker v 3.0.4		IE	y	n	✓	✓			✓	✓							✓	✓			✓	✓	✓	✓	✓
	ChatGPT con 1		PV	y	n	-	🔒			🔒								🔒				-	🔒			
	ChatGPT con 2		PV	y	n		✓			✓	✓							✓	✓			✓	✓		✓	✓
	Chime		n	y	n		✓		✓	✓	✓	✓	✓					✓	✓	✓	✓	✓	✓		✓	✓
	Clickmeeting con 1		IE	y	n		✓			✓	✓							✓	✓			✓	✓	✓	✓	✓
	Discord		n	y	n		✓		✓	✓	✓	✓	✓	✓				✓	✓	✓	✓	-	✓			
	eduMEET		IE	y	n	✓	✓			✓	✓							✓	✓			✓	✓	✓	✓	✓
	Google Meet		PV	y	n		✓			✓	✓							✓	✓			✓				✓
	Instagram		PV	y	n		✓			✓	✓	✓	✓		✓			✓	✓	✓	✓	-	✓			
	Ionos		n	y	n		✓			✓	✓							✓				✓	✓		✓	
	Janus		PV	y	n		✓			✓	✓							✓	✓			✓	✓	✓	✓	✓
	Jitsi		n	y	n		✓			✓	✓							✓				✓	✓		✓	
	LiveKit con 1		PV	y	n	-	🔒			🔒								🔒				-	🔒			
	Ringcentral		n	y	n		✓			✓								✓	✓			✓	✓			
	Slack		n	y	n		✓		✓	✓	✓	✓	✓					✓	✓	✓	✓	✓	✓		✓	✓
	Slack con 1		HF	y	n		✓		✓	✓	✓	✓	✓					✓	✓	✓	✓	✓	✓		✓	✓
	Slack con 2		n	y	n	ERR	✓		✓	✓		✓	✓					✓	✓	✓		-	-	-	-	-
	Snapchat		PV	y	n		✓			✓	✓							✓	✓			✓	✓	✓	✓	✓
	Steam		PV	y	n		✓			✓								✓	✓	✓	✓	✓	✓	✓		
	Teams		n	y	n		✓		✓	✓								✓	✓			✓	✓			✓
	Vonage con 2		PV	y	n		✓		✓	✓	✓							✓	✓			✓	✓		✓	
	Webex		n	y	n		✓			✓								✓	✓					✓	✓	✓
	Wickr		n	y	n	-	✓			✓								✓				-	-	-	-	-
	Zoho con 1		n	y	n	✓	✓			✓								✓	✓			-	-	-	-	-
	Zoho con 2		n	y	n		✓		✓	✓	✓							✓	✓			✓	✓			
	Zoom		PV	y	n	✓	✓			✓	✓							✓	✓			-	-	-	-	-

Table 1: Supported DTLS versions and DTLS/SRTP cipher suites offered by clients and supported by servers. Local endpoints are shaded in gray. An empty cell indicates that a client did not offer a given suite or a server did not accept it, while ✓ denotes that the suite was offered (client) or accepted (server); ✓ marks the default selected when multiple suites are supported. 🔒 indicates that the server picks a single cipher suite even if the client does not offer it. A dash (-) denotes "not applicable": SRTP may not be negotiated at all and order enforcement may be irrelevant. When forcing a DTLS version, the alerts seen are `PV` (`PROTOCOL_VERSION`), `IP` (`ILLEGAL_PARAMETER`), `UM` (`UNEXPECTED_MESSAGE`), `IE` (`INTERNAL_ERROR`), and `HF` (`HANDSHAKE_FAILURE`), with `n` indicating a silent abort. Our tool failed to determine DTLS cipher suite order enforcement for Slack's Android (marked `ERR`). Missing DTLS v1.3 support in Firefox is attributed to a bug in Firefox 137 (see Section 5.1).

cord server was accepting DTLS connections with optional client authentication. We reported the problems to Discord, and they acknowledged and fixed the issue, awarding us a bug bounty (severity: medium).

Zoom. Our analysis revealed that the Zoom web application's media server failed to request authentication from the browser, leading to missing client authentication. We reported the issue to Zoom, which acknowledged and fixed it (severity: high) and awarded us a bug bounty.

Teams. We observed that Microsoft Teams allows a peer to complete DTLS authentication using either an empty certificate or an arbitrary client certificate. In both cases, once the DTLS handshake completes, we receive *plaintext RTCP Source Descriptions*, behavior that the WebRTC standard explicitly prohibits [45, Section 6.5], as well as continued ICE connectivity checks (e.g., STUN binding requests/responses). However, we did not observe any true RTP media traffic. One possible explanation is that the media server cannot properly associate our connection with the correct call. Microsoft investigated our report but determined the reported vulnerabilities were out of scope because no RTP leakage was observed.

FreeSWITCH. Internally, BigBlueButton uses FreeSWITCH [52], an open-source telephony framework used by multiple media servers for a variety of purposes (e.g., SIP, call routing, and WebRTC). While testing version 2.4 of BigBlueButton, we noticed that FreeSWITCH accepts any client certificate. In the code, FreeSWITCH mistakenly overwrites the remote fingerprint received in the SDP while attempting to extract the client certificate's fingerprint. Thus, the peer certificate verification always returns true. This issue was reported independently of our research in 2023 [40], but the related pull request was never merged. In version 2.5, BigBlueButton transitioned from using FreeSWITCH for WebRTC calls to Mediasoup.

Zoho. In our tests, Zoho established multiple DTLS connections in one call. We found one of them to be vulnerable, as Zoho's media server accepted any client certificate in the handshake. We were able to MitM the connection and observed the exchange of call metadata on a WebRTC data channel (SCTP). We therefore consider Zoho as exploitable. As mentioned in Section 5.1, Zoho used a 512-bit RSA certificate. To demonstrate that this allows an attacker to bypass authentication, we used `cado-nfs` [58] to factor the key and retrieve the private key for the certificate in 4.5 hours with an AMD EPYC 7763. While Zoho did not negotiate RSA KEX cipher suites by default (which would have allowed for passive attacks), the certificate was used for all incoming clients, allowing an attacker to break the certificate once and then use the keys for active server impersonation attacks. We notified Zoho about the vulnerability, and they replaced the certificate with an ECDSA one. Zoho determined the missing identity check to be an out-of-scope vulnerability.

Steam. For Steam, the media server did not request a client certificate during the DTLS handshake. After we completed the handshake, the server immediately sent us RTP from the call, which we could decrypt. Therefore, we classified Steam as exploitable.

Vonage. With Vonage, we observed that the media server neither requested nor validated a web client's certificate when one was presented. Additionally, Vonage emits media data directly after our authentication bypass, confirming exploitability.

RingCentral. The RingCentral WebRTC gateway did not request client authentication from the browser. We directly received RTP data from the DTLS endpoint without further interaction.

Browsers. We evaluated the security of browser implementations in both DTLS roles: as a DTLS client and as a DTLS server. Our analysis reveals that none of the tested browsers were directly affected by a direct authentication bypass vulnerability.

Non-Exploitable. For *11* implementations (marked ✗ in Table 2), we could not provoke media data transmission, or we are missing confirmation from the vendor. We therefore manually investigated individual applications and found, for example, that mediasoup [9] actually uses a delayed client authentication check by reviewing the source code. Similarly, after contacting the Amazon [5] team, we learned that Chime is not using DTLS-SRTP for all of its connections, but is *also* sometimes using regular DTLS connections that see a custom authentication after the handshake. We likewise observed the Snapchat web client performing custom authentication by including a token in the first protected data-channel message. For Amazon Wickr, we assume that an authenticated key exchange occurs after the DTLS handshake to establish keys for end-to-end encryption (E2EE). Among our tests, some were unsuccessful for all applications. Concretely, we could not find evidence for *Flow Bypasses*, *Missing Signature Verification*, or *Missing Public Key Protection*. Additionally, when testing if the *OS Trust Store* was used, we observed that the application did not verify the peer's identity in all cases, leading us to conclude that no application was utilizing the OS trust store.

5.3 Non-Security Bugs

During our testing, we encountered several non-security critical bugs in the tested applications that are worth mentioning. We noticed that our analysis of Discord only functions when DTLS retransmissions are used, as the Discord media server sends only half of the *ServerHello* flight in its first packet, without following up with the rest of the flight. Only after a retransmission were we able to receive the full flight. This behavior causes unnecessary delay for real users, as it adds an additional round-trip time to the connection establishment. Another observation we made is that five applications are configured to use the DTLS Denial-of-Service (DoS) countermeasure, which involves adding an additional cookie exchange

DTLS Role **Server**	Platform	Cert. Requested	Auth. Required	Identity Check	Signature Verified	No Flow Bypass	Public Key Protected	No OS Trust Store	Assessment
Chromium		✓	✓	✓	✓	✓	✓	✓	✓
Firefox		✓	✓	✓	✓	✓	✓	✓	✓
Safari		✓	✓	✓	✓	✓	✓	✓	✓
Adobe Connect		✓	✓	✓	✓	✓	✓	✓	✓
BBB v3.0.4		✓	✓	✗	✓	✓	✓	✗	✗
BBB v3.0.12 con 1		✓	✓	✗	✓	✓	✓	✗	✗
ChatGPT con 1		✓	✓	✗	✓	✓	✓	✗	✗
ChatGPT con 2		✓	✓	✓	✓	✓	✓	✓	✓
Chime		✓	✓	✓	✓	✓	✓	✓	✓
Clickmeeting con 1		✓	✓	✗	✓	✓	✓	✓	∅
Discord 2022		✓	✓	✗	✓	✓	✓	✗	💣
Discord 2024		✓	✗	✓	✓	✓	✓	✓	💣
Discord		✓	✓	✓	✓	✓	✓	✓	✓
eduMEET		✓	✓	✗	✓	✓	✓	✗	✗
Google Meet		✓	✓	✓	✓	✓	✓	✓	✓
Instagram		✓	✓	✓	✓	✓	✓	✓	✓
Ionos		✓	✓	✓	✓	✓	✓	-	✓
Janus		✓	✓	✗	✓	✓	✓	✓	∅
Jitsi		✓	✓	✓	✓	✓	✓	-	✓
LiveKit con 1		✓	✓	✗	✓	✓	✓	✗	✗
Ringcentral		✗	-	-	-	-	-	-	💣
Slack		✓	✓	✓	✓	✓	✓	✓	✓
Slack con 1		✓	✓	✓	✓	✓	✓	✓	✓
Slack con 2		✗	-	-	-	-	-	-	🔧
Snapchat		✗	-	-	-	-	-	-	🔧
Steam		✗	-	-	-	-	-	-	💣
Teams		✓	✗	✗	✓	✓	✓	✗	∅
Vonage con 2		✗	-	-	-	-	-	-	💣
Webex 2024		✓	✗	✓	✓	✓	✓	✓	💣
Webex		✓	✓	✓	✓	✓	✓	✓	✓
Wickr		✗	-	-	-	-	-	-	🔧
Zoho con 1		✓	✗	✗	✓	✓	✓	✗	💣
Zoho con 2		✓	✓	✓	✓	✓	✓	✓	✓
Zoom 2024		✗	-	-	-	-	-	-	💣
Zoom		✓	✓	✓	✓	✓	✓	✓	✓

DTLS Role **Client**	Platform	Identity Check	Signature Verified	Public Key Protected	No OS Trust Store	Assessment
Chromium		✓	✓	✓	✓	✓
Firefox		✓	✓	✓	✓	✓
Safari		✓	✓	✓	✓	✓
BBB v2.4		✗	✓	✓	✗	💣
BBB v3.0.12 con 2		✗	✓	✓	✗	✗
ChatGPT con 1		✓	✓	✓	✓	✓
ChatGPT con 2		✗	✓	✓	✗	✗
Chime		✓	✓	✓	✓	✓
Clickmeeting con 2		✗	✓	✓	✓	∅
Goto Meet		✓	✓	✓	✓	✓
LiveKit con 2		✗	✓	✓	✗	✗
MatterMost		✗	✓	✓	✗	✗
Slack		✓	✓	✓	✓	✓
Slack con 1		✓	✓	✓	✓	✓
Slack con 2		✓	✓	✓	✓	🔧
Steam		✓	✓	✓	✓	✓
Vonage con 1		✗	✓	✓	✓	💣
Wickr		✓	✓	✓	✓	🔧

Table 2: Overview of our DTLS authentication tests across tested applications. Local endpoints are shaded in gray. ✓ indicates an expected and correct behavior, ✗ indicates that the application fails this test on the DTLS layer. As for the **Assessment** column: 💣 indicates a failed test that results in an exploitable vulnerability, ✗ denotes that the application sends an encrypted alert directly after the handshake, ∅ indicates that the endpoint abandoned all communication to us, except for ICE connectivity checks, and 🔧 denotes that the application performs, or is highly likely to perform, a custom authentication protocol after the DTLS handshake.

to the DTLS protocol. However, in the case of DTLS-SRTP, this addition is arguably not necessary. Namely, in DTLS-SRTP, the server knows from where it expects a connection and can limit incoming *ClientHello* messages to the expected endpoints, preventing DoS attacks. By adding the countermeasure, implementations add an additional round-trip time to the connection establishment, which unnecessarily slows down the connection. In Webex, the mitigation is not implemented correctly: the cookie is hard-coded to the ASCII string `session id`, defeating its purpose. We also observed that many applications that perform a delayed fingerprint check and terminate the DTLS-SRTP connection after the handshake still leave the ICE candidate pair active, and we continue to receive STUN *Binding Success Responses* when forwarding *Binding Requests*.

6 Proof-of-Concept Exploit

To demonstrate that vulnerabilities on the DTLS layer can lead to an exploit that leaks media data to the attacker, we developed a proof-of-concept exploit for Webex. In this exploit, we wait for a client to establish a connection to Webex, but then the exploit authenticates using an empty certificate and an attacker-chosen public key. The exploit then finishes the DTLS handshake. After the DTLS handshake is completed, we then request media data for the client's meeting, using Cisco's Multistreaming protocol [15]. After that, the Webex media server sends the audio stream of the targeted call to us, which we decrypt and decode. This allows us to listen in on the call. From the view of other users in the Webex meeting, the real user joined the call. After a certain amount of time (~30s), the attacker gets disconnected from the call and will not receive further media data. We assume that this is simply a limitation of our simplistic PoC, as we did not prevent the real client from signaling to the media server on the application layer that it needs to reconnect.

7 Browser API Evaluation

Beyond server-side implementations, we investigated the extent to which the DTLS channel can be influenced through the JavaScript WebRTC API, as well as the permissiveness of browser certificate acceptance policies. The WebRTC API hides most of the DTLS connection internals; the main interface for users to influence the DTLS channel is through the `generateCertificate()` function and by manually modifying the SDP sent during signaling.

Certificate Generation. The WebRTC API function `generateCertificate()` generates a self-signed X.509 certificate and the corresponding private key. The standard dictates that RSASSA-PKCS1-v1_5 with 2048-bit modulus and 65537 exponent and ECDSA with P-256 curve *must* be supported, while other algorithms are optional. All tested browsers support this. Aside from that, the specification also mentions RSA-PSS certificate as a permitted option.

We tested the `generateCertificate()` function in Chrome, Safari, Edge, Firefox, and Opera to check which algorithms and cryptographic parameters are allowed. These browsers use one of three browser engines: Blink, Gecko, or WebKit. All browser engines use the native WebRTC library[2]; however, each engine typically adds additional functionality, leading to browsers potentially behaving differently in identical situations.

For RSA-based signature schemes, we tested the minimum and maximum supported modulus sizes and the smallest supported exponent. We test this since allowing small moduli may permit factoring attacks [63], and allowing a small exponent can lead to signature forgery vulnerabilities [11]. For ECDSA, we tested which curves out of the SECG curves over prime and binary fields[3] and the Brainpool curves are supported. Some of these curves are too small and may allow motivated attackers to recover private keys (e.g., [64]). The complete list of curves we tested is available in the artifacts.

We present the results in Table 3. Chrome, Edge, and Opera behave identically, consistent with all three using Google's Blink browser engine. Firefox is the only browser supporting ECDSA curves other than P-256. However, Firefox allows users to generate potentially weak RSA exponents ($e = 3$), which may enable signature forgery attacks if signature validation is not strictly implemented [11].

Certificate Permissiveness. We also tested whether browsers accept weak RSA certificates from peers, specifically those with a modulus size smaller than 1024 bits (e.g., 512 bits). All tested browsers accept weak RSA certificates provided by the media server (Table 3, last column). We performed this test using a custom Janus [6] media server configured to offer a 512-bit RSA certificate to clients.

SDP Munging. In all browsers, it is possible to modify the generated SDP offer from the API before sending it to the server, a practice commonly known as *SDP munging*. Although the specification explicitly forbids this practice [62], it remains widely used by developers to work around API limitations. Browser vendors are actively working toward its deprecation [14].

The extent to which modifications can be made differs across browsers. In all browsers except Firefox, it is not possible to change the certificate fingerprint in the SDP. This would have been beneficial for our testing framework, as it would allow us to claim custom certificates for analyzing peers without modifying the browser's source code.

[2] https://webrtc.github.io/webrtc-org/native-code

[3] The NIST curves are a subset of curves defined by SECG.

Browser	Version	RSA-PKCS1-v1.5			RSA-PSS	ECDSA Supported Curves	Rejects 512-bit RSA certificate
		min. N	max. N	min. e			
Chrome	121.0.6167.184	1024	8192	1025	✗	P-256	✗
Safari	18.6 (20621.3.11.11.3)	1024	8192	260	✗	P-256	✗
Edge	121.0.2277.128	1024	8192	1025	✗	P-256	✗
Firefox	122.0.1	1024	16384	3	✗	P-256, P-384, P-521	✗
Opera	107.0.5045.21	1024	8192	1025	✗	P-256	✗

Table 3: Results of testing the `generateCertificate()` WebRTC API function. Safari was tested on macOS Sequoia v15.6, and other browsers on Ubuntu 20.04.6 LTS. The minimum public exponent size was tested with a 1024-bit modulus. For ECDSA, the function always generates a certificate that uses the SHA-256 hash function, irrespective of what is provided as an argument for the hash function name.

8 Discussion

Missing *CertificateRequest* Acceptance. A surprising finding that we observed in many applications is that they did not request a certificate at all. Browsers (and other clients) generally accept such DTLS connections because, from the perspective of a DTLS library, it is unaware that it is being used in a WebRTC context where client authentication is mandatory. We therefore propose a new hardening mechanism for (D)TLS libraries, where the same semantics of *optional* and *required* authentication, currently employed on the server (see Section 4.2), be replicated on the client. Specifically, when client authentication is set to required on a client, it should abort the connection when the server does not request the client to authenticate. Deploying this defense-in-depth mechanism would break misconfigured applications, forcing them to correct their configurations.

DTLS-SRTP vs SDES-SRTP. Earlier versions of WebRTC used SDES-SRTP (Session Description Protocol Security Descriptions) instead of DTLS-SRTP. With SDES, the symmetric keys for the SRTP connection are directly exchanged in the signaling phase (via SDP) without the use of public key cryptography. Eventually, DTLS-SRTP was chosen as a replacement as it offers better security against an honest-but-curious signaling server. Neither approach protects against an actively-malicious signaling server. In our view, plugging in the whole DTLS technology stack (for both clients and servers), including X.509 implementations, instead of designing a dedicated key exchange, had its downsides. The (D)TLS standard introduces unnecessary technological complexity, as many of its features are unused in WebRTC. Moreover, DTLS introduced two additional round-trips before a connection is established, which significantly increases latency. Since only the key exchange component of DTLS is used, WebRTC could achieve the same goals by exchanging public keys instead of X.509 certificate fingerprints in the SDP. This would have allowed peers to do the key exchange directly, preventing many unnecessary computations, round-trips, and technical overhead.

Modularity at the Cost of Complexity. Reliable, scalable and portable real-time communication is challenging. Fortunately, through the effort of the WebRTC framework, anyone can easily create an RTC application capable of running in everyone's browser in little time and without expensive hosting costs [31]. To accomplish this, the WebRTC framework made heavy use of "off-the-shelf" components. This allows WebRTC to easily interoperate with many existing, older technologies, like VoIP and SIP. However, this also introduced all the weight and complexity that these older technologies bring. For instance, SDP was preferred over JSON or Protobuf for exchanging connection parameters. Instead of exchanging public keys in signaling, WebRTC uses DTLS to exchange them. Even the choice of SRTP may be re-evaluated with alternatives like RTP over QUIC evolving [17]. Thus, while these component choices allow for modularity and rapid development, the added complexity makes these systems harder to analyze and test, which is probably why these basic DTLS flaws were not discovered earlier. WebRTC is yet another example that, in the long run, a complex design has a toll on the development lifecycle of such systems and ultimately their security. Our open-source testing framework DMS is a first step toward remedying this, and developers and admins can use it to test their systems and configurations. However, in this work, we only analyzed a small portion of the attack surface of this ecosystem and there are likely many more vulnerabilities to be uncovered.

9 Related Work

WebRTC Security. WebRTC's security architecture is detailed in RFC 8826 [44] and RFC 8827 [45]. However, both RFCs cover primarily direct peer-to-peer connections between two clients. Johnston demonstrated successful MitM attacks against naive WebRTC deployments that rely on a compromised signaling server and recommended authenticating certificate fingerprints via an authenticated signaling path [59]. A broader community study similarly argued that WebRTC's self-signed certificate model makes fingerprint verification via

secure signaling essential to prevent MitM attacks [65]. Reiter et al. also explore untrusted signaling channels and present privacy leaks where ICE/SDP flows can expose local and public IPs and enable in-browser network reconnaissance [41]. Notably, none of these works provides concrete guidance for the security of media servers.

RTC Protocols Security. Early VoIP security research by Gupta and Shmatikov revealed critical weaknesses in how session keys are established for SRTP. In particular, when SRTP is keyed via the older SDES mechanism, a replay attack can cause reuse of keystream material, completely breaking transport-layer encryption [24]. They also demonstrated a MitM attack on the ZRTP key exchange protocol, exploiting the case where users cannot perform the Short Authentication String (SAS) verification (e.g., devices without a display), effectively downgrading the session. Bresciani and Butterfield provided a formal security proof for ZRTP, confirming that the Diffie-Hellman key agreement (with SAS verification) can indeed prevent MitM attacks and strengthen SRTP's end-to-end authenticity [13].

Vulnerabilities in Video Conferencing Systems. Beyond core WebRTC issues, conferencing apps show application-layer and deployment flaws. A study of BigBlueButton and eduMEET found 57 flaws across access control and media handling [25]. Other bugs in proprietary stacks include Zoom's crypto and E2EE design [34], an XMPP "stanza smuggling" chain that enabled code execution [23], and a media router overflow [1]. Similar issues are reported for Microsoft Teams [12] and Electron-based clients like Jitsi [3] and Discord [29]. Google Project Zero fuzzed consumer WebRTC apps, such as FaceTime and WhatsApp, which surfaced memory safety bugs in media processing [53, 54, 55].

DTLS Implementations. Although the DTLS protocol is closely related to the TLS protocol, its implementations have not received the same level of scrutiny as TLS until recently. The state machine of the DTLS implementations has been analyzed by Fiterau-Brostean et al. [21], who used state-machine fuzzing to automatically create a model of the state machine. The concept was later extended by Fiterau-Brostean et al. [22] to avoid manual analysis of the state machine for already known vulnerability types. Since state machine fuzzing is inherently tricky outside of a controlled environment, we did not explore applying this approach to DTLS-SRTP. Besides the state machine, the DTLS ecosystem has been analyzed by Erinola et al. [18] in a first Internet-wide ecosystem study. Although the study was extensive, it was unable to capture DTLS as used in DTLS-SRTP because DTLS servers are not permanently located on specific endpoints and may only respond to messages from a previously established ICE candidate pair. Related to this limitation, work by Enable Security [19] examined which remote RTC endpoints are willing to accept DTLS *ClientHello* messages outside of the selected ICE candidate pair. A symbolic analysis of DTLS implementations has been performed by Asadian et al. [7], who analyzed four DTLS server implementations and uncovered non-conformant behavior and security issues in OpenSSL and TinyDTLS.

10 Conclusion

In this work, we presented the first WebRTC/DTLS-SRTP analysis platform DMS. Setting up our platform in a MitM position, using TLS-Attacker's MitM module, allowed us to implement complex testing strategies without needing to access key material, enabling the systematic evaluation of *24* service providers and *5* browsers.

Returning to the research questions that we set out to explore, we observe the following. With respect to **RQ1**, we find a maturing ecosystem with universal DTLS 1.2 support but negligible DTLS 1.3 adoption, and a consistent preference for forward-secure key exchange and modern AEAD cipher suites. On the other hand, the answer to **RQ2** is somewhat less satisfactory. While all browsers implement DTLS-SRTP securely, *19* server implementations contained authentication bypasses, of which *9* were confirmed to be exploitable—allowing attackers to decrypt media from a pure MitM position. These findings reveal severe issues in the WebRTC ecosystem that affect the security of media connections for hundreds of millions of users and need to be addressed, either through systematic testing or a technology change. In addition, these findings suggest a gap in WebRTC/DTLS-SRTP proficiency between browser providers and application providers. This is perhaps expected, since WebRTC was primarily developed by the former community.

Future Work. In this work, we have not yet analyzed DTLS implementations with the same level of scrutiny that TLS implementations have been subjected to. Works like Maehren et al. [32], Fiterau-Brostean et al. [21], and Fiterau-Brostean et al. [22] use more advanced techniques in their analysis. Applying these more advanced techniques to WebRTC and DTLS-SRTP is more challenging as it requires hooking into the signaling phase in order to extract or manipulate keys and increasing the level of automation in the initiation of connections, but it is likely to be a fruitful endeavor.

Acknowledgment

The authors would like to thank the reviewers for their insightful comments. Lukas Knittel was supported by the research project "North-Rhine Westphalian Experts in Research on Digitalization (NERD II)", sponsored by the state of North Rhine-Westphalia – NERD II 005-2201-0014. Vukašin Karadžić was supported by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

Ethical Considerations

Our research involves testing the security of web applications by manipulating DTLS and media messages in our own WebRTC connections. We identified the following stakeholders: (1) the web application service providers whose systems we tested, (2) other users of these services, (3) our research team members, and (4) the broader community that relies on secure WebRTC implementations. To ensure our research maximizes benefits while minimizing potential harms, we implemented several safeguards:

Limited Scope. We exclusively manipulated our own connections and authentication credentials, ensuring no impact on other users' sessions or data.

Resource Consumption. Our tests are severely rate-limited, with at most one connection every few seconds, minimizing impact on the network and computation resources, typically totaling less than 500 short-lived connections. The performed tests were not expected to bind many computational resources.

No Exploitation. While we identified vulnerabilities, we did not exploit them beyond what was necessary for a proof of concept, and we never accessed or modified data belonging to other users.

Broader Impact Analysis. We considered both positive and negative potential outcomes of our research. Our research has improved the security of WebRTC connections for hundreds of millions of users worldwide. Additionally, our research advanced the field of practical communication protocol research, showcasing how to perform studies in highly complicated communication protocols, providing a prime case study for research and industry alike. On the negative side, our research may have temporarily consumed some amount of server computation and may have triggered warnings at tested applications, which temporarily binds security team resources.

Vendor Permission and Testing Scope. Where vendors had public coordinated vulnerability disclosure (CVD) or bug bounty programs, we operated within those programs' terms, which expressly permit external security testing. For all services, we limited experiments exclusively to our own sessions and credentials. Our methodology was designed to minimize operational risk: we (1) manipulated only our own connections and authentication credentials, (2) rate-limited to at most one call every few seconds, and (3) did not exploit beyond what was necessary to show exploitability, nor did we access or modify other users' data. These probes target DTLS handshake-layer behaviors rather than high-load paths, minimizing crash risk and operational impact.

Responsible Disclosure. We responsibly disclosed all findings to the respective vendors in accordance with their vulnerability disclosure guidelines, and continuously assisted them by retesting deployed patches and providing feedback.

Discord and Zoom confirmed our reports, fixed the reported issues, and awarded us bug bounties. Webex acknowledged the reported vulnerabilities, fixed the issues, and assigned CVE-2025-20215 (severity medium). Our proof-of-concept only captured audio from our own test meetings that we initiated. Microsoft (Teams) considered the reported vulnerabilities to be out of scope for their threat model. Zoho removed the weak certificate after our initial report and awarded us a bounty. Steam and Ringcentral confirmed the exploits and awarded us bounties.

Open Science

We provide both our testing framework, DTLS MitM Scanner (cf. Section 3), and the results of our evaluation as artifacts. Furthermore, we provide PCAP recordings of all our executed tests, as well as the textual report, which is output by our framework. We also provide a patch file to modify Chromium as described in Section 4.3. In addition, our artifacts contain scripts and instructions to reproduce our browser-side tests: JavaScript snippets to replace the SDP fingerprint, a Janus setup using a 512-bit RSA certificate for DTLS, and materials for examining which parameters browsers permit when generating a certificate (cf. Table 3). Finally, we include a video recording of the exploit for Cisco Webex (cf. Section 6).

Our artifact can be found at https://doi.org/10.5281/zenodo.17880120.

References

[1] Thijs Alkemade and Daan Keuper. Zoom RCE from Pwn2Own 2021. Sector 7 research blog, August 2021. URL https://sector7.computest.nl/post/2021-08-zoom/.

[2] John Althouse. TLS Fingerprinting with JA3 and JA3S. https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967/, 2019. Salesforce Engineering Blog.

[3] Benjamin Altpeter. RCE in Jitsi Meet Electron prior to 2.3.0 due to insecure use of `shell.openExternal()` (CVE-2020-25019). https://benjamin-altpeter.de/jitsi-meet-electron-rce-shell-openexternal/, August 2020.

[4] Harald Alvestrand. Google release of WebRTC source code. URL https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html.

[5] Amazon Web Services, Inc. Amazon chime. https://aws.amazon.com/chime/, 2025.

[6] Amirante, A. and Castaldi, T. and Miniero, L. and Romano, S. P. Janus: a general purpose WebRTC gateway. In *Proceedings of the Conference on*

Principles, Systems and Applications of IP Telecommunications, IPTComm '14, New York, NY, USA, 2014. Association for Computing Machinery. URL https://doi.org/10.1145/2670386.2670389.

[7] Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson, and Konstantinos Sagonas. Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification. In *IEEE Conference on Software Testing, Verification and Validation, ICST*, 2022.

[8] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc3711.txt. Updated by RFCs 5506, 6904, 9335.

[9] Iñaki Baz Castillo, José Luis Millán, and Nazar Mokynskyi. mediasoup. https://mediasoup.org/, 2025.

[10] A. Begen, P. Kyzivat, C. Perkins, and M. Handley. SDP: Session Description Protocol. RFC 8866 (Proposed Standard), January 2021. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8866.txt.

[11] Daniel Bleichenbacher. Forging some RSA signatures with pencil and paper, 2006. Presented at CRYPTO 2006 rump session.

[12] Fabian Bräunlein. MS teams: 1 feature, 4 vulnerabilities. Positive Security blog, December 2021. URL https://positive.security/blog/ms-teams-1-feature-4-vulns.

[13] Riccardo Bresciani and Andrew Butterfield. A formal security proof for the ZRTP Protocol. In *2009 International Conference for Internet Technology and Secured Transactions,(ICITST)*, pages 1–6. IEEE, 2009.

[14] Chromium Project. Issue 40567530: Deprecate and remove ability to modify SDP before SetLocalDescription. https://issues.chromium.org/issues/40567530.

[15] Cisco. Announcing the Multistream Feature in Webex Web Meetings SDK. https://developer.webex.com/blog/announcing-the-multistream-feature-in-webex-web-meetings-sdk, 2024.

[16] Cisco Systems, Inc. Cisco Security Advisory CVE-2025-20215. https://sec.cloudapps.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-webex-join-yNXfqHk4, 2025.

[17] Mathis Engelbart, Joerg Ott, and Spencer Dawkins. RTP over QUIC (RoQ). Internet-Draft draft-ietf-avtcore-rtp-over-quic-14, Internet Engineering Task Force, March 2025. URL https://datatracker.ietf.org/doc/draft-ietf-avtcore-rtp-over-quic/14/. Work in Progress.

[18] Nurullah Erinola, Marcel Maehren, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. Exploring the unknown DTLS universe: Analysis of the DTLS server ecosystem on the internet. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4859–4876, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/erinola.

[19] Alfred Farrugia and Sandro Gauci. DTLS "ClientHello" Race Conditions in WebRTC Implementations. https://www.enablesecurity.com/research/webrtc-hello-race-conditions-paper.pdf, October 2024. White paper, Enable Security GmbH.

[20] J. Fischl, H. Tschofenig, and E. Rescorla. Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS). RFC 5763 (Proposed Standard), May 2010. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc5763.txt. Updated by RFC 8842.

[21] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[22] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Automata-based automated detection of state machine bugs in protocol implementations. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL https://www.ndss-symposium.org/ndss-paper/automata-based-automated-detection-of-state-machine-bugs-in-protocol-implementations/.

[23] Ivan Fratric. XMPP stanza smuggling or how i hacked zoom. Black Hat USA 2022 talk (slides), August 2022. URL https://i.blackhat.com/USA-22/Thursday/US-22-Fratric-XMPP-Stanza-Smuggling.pdf. See also: Project Zero issue 2254.

[24] Prateek Gupta and Vitaly Shmatikov. Security analysis of voice-over-IP protocols. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 49–63. IEEE, 2007.

[25] Nico Heitmann, Hendrik Siewert, Sven Moog, and Juraj Somorovsky. Security analysis of bigbluebutton and edumeet. In *International Conference on Applied Cryptography and Network Security*, pages 190–216. Springer, 2024.

[26] C. Holmberg and R. Shpount. Session Description Protocol (SDP) Offer/Answer Considerations for Datagram Transport Layer Security (DTLS) and Transport Layer Security (TLS). RFC 8842 (Proposed Standard), January 2021. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8842.txt.

[27] Jitsi. jitsi-srtp: SRTP implementation for Jitsi. https://github.com/jitsi/jitsi-srtp, 2021.

[28] A. Keranen, C. Holmberg, and J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445 (Proposed Standard), July 2018. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8445.txt. Updated by RFC 8863.

[29] Masato Kinugawa. Discord desktop app RCE. Masato Kinugawa's Security Blog, October 2020. URL https://mksben.l0.cm/2020/10/discord-desktop-rce.html.

[30] Let's Encrypt. Let's Encrypt. https://letsencrypt.org/, 2025.

[31] Tsahi Levent-Levi. Is WebRTC really free? the costs of running a WebRTC application. URL https://bloggeek.me/is-webrtc-really-free/.

[32] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. TLS-Anvil: Adapting combinatorial testing for TLS libraries. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 215–232, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/maehren.

[33] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc5766.txt. Obsoleted by RFC 8656, updated by RFCs 8155, 8553.

[34] Bill Marczak and John Scott-Railton. Move fast and roll your own crypto: A quick look at the confidentiality of zoom meetings. Citizen Lab Report, April 2020. URL https://citizenlab.ca/2020/04/move-fast-roll-your-own-crypto-a-quick-look-at-the-confidentiality-of-zoom-meetings/.

[35] D. McGrew and E. Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). RFC 5764 (Proposed Standard), May 2010. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc5764.txt. Updated by RFCs 7983, 9443.

[36] NIST National Vulnerability Database. CVE-2020-2655. https://nvd.nist.gov/vuln/detail/CVE-2020-2655, 2020.

[37] P. Patil, T. Reddy, and D. Wing. Traversal Using Relays around NAT (TURN) Server Auto Discovery. RFC 8155 (Proposed Standard), April 2017. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8155.txt.

[38] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews. Session Traversal Utilities for NAT (STUN). RFC 8489 (Proposed Standard), February 2020. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8489.txt.

[39] M. Petit-Huguenin, S. Nandakumar, C. Holmberg, A. Keränen, and R. Shpount. Session Description Protocol (SDP) Offer/Answer Procedures for Interactive Connectivity Establishment (ICE). RFC 8839 (Proposed Standard), January 2021. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8839.txt.

[40] praveen-kd-23. Wrong DTLS Peer Certificate verification (Issue #2076, signalwire/freeswitch). https://github.com/signalwire/freeswitch/issues/2076, May 2023. GitHub issue.

[41] Andreas Reiter and Alexander Marsalek. WebRTC: your privacy is at risk. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 664–669, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344869. doi: 10.1145/3019612.3019844. URL https://doi.org/10.1145/3019612.3019844.

[42] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). RFC 5705 (Proposed Standard), March 2010. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc5705.txt. Updated by RFCs 8446, 8447.

[43] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8446.txt.

[44] E. Rescorla. Security Considerations for WebRTC. RFC 8826 (Proposed Standard), January 2021. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8826.txt.

[45] E. Rescorla. WebRTC Security Architecture. RFC 8827 (Proposed Standard), January 2021. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8827.txt.

[46] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Historic), April 2006. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc4347.txt. Obsoleted by RFC 6347, updated by RFCs 5746, 7507.

[47] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc6347.txt. Obsoleted by RFC 9147, updated by RFCs 7507, 7905, 8996, 9146.

[48] E. Rescorla, H. Tschofenig, and N. Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. RFC 9147 (Proposed Standard), April 2022. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc9147.txt.

[49] Salesforce. JA3. https://github.com/salesforce/ja3, 2020.

[50] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Internet Standard), July 2003. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc3550.txt. Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164, 8083, 8108, 8860.

[51] Selenium Project. Selenium. https://www.selenium.dev/.

[52] SignalWire, Inc. Freeswitch. https://signalwire.com/freeswitch, 2025.

[53] Natalie Silvanovich. Adventures in video conferencing part 2: Fun with FaceTime. Google Project Zero blog, December 2018. URL https://googleprojectzero.blogspot.com/2018/12/adventures-in-video-conferencing-part-2.html.

[54] Natalie Silvanovich. Adventures in video conferencing part 1: The wild world of WebRTC. Google Project Zero blog, December 2018. URL https://googleprojectzero.blogspot.com/2018/12/adventures-in-video-conferencing-part-1.html.

[55] Natalie Silvanovich. Adventures in video conferencing part 3: The even wilder world of WhatsApp. Google Project Zero blog, December 2018. URL https://googleprojectzero.blogspot.com/2018/12/adventures-in-video-conferencing-part-3.html.

[56] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016. ISBN 9781450341394. doi: 10.1145/2976749.2978411. URL https://doi.org/10.1145/2976749.2978411.

[57] R. Stewart, M. Tüxen, and K. Nielsen. Stream Control Transmission Protocol. RFC 9260 (Proposed Standard), June 2022. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc9260.txt.

[58] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm, 2017. URL http://cado-nfs.inria.fr/. Release 2.3.0.

[59] Tsahi Levent-Levi. WebRTC and Man-in-the-Middle Attacks. https://webrtchacks.com/webrtc-and-man-in-the-middle-attacks/, June 2015.

[60] Cristiana Tudor. The impact of the COVID-19 pandemic on the global web and video conferencing SaaS market. *Electronics*, 11(16), 2022. ISSN 2079-9292. doi: 10.3390/electronics11162633. URL https://www.mdpi.com/2079-9292/11/16/2633.

[61] M. Tuexen, R. Stewart, R. Jesup, and S. Loreto. Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets. RFC 8261 (Proposed Standard), November 2017. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8261.txt. Updated by RFCs 8899, 8996.

[62] J. Uberti, C. Jennings, and E. Rescorla (Ed.). JavaScript Session Establishment Protocol (JSEP). RFC 8829 (Proposed Standard), January 2021. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc8829.txt.

[63] Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. Factoring as a service. Cryptology ePrint Archive, Paper 2015/1000, 2015. URL https://eprint.iacr.org/2015/1000.

[64] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. 12(1): 1–28, January 1999. doi: 10.1007/PL00003816.

[65] WebRTC-Security contributors. A study of WebRTC security. https://webrtc-security.github.io/, 2015.

[66] Kaito Yamada. Pcap4J. https://www.pcap4j.org/.

A Evaluation Setup

Our testbed consists of an analysis host connected via Ethernet to the backbone network and exposing a local interface to the target device. For mobile applications, the phone connects to the analysis host's Wi-Fi hotspot, which analyzes and forwards traffic through our tool to the backbone network. For desktop and browser applications, we run a virtual machine (VM) bridged to an interface on the analysis host. All web application DTLS tests were conducted using Google Chrome version 123. Each application was tested with a 300 ms timeout for DTLS messages. If a run failed, e.g., due to packet loss or a switch to a different ICE candidate, we repeated the test flow up to five times. In our evaluation, such test failures were typically attributed to unhandled protocol features (e.g., STUN as a transport), which we then implemented for the final evaluation. Many applications do not surface alerts on handshake failure and instead silently abort. In such cases, we performed five retries and recorded the missing response. Profiling an application can take between 5 minutes and multiple hours, depending on the complexity of the call flow, the number of distinct connections initiated in each call, and the target's DTLS configuration. Where applicable, we automated the connection setup using Selenium, with the application running in a browser within the VM. Developing robust Selenium scripts proved challenging due to nondeterministic site behavior (e.g., unsolicited feedback prompts). Consequently, we profiled most applications via manual call initiation or lightweight click-automation scripts. In total, we performed approximately 17 000 calls during our final evaluation. Some services do not issue (public) version numbers, so we are not able to pinpoint each of them to a release tag. Unless otherwise stated, we performed the tests in July of 2025.