2017 International Conference on Computational Science and Computational Intelligence

Security Mechanisms for Signaling in WebRTC-based Peer-to-Peer Networks

Dennis Boldt and Sebastian Ebers
Institute of Telematics
University of Lübeck
Lübeck, Germany
Email: {boldt,ebers}@itm.uni-luebeck.de

***Abstract*—Chord is an efficient and well-known way to create an overlay for a structured peer-to-peer network. We use Chord for a peer-to-peer network built on WebRTC, a set of protocols for direct connections between web browsers. However, Chord lacks mechanisms for authentication and end-to-end confidentiality. Thus, a man-in-the-middle attack could occur when two peers negotiate WebRTC parameters for a direct connection. We solve this security vulnerability with hybrid encryption: Each host generates a unique long-term asynchronous key pair for authentication and short-term asynchronous key pairs to establish synchronous secret keys. With these, peers can exchange WebRTC connection parameters via end-to-end authenticated and encrypted messages over multiple hops and thus establish a direct connection in a secure fashion.**

***Index Terms*—Peer-to-Peer; WebRTC; Encryption; Key-Exchange; Authentication**

I. Introduction

A *Peer-to-peer* (P2P) network is an example for a distributed system architecture, which does not necessarily rely on central servers. A *peer* is a node in a P2P network, acting as client and server by providing and consuming content to and from other peers via direct connections. While in a centralized P2P network a server is used for coordination and content lookup, a fully decentralized P2P network is comprised of peers only. Hybrid P2P networks make use of so-called *super peers*, which are not pure servers but special peers providing additional functionality. A so-called overlay network is used for coordination of peers as well as content lookup and distribution.

P2P networks are further distinguished in *unstructured* and *structured* networks. In the former, there is no specific relation between a peer and the content it provides. In the latter, a deterministic function determines which peer is responsible for which content. A well-known example for a structured P2P network protocol is Chord [1], which we use in our approach.

While Chord organizes content and peers, it does not define how to establish the actual connection between peers. For this, *Web Real-Time Communication* (WebRTC) [2] can be used, which allows for direct P2P connections between browsers.

Imagine a requesting peer p_a wants to establish a WebRTC connection with peer p_b. First, p_a employs Chord to locate p_b. Second, both peers exchange so-called *signaling messages* comprised of transport addresses along the Chord-ring. Finally, p_a establishes a direct WebRTC connection to p_b.

While WebRTC connections are encrypted to provide confidentiality between two peers, Chord neither provides peer authentication nor end-to-end confidentiality over multiple hops. Thus, an attacker p_e who learns the transport addresses from p_a and p_b could intercept the peers' messages and perform a *man-in-the-middle* (MITM) attack. For this, p_e would open a WebRTC connection to p_b in the name of p_a and vice versa. By relaying the messages to the legitimate peer, p_e could eavesdrop and even modify the exchanged messages.

This paper presents an approach to enrich a WebRTC-based Chord network by peer authentication and end-to-end confidentiality. To communicate in an authenticated and confidential fashion, each peer creates a unique long-term asymmetric key pair based on the *Elliptic Curve Digital Signature Algorithm* (ECDSA) and uses the public key's hash value as its own identifier in the Chord-ring. Before exchanging signaling messages, the peers use *Elliptic Curve Diffie-Hellman* with *ephemeral keys* (ECDHE)—as in *Transport Layer Security* (TLS) [3]—to establish a secret key for encrypting the signaling messages.

However, our approach also introduces a message overhead for the authenticated key exchange and a computational overhead for securing the message exchange.

After briefly introducing the fundamentals in Section II, we present the identified problems in Section III, our solutions in Section IV and the evaluation of our approach in Section V.

II. Fundamentals

This sections provides the fundamental technologies and mechanisms used in our approach.

A. Chord

Chord [1] is a common structured P2P network protocol which forms a *Distributed Hash Table* (DHT) for organizing peers and data items. $P = \{p_1, \ldots, p_N\}$ is a set of N peers and $D = \{d_1, \ldots, d_L\}$ is a set of L data items. Chord employs a cryptographic hash function $H\colon \{0,1\}^* \rightarrow \{0,1\}^n$ to compute a fixed length identifier (ID) of n bits for every peer $p_i \in P$ and every data item $d_k \in D$. The identifiers are denoted as h_{p_i} and h_{d_k}, respectively. Based on their identifiers, which are called *keys* for data items, all peers are arranged in a ring, the so-called Chord-ring. Subsequently, peer p_i is in

978-1-5386-2652-8/17 $31.00 © 2017 IEEE
DOI 10.1109/CSCI.2017.4

Authorized licensed use limited to: Hochschule RheinMain (formerly FH Wiesbaden). Downloaded on November 07,2024 at 13:24:44 UTC from IEEE Xplore. Restrictions apply.

charge of data items in the key space between the predecessor's ID and its own ID: $(\, h_{\text{pred}(p_i)},\ h_{p_i}\,]$.

Chord's core operation is to find p_i, which is in charge of a given hash value h_x: $p_i = \text{find\_successor}(h_x)$. Since every peer knows how to contact its successor, this operation can be naïvely implemented by walking along the ring of peers in $O(N)$ steps (see Figure 2). For scalability, Chord improves the inefficient lookup to $\log_2(N)$ steps. Since this is not required for our approach, we refer the interested reader to [1].

One can use Chord to store and retrieve *key-value pairs*, where data items are the values and hashes are their corresponding keys: $\text{put}(h_d, d)$ and $d = \text{get}(h_d)$.

B. WebRTC

Web Real-Time Communication (WebRTC) [2] is set of protocols used to establish direct P2P connections between two browsers comprising, among others, data channels [4]. Figure 1 shows all required steps to establish a data channel with the offer/answer model [5]. For this, WebRTC uses *Interactive Connectivity Establishment* (ICE) [6], which can be described in three phases:

1) Gather *signaling messages* comprising offer, answer and ICE candidates, which contain transport addresses. They can be gathered using a server offering the protocols *Session Traversal Utilities for NAT* (STUN) [7] and *Traversal Using Relays around NAT* (TURN) [8].
2) Exchange of the signaling messages between both peers through a so-called *signaling channel*.
3) Connectivity checks with all ICE candidates performed by both peers to establish a data channel.

Further details on WebRTC are described in [9].

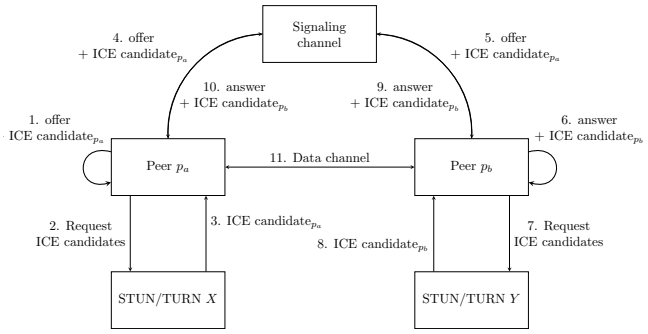

Fig. 1: WebRTC offer/answer model, including offer, answer, ICE candidates and signaling channel.

C. Bootstrapping of WebRTC-based Peer-to-Peer Networks

In our previous work we combined Chord and WebRTC into a WebRTC-based Peer-to-Peer network [9]. For this, we presented a decentralized bootstrapping architecture, which uses the *Domain Name System* (DNS) [10] as a highly decentralized architecture to handle bootstrapping. We defined that a peer can operate in two ways: *Slaves* are regular peers in the Chord-ring used by browsers, which don't have any special capabilities, except those which are needed for the Chord protocol. *Masters* are special peers, which additionally offer a STUN/TURN server and a WebSocket [11] server with access to the Chord-ring via WebRTC.

To join the network, called bootstrapping, p_a connects via WebSockets to a known master p_m in the network and sends a $\text{find\_successor}(h_{p_a})$ request, which p_m forwards in-network via WebRTC data channels. The message is forwarded until an appropriate peer handles the request (see Figure 2).

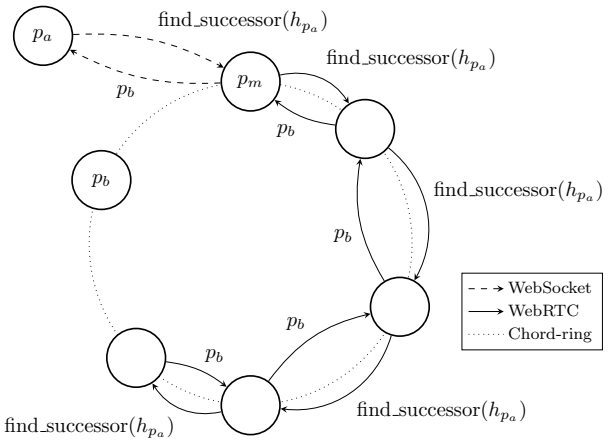

Fig. 2: p_a connects via WebSockets to a master p_m and sends a $\text{find\_successor}(h_{p_a})$ request, which p_m forwards in-network via WebRTC data channels.

Afterwards, the signaling messages are exchanged between p_a and p_b, which can be seen for offer and answer in Figure 3. The WebSocket connection between p_a and p_m together with the in-network WebRTC data channels form the signaling channel between p_a and p_b.

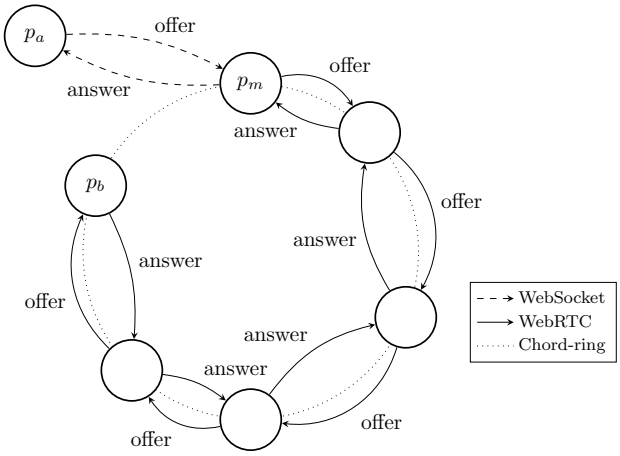

Fig. 3: Offer and answer messages are exchanged between p_a and p_b.

To establish direct connections via data channels between peers within the network, the signaling messages are exchanged in-network using the data channels only, i.e., no server is required.

Authorized licensed use limited to: Hochschule RheinMain (formerly FH Wiesbaden). Downloaded on November 07,2024 at 13:24:44 UTC from IEEE Xplore. Restrictions apply.

D. Cryptographic Methods

This section gives an overview of the cryptographic methods and their definitions used in this paper. We use H to refer to a *cryptographic hash* function and K to refer to a *secret key* employed by symmetric cryptography. KR and KU refer to a *private* and *public key*, respectively, which form a key pair employed by asymmetric cryptography. We use σ as a container for the cryptographic parameters.

The *Elliptic Curve Digital Signature Algorithm* (ECDSA) [12], [13] is used to sign and verify messages. We define $\sigma^{\text{ECDSA}} = (C^{\text{ECDSA}}, H)$, where C^{ECDSA} is an elliptic curve and H is a hash function. A peer p_i generates a long-term asymmetric key pair:

$$(KR_{p_i}^{\text{DSA}}, KU_{p_i}^{\text{DSA}}) = \text{ECDSA}_{\text{keygen}}(\sigma^{\text{ECDSA}}) \tag{1}$$

The private key can be used for signing a message, which yields a signature:

$$\text{sig}_{p_i}(m) = \text{ECDSA}_{\text{sign}}(KR_{p_i}^{\text{DSA}}, m, \sigma^{\text{ECDSA}}) \tag{2}$$

The corresponding public key can be used to verify the signature:

$$\text{ECDSA}_{\text{verify}}(KU_{p_i}^{\text{DSA}}, \text{sig}_{p_i}(m), m, \sigma^{\text{ECDSA}}) \tag{3}$$

To enable forward secrecy, we employ the *Elliptic Curve Diffie-Hellman* with *ephemeral keys* (ECDHE) [14] to establish a shared secret S between two peers $p_i, p_j \in P,\ i \neq j$. We define $\sigma^{\text{ECDHE}} = (C^{\text{ECDHE}})$, where C^{ECDHE} is an elliptic curve. Both peers generate an ephemeral, i.e., short-term asymmetric key pair:

$$(KR_{p_i}^{\text{DHE}}, KU_{p_i}^{\text{DHE}}) = \text{ECDHE}_{\text{keygen}}(\sigma^{\text{ECDHE}}) \tag{4}$$

$$(KR_{p_j}^{\text{DHE}}, KU_{p_j}^{\text{DHE}}) = \text{ECDHE}_{\text{keygen}}(\sigma^{\text{ECDHE}})$$

In contrast to the previously generated ECDSA key pair, this one is only used for establishing the shared secret S and directly discarded afterwards. Finally, the shared secret can be derived by both peers from their own private key and the other peer's public key:

$$S = \text{ECDHE}_{\text{deriveSecret}}(KR_{p_i}^{\text{DHE}}, KU_{p_j}^{\text{DHE}}, \sigma^{\text{ECDHE}}) \tag{5}$$

The *Password-Based Key Derivation Function 2* (PBKDF2) [15] takes a shared secret S of arbitrary length and derives a secret key K of fixed length for a symmetric-key algorithm. We define $\sigma^{\text{PBKDF2}} = (H, n)$, where H is a cryptographic hash function applied n-times. Additionally, PBKDF2 requires a cryptographically secure pseudorandom number s, a so-called salt. If freshly generated, it ensures that PBKDF2 always derives a different secret key for the same shared secret:

$$K = \text{PBKDF2}(S, s, \sigma^{\text{PBKDF2}}) \tag{6}$$

The *Advanced Encryption Standard* (AES) [16] is a symmetric-key algorithm. We define $\sigma^{\text{AES}} = (\rho, l)$, where ρ is the cryptographic mode of operation and l as the length of K. Both peers must use the same secret key K, cryptographically secure pseudorandom number IV, a so-called initialization vector, and σ^{AES}. A plain text message m can be encrypted with the secret key K:

$$m' = \text{AES}_{\text{enc}}(m, K, IV, \sigma^{\text{AES}}) \tag{7}$$

The cipher text message m' can be decrypted with the same key K:

$$m = \text{AES}_{\text{dec}}(m', K, IV, \sigma^{\text{AES}}) \tag{8}$$

Since the IV is required decryption, it must be transferred unencrypted.

We denote the tuple of all volatile short-term cryptographic parameters as Σ:

$$\Sigma = (\sigma^{\text{ECDHE}}, \sigma^{\text{PBKDF2}}, \sigma^{\text{AES}}) \tag{9}$$

Note that Σ does not contain σ^{ECDSA}, since we use ECDSA to achieve long-term endpoint authentication (see Section IV-A).

E. Web Cryptography API

The *Web Cryptography API* (WebCrypto API) [17] allows JavaScript to use cryptographic methods natively in a browser. It supports the generation, import and export of symmetric and asymmetric keys, to sign messages and verify signatures, to encrypt and decrypt messages, to calculate hash values and to generate cryptographically secure pseudorandom numbers. Thus, this API supports all previously defined cryptographic methods.

III. Problem Statement

Since we use DNS for bootstrapping, we can use *Domain Name System Security Extensions* (DNSSEC) [18] to secure the DNS requests. Because STUN and TURN support TLS-over-TCP, a secure connection to the STUN/TURN server can be used by ICE to gather the ICE candidates. WebSockets support TLS-over-TCP as well. Thus, the connection between a joining and a master peer is secured. WebRTC data channels between directly connected peers in the Chord-ring are secured with *Datagram Transport Layer Security* (DTLS) [19].

However, even if all data channels are secured, intermediate nodes along the signaling channel are able to eavesdrop, modify, forge or drop the signaling messages transferred via multi-hop, as no end-to-end encryption exists.

Thus, we consider the signaling channel to be insecure and mechanisms for authentication and end-to-end confidentiality are required. Fortunately, modern browsers support the WebCrypto API, which we use in our approach.

IV. Security Mechanisms

This section provides a solution for endpoint authentication, key exchange, mutual authentication and end-to-end confidentiality.

Authorized licensed use limited to: Hochschule RheinMain (formerly FH Wiesbaden). Downloaded on November 07,2024 at 13:24:44 UTC from IEEE Xplore. Restrictions apply.

A. Endpoint Authentication

For long-term endpoint authentication, p_i creates an ECDSA key pair $(KR_{p_i}^{\text{DSA}}, KU_{p_i}^{\text{DSA}})$. The endpoint is identified by its public key's hash value :

$$h_{p_i} = h_{KU_{p_i}^{\text{DSA}}} = H(KU_{p_i}^{\text{DSA}}) \tag{10}$$

B. Mutual Authentication, Key Exchange and Confidential Signaling in 3 Phases

Assume p_a, p_b with their corresponding long-term ECDSA key pairs $(KR_{p_a}^{\text{DSA}}, KU_{p_a}^{\text{DSA}})$ and $(KR_{p_b}^{\text{DSA}}, KU_{p_b}^{\text{DSA}})$. If p_a wants to establish a data channel to p_b, both peers will have to exchange signaling messages (see Section II-B). As described in Section III, the signaling channel is vulnerable to a MITM attacker p_e. This could be the master peer or any other peer along the signaling channel. To achieve mutual authentication and end-to-end confidentiality between p_a and p_b, we perform the following three phases. We use $\|$ to denote concatenation.

Phase 1: p_a and p_b perform mutual authentication and exchange of cryptographic parameter tuples (see Figure 4):

1) p_a creates a set of supported cryptographic parameter tuples $\Sigma_{p_a}^*$ and signs them with its long-term private key: $\text{sig}_{p_a}(\Sigma_{p_a}^*) = \text{ECDSA}_{\text{sign}}(KR_{p_a}^{\text{DSA}}, \Sigma_{p_a}^*, \sigma^{\text{ECDSA}})$.
2) p_a sends $[KU_{p_a}^{\text{DSA}} \parallel \Sigma_{p_a}^* \parallel \text{sig}_{p_a}(\Sigma_{p_a}^*)]$ over the signaling channel to p_b.
3) p_b takes p_a's ECDSA public key and calculates the hash $h_{p_a} = H(KU_{p_a}^{\text{DSA}})$ to authenticate p_a.
4) p_b verifies the signature of the cryptographic parameter tuples: $\text{ECDSA}_{\text{verify}}(KU_{p_a}^{\text{DSA}}, \text{sig}_{p_a}(\Sigma_{p_a}^*), \Sigma_{p_a}^*, \sigma^{\text{ECDSA}})$. If it fails, p_b will reject the connection.
5) p_b selects a cryptographic parameter tuple $\Sigma_{p_b} \in \Sigma_{p_a}^*$.
6) p_b signs Σ_{p_b} which yields the signature: $\text{sig}_{p_b}(\Sigma_{p_b}) = \text{ECDSA}_{\text{sign}}(KR_{p_b}^{\text{DSA}}, \Sigma_{p_b}, \sigma^{\text{ECDSA}})$.
7) p_b returns $[KU_{p_b}^{\text{DSA}} \parallel \Sigma_{p_b} \parallel \text{sig}_{p_b}(\Sigma_{p_b})]$ over the signaling channel to p_a.
8) p_a takes p_b's ECDSA public key and calculates the hash $h_{p_b} = H(KU_{p_b}^{\text{DSA}})$ to authenticate p_b.
9) p_a verifies the signature of the selected cryptographic parameters: $\text{ECDSA}_{\text{verify}}(KU_{p_b}^{\text{DSA}}, \text{sig}_{p_b}(\Sigma_{p_b}), \Sigma_{p_b}, \sigma^{\text{ECDSA}})$. If it fails, p_a will reject the connection.

Phase 2: Based on the negotiated Σ_{p_b} from Phase 1, p_a and p_b perform an authenticated exchange of ECDHE public keys to derive a secret key (see Figure 5):

10) p_a generates a short-term ECDHE key pair $(KR_{p_a}^{\text{DHE}}, KU_{p_a}^{\text{DHE}}) = \text{ECDHE}_{\text{keygen}}(\sigma^{\text{ECDHE}})$.
11) p_a signs the public key with its ECDSA private key $\text{sig}_{p_a}(KU_{p_a}^{\text{DHE}}) = \text{ECDSA}_{\text{sign}}(KR_{p_a}^{\text{DSA}}, KU_{p_a}^{\text{DHE}}, \sigma^{\text{ECDSA}})$.
12) p_a sends $[KU_{p_a}^{\text{DHE}} \parallel \text{sig}_{p_a}(KU_{p_a}^{\text{DHE}})]$ over the signaling channel to p_b.
13) p_b verifies the signature of p_a's ECDHE public key to confirm its integrity: $\text{ECDSA}_{\text{verify}}(KU_{p_a}^{\text{DSA}}, \text{sig}_{p_a}(KU_{p_a}^{\text{DHE}}), KU_{p_a}^{\text{DHE}}, \sigma^{\text{ECDSA}})$. If it fails, p_b will reject the connection.
14) p_b generates its ECDHE key pair $(KR_{p_b}^{\text{DHE}}, KU_{p_b}^{\text{DHE}}) = \text{ECDHE}_{\text{keygen}}(\sigma^{\text{ECDHE}})$.

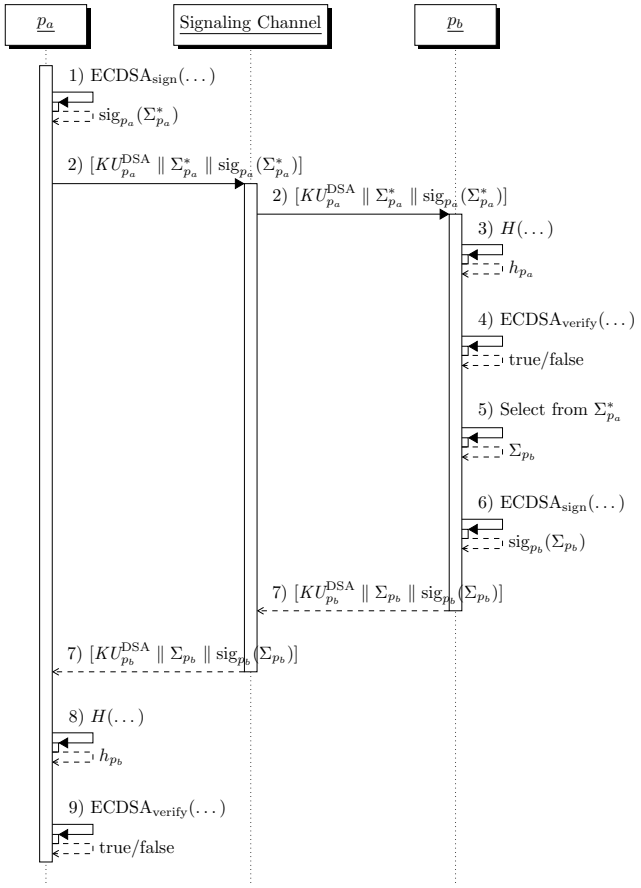

Fig. 4: In phase 1, p_a and p_b perform mutual authentication and exchange of cryptographic parameter tuples.

15) p_b derives the shared secret $S = \text{ECDHE}_{\text{deriveSecret}}(KR_{p_b}^{\text{DHE}}, KU_{p_a}^{\text{DHE}}, \sigma^{\text{ECDHE}})$.
16) p_b generates a fresh random salt s.
17) p_b derives the secret key $K = \text{PBKDF2}(S, s, \sigma^{\text{PBKDF2}})$ with the shared secret S and the salt s.
18) p_b calculates a signature for the concatenation of its short-term public key and the salt: $\text{sig}_{p_b}(KU_{p_b}^{\text{DHE}} \parallel s) = \text{ECDSA}_{\text{sign}}(KR_{p_b}^{\text{DSA}}, KU_{p_b}^{\text{DHE}} \parallel s, \sigma^{\text{ECDSA}})$.
19) p_b sends $[KU_{p_b}^{\text{DHE}} \parallel s \parallel \text{sig}_{p_b}(KU_{p_b}^{\text{DHE}} \parallel s)]$ over the signaling channel back to p_a.
20) p_a verifies the signature of p_b's ECDHE public key and the salt: $\text{ECDSA}_{\text{verify}}(KU_{p_b}^{\text{DSA}}, \text{sig}_{p_b}(KU_{p_b}^{\text{DHE}} \parallel s), KU_{p_b}^{\text{DHE}} \parallel s, \sigma^{\text{ECDSA}})$. If it fails, p_a will reject the connection.
21) p_a derives the shared secret $S = \text{ECDHE}_{\text{deriveSecret}}(KR_{p_a}^{\text{DHE}}, KU_{p_b}^{\text{DHE}}, \sigma^{\text{ECDHE}})$.
22) p_a derives the secret key $K = \text{PBKDF2}(S, s, \sigma^{\text{PBKDF2}})$.

Now the key-exchange is finished and both peers are mutually authenticated. All ECDHE keys are discarded. A MITM p_e has no known way to modify any of the exchanged messages undetected. If p_e changes any of p_a's or p_b's public keys, it will have to change all signatures as well.

Authorized licensed use limited to: Hochschule RheinMain (formerly FH Wiesbaden). Downloaded on November 07,2024 at 13:24:44 UTC from IEEE Xplore. Restrictions apply.

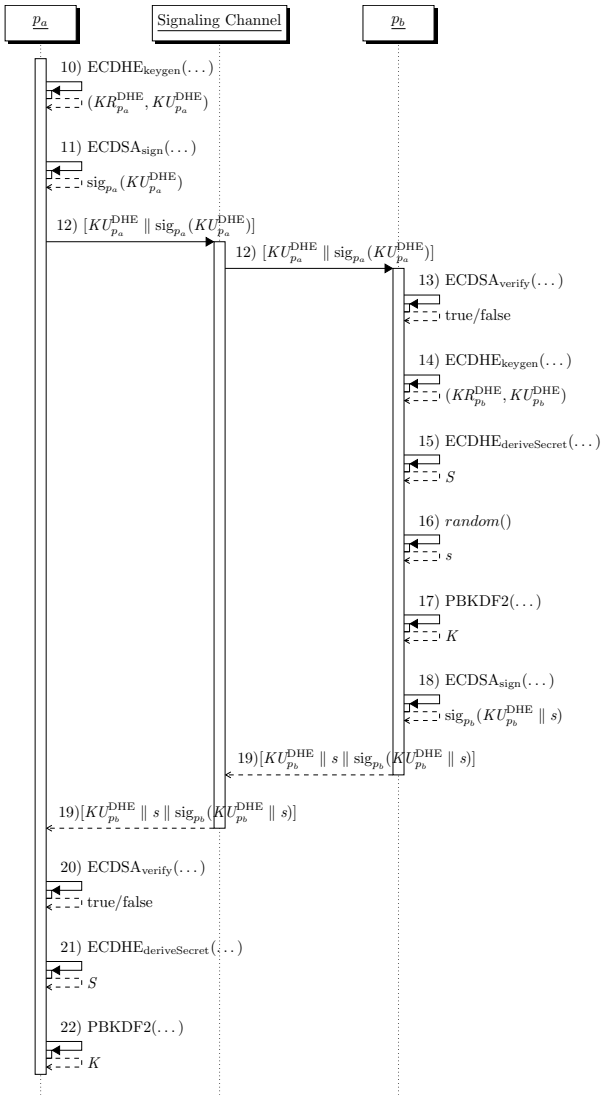

Fig. 5: In phase 2, p_a and p_b exchange authenticated keys to derive a secret key.

Phase 3: The signaling messages $M = \{m_1, \dots, m_n\}$ are each exchanged confidentially between p_a and p_b (see Figure 6):

23) p_a generates a random initialization vector IV_i.
24) p_a encrypts $m_i \in M$ applying AES with the exchanged secret key K and IV_i: $m'_i = \text{AES}_{\text{enc}}(m_i, K, IV_i, \sigma^{\text{AES}})$.
25) p_a signs the concatenation of m'_i and IV_i with its ECDSA private key: $\text{sig}_{p_a}(m'_i \parallel IV_i) = \text{ECDSA}_{\text{sign}}(KR^{\text{DSA}}_{p_a}, m'_i \parallel IV_i, \sigma^{\text{ECDSA}})$.
26) p_a sends $[m'_i \parallel IV_i \parallel \text{sig}_{p_a}(m'_i \parallel IV_i)]$ over the signaling channel to p_b.
27) p_b verifies the signature of the encrypted message and the IV to confirm the integrity and authentication: $\text{ECDSA}_{\text{verify}}(KU^{\text{DSA}}_{p_a}, \text{sig}_{p_a}(m'_i \parallel IV_i), m'_i \parallel IV_i, \sigma^{\text{ECDSA}})$. If it fails, p_b will reject the connection.
28) p_b decrypts m'_i with $m_i = \text{AES}_{\text{dec}}(m'_i, K, IV_i, \sigma^{\text{AES}})$.
29) p_b sends an acknowledgement.

p_b sends its signaling messages to p_a in the same way. Overall, p_a and p_b exchange 8 signaling messages with the corresponding acknowledgements, i.e., 8 round-trips [9].

Thus, we achieved a confidential end-to-end exchange of the signaling messages between p_a and p_b. This fulfills the requirements described in Section III.

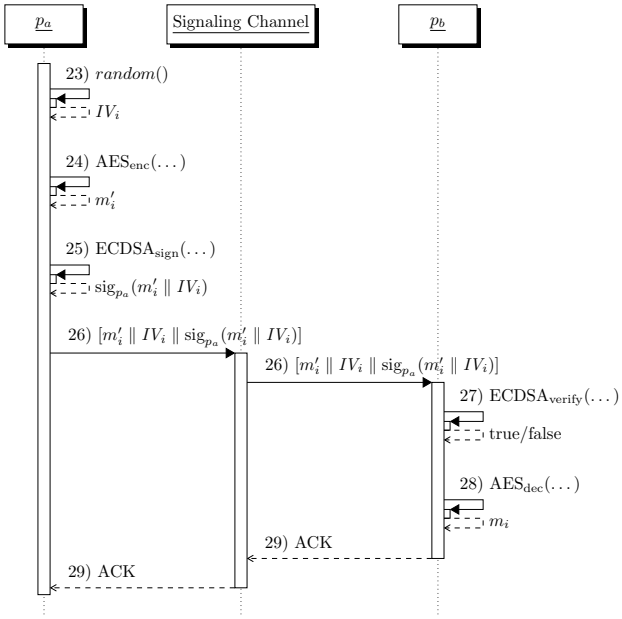

Fig. 6: In phase 3, p_a and p_b exchange the signaling messages confidentially (only one round-trip is shown).

V. Performance Evaluation

To evaluate our approach, we created the JavaScript framework *WebCrypto.js* [20], which simplifies the usage of the WebCrypto API. It can be used within browsers and with node.js. Former provides the API natively, while the latter requires *node-webcrypto-ossl* [21] as an additional library.

A. Cryptographic Parameters

Based on the recommendations from BSI [22], NIST [23], [24], Cisco [25], and LastPass [26], we selected the parameters shown in Table I, since they are also supported by Chrome 61, Firefox 58 and the library *node-webcrypto-ossl*.

TABLE I. Recommended, supported and selected cryptographic parameters.

Parameter	Recommendations	Supported + Selected
H	H (bit): 224[23], 256 [22], [25]	H: SHA-256
σ^{ECDSA}	C (bit): 224 [23], 250 [22], 256 [25]	C^{ECDSA}: P-256
σ^{ECDHE}	C (bit): 224 [23], 250 [22], 256 [25]	C^{ECDHE}: P-256
σ^{PBKDF2}	n (rounds): 1000 [24], 10 000 [26]	n: 10 000
σ^{AES}	ρ (mode): CBC or GCM [25] l (bit): 112 [22], 128 [23], [25]	ρ: GCM l: 128

Authorized licensed use limited to: Hochschule RheinMain (formerly FH Wiesbaden). Downloaded on November 07,2024 at 13:24:44 UTC from IEEE Xplore. Restrictions apply.

B. Experimental Results

To evaluate our approach, we implemented the security mechanisms from Section IV. We did not measure the time required to transfer the messages over the signaling channel between p_a and p_b. This would introduce bias since it depends on the logical and physical distance between the involved peers. Thus we used an in-memory Signaling Channel (without actually sending the messages). We ran $1\,000$ iterations in Chrome 61, Firefox 58 and Node 8.7.0 on a Dell Latitude E6420 with an Intel Core i5-2520M CPU with 2.50 GHz and 8 GB RAM.

Table II shows the average computational overhead for each phase. The most significant difference can be seen in phase 2, where Firefox is by far the slowest. In comparison to the 8 round-trips required to establish a WebRTC connection [9], we just need 2 additional round-trips to establish the shared key, which is a message overhead of $25\,\%$. Thus, we consider the additional overheads to be acceptable.

TABLE II. Average values for all three phases in ms. Phase 3^* includes it 8 times. Sum* includes phase 1, 2 and phase 3^*.

Environment	Phase 1	Phase 2	Phase 3	Phase 3^*	Sum*
Chrome	5.05	27.36	4.93	39.44	71.85
Firefox	8.03	64.19	4.10	32.80	105.02
Node	9.23	46.26	6.18	49.44	104.93

VI. Conclusion and Future Work

In this paper, we presented an approach to prevent MITM attacks on peers in a Chord network when exchanging signaling messages over multiple hops. We use hybrid cryptography to enable peers to exchange end-to-end encrypted and authenticated messages comprising parameters to subsequently establish a direct WebRTC connection.

Our evaluation reveals overheads in terms of time and additional messages. The time for cryptographically securing the messages takes less than 110 ms in average for the slowest JavaScript engine. The negotiation of cryptographic parameters and the establishment of cryptographic keys requires two RTTs over the signaling channel. We consider both acceptable in comparison to the security benefit.

In future work, we will integrate our solution in the approach presented in [9] and evaluate the delay introduced by the additional round trip times in a wide area network.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in ACM SIGCOMM Computer Communication Review, vol. 31, no. 4. ACM, 2001, pp. 149–160.

[2] H. Alvestrand, "Overview: Real Time Protocols for Browser-based Applications," Internet Engineering Task Force, Internet Draft, draft-ietf-rtcweb-overview-18, Mar. 2017.

[3] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008.

[4] R. Jesup, S. Loreto, and M. Tuexen, "WebRTC Data Channels," Internet Engineering Task Force, Internet Draft, draft-ietf-rtcweb-data-channel-13, Jan. 2015.

[5] J. Rosenberg and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)," RFC 3264 (Proposed Standard), Internet Engineering Task Force, Jun. 2002.

[6] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245 (Proposed Standard), Internet Engineering Task Force, Apr. 2010.

[7] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389 (Proposed Standard), Internet Engineering Task Force, Oct. 2008.

[8] 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), Internet Engineering Task Force, Apr. 2010.

[9] D. Boldt, F. Kaminski, and S. Fischer, "Decentralized Bootstrapping for WebRTC-based P2P Networks," The Fifth International Conference on Building and Exploring Web Based Environments (WEB2017), May 2017, pp. 17–23.

[10] P. Mockapetris, "Domain names - concepts and facilities," RFC 1034 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1987.

[11] I. Hickson, "The Websocket API," W3C Candidate Recommendation, Sep. 2012. [Online]. Available: http://www.w3.org/TR/websockets/ [retrieved: November, 2017]

[12] D. McGrew, K. Igoe, and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms," RFC 6090 (Informational), Internet Engineering Task Force, Feb. 2011.

[13] National Institute of Standards and Technology (NIST), "FIPS PUB 186-4: Digital Signature Standard (DSS)," Federal Information Processing Standard, vol. 186-4, Jul. 2013.

[14] D. R. L. Brown, "Standards for Efficient Cryptography 1 (SEC 1): Elliptic Curve Cryptography," http://www.secg.org/sec1-v2.pdf, 5 2009.

[15] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RFC 2898 (Informational), Internet Engineering Task Force, Sep. 2000.

[16] National Institute of Standards and Technology (NIST), "FIPS PUB 197: Advanced Encryption Standard (AES)," Federal Information Processing Standard, vol. 197, 2001.

[17] M. Watson, "Web Cryptography API," W3C Working Group Note, Jan. 2017. [Online]. Available: https://www.w3.org/TR/WebCryptoAPI/ [retrieved: November, 2017]

[18] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," RFC 4033 (Proposed Standard), Internet Engineering Task Force, Mar. 2005.

[19] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347 (Proposed Standard), Internet Engineering Task Force, Jan. 2012.

[20] D. Boldt, "WebCrypto.js," Github, Oct. 2017. [Online]. Available: https://github.com/boldt/WebCrypto.js [retrieved: November, 2017]

[21] Peculiar Ventures Inc, "WebCrypto.js," Github, Oct. 2017. [Online]. Available: https://github.com/PeculiarVentures/node-webcrypto-ossl/ [retrieved: November, 2017]

[22] Bundesamt fur Sicherheit in der Informationstechnik (BSI), "Kryptographische Verfahren: Empfehlungen und Schlüssellangen, BSI TR-02102-1," Technische Richtlinie, vol. 2017-01, Feb. 2017.

[23] National Institute of Standards and Technology (NIST), "SP 800-57: Recommendation for Key Mamangement – Part 1: General," NIST Special Publication, vol. 800-57 Part 1, Rev. 4, 2001.

[24] ——, "SP 800-132: Recommendation for Password-Based Key Derivation – Part 1: Storage Applications," NIST Special Publication, vol. 800-132, 2010.

[25] Cisco, "Next Generation Encryption," Cisco Security Research & Operations, Oct. 2015. [Online]. Available: https://www.cisco.com/c/en/us/about/security-center/next-generation-cryptography.html [retrieved: November, 2017]

[26] LastPass, "Password Iterations (PBKDF2)," LastPass, Oct. 2017. [Online]. Available: https://helpdesk.lastpass.com/en/account-settings/general/password-iterations-pbkdf2/ [retrieved: November, 2017]

Authorized licensed use limited to: Hochschule RheinMain (formerly FH Wiesbaden). Downloaded on November 07,2024 at 13:24:44 UTC from IEEE Xplore. Restrictions apply.