Anonymous Communication in the Browser via Onion-Routing

Florian Burgstaller, Andreas Derler, Stefan Kern, Gabriel Schanner, Andreas Reiter
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Austria
{florian.burgstaller, andreas.derler, s.kern, gabriel.schanner}@student.tugraz.at, andreas.reiter@iaik.tugraz.at

***Abstract*—Every single communication on the Internet reveals private and sensitive information of the communicating parties if no further measures are applied. Various applications and measures are already available to e.g. tunnel traffic through other nodes to obscure the original sender and receiver. Existing frameworks require external applications, running on the particular nodes. We propose a flexible architecture for an anonymous communication framework that supports the interoperability among different platforms. Our proof-of-concept implementation, based on web standards and web technologies shows the feasibility of the framework in terms of usability and interoperability. The framework is running completely in the web-browser and does not have requirements on external applications. The evaluation results show that our framework brings great benefits to user's privacy and security.**

I. Introduction

Communication on the Internet is still a hot topic, and gives users the feeling of communicating anonymously. Nevertheless, the genuine truth is, that with every single packet users reveal sensitive information in terms of public IP address information. This information can be used e.g. to geo-locate users and compromice their privacy. With more advanced and application oriented approaches like Web-Browser fingerprinting [7, 8, 11] attackers can track users activity in a reliable manner.

Till now, anonymity on application level is provided by external applications by e.g. tunnelling the traffic through an anonymity network. This increases the burden of using such a technology, as users have to actively install anonymity-enabling systems and configure applications to use it. We are heading for a solution that can seamlessly be integrated into cross-platform and web-based applications without external dependencies. With the emergence of new Web-technologies, capabilities of Web-applications are constantly increasing. Currently we see a movement of bringing all applications to the web, backed by cloud computing resources. New web technologies bring large advancements in terms of asynchronous communication with multiple servers or even direct browser-to-browser communication. Hence, the technological basis to enable our vision is available.

In this paper we propose an architecture of an anonymity network that is based on peer-to-peer technology and orients on already established anonymity layers. Our main goal beside protecting user's privacy is to maintain interoperability among different devices and operating systems. The proof-of-concept implementation is based on new and already established web standards and therefore is applicable in various scenarios for a broad range of desktop and mobile devices.

The remainder of this paper is structured as follows. In Section II we give relevant background information and related work on this topic. In Section III we present our general architecture, where the implementation as presented in Section IV is based on. Evaluations in terms of performance and security are provided in Section V. In Section VI we highlight possible future work and draw conclusions.

II. Background and Related Work

Chaum [4] first proposed a framework of an untraceable, anonymous electronic mail system using mixes and public key cryptography. A mix is a computer that processes mail items before being delivered and is used to hide the correspondences between the actual sender and recipient of a mail item. If user A wants to send a message to user B, she has to create a packet by first selecting a mix (or a number of mixes) and then seal the message with the public key of B, the public key of the last mix (the mix that will deliver the message to user B), the public key of the second to last mix and so forth. Furthermore, address information about the next destination of the packet has to be included as well. When user A finishes creating the packet, she sends it to the first mix, which will decrypt the packet using her private key, extract the next destination address and forward the rest of the packet. The proposed system also included the following features: thwarting statistical attacks by uniform padding of messages as well as forwarding messages only in bulk, including an untraceable return address in a message, and verified digital pseudonyms in conjunction with a voting mechanism.

Dingledine et al. [5] implemented Tor, a low latency onion routing service based on a network of onion routers (OR). The function of these ORs can be compared to that of mixes described in [4]. Clients use a locally installed onion proxy (OP) software to access directory information of the network, build circuits using a set of ORs and manage connections from user applications. Directory information is required by the user's OP to retrieve state information of the currently registered ORs and then choose suitable ORs for building a circuit. A concrete circuit is then determined and built incrementally by the users OP, which negotiates symmetric keys with each of the ORs used in the circuit. After a Circuit has been established, it can be used for multiple TCP stream relays across the *onion network*. By routing arbitrary TCP traffic through the *onion network*, the user can access resources

on the web anonymously, as each OR along the way only knows about his immediate predecessor and successor.

MorphMix [9] is a peer-to-peer (P2P) based mix network for anonymous internet usage. It builds on the system described by Chaum [4] with the distinction that each node of the network can be a user and a mix at the same time. The authors argue that the P2P approach has an inherent advantage over traditional mix networks concerning traffic analysis attacks, even without using techniques such as cover traffic[1]. Another difference, when compared to other mix network implementations such as Tor [5], is the fact that the user initiating a tunnel (a tunnel is comparable with circuits described in [5] and mix cascades described in [4]) only chooses the first node of the tunnel. Each additional node is then selected by the previous node in the tunnel. Furthermore, emphasis is put on the detection of subnetworks of malicious nodes, which try to gain information about users in the network.

While the mix network implementations discussed so far were all implemented as stand alone applications relaying TCP streams, this work focuses on using a browser based P2P standard called WebRTC [2] for communicating between nodes. WebRTC allows for direct browser-to-browser communication using its JavaScript API and enables applications using specific APIs the exchange of video, audio, and generic data. Once a WebRTC connection is established, no intermediary servers are required. Means are provided to hole-punch Network-Address-Translation (NAT) gateways to even enable a direct connection for nodes behind NATs. Only as a wors-case fallback solution external relay servers are used. The following is a selection of works that already used WebRTC for the creation of P2P networks.

Vogt et al. [10] implemented a P2P mesh using WebRTCs RTCDataChannel API. This P2P network acts as a means of content sharing between all registered clients. Clients can join this mesh using a bootstrap server, which helps the client in establishing a direct connection to one of the peers.

Bevilacqua et al. [3] created ufo.js, a browser oriented P2P network using WebRTCs RTCDataChannel API which allows connected nodes to exchange either binary or string data. The network is composed of nodes, supernodes, and (optionally) a server. Nodes are the basic component of the network and represent a webpage executing the ufo.js library in a browser, while being connected to a number of other nodes. Supernodes are nodes that published their public address and thus create an entry point for new nodes wanting to join the network. The optional server component holds a list of supernodes and thus simplifies their access.

Webtorrent [1] is a streaming torrent client usable in the Browser. It is completely implemented in JavaScript and uses WebRTC's RTCDataChannel API for the P2P part of the BitTorrent protocol.

All of them [1, 3, 10] use WebRTC's datachannel API to set up P2P overlay networks. One shortcoming of the WebRTC technology in terms of usability is the mandatory requirement to have a separate channel to exchange handshake and connection information. All of the described frameworks use a central or semi-central entity to keep track of all or parts of the connected nodes and assists them in establishing a direct connection.

[1]Creating dummy traffic between nodes to cloud actual transmitted data in a steady stream of similar looking data.

III. Architecture

The goal of this work is to achieve anonymity within a network of communicating parties. In this context anonymity is defined by obscuring the user by hiding its IP address. To this end, the main focus is to enable anonymous web communication. This is done by introducing a so called *onion network*. The *onion network* provides anonymity for each client within the network. Providing the anonymity is achieved by building so called onion chains. An onion chain consists of a minimum of three nodes, whereas each node shares a dedicated communication key with the client. Since the nodes are structured in a chain, each node only possesses information about its predecessor and successor, consequently, no node in the network has further knowledge about the other communicating parties.

Initially, each onion chain is established during the chain set-up. In the first step of the chain set-up the client chooses a fixed amount of nodes from a public node list, which contains currently available nodes and their public keys. Subsequently, the client uses the public key information of the first node, which will be further called *entry node*, to invite it into the chain. One step of this invite is to establish a shared key between the client and node, which will be subsequently used for communication within the onion chain. The invitation for the second node (*intermediate node*) is again encrypted with the corresponding public key information and sent to the entry node, which subsequently forwards the invitation to the intermediate node. Analogously, this process is repeated for each further node in the chain. Furthermore, the last node in a chain is called *exit node*. In this way, each node only communicates with its chain neighbors[2], which is crucial to guarantee anonymity.

Figure 1 illustrates how an onion message is sent through the network. The client encrypts the message content for each node with their shared key. Subsequently, whenever a message is sent from the client to the *exit node*, each node unwraps the outermost layer of the encrypted message, by decrypting the message payload. As a consequence, the *exit node* is able to access the plain content of the message after decryption. Analogously, each node wraps the response by encrypting the message. Since each node maintains its own key with the client, the content of the packet differs between any two nodes in the network. On arrival at the client, the response needs to be decrypted for each node in the onion chain.

In order to divide the protocol complexity into separate parts a protocol stack is introduced. The protocol stack consists of three main layers, namely, communication layer, onion layer and end-to-end communication layer. The communication layer is responsible for the transport of data between two entities. Since the protocol is designed to demand minimal trust on external servers, this is done via a peer-to-peer channel. To this end, introducing a peer-to-peer communication channel over the internet is necessary. Nevertheless, there is a need for a central registration point of available nodes in the network.

[2]The predecessor and successor within the chain

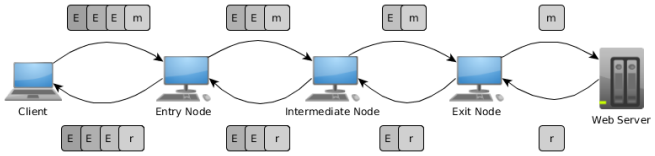

Fig. 1. Flow of onion messages sent through the chain. Message m is sent from the client to the exit node encapsulated in layers of encryption. Response r is sent from exit node to the client, whereas each node adds an encryption layer.

The registration point is called *directory server* and maintains the necessary address information, which enables peer-to-peer connection establishment between nodes.

The onion layer is responsible to manage the onion chain. Since nodes within an onion chain are connected through peer-to-peer channels, it is crucial to handle disconnections of individual nodes. To this end, whenever a node disconnects from the network it is necessary to re-build the chain. This needs to be done fully automatically and transparent to the user. Furthermore, each node can be part of multiple chains and is responsible for encrypting or decrypting each received message of each chain. Therefore, it is crucial to consider performance drawbacks. To this end, encryption and decryption should be done in parallel.

With the help of the former two layers, an encrypted and anonymous communication channel to the designated exit node is provided. To link two onion chains and two exit nodes respectively together, the end-to-end communication layer is introduced as illustrated in Figure 2. By connecting two onion chains, it allows to anonymously connect two clients through the *onion network*. Consequently, such a connection enables the clients to anonymously exchange data. Each encrypted end-to-end packet is transferred as onion layer packet payload. The content of the end-to-end communication is transferred between the two exit nodes of the individual chains. Therefore, it is necessary to apply additional encryption in the end-to-end communication layer, since otherwise the content would be exchanged in plain between the exit nodes.

Due to the high complexity of the protocol it is crucial to provide an easy to use interface to the user. Furthermore, accessing the network should be independent of the underlying platform. This consequently allows to anonymously send data on desktops as well as mobile devices. In addition, installation of the required end-user applications should be as easy as possible.

IV. Implementation

As part of this paper a proof-of-concept library was developed. The library offers anonymous AJAX[3] requests as well as downloading files anonymously. Furthermore, the library enables anonymous data exchange of two clients via end to end communication. All the core functionalities, such as chain establishment is done transparently to the user of the library. After evaluating the requirements described in the previous section in terms of usability and cross-platform applicability, we concluded to go for a web-based solution, relying on

[3]Asynchronous JavaScript And XML

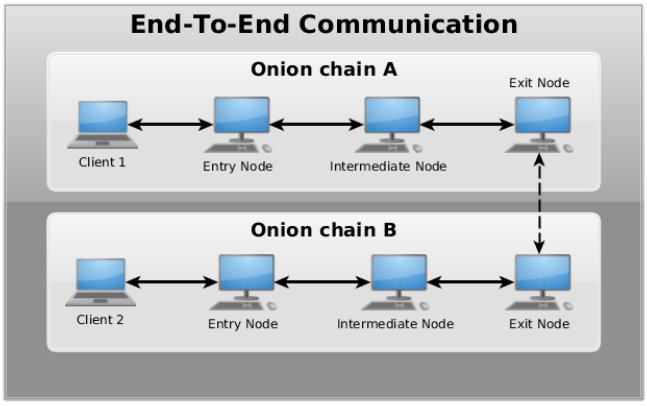

Fig. 2. End-to-End communication overview of participating entities, communication paths and high level layer abstraction

well-established and state-of-the-art web-technologies. Due to various restrictions on web-applications, intermediary- and exit-nodes run inside a chrome browser extension, whereas clients can access the onion network without the need of any software installation.

For applying cryptographic primitives, the browser built-in Web Cryptography API[4](also known as WebCrypto API) is used. The WebCrypto API provides a standardized interface to apply cryptography in web-applications, without requiring additional JavaScript libraries. The WebCrypto API is supported on all major browsers including recent versions of mobile browsers. In addition we use Web Workers[5] to provide parallel encryption and decryption functionality are employed. Web Workers are a browser built-in feature, which allow to asynchronously perform computations without blocking the main threat. In our case, each encryption or decryption operation is handled in a separate Web Worker.

A. Communication Layer

The communication layer is responsible for the transport of data between peers in the network. Therefore, this layer has to establish a connection between two peers as well as handle the connection state. To directly transport the data from one browser to another the WebRTC API is used. WebRTC is a new and still emerging technology, nevertheless it is currently supported by various different browser, including mobile browsers. Although the WebRTC technology mainly utilizes the unreliable UDP protocol, there are measures in place to establish a reliable connection, based on the Stream Control Transmission Protocol (SCTP). Furthermore, Datagram Transport Layer Security (DTLS), which is based on Transport Layer Security (TLS), is used to ensure an encrypted communication via WebRTC. However, as each peer only has a self signed certificate, authentication is not provided.

Signaling: The WebRTC API needs an external channel to establish the connection between two peers. Therefore, a new peer initially connects to the *signaling server* that is part of this paper. To this end, Websockets[6] are used to establish

[4]http://www.w3.org/TR/WebCryptoAPI/
[5]http://www.w3.org/TR/workers/
[6]http://www.w3.org/TR/2011/WD-websockets-20110419/

a bidirectional connection with the *signalling server*. The *signaling server* responds with the IP address and port of the peer and saves this information on the server as well as keeping the Websocket connection alive. If another peer wants to establish a connection it uses his own Websocket connection to send his WebRTC session description to the *signaling server*, which forwards the data to the other peer. This initiates a connection establishment procedure where public address information is exchanged, and obstacles like network address translation (NAT) gateways are circumvented.

Datachannel Communication: After the peers exchanged the needed settings with the help of the *signaling server* they can open the WebRTC RTCDataChannel. This data channel API provides send and retrieve functions that are used to transport the chunks of data. As of now the WebRTC implementations in browsers have a limited amount of data that can be sent in one message. Therefore, the communication layer has to split the data into chunks smaller than the maximum transport capacity. These chunks need to be concatenated on the receiving end of the data channel before the data is passed on to the layers above. To prevent statistical attacks on the connection each chunk is padded with random data to have the same size. Moreover, the current browser implementations of WebRTC in Firefox and Chrome have a limited amount of data that can be buffered before sending. Therefore, the data channel has to be queried before sending a chunk of data to make sure that the buffer is not full. If the buffered amount is over a certain threshold, sending of the chunk of data is delayed to allow the data channel to complete buffered operations. If the data channel is closed by the other peer or an error occurred, the connection is terminated and the layers above are notified that the connection was closed.

B. Onion Layer

If a node wants to join the *onion network* it has to register at the *directory server*. Consequently, the *directory server* maintains a list of available nodes, which can be retrieved by all clients. As a result, clients can use the node list in order to build a chain. This is illustrated in Figure 3. During registration, each node needs to provide a public key. Consequently, the *directory server* uses the socket information[7] of the registering node and maps it to the provided public key. Therefore, each entry consists of socket information and a public key. In addition, the *directory server* establishes a session identifier during registration. To counterbalance accumulation of disconnected nodes, the *directory server* expects regular heartbeat messages from each node. If a node fails to provide heartbeats within a certain time period, it is removed from the *onion network*.

Chain build-up: Initially, each node generates a session based unique RSA-Key pair. This key pair serves as the basis of cryptographic security for the node within the *onion network*. To this end, the public key is sent to the *directory server* at registration. The *directory server* distributes the public keys of nodes to a client during onion chain establishment. The client starts to build an onion chain by randomly selecting a minimum of three random nodes from the node list retrieved from the *directory server*. To this end, a minimum size of the node list is required in order to prevent the risk of

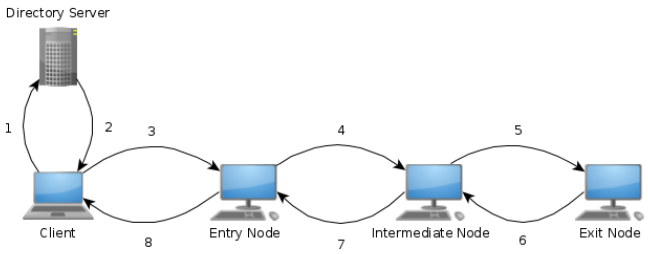

Fig. 3. Sequence of onion chain establishment. Clients can retrieve a node list from the *directory server*. Subsequently, the client randomly selects three nodes from the list, using them as chain nodes.

compromised nodes. For each selected node within the chain a dedicated AES-Key is generated and subsequently wrapped with the public RSA-Key of the node. Furthermore, a random 128-bit chain identifier is generated for each node separately, which enables each node to identify the chain during communication. During the chain set-up process the client encrypts the generated chain identifier of each node with the AES-Key and subsequently concatenates the encrypted data with the wrapped AES-key plus the public socket information of the next node within the chain. This information is encrypted in layers in succession of the nodes within the chain. Each node maps its own chain identifier to the information required for chain communication. This information includes a sequence number for each connection. The sequence number is initialized to zero during build up. To prevent duplicated sequence numbers due to race conditions. There are dedicated sequence numbers for outgoing and incoming messages.

Sending chain identifiers or sequence numbers in plain format would enable adversaries to deploy statistical attacks by mapping messages to the specific chains. Therefore, chain identifiers are concatenated with the sequence number and subsequently hashed.

$$SecureIdentifier = H(identifier||sequencenumber)$$

As a result adversaries are not able to map messages to a specific chain. However, this implies that each node needs to precompute the hash of the chain identifier concatenated with the next expected sequence number for each message.

For the sake of preventing replay attacks a random 128 bit nonce is generated on the client and exit node respectively during build up. The nonces are verified at the end of the chain build up and will be sent along each message on the onion layer. This kind of replay attack is covered in Section V-B. In the interest of facilitating communication between separate onion chains, exit nodes need to provide means to enable nodes outside of the chain to anonymously communicate with the chain. This is done by generating a *public chain identifier* during build up. The reason for the *public chain identifier* is discussed in section V-B.

Node Communication: On the client-side, each message is encrypted for each node in the chain using the shared keys[8]. Each encryption layer is subsequently removed from the message on each node by decrypting the payload. Once the

[7] Socket information consists of IP address and port

[8] AES-GCM[6] is used for symmetric encryption

message is received by the exit node, it is decrypted and the original content can used by the exit node to perform specific operations according to the content. For decryption every node needs an initialization vector (IV). Each node in the chain has access to the IV for the next node after decryption, so it can be sent in plain together with the message.
Example layered encryption for a chain with three nodes:

$$ENC = (IV_1, E_{k_1}(IV_2, E_{k_2}(IV_3, E_{k_3}(msg, nonce))))$$

As already mentioned, the above nonce is received from the exit node in the build-up process and sent along each message through the chain.

In order to increase performance, encryption and decryption is done in parallel by using Web Worker threads. As a consequence, ordering of packets is no longer assured, which means that outgoing encrypted packets are not necessarily ordered in the same sequence as they were upon receipt. Therefore, each message is assigned with a specific message identifier, which allows to map a received message response on the client-side to the corresponding request.

Since the *Secure identifier* requires the precomputation of the next expected sequence number it is necessary to introduce a synchronization point for received messages. Message queuing presents this necessary synchronization point, whereupon *Secure identifiers* of received messages are checked sequentially. After a *secure identifier* of a message is verified, the next expected *secure identifier* is precomputed, which in turn is compared to the next message in the message queue.

In order to prevent an attacker from replacing the encrypted payload of an onion message it is necessary to cryptographically link the encrypted payload with the sequence number and chain identifier. Unfortunately, since message encryption is done in parallel, the sequence number can not be used as additional data, since it is not known during encryption time. Therefore, a message authentication code is computed after the encryption. This is done by hashing the concatenation of the sequence number, chain identifier and the encrypted payload.

$$MAC = H(sequence\,number \,||\, chain\,id \,||\, enc)$$

The computed hash is appended to the message and verified by the receiver.

Communication with third parties: Depending on the message type, exit nodes communicate with third party entities in respect to the onion chain. This includes communication with remote web servers when handling anonymous AJAX (*AAJAX*) messages. Upon receipt of an *AAJAX* message the exit node decrypts the content and subsequently is able to access the contained jQuery AJAX request configuration. This configuration is used to create a jQuery AJAX request to the corresponding web server. The corresponding response is consequently encrypted and send over the chain to the client.

In addition, when an exit node receives a end-to-end connection message it uses the contained socket and public chain information to create a WebRTC connection to a remote exit node. This is done by initiating the connection establishment of the WebRTC layer. Since the remote exit node is part of another chain it sends the data content of the message to the client of the remote chain.

Failure Recovery: Operating the onion chain within a browser extension implies frequent disconnections of chain nodes. As a consequence, sophisticated failure recovery of onion chains is essential. To this end, onion chains utilize the message type *error*. Since error messages are created under various circumstances different error types are introduced. These are *buildError*, *chainError*, *messageError* and *nodeError*. Errors that occur during onion chain build up are of type *buildError*. This error type is potentially recoverable, therefore, chain build up is attempted repeatedly in increasing time intervals. Errors of type *chainError* are critical errors, which imply that it is not possible to create an onion chain. Errors that occur with individual onion messages are of type *messageError* and enable to individually re-send faulty messages. *NodeErrors* are triggered whenever a node is no longer responsive. This type of error can occur frequently and requires a rebuild of the onion chain. To this end, rebuilding an onion resets current chain information and invokes a new build up.

Tear Down: Clients can intentionally close onion chains by sending a *close* message through the chain. Consequently, each node in the chain clears chain specific information upon receiving a *close* message.

C. End-to-End Communication

Figure 2 illustrates the structure of two connected chains. In the last section the protocol for communication between clients, building an onion chain, was described (*intra-chain communication*). In contrast, this section covers the communication between individual chains (*inter-chain communication*). When clients are willing to directly communicate with each other, their onion chains are connected, this is called *end-to-end communication*. Since the end-to-end communication protocol is separated from the onion protocol, several sequences like connection establishment and properties like failure recovery have to be described for this abstraction layer as well. Since the data which is exchanged between the exit nodes is not protected on onion layer level, the communication has to be encrypted on the clients. The message meta data is asymmetrically encrypted with RSA keys. Those keys have to be exchanged by the client via a secure channel outside of the onion layer. The exchanged messages itself are AES encrypted.

Connection Establishment: The build up process has to be supplied with a public key and the public chain information from the remote chain. In order to enable the clients to map messages to specific *end-to-end communications* a *connection identifier* is introduced. To build up a new connection with a remote client, the following information is sent over the onion chain:

$$E_{K_s}(K_{pubA}, S_A, CHID_A, CID), E_{K_{pubB}}(K_s), CHID_B, S_B$$

Where K_s denotes the generated symmetric key, which is wrapped using the asymmetric public key of the remote client. Furthermore, the symmetric key is used to encrypt the local connection data. This includes the asymmetric public key of the local client K_{pubA}, the connection identifier CID, the public socket information S_A and the public chain information $CHID_A$ of the local chain. In addition, the message includes the remote public chain information, consisting of the public chain identifier $CHID_B$ and the public socket information S_B.

When the exit node of the local chain receives the end-to-end communication message, it forwards the encrypted message to the given socket with the *public chain identifier* of the remote client. Once the message was sent through the remote chain and received by the client, the symmetric key is unwrapped using the corresponding private key. After the encrypted payload is decrypted using the symmetric key, the connection data[9] is stored. As a last step, the remote client uses the public chain information to reply with a newly generated nonce. The reason for this nonce is discussed in Section V-B. When the response is received by the local exit node, it extracts the *public chain identifier* in order to forward the message back to the client on the onion layer.

Communication: Data sent via end-to-end connections is encrypted using the symmetric key, which was shared between the clients during build up. The connection data of a received end-to-end connection message is decrypted using the private asymmetric key. Consequently, the connection data can be used to access the corresponding symmetric key, which enables to decrypt the message payload. Since messages sent through the onion chain possibly arrive in random order, a sequence number is introduced. The sequence number allows to reorder received messages. Each end-to-end connection message contains connection specific data, which is encrypted with the public key of the remote client. The connection specific data consists of the *sequence number*, the *connection identifier* and the *message type*. Since the sequence number is part of the connection data, the connection data is different for each message. Because *end-to-end connections* provide reliable data transmissions, it is necessary to introduce acknowledge (ACK) messages. Each end-to-end connection message is answered with an ACK message. Unless the ACK message is received within a certain time period the message is considered lost and is resent. The *message type* describes, whether the message is a content message or an acknowledge (ACK) message. Messages which are exchanged between two exit nodes are not encrypted on the onion layer and therefore do not contain onion layer padding. To prevent an attacker from observing protocol ACKs[10], those messages are padded to a random size.

Failure Recovery: The responsibility of the failure recovery on the onion layer is to create a new onion chain, when one of the nodes in the current chain no longer responds. Since the *end-to-end communication* involves two independent chains, update messages have to be exchanged when one of those chains are rebuilt. If the client of a chain is notified that its chain was rebuilt, it sends a message to the remote client with the public chain information. Therefore the remote client is able to update its connection information and resend all unacknowledged messages. In this case, the failure recovery is fully transparent to the application using the *end-to-end communication*. If a node disconnects from the *onion network*, it may as well happen that it was part of both of the communicating chains. If this is the case, the *end-to-end communication* is not able to recover the connection and therefore notifies the application. The application then has to recover the connection on its own.

[9]Public key, socket information, chain and connection identifier of the remote client

[10]ACK messages always have the same size, and therefore can be traced

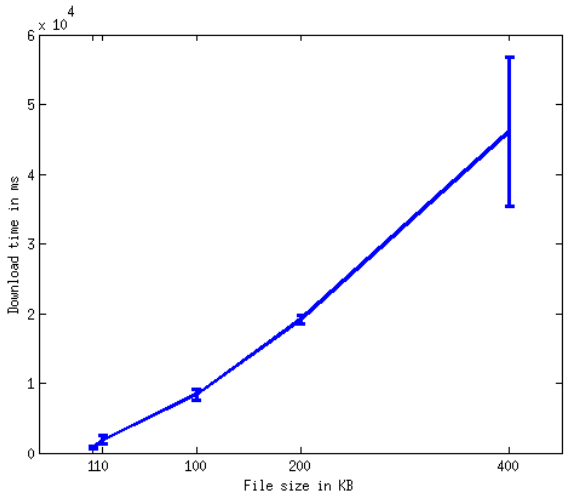

Fig. 4. Performance evaluation using file downloads

V. Evaluation

A. Performance Analysis

To evaluate the extent of performance deprivation caused by routing data through the *onion network* instead of accessing it directly, a set of benchmarks were performed. All tests were performed using the following hardware setup:

- A Lenovo Carbon X1 2nd Gen,CPU i7-4550U 1.50GHz, OS: Arch Linux, Browser: Google Chrome (Version 39.0.2171.95 64-bit)
- A Raspberry Pi hosting the WebRTC *Signaling Server*, Average ping response time: 25.446ms
- A Debian virtual machine, CPU 2,4GHZ, hosting the onion *directory server*, Average ping response time: 49.119ms

1) File download: For this purpose, the AAJAX file download functionality was used to retrieve files of various sizes through the *onion network* and then compared to retrieving the file using the browsers built-in AJAX functionality. Files of sizes between 1 and 400 kilobytes were downloaded five times, the results are depicted in figure 4. The figure displays only the results for retrieving the files using the *onion network*. Direct download of files resulted in an average download time of 63 ms with only $<$1ms deviation for different file sizes. Thus it seems that the chunking, which is required for the transmission of large files through WebRTC, combined with the de- and encryption operations of those chunks leads to a rather significant performance loss for large files.

2) End-to-End latency: This test was performed using a total of four browser tabs, each hosting the library. Two tabs were used as clients, each building up a chain using the other 3 tabs. The other two tabs were used as intermediary nodes. Client A then proceeded to send a text message to client B, who again replied with a text message. This process was repeated 100 times, with measurement starting when client A starts sending a message (before encryption) and ending when client A received the response from client B (after decryption).

	Idle	Executing
Browser	2.22%	14.84%
Tab 1	0.64%	26.83%
Tab 2	0.64%	18.15%
Tab 3	0.77%	39.05%
Tab 4	0.72%	15.80%

TABLE I. AVERAGE CPU LOAD FOR BROWSER PROCESSES IN VARIOUS STATES.

The mean duration was 1204.1 ms with a standard deviation of 213.8594.

3) CPU load: In conjunction with the measurements performed in section V-A2 the CPU load of the four tabs as well as the browser process was measured. This was done using the "pidstat" tool[11].The process state was measured each second for a total of 100 seconds. Measurements were taken in two states:

- **Idle**: All tabs loaded the library and a successful connection between the chains of client A and B was established, however no data was currently being transmitted.
- **Executing**: The performance test described in the previous section V-A2 is being executed. Data is constantly transmitted between client A and B.

The final results are listed in Table I. To put this into perspective, playing a 360p video on YouTube resulted in 10.94% CPU load for the tab and 4.91% CPU load for the browser process.

B. Security Analysis

Before declaring the various attack scenarios, the power of an attacker is defined.

The WebRTC channels between nodes are encrypted via Datagram Transport Layer Security (DTLS) with self signed certificates. The fingerprints of the self-signed certificates are exchanged via the signalling channel. That means, an attacker that is not able to manipulate the channel, can not eavesdrop the communication. In this work, a potential attacker is assumed to be able to deploy man in the middle attacks on WebRTC connections and therefore, those connections are considered to be unprotected.

Packet Tracing: The level of anonymity in the network depends on hiding the chain of a given origin node. If an attacker is able to trace a packet though the network by observing some unique property, the anonymity of the network is broken.

In this work, an attacker is assumed to be able to read the content of all node connections at the same time. This means that the size of each packet is known and could be used to trace packets through the network. Even if the size of packets varies, while traveling through the network, statistical attacks could be mounted to find different traces with certain possibilities.

Observing the size of encrypted messages sent through the *onion network* can be used by adversaries to deploy statistical attacks, and consequently detect onion chains, which would destroy anonymity. To this end, messages are split into fixed size message chunks. If a message part is smaller than the fixed size, random data padding is added. Packets on the WebRTC layer are chunked to a fixed size, but they contain plain meta data with instructions on how those chunks should be assembled back into onion layer messages. This chunking was implemented to circumvent technical limitations of the WebRTC's data channel, but it does not add any security since attackers can simply read the assembly information to learn the original packet sizes. As a consequence, a chunking on the onion layer is introduced in this work. Since this is done prior encryption, no size information can be retrieved by an attacker. In Figure 5 the preprocessing of messages on the onion layer is illustrated. Upon receipt, message chunks are stored in a buffer, whereas the buffer of the message is flushed once all chunks are received.

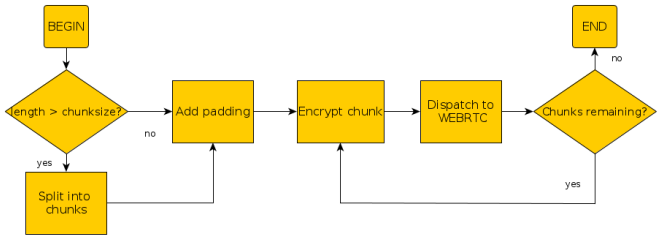

Fig. 5. Message preprocessing on the onion layer

The *secure identifier* from Section IV-B, which is used to identify packets in order to find the corresponding AES key for decryption, is the result of hashing a unique identifier together with a sequence number. As a result, this identifier is guaranteed to change at every hop in the network and is therefore not traceable.

The communication between two exit nodes in a end-to-end connection can be observed as such because those messages have not the same shape as onion layer messages. As a result, an attacker is able to gather the information which nodes in the network are (also) exit nodes and which of them communicate with each other. Since exit nodes of chains are public knowledge anyway, only the information about which exit node communicates with each other is relevant. However, no process could be found to turn this information into a risk, harming the anonymity of the system.

If a node is compromised, only information about the neighbors is available. In addition, only an exit node is aware of his role in the chain. As a result, a node can not distinguish of being an entry or intermediate node. In order to break anonymity, it is necessary to control each node in the chain.

Replay Attacks: For the sake of preventing replay attacks, each connection maintains its own sequence number, which is concatenated and hashed with the chain identifier. Consequently, repeatedly sending an individual message of an established onion chain with the identical *secure identifier* is not possible.

Furthermore, to circumvent replay attacks on a session

[11] http://pagesperso-orange.fr/sebastien.godard/

basis on the onion layer[12], nonces are exchanged between client and exit node during chain establishment. The initial packet of the client to the exit node contains a nonce, which is sent back along another nonce generated by the exit node to the client. Subsequently, each following packet sent by the client contains the nonce generated by the exit node and is verified by the exit node.

The same principles are used for end-to-end connections. Nonces are exchanged during connection establishment and verified for subsequent messages. In addition, since each message contains an encrypted sequence number, messages containing a sequence number previously received are ignored.

Data Manipulation: Since WebRTC communication is considered to be insecure, the protocol is designed to counterbalance any kind of data manipulation. Manipulating bytes in the encrypted payload is detected by the authenticated decryption[13]. In addition, exchanging a payload with a previously sent payload is prevented by the appended MAC. Since the chain identifier is secret, it is unfeasible for an attacker to forge a new MAC. In general, manipulated messages are discarded by the receiving node.

Manipulating the *secure identifier* results in the receiving node not being able to find the corresponding AES key and consequently discarding the message.

End-to-end connections involve messages being exchanged between the two exit nodes. Similar to the onion layer, manipulating the encrypted payload is detected by the AES GCM algorithm. In contrast to the onion layer it is possible to use the sequence number as additional data in order to prevent exchanging a payload with a previously sent payload. The public information of an exit node is used to anonymously identify a chain by a remote connection. If the public information is manipulated while sending a message from exit node to exit node, the public information is either invalid or mapped to another chain with the same exit node. In case the public information is invalid the message is discarded. On the contrary, if the public information identifies another chain the message is sent through this chain. However, the client in any case discards this message, since neither AES key nor RSA key match with ones used for encrypting the message.

VI. Future Work and Conclusions

In this paper we presented a novel architecture with a focus on interoperability and usability, to enable an anonymous communication among different parties. Our proof-of-concept implementation shows the feasibility of the proposed architecture and is solely based on established web-standards and therefore can easily be used in cross-platform and web-application scenarios. The evaluation again highlights the protection of users anonymity even under the assumptions that particular components in the system cloud be compromised.

As the current work still depends on using servers to establish WebRTC connections or store the node information, it might be a good idea to get rid of these servers, because they are a single points of failure. Also, it is easier for an attacker to take over a server than the whole network. Therefore, it could be tried to use peers in the network as a distributed way to manage information about all registered nodes, which currently is the task of the *directory server*.

Another way to improve the work against attacks is to use multiple *directory servers* instead of only one. This helps if one of the *directory servers* is controlled by an attacker. The attacker then could not route all traffic to nodes under her control, as the node list would contain nodes of all different *directory servers*.

Also if the browser implementations of WebRTC gets improved and evolves, some limitations, such as message size, might be removed, the transfer of bigger data will get a lot faster, because chunking of the data will not be necessary.

[12] A onion layer session involves chain establishment and consequent messages

[13] AES GCM

References

[1] Feross Aboukhadijeh. Webtorrent - streaming torrent client for node and the browser. https://github.com/feross/webtorrent. Accessed: 2015-06-08.

[2] Cullen Jennings Adam Bergkvist, Daniel C. Burnett and Anant Narayanan. Webrtc 1.0: Real-time communication between browsers. https://w3c.github.io/webrtc-pc/. Accessed: 2015-06-32.

[3] A. Bevilacqua, P. Boemio, and S.P. Romano. Introducing ufo.js: A browser-oriented p2p network. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, pages 353–357, Feb 2014.

[4] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *COMMUNICATIONS OF THE ACM*, 24:84–88, 1981.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[6] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. *NIST Special Publication*, 2007.

[7] Peter Eckersley. How Unique Is Your Web Browser ? pages 1–18, 2010.

[8] Martin Mulazzani and Philipp Reschl. Fast and reliable browser identification with javascript engine fingerprinting. *Web 2.0 Workshop on ...*, 2013.

[9] Marc Rennhard and Bernhard Plattner. Introducing morphmix: Peer-to-peer based anonymous internet usage with collusion detection. In *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, WPES '02, pages 91–102, New York, NY, USA, 2002. ACM.

[10] C. Vogt, M.J. Werner, and T.C. Schmidt. Leveraging webrtc for p2p content distribution in web browsers. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–2, Oct 2013.

[11] Ting Fang Yen, Xin Huang, Fabian Monrose, and Michael K. Reiter. Browser fingerprinting from coarse traffic summaries: Techniques and implications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5587 LNCS:157–175, 2009.