Message-Layer Encryption in Ricochet

by Liam Kirsh

Computer Science Department
College of Engineering
California Polytechnic State University
2017

Date submitted: 06/07/17
Advisor: Dr. Bruce DeBruhl

Table of Contents

Background

Ricochet is an experimental anonymous peer-to-peer instant messaging client. It uses Tor Network hidden services to allow contacts to connect to each other without a centralized server. No personal information is exposed to eavesdroppers, and communication is authenticated and private.[1] The initial version of Ricochet was released in March 2014, and its source code is publicly available on GitHub.[2]

The formal Ricochet project goals are to implement a real-time messaging system with the following properties:

a) Users aren't personally identifiable by contacts or their address

b) Communication is authenticated and private

c) No person or server can access contact lists, message history, or other metadata

d) Resist censorship and monitoring at the local network level

e) Resist blacklisting or denial of service against users

f) Accessible and understandable for non-technical users

g) Reliability and interactivity comparable with traditional IM services

The Ricochet protocol is defined in three layers.[3] The first of these is the *connection layer*, which allows an anonymized TCP-style connection between peers. Next is the *packet layer*, which separates communications into a series of packets delivered to channels. Finally, the *channel layer* handles packets according to the channel type and channel state.

Upon initial launch, the Ricochet client establishes a Tor hidden service, generating a 1024-bit RSA keypair. The keypair is stored for permanence in a JSON configuration file. The Tor hidden service computes its *onion name* by performing a SHA-1 hash of the public key and truncating it to 80 bits, or 16 base32 characters. The service announces this 16-character address to a distributed hash table in the Tor network. To connect between clients, Tor hidden services allow for the creation of end-to-end connections, encrypted using a Diffie-Hellman handshake to provide forward secrecy. Connections are bidirectional and held open unless explicitly closed.[4]

A user (Alice) sends a contact request by entering her name, a message, and the recipient's (Bob's) ID into the Add Contact dialogue. After a connection is established from each user's client to the other's hidden service, Alice's client sends a sequence of bytes to indicate versions of the protocol her client supports. Bob's client responds to Alice's service with the highest mutual version supported or an error if no suitable version is found.

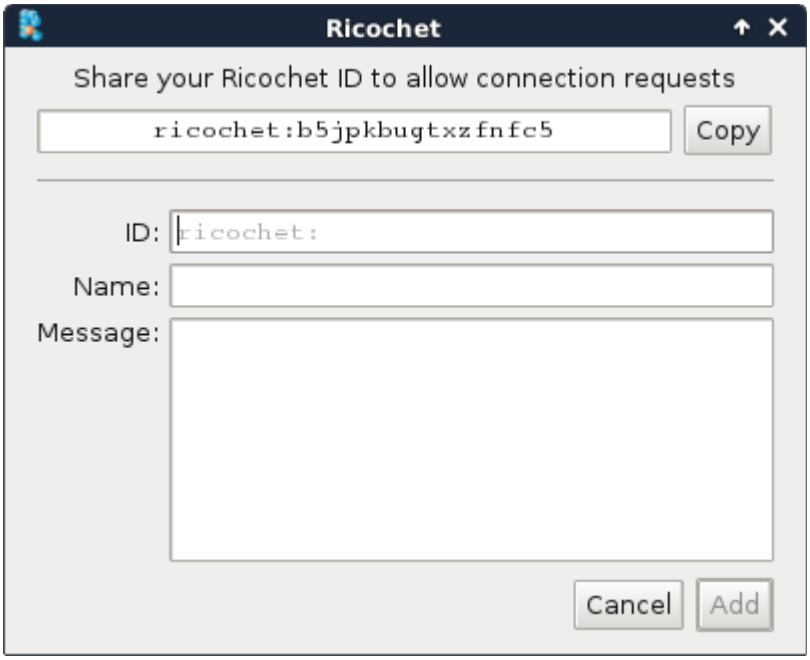

Figure 1: Ricochet's contact reguest dialogue.

Both Alice and Bob must send an *OpenChannel* message to the other user to create an instance of a channel. Channels only exist within a connection between peers. Distinct features have separate channel types, such as *im.ricochet.chat* and *im.ricochet.file-transfer*[1], and the receiving client of the *OpenChannel* message can choose to accept or reject the channel. Channel instances also provide a state for messages. For example, each file transferred would have its own channel, and the channel must be closed after transfer completion.

At first, the initiator of the contact request attempts to open a *im.ricochet.contact.request* channel to the recipient. A contact request channel is used by Alice to introduce herself to Bob and ask for further contact, including being added to Bob's contact list. In a contact request channel, *ContactRequest* and *Result* packets are sent. A *ContactRequest* packet can optionally include a nickname string and message string (Alice's server ID was transmitted in the initial connection). A *Result* packet, sent by Bob's client, responds to Alice's request with a single enumerated type: Pending, Accepted, Rejected, or Error.

1 Support for file transfers in Ricochet has not yet been added.

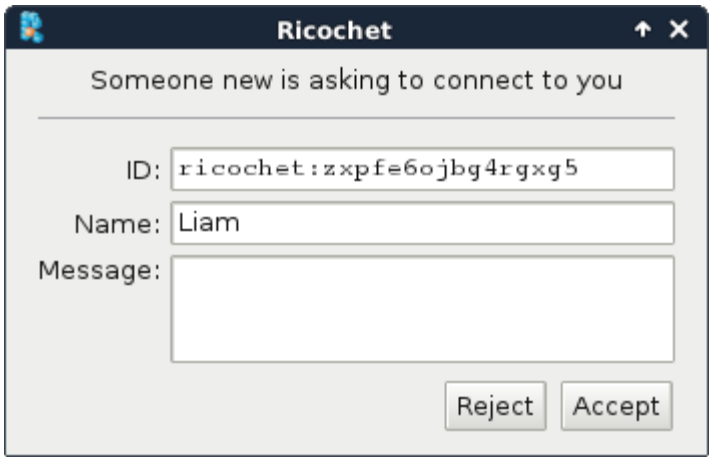

Figure 2: Contact request alert in Ricochet.

To send chat messages, each client creates a channel of type *im.ricochet.chat*. Only the initiator, or peer who created the channel, can send messages and receive acknowledgement on a channel; the opposing peer must create a chat channel to send its own messages. Within a chat channel, the initiator transmits *ChatMessage* packets. A *ChatMessage* contains a string representing the message text, an optional unsigned 32-bit message ID for acknowledgement, and an optional 64-bit *time_delta* value indicating the number of seconds between when the message was composed and when it is transmitted.

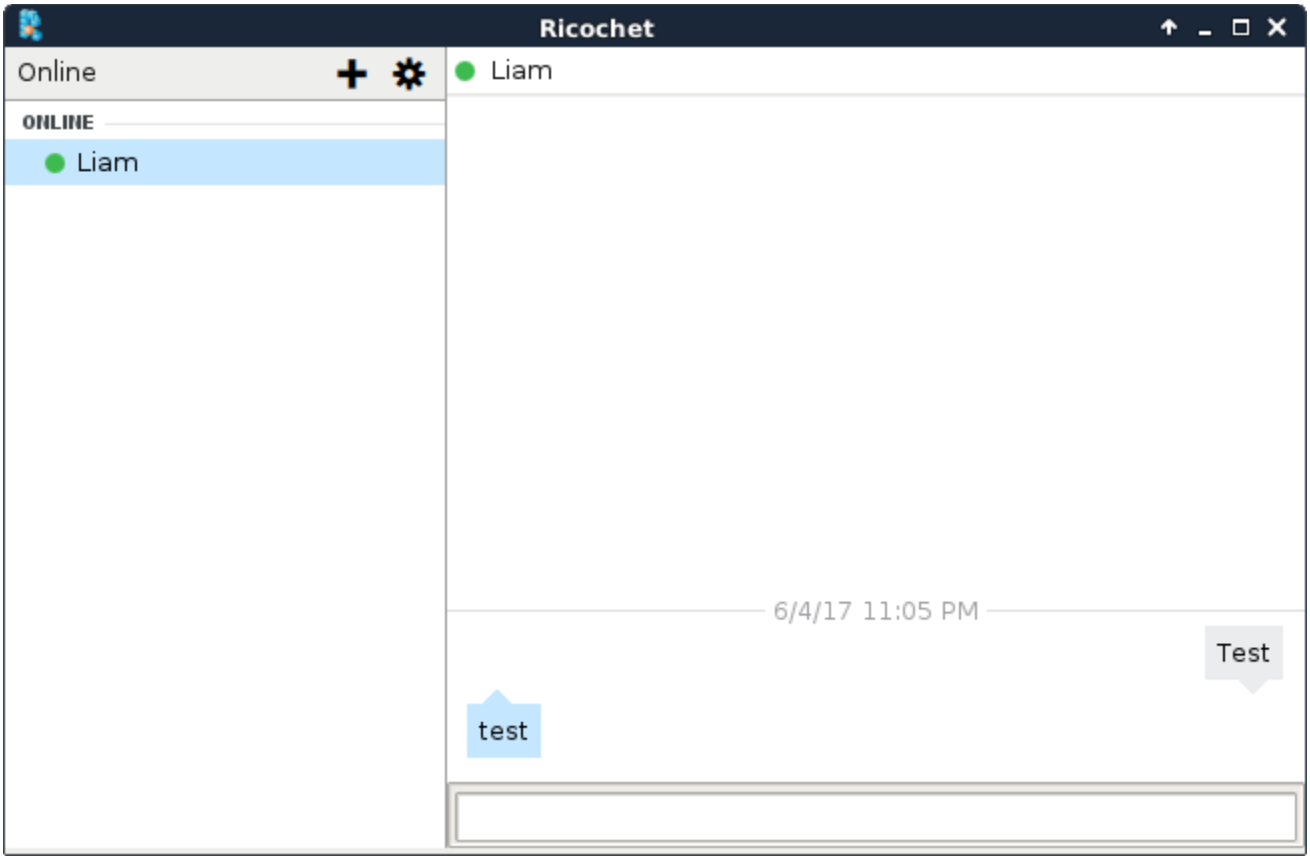

Figure 3: Sending and receiving messages in Ricochet.

Project Goals

My intention for this project was to implement encryption and authentication on Ricochet's message layer. This provides two major benefits. The first is a higher level of encryption in the event that a flaw is found in the existing channel-layer encryption, or if increases in computing power make attacks on that encryption feasible. Tor is used by the military, journalists, law enforcement, researchers, and activists, whose communications may be interesting to state-level adversaries and criminal organizations with access to great amount of computing resources.[5] Additionally, message-layer encryption would allow users to specify relay nodes that can temporarily store messages and forward them once the recipient comes online. Because the channel-layer encryption in hidden services only encrypts data between end users, the addition of authentication and encryption within messages would enable the application to conceal communication data from relay nodes. To preserve backward compatibility, the message-layer encryption would be opportunistic and only used when both contacts support it.

Stronger cryptography

Numerous groups have noted the potential weakness in relying on Tor hidden services for message encryption. The results of a security audit were published in February 2016 in which the information assurance firm NCC Group noted three cryptography-related vulnerabilities in Ricochet, two of which relate to the weak cryptography in Tor.[6]

The first relevant vulnerability, entitled “Host Verification Weak Against State Level Adversaries”, highlights the risk of impersonation of Ricochet users due to weak cryptography in Tor. As was discussed in the *Background* section of this document, Tor hidden service onion names consist of the leading 80 bits of a SHA-1 hash. To impersonate a Ricochet user, an attacker must perform a *preimage attack* on the victim's Ricochet ID. Although a preimage attack has not yet been successfully completed on SHA-1, NCC Group remarks that state-level adversaries may have better attacks than publicly known ones. They cite weaknesses in SHA-1's collision resistance to demonstrate it should not be considered cryptographically strong.

Ricochet's audit also notes a medium impact, low exploitability risk posed by the lack of application layer message encryption. NCC notes that “In order to maximize protection against state-level adversaries while waiting for the next generation of hidden/onion services protocol to address some of known short-comings, it may be beneficial to add application-layer encryption of messages as well.”

Support for relay nodes

Currently, if a message is sent to a known contact while the recipient is offline, the message is queued on the client until the recipient returns online. However, if the sender closes her client while the message is still queued, all messages are cleared. This is counterintuitive because it differs from the way that messages are handled in other

communication services when the recipient is offline. To solve this problem, in Facebook[2], SMS, and email, messages are stored in plaintext on a centralized server. Such an approach would violate goals *b-e* of the Ricochet project. Instead, this project aims to create a protocol under which users can designate a *Ricochet relay* client to temporarily store their messages while the recipient is offline. Since the relay is itself a hidden service, the channel-layer encryption in Tor encrypts data for the relay, not the final recipient. Implementing message-layer encryption would ensure that minimal data and metadata[3] is visible to relays and that the ability to authenticate messages is preserved.

It is important to distinguish between the relays defined by the Tor protocol that are used for communication between Tor hidden services (*Tor relays*) and *Ricochet relays* per the specifications in this paper. The current design of the Tor network declares that for communication from a client to a Tor hidden service, the client and server each must randomly choose three Tor relays on the network to create a circuit for anonymous communication. Data passes along the relays in the circuit in fixed-size cells which are immediately forwarded from one relay to the next. *Tor relays* do not allow the client to request that data be stored until the recipient fulfills certain conditions, and thus *Tor relays* are unsuitable for the previously described purposes without introducing storage commitments.

Implementation

Choice of cryptographic protocol

Three different cryptographic protocols were considered that could meet the goals of the project while enhancing security of relayed messages: the Signal Protocol, Off-the-Record Messaging, and static Diffie-Hellman. RSA was not evaluated, as it requires additional CPU consumption, memory usage, and key generation time relative to Diffie-Hellman.

The Signal Protocol provides asynchronous messaging as well as perfect forward secrecy, but relies on a centralized server and does not preserve anonymity. Using a *double ratchet* system, the application generates a new encryption key for each message. Unfortunately, *Extended Triple Diffie-Hellman* and the *double ratchet* algorithm dictate that upon registration, each user preemptively generates a set of public keys and sends them to a centralized server in order for contacts to generate each new encryption key. [7][8] Although a trusted *Ricochet relay* could double as a key server, a compromised server would leak the identity of the recipient when he publishes his keys, as well as the identity of the sender when she requests the recipient's keys.[9] This violates goals *c*, *d*, and *e* of the project.

2 Facebook offers a private messaging mode using the Signal Protocol.
3 In a simple Ricochet relay system, relays would know a message's timestamp and final recipient ID.

Off-the-Record Messaging (OTR) is a cryptographic protocol for instant messaging conversations that predates the Signal Protocol. Like Signal, OTR offers perfect forward secrecy using per-message encryption keys.[10] Instead of distributing per-message key data on a centralized server, each message includes a Diffie-Hellman (DH) public key that must be used to derive the key for subsequent messages. Thus, the protocol does not require a centralized server, but only maintains perfect forward secrecy synchronously – if Alice and Bob take turns sending messages. If Bob is offline or does not reply for a long time, Alice must remember the entire sequence of DH keys for her messages, since she cannot be sure which key the next message from Bob will be encrypted under. In addition to this vulnerability, the author of the Signal Protocol notes unnecessary complexity in OTR.[11]

Finally, static Diffie-Hellman (D-H) offers strong encryption of messages without the forward secrecy provided by OTR. Unlike the ephemeral Diffie-Hellman protocol used in OTR, static D-H dictates that the sender and recipient establish a long-term shared key. As long as each user exchanges her respective public key over an authenticated channel (the Tor hidden service connection), messages cannot be forged in a man-in-the-middle attack by a compromised *Ricochet relay*. Researchers recommend that elliptic curve Diffie-Hellman (ECDH) be used for greater protection against state-level actors.[12]

Although the Signal Protocol offers perfect forward secrecy, it requires centralized key distribution servers, which could allow an adversary with access to the server to determine sender-recipient patterns based on key retrieval. OTR offers advantages over static ECDH without the requirement of centralized servers, but the complexity of implementing it in Ricochet falls outside the scope of this project. Thus, for the purposes of this project, **static ECDH** was implemented as the message-layer cryptographic protocol in Ricochet for its strong encryption and minimal key generation and exchange overhead. Because some cryptographers distrust other curve constants[13], the **Curve25519** ECDH function has become the industry standard.[14]

GPGME cryptographic library

GPGME is a cryptographic library that provides a wrapper for the Gnu Privacy Guard (GPG) command-line interface. The development version of GPG, version 2.1.*, offers the option to generate and manage ECDH encryption keys, including Curve25519 ECDH keys.. This functionality can be controlled from the GPGME library.

Including the header file *gpgme.h* in Ricochet source files and compiling with the -lgpgme, -lassuan, and -lgpg-error compiler flags gives the developer access to the GPGME API. To initialize the library functionality and create a GPGME *context*, the developer calls a set of initialization functions as follows (error checking not shown):

```
gpgme_ctx_t context;
gpgme_check_version(NULL);
```

```
gpgme_set_locale(NULL, LC_CTYPE, setlocale(LC_CTYPE,
NULL));
gpgme_new(&context);
gpgme_set_protocol(context, GPGME_PROTOCOL_OpenPGP);
gpgme_ctx_set_engine_info(context,
GPGME_PROTOCOL_OpenPGP, “<GPG executable file path>”,
“<GPG configuration directory>”);
gpgme_set_armor(context, 1);
```

The *gpgme_ctx_set_engine_info* function specifies the file path of the gpg2 executable as well as the GPG configuration directory to be used. The gpg2 executable and necessary libraries would be distributed alongside a release of Ricochet. In order to segregate Ricochet keys from the user's personal GPG keychain, Ricochet would have its own GnuPG configuration directory to be used for this encryption protocol. *gpgme_set_armor* dictates that key data will be exported as ASCII text as opposed to binary data, which allows the data to be stored in a QString type in Ricochet.

The next step upon initial launch of the Ricochet application is to generate an ECDH keypair. The GPGME library can trigger GPG to generate a keypair and store it in the keychain using the following command:

```
gpgme_op_genkey(context, parms, NULL, NULL);
```

where parms is a string defining the key generation parameters:

- Key-Type: ecdh
  - Elliptic Curve Diffie-Hellman is used as the public key algorithm.
- Key-Curve: Curve25519
  - The industry standard elliptic curve.
- Key-Usage: encrypt
  - Because public keys are shared over an authenticated channel, signatures are not necessary.
- Name-Real: <Ricochet ID>
  - This field is used to associate keys with the sending Ricochet client. The Name-Real will be the sender's Ricochet ID.
- Expire-Date: 0
  - For the purposes of this project, static ECDH keys with no expiration date will be used; however, in the *Future Improvements* section of this document other approaches are discussed.

To export generated keys, the application must perform a search through the keychain for the key using a C-style string pattern (the user's Ricochet ID) and rewind the data buffer:

```
gpgme_data_t result = NULL;
```

```
gpgme_op_export(context, pattern, 0, result);
gpgme_data_rewind(result);
```

The key data can then be read into memory using the *gpgme_data_read* function. The application may import a contact's key into the GnuPG keychain using the *gpgme_op_import* function.

Modifications to the Ricochet client

To implement the ECDH key exchange in Ricochet, it was necessary to modify two object types: *ContactRequest* and *Response*. The *ContactRequest* packet is sent over the *im.ricochet.contact.request* channel as part of the *OpenChannel* message, while the *Response* is part of the ChannelResult message sent over the same channel. Both object types require the addition a single additional property, an optional string in which the publicKey is stored. The new types are identified as follows in red.

```
message ContactRequest {
    optional string nickname = 1;
    optional string message_text = 2;
    optional string publicKey = 3;
} // src/protocol/ContactRequestChannel.proto

message Response {
    enum Status {
        [...]
    }
    required Status status = 1;
    optional string publicKey = 2;
} // src/protocol/ContactRequestChannel.proto
```

These types are defined in the *ContactRequestChannel* protocol buffer. The *unique numbered tag* following each field identifies it in the *message binary format* used to encode the messages. After this, clients choose whether or not to use message-layer encryption based on the results of the version negotiation.

Messages can be encrypted by the GPGME library with a call to the *gpgme_op_encrypt* function.

Future Improvements

Use of the Signal Protocol in Ricochet

In a modified version of the Signal Protocol, end-user clients could be responsible for distributing a set of ephemeral keys to contacts periodically when both contacts are online. This would allow for perfect forward secrecy without the vulnerabilities detailed

in Off-the-Record Messaging in the *Implementation* section of this document. More research can be done as to the potential vulnerabilities created by relying on user clients to distribute their own keys.

Use of Off-the-Record Messaging in Ricochet

In a project of longer timespan, the OTR Protocol could replace ECDH in Ricochet. This would require the use of an additional configuration file or database to preserve sequences of DH keys for periods in which one user sends several messages to a contact without receiving a response. It might be necessary to place a limit on the number of messages that can be sent to a given recipient while the recipient is offline in order to preserve disk space.

Ephemerality in D-H

In the ECDH encryption protocol specified in this document, OpenPGP allows keys to have a set expiration date. To offer weak forward secrecy, Ricochet clients could limit the number of messages sent to a *Ricochet relay* while the true recipient is offline, requiring that a D-H exchange be performed directly between two contacts before continuing. Alternatively, to favor user friendliness over security, the D-H exchange could be performed at specified intervals when both parties are online.

Ricochet Relays

After an encryption protocol is implemented and the changes are accepted to the Ricochet project, one could begin implementing the Ricochet relay software. Standard Ricochet clients might have an additional configuration panel in the graphic user interface where users could specify the Ricochet ID of a trusted relay. If two contacts are named *Alice* and *Bob*, the relay could be named *Richard*. If Alice tries to send a message encrypted for Bob to him but he is offline, her software would send a message to Richard asking him to forward her message to Bob. Richard would run a modified version of Ricochet with a minimal interface, and his software would automatically forward any encrypted messages received to their designated recipient. If Richard's machine were compromised, the attacker would see only the metadata and final recipient of the messages stored on his machine.

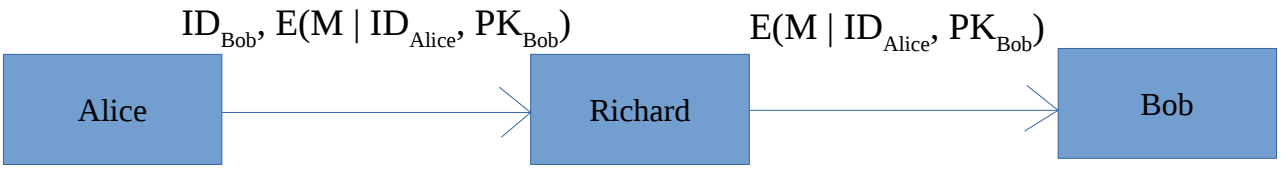

Figure 4: Alice sends her message to Richard, who forwards it to Bob once Bob is online.

References

[1] J. Brooks, “Technical design of Ricochet.” [Online]. https://github.com/ricochet-im/ricochet/blob/4294b6b2b21c907ba87041bcd9c2a1ddb6361080/doc/design.md

[2] “Github – ricochet-im/ricochet: Anonymous peer-to-peer instant messaging.” [Online]. https://github.com/ricochet-im/ricochet/

[3] “ricochet/protocol.md at master – ricochet-im/ricochet – GitHub.” [Online]. https://github.com/ricochet-im/ricochet/blob/4294b6b2b21c907ba87041bcd9c2a1ddb6361080/doc/protocol.md

[4] “Tor: Hidden Service Protocol.” The Tor Project. [Online]. https://www.torproject.org/docs/hidden-services.html.en

[5] “Why Use Tor?” The Tor Project. [Online]. https://www.torproject.org/about/torusers.html.en

[6] J. Hertz et al, “Ricochet Security Assessment.” NCC Group. Feb. 15, 2016. [Online]. https://ricochet.im/files/ricochet-ncc-audit-2016-01.pdf

[7] M. Marlinspike and T. Perrin, “The X3DH Key Agreement Protocol.” Open Whisper Systems. Nov. 4, 2016. [Online]. https://whispersystems.org/docs/specifications/x3dh/

[8] M. Marlinspike, “Forward Secrecy for Asynchronous Messages.” Open Whisper Systems Blog. Aug. 22, 2013. [Online]. https://whispersystems.org/blog/asynchronous-security/

[9] N. Unger et al, “SoK: Secure Messsaging.” 2015 IEEE Symposium on Security and Privacy. [Online]. http://www.ieee-security.org/TC/SP2015/papers-archived/6949a232.pdf

[10] N. Borisov et al, “Off-the-Record Communication, or, Why Not To Use PGP.” [Online]. https://otr.cypherpunks.ca/otr-wpes.pdf

[11] M. Marlinspike, “Simplifying OTR deniability.” Open Whisper Systems Blog. Jul. 27, 2013. [Online]. https://whispersystems.org/blog/simplifying-otr-deniability/

[12] D. Adrian et al, “Imperfect Forward Secrecy: How Diffie-Hellman Fails in

Practice.” [Online]. https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf

[13] B. Schneier, “The NSA Is Breaking Most Encryption on the Internet.” Schneier on Security. Sep. 5, 2013. [Online]. https://www.schneier.com/blog/archives/2013/09/the_nsa_is_brea.html

[14] “Things that use Curve25519.” IANIX. [Online]. https://ianix.com/pub/curve25519-deployment.html