

Secrecy and Authenticity Properties of the Lightning Network Protocol

Hans Hüttel^a and Vilim Staroveski

Department of Computer Science, Aalborg University, Denmark

Keywords: Cryptographic Protocols, Protocol Verification, Blockchain.

Abstract: The Lightning Network is a second layer protocol that sits on top of the Bitcoin cryptocurrency. It is a decentralized network of payment channels first conceptualized in 2014 and its first implementation was released in 2017. Being a fairly new technology, it may have security issues that we do not know of and the goal of this report is to analyse the Lightning Network to further investigate its security properties. The focus of this analysis is on answering whether the confidential data is kept secret and whether the user authenticity holds in the protocol. In the analysis we use the process algebra to formally describe cryptographic protocols that form the Lightning Network and an automatic cryptographic protocol analyser called ProVerif for their analysis.

1 INTRODUCTION

The Lightning Network is a decentralized network of bidirectional payment channels that sits on top of the Bitcoin blockchain. It was conceptualized in 2014 and a release candidate for version 1.0 of the Lightning protocol was released in December 2017. At the time of writing, the Lightning Network consists of about 8000 nodes, with about 37000 payment channels that altogether contain about 1000 bitcoins (worth about 8 million US dollars with the current exchange rate).

Unfortunately there is still little work done on analysing the security properties of the Lightning Network. Kiayias et al. (Kiayias and Litos, 2019) have studied the protocols from the point of view of composable protocols, but some aspects of the protocol have not been addressed, and the focus is on the properties of the cryptographic primitives involved.

One of the limitations of the Lightning Network is that users need to be online when receiving a payment. This could potentially expose their private keys, which would give an attacker full access to user funds. The secrecy of the user private key is therefore the main security property of the Lightning Network. Another important security property that the Lightning Network must satisfy is user authenticity. The protocol must be able to guarantee that if user *A* believes that is interacting with user *B*, then this is the case.

In this paper we use the automatic protocol verifier ProVerif (Blanchet et al., 2018) to give a fully

automated analysis of these security properties of the Lightning Network.


The Lightning Network is a collection of four cryptographic protocols, and each of these has to be described and analyzed separately.

The four subprotocols constituting the Lightning Network are: The *key agreement protocol* that establishes a secure and authenticated connection to another user, the *channel opening protocol* opening a payment channel to another user, the *onion routing protocol* for making a payment to a user and the *channel closing protocol* closing an existing payment channel. In this paper we are analysing these protocols with the exception of the channel closing protocol. Due to its simple behaviour (which simply is that a channel can be closed independently from the other participant), we omit its analysis in this paper.

The rest of our paper is organized as follows. In Section 2 we describe the analysis methodology that we follow, and in particular the Dolev-Yao attacker model and the applied π -calculus that form the basis of ProVerif. Section 3 contains the specifications of each of the subprotocols of the Lightning Network and the results of our analysis.

2 THE SYMBOLIC APPROACH TO SECURITY ANALYSIS

In this paper we use the symbolic approach to protocol analysis based on the Dolev-Yao assumptions (Dolev and Yao, 1981).

^a  <https://orcid.org/0000-0002-4603-5407>

2.1 The Dolev-Yao Model

The underlying assumption of the symbolic approach is that all messages exchanged in the protocol will be sent through public channels. These are assumed to be controlled by an attacker that can

- obtain any message passing through the network
- is a legitimate user of the network and can interact with other users
- will have an opportunity to be a receiver to any user

Moreover, an attacker will also be able to modify, delete and inject messages as well as use the user defined cryptographic functions for manipulating the obtained data (e.g. accessing the i th element in a tuple, constructing a tuple, decrypting a message, etc.). However, the attacker we are considering will not be able to perform any sort of cryptanalysis or any operation that would break the cryptography properties of the defined cryptographic functions (e.g. finding an inverse of an one-way hash function used in the protocol).

2.2 The Applied π -calculus

We use the applied π -calculus for modelling the cryptographic protocols. The choice of using a process calculus lends itself well to a protocol analysis that uses the Dolev-Yao-assumptions; as pointed out by Abadi and Gordon (Abadi and Gordon, 1999), we can then view any attacker as a process that interacts with the protocol that is the protocol itself. Our version of the applied π -calculus is that of (Blanchet, 2002), which is used in the ProVerif protocol analyzer described in Section 2.4.

We consider an infinite set of *names* (channels) \mathcal{N} and an infinite set of *variables* \mathcal{X} . We let a, b and c range over \mathcal{N} and x, y and z range over \mathcal{X} . We will use the letters M and N for *terms* and the letters P, Q and R for *processes*.

The syntax of processes is given in Table 1. The process $\mathbf{0}$ is the inactive process. The output process $\bar{a}\langle N \rangle.P$ outputs the term N on channel a and continues as P . The input process $a(x).P$ allows a term to be input on the channel a and binds the term to x within the continuation P . The process $P \mid Q$ denotes the parallel execution of processes P and Q . The *replication* process $!P$ denotes an infinite supply of copies of the process P . The *match* process **if** $M = N$ **then** P compares the terms M and N .

A name n occurring in a process P is called *free* if it is not bound by a restriction or an input. The set of free names in P is denoted by $\text{fn}(P)$.

Table 1: Formation rules for terms and processes in the applied π -calculus.

$$\begin{aligned} M, N &::= x, y, z \mid a, b, c \mid f(M_1, \dots, M_n) \\ P, Q &::= \mathbf{0} \mid \bar{a}\langle N \rangle.P \mid a(x).P \mid P \mid Q \mid !P \mid (\nu a)P \\ &\mid \mathbf{if} \ M = N \ \mathbf{then} \ P \mid \mathbf{let} \ x = g(M_1, \dots, M_n) \ \mathbf{in} \ P \end{aligned}$$

Table 2: Structural congruence rules.

$$\begin{aligned} P &\equiv P & P \mid \mathbf{0} &\equiv P \\ P \mid Q &\equiv Q \mid P & P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\ (\nu a)(\nu b)P &\equiv (\nu b)(\nu a)P & (\nu a)\mathbf{0} &\equiv \mathbf{0} \end{aligned}$$

$$(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q \quad \text{if } a \notin \text{fn}(P)$$

$$\frac{P \rightarrow Q}{P \equiv Q} \quad \frac{P \equiv Q}{Q \equiv P} \quad \frac{P \equiv Q \quad Q \equiv R}{P \equiv R}$$

$$\frac{P \equiv Q}{P \mid R \equiv Q \mid R} \quad \frac{P \equiv Q}{(\nu a)P \equiv (\nu a)Q}$$

In the original π -calculus, the only terms that can be transmitted along channels are names, but since the applied π -calculus is used for describing cryptographic protocols, we here allow a richer set of data terms. Terms are now built from a *signature*, a finite set of function symbols Σ that represent the cryptographic primitives. Some function symbols are *constructors* that are used for building terms (e.g. in encryption), while others are *destructors* used for taking terms apart (e.g. in decryption).

In Table 1, f represents a constructor, while g represents a destructor ($f, g \in \Sigma$). The evaluation of data terms is defined by a collection of term reduction rules that are specific to a given signature. We write $M \rightarrow M'$ if the term M evaluates to M' and $M \dashrightarrow$ if the term M cannot be evaluated.

The notion of *structural congruence* is used to identify processes that are identical up to structure. The rules defining the relation are presented in Table 2.

The semantics of the π -calculus is given by a *reduction* relation, \rightarrow defined inductively by a collection of reduction rules on closed processes, presented in Table 3. We write $P \rightarrow^* P'$ if either $P = P'$ or P reduces to P' in 1 or more reduction steps.

To the reduction rules for the π -calculus we add the reduction rules in Table 4; they describe how terms are evaluated by applying destructors in let-expressions. If a term $g(M_1, \dots, M_n)$ evaluates to M' , the process continues as $P\{M'/x\}$, otherwise, it terminates.

Table 3: Reduction rules of the π -calculus.

$$\bar{a}\langle M \rangle.P \mid a(x).Q \rightarrow P \mid Q\{M/x\}$$

$$\frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R} \quad !P \rightarrow P \mid !P$$

$$\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \quad \frac{P \rightarrow Q}{(\nu a)P \rightarrow (\nu a)Q}$$

$$\text{if } M = M \text{ then } P \rightarrow P$$

Table 4: Reduction rules for let-expressions.

$$\frac{g(M_1, \dots, M_n) \rightarrow M'}{\text{let } x = g(M_1, \dots, M_n) \text{ in } P \rightarrow P\{M'/x\}}$$

$$\frac{g(M_1, \dots, M_n) \dashrightarrow}{\text{let } x = g(M_1, \dots, M_n) \text{ in } P \rightarrow 0}$$

2.3 Secrecy and Authenticity

In order to define security properties, we need to define an adversary present in the protocol. We are assuming an adversary that has Dolev-Yao capabilities previously defined. An adversary is any process that has a set of public names S in its initial knowledge and does not contain correspondence assertions (defined below).

Definition 1. Let S be a finite set of names. A closed process Q is an S -adversary if and only if $fn(Q) \subseteq S$ and Q does not contain correspondence assertions or executed correspondence assertions.

Informally, the secrecy of a term M is preserved in the protocol, if M is never output to the public channels and therefore obtained by the attacker.

Definition 2 ((Blanchet, 2002)). Let P be a closed process and M be a message. We say that P outputs M on c if and only if $P \rightarrow^* \bar{a}\langle M \rangle.R \mid R'$ for some process R and R' and channel a . We say that P preserves the secrecy of M from S if and only if $P \mid Q$ does not output M on c for any S -adversary Q and any $c \in S$.

We say that principal A is authentic to principal B , if whenever B has completed a protocol run believing that A was the initiator, then this was indeed the case. To express this, we use *correspondence assertions* that are labelled with terms and are of the form **begin**(M) and **end**(M). The intention is that in every run of a protocol, if an end-expression **end**(M) appears, a begin-expression **begin**(M) with the same label M must have appeared earlier in the run. We add correspondence assertions to the process calculus syntax as shown in Table 5.

The reduction rules for correspondence assertions are shown in Table 6; they tell us how as-

Table 5: Formation rules for correspondence assertions.

$$P ::= \mathbf{begin}(M).P \mid \mathbf{end}(M).P \\ \mid \mathbf{begin_ex}(M).P \mid \mathbf{end_ex}(M).P$$

Table 6: Reduction rules for correspondence assertions.

$$\mathbf{begin}(M).P \rightarrow \mathbf{begin_ex}(M) \mid P \\ \mathbf{end}(M).P \rightarrow \mathbf{end_ex}(M) \mid P$$

sertions evolve to executed correspondence assertions: **begin**(M). P evolves to **begin_ex**(M) \mid P and **end**(M). P evolves to **end_ex**(M) \mid P .

We distinguish between non-injective and injective agreement.

Definition 3. The closed starting process P satisfies non-injective agreement with respect to S -adversaries if and only if for any S -adversary Q , for any P' such that $P \mid Q \rightarrow^* P'$, for any M , if **end_ex**(M) occurs in P' , then **begin_ex**(M) also occurs in P' . (In P' , the restrictions are renamed such that they bind pairwise different names, and names different from free names.)(Blanchet, 2002)

If we need to express that the event **end**(M) happened after the event **begin**(M) and at most as many times as event **begin**(M), we use injective agreement.

Definition 4. The closed starting process P satisfies injective agreement with respect to S -adversaries if and only if for any S -adversary Q , for any P' such that $P \mid Q \rightarrow^* P'$, for any M , the number of occurrences of **end_ex**(M) in P' is at most the number of occurrences of **begin_ex**(M) in P . (In P' , the restrictions are renamed such that they bind pairwise different names, and names different from free names.)(Blanchet, 2002)

2.4 ProVerif

ProVerif is a tool for automatically analyzing the security of cryptographic protocols (Blanchet, 2014) that is able to prove secrecy, authenticity and observational equivalence properties of cryptographic protocols. Its analysis considers a unbounded number of sessions and unbounded message space and performed on a symbolic model of a protocol (Blanchet, 2014). ProVerif automatically translates a protocol described in the applied π -calculus into an abstract model using Horn clauses and determines whether the required security properties hold based on the resolution of these clauses.

3 ANALYZING THE LIGHTNING NETWORK PROTOCOLS IN ProVerif

An important part of our work has been the description of each subprotocol in ProVerif.

3.1 Protocol Signature

For modeling the protocols will use the *signature* (\mathcal{S}, Σ), where \mathcal{S} is a set of sorts and Σ is a set of function symbols. \mathcal{S} contains two sorts: $\mathcal{S} = \{term, key\}$. The sort *term* will be used for all hashes, encrypted data and messages exchanged between the nodes, while the sort *key* will be used for all types of keys used (e.g. ephemeral and static keys, encryption keys, etc). The function symbols are: $\Sigma = \{\perp, pk, hash, ECDH, HKDF, senc, sdec, [], ith_i\}$ and their definitions are given in Table 7 where:

- \perp represents an empty data
- $ECDH(k_1, k_2)$ is an abstraction of the Elliptic Curve Diffie-Hellman (Diffie and Hellman, 1976) operation which combines two keys k_1 and k_2 and produces a new key
- $senc(d, k)$ encrypts data d with a symmetric key k and produces a ciphertext
- $sdec(d, k)$ decrypts encrypted data d with a symmetric key k and produces its plaintext version
- $pk(k)$ is a function that generates a public key from a private key k
- $HKDF(k, x)$ represents a Hashed Message Authentication Code (HMAC)-based key derivation function (Krawczyk and Eronen, 2010), where a key k and data x are combined to create a new term; newly created term is actually a key which is used to derive more additional keys
- $hash(d)$ is a representation of a standard hash function that takes a data d and returns its hash value
- $[x, y]$ is a tuple creation function in which we also consider the short form $[x_1, x_2, \dots, x_{n-1}, x_n]$ for the expression $[x_1, [x_2, [\dots, [x_{n-1}, x_n]]]]$
- $ith_i(M)$ function that returns i -th element of a tuple M
- $getmess(s)$ returns the data that is being signed with the signature s
- $sign(d, k)$ signs a data d with a key k
- $checksig(s, k)$ returns true, if the signature s was made using a private key that corresponds to the public key k

Table 7: Cryptographic primitives.

| | |
|--------------|-------------------------------------|
| \perp : | $\rightarrow term$ |
| $ECDH$: | $key \times key \rightarrow key$ |
| $senc$: | $term \times key \rightarrow term$ |
| $sdec$: | $term \times key \rightarrow term$ |
| pk : | $key \rightarrow key$ |
| $HKDF$: | $key \times term \rightarrow term$ |
| $hash$: | $term \rightarrow term$ |
| $[\]$: | $term \times term \rightarrow term$ |
| ith_i : | $term \rightarrow term$ |
| $getmess$: | $term \rightarrow term$ |
| $sign$: | $term \times key \rightarrow term$ |
| $checksig$: | $term \times key \rightarrow term$ |
| bk : | $key \times key \rightarrow key$ |
| $HMAC$: | $term \times key \rightarrow term$ |

Table 8: Additional reduction rules.

| | |
|-------------------------------|--------------------|
| $sdec(senc(M, N), N)$ | $\rightarrow M$ |
| $ith_i([M_1, \dots, M_n])$ | $\rightarrow M_i$ |
| $getmess(sign(M, N), N)$ | $\rightarrow M$ |
| $checksig(sign(M, N), pk(N))$ | $\rightarrow true$ |

- $bk(k, b)$, creates a new key by combining a key k with a blinding factor b , which can be considered as another key
- $HMAC(d, k)$ represents the keyed hash message authentication code computed over the data d and using the key k

We will also extend the applied π -calculus reduction system with additional reduction rules for the destructors we will use. The rules are given in Table 8.

3.2 The Key Agreement Protocol

The first subprotocol of the Lightning Network that we are going to analyse is *the key agreement protocol*. Its goals are to provide mutual authentication between two participants (i.e. nodes) and to create secret session keys (asymmetric encryption key-pairs: ek_I, dk_I for the protocol initiator, and ek_R, dk_R for the responder). The keys will be used to encrypt all future messages exchanged between the participants.

3.2.1 Properties

The protocol is expected to satisfy the following properties:

Table 9: The key agreement protocol message sequence.

$$\begin{aligned}
 node_R &\rightarrow node_I : && s_pub_R & (1) \\
 node_I &\rightarrow node_R : && [e_pub_I, senc(\perp, [k_1, 0, h_1])] & (2) \\
 node_R &\rightarrow node_I : && [e_pub_R, senc(\perp, [k_2, 0, h_2])] & (3) \\
 node_I &\rightarrow node_R : && [senc(s_pub_I, [k_2, 1, h_3]), & (4) \\
 &&& senc(\perp, [k_3, 0, h_4])] & (5)
 \end{aligned}$$

- the initiating node must be authentic to the responding node - if the responding node reaches the end of the protocol with a belief that it has done so with the initiating node, then the initiating node has actually engaged a session with the responding node
- the responding node must be authentic to the initiating node - similarly to the above
- the secrecy of the generated secret session keys ek_I and dk_I must hold - if the initiating node has reached the end of the protocol with the responding node, then the keys ek_I and dk_I , that the initiating node has generated, are secret and can be used for encrypting and decrypting future communication with the responding node
- the secrecy of the generated secret session keys ek_R and dk_R must hold - similarly to the above

3.2.2 Protocol Description

The key agreement protocol chosen for the Lightning Network is Noise.XK (Perrin, 2016), (lightningnetwork, 2017b). Throughout the handshake process, each side maintains the following variables:

- e : a freshly generated ephemeral key-pair
- s : a static key-pair
- ck : the chaining key (a key that is built iteratively until the end of the protocol when it is used to derive the encryption keys)
- h : the handshake hash (accumulated hash of all handshake data sent and received; never transmitted, but used as the Associated Data in the AEAD messages)
- k_1, k_2, k_3 : the intermediate keys which are used for encrypting messages during the handshake (lightningnetwork, 2017b).

Key-pairs will have a private and public components which we will differentiate by appending the word *priv* for private and *pub* for public. In table 9 we can see the message sequence of the protocol.

The protocol starts with the initialization phase, where both parties initialize their handshake state, namely, ck and h . The participants also acquire the

Table 10: The initiating node process.

$$\begin{aligned}
 &node_I(s_priv_I, s_pub_R, h_0, ck_0) = & (1) \\
 &(ve_priv_I)c(s_pub_X).begin(s_pub_X). & (2) \\
 &\mathbf{let} k_1 = ith_0(HKDF(ck_0, es)) \mathbf{in} & (3) \\
 &\mathbf{let} ck_1 = ith_1(HKDF(ck_0, es)) \mathbf{in} & (4) \\
 &\bar{c}\langle [pk(e_priv_I), d_1] \rangle.c(m_2). & (5) \\
 &\mathbf{let} e_pub_R = ith_0(m_2) \mathbf{in} & (6) \\
 &\mathbf{let} k_2 = ith_0(HKDF(ck_1, ee)) \mathbf{in} & (7) \\
 &\mathbf{let} ck_2 = ith_1(HKDF(ck_1, ee)) \mathbf{in} & (8) \\
 &\mathbf{let} d_2 = ith_1(m_2) \mathbf{in} & (9) \\
 &\mathbf{let} k_3 = ith_0(HKDF(ck_2, se)) \mathbf{in} & (10) \\
 &\mathbf{let} ck_3 = ith_1(HKDF(ck_2, se)) \mathbf{in} & (11) \\
 &\mathbf{let} dd_2 = sdec(d_2, [k_2, 0, h_3]) \mathbf{in} & (12) \\
 &\mathbf{if} dd_2 = \perp \mathbf{then} \bar{c}\langle m_3 \rangle. & (13) \\
 &\mathbf{if} s_pub_X = s_pub_R \mathbf{then} \mathbf{end}(pk(s_priv_I)) & (14)
 \end{aligned}$$

where

$$\begin{cases}
 h_1 = hash([h_0, pk(e_priv_I)]) \\
 es = ECDH(e_priv_I, s_pub_X) \\
 d_1 = senc(\perp, [k_1, 0, h_1]) \\
 h_2 = hash([h_1, d_1]) \\
 h_3 = hash([h_2, e_pub_R]) \\
 ee = ECDH(e_priv_I, e_pub_R) \\
 h_4 = hash([h_3, d_2]) \\
 d_3 = senc(pk(s_priv_I), [k_2, 1, h_4]) \\
 h_5 = hash([h_4, d_3]) \\
 se = ECDH(s_priv_I, e_pub_R) \\
 ek_I = ith_0(HKDF(ck_3, \perp)) \\
 dk_I = ith_1(HKDF(ck_3, \perp)) \\
 m_3 = [d_3, d_4]
 \end{cases}$$

static public key of the other party and hash it together with some additional data known to both sides.

We first describe the actions of the initiating node. In Table 10, we can see the applied π -calculus model of the initiating node process.

The actions of the initiating node are

- Generate a new ephemeral key pair e (line 2)
- Compute the first intermediary key and update the chaining key ck_I using the outputs from HKDF function (lines 3,4)

$$\begin{aligned}
 k_1 &= ith_0(HKDF(ck_I, ECDH(e_priv_I, s_pub_R))) \\
 ck_I &= ith_1(HKDF(ck_I, ECDH(e_priv_I, s_pub_R)))
 \end{aligned}$$

- Encrypt an empty text (\perp) with k_1 and send it together with the ephemeral public key (e_pub_I) to the responding node (line 5)

- Receive a response from the responding node and extract e_pub_R and the encrypted data from it (line 6)
- Compute a new intermediary key and update the chaining key ck_I using the outputs from HKDF function (line 7):

$$k_2 = ith_0(HKDF(ck_I, ECDH(e_pub_R, e_priv_I)))$$

$$ck_I = ith_1(HKDF(ck_I, ECDH(e_pub_R, e_priv_I)))$$

- Validate the encrypted data; if it fails, stop (line 8)
- Encrypt the static public key with k_2 (line 9)
- Compute a new intermediary key and update the chaining key ck using the outputs from HKDF function (lines 10,11)

$$k_3 = ith_0(HKDF(ck_I, ECDH(s_priv_I, e_pub_R)))$$

$$ck_I = ith_1(HKDF(ck_I, ECDH(s_priv_I, e_pub_R)))$$

- Encrypt an empty text (\perp) with k_3 and send it together with encrypted static public key to the responding node (line 12)
- Derive the session keys from the chaining key using the HKDF function

In Table 11, we can see the applied π -calculus model of the responding node process. The actions of the responding node are

- Receive a message from the initiating node and extract e_pub_I (line 16) and encrypted data from it
- Compute the first intermediary key (line 17) and update the chaining key ck_R using the outputs from the HKDF function (lines 18,19)

$$k_1 = ith_0(HKDF(ck_R, ECDH(e_pub_I, s_priv_R)))$$

$$ck_R = ith_1(HKDF(ck_R, ECDH(e_pub_I, s_priv_R)))$$

- Validate the encrypted data (line 20); if it fails, stop
- Generate a new ephemeral key pair e
- Compute a new intermediary key and update the chaining key ck_R using the outputs from HKDF function (lines 21,22)

$$k_2 = ith_0(HKDF(ck_R, ECDH(e_priv_R, e_pub_I)))$$

$$ck_R = ith_1(HKDF(ck_R, ECDH(e_priv_R, e_pub_I)))$$

- Encrypt an empty text (\perp) with k_2 and send it together with the ephemeral public key (e_pub_R) to the initiating node (line 23)
- Receive a message from the initiating node and decrypt s_pub_I from and save the additional encrypted data (line 24)

- Compute a new intermediary key and update the chaining key ck_R using the outputs from HKDF function (lines 23,24)

$$k_3 = ith_0(HKDF(ck_R, ECDH(e_priv_R, s_pub_I)))$$

$$ck_R = ith_1(HKDF(ck_R, ECDH(e_priv_R, s_pub_I)))$$

- Validate the encrypted data from the previous message; if it fails, stop (line 30)
- Derive the session keys from the chaining key using the HKDF function (line 31)

We have omitted some of the steps in both of the descriptions (e.g. updating the handshake hash h) but they can be easily read from the corresponding models in Table 10 and Table 11.

Lastly, in Table 12, we can see the main process that will start the initiating and responding nodes processes. Additionally, the participants static public keys are outputted to the public channel to make sure that the attacker receives them.

3.2.3 Analysing the Protocol in ProVerif

The representation of the protocol in ProVerif is a straightforward translation of the specifications in Table 10, 11 and 12.

To test the secrecy properties we use the queries

```
query attacker(secret_ek_I);
attacker(secret_dk_I);
attacker(secret_ek_R);
attacker(secret_dk_R).
```

The free names $secret_ek_I$, $secret_dk_I$, $secret_ek_R$ and $secret_dk_R$ denote the secret keys that are encrypted using session keys ek_I , dk_I , ek_R and dk_R respectively and output to the public channel. If the attacker is able to obtain (decrypt) the encrypted form of these names, then we know that it has obtained some of the session keys.

To test for authenticity properties we use the queries

```
query x:public_key, y:public_key;
inj-event(end_I(x,y)) ==>
inj-event(begin_I(x,y)).
query x:public_key, y:public_key;
inj-event(end_R(x,y)) ==>
inj-event(begin_R(x,y)).
```

These queries test for injective agreement and work as described in Section 2.3. The first query tests if the responding node authentic to the initiating node, while the second tests if the initiating node is authentic to the responding node.

ProVerif outputs the results:

RESULT not attacker(secret_ek_I[]) is true.
 RESULT not attacker(secret_dk_I[]) is true.
 RESULT not attacker(secret_ek_R[]) is true.
 RESULT not attacker(secret_dk_R[]) is true.
 RESULT inj-event(end_I(x_229,y_230)) ==>
 inj-event(begin_I(x_229,y_230)) is false.
 RESULT inj-event(end_R(x_231,y_232)) ==>
 inj-event(begin_R(x_231,y_232)) is true.

The secrecy of the session keys is proven to hold for the protocol. Neither the keys owned by the initiating node and the ones owned by the responding node were leaked to the attacker. However, while the ini-

Table 11: Responding node process.

(15) $node_R(s_priv_R, s_pub_I, h_0, ck_0) =$
 (16) $(\vee e_priv_R)c(m_1).$
 (17) **let** $e_pub_X = ith_0(m_1)$ **in**
 (18) **let** $d_1 = ith_1(m_2)$ **in**
 (19) **let** $k_1 = ith_0(HKDF(ck_0, se))$ **in**
 (20) **let** $ck_1 = ith_1(HKDF(ck_0, se))$ **in**
 (21) **let** $dd_1 = sdec(d_1, [k_1, 0, h_1])$ **in**
 (22) **if** $dd_1 = \perp$ **then**
 (23) **let** $k_2 = ith_0(HKDF(ck_1, ee))$ **in**
 (24) **let** $ck_2 = ith_1(HKDF(ck_1, ee))$ **in**
 (25) $\bar{c}(\langle pk(e_priv_R), d_2 \rangle).c(m_3).$
 (26) **let** $d_3 = ith_0(m_3)$ **in**
 (27) **let** $d_4 = ith_1(m_3)$ **in**
 (28) **let** $s_pub_X = sdec(d_3, [k_2, 1, h_4])$ **in begin**(s_pub_X).
 (29) **let** $k_3 = ith_0(HKDF(ck_2, es))$ **in**
 (30) **let** $ck_3 = ith_1(HKDF(ck_2, es))$ **in**
 (31) **let** $dd_4 = sdec(d_4, [k_3, 0, h_5])$ **in**
 (32) **if** $dd_4 = \perp$ **then**
 (33) **if** $s_pub_X = s_pub_I$ **then end**($pk(s_priv_R)$)

where $\left\{ \begin{array}{l} h_1 = hash([h_0, e_pub_X]) \\ se = ECDH(s_priv_R, e_pub_X) \\ h_2 = hash([h_1, d_1]) \\ h_3 = hash([h_2, pk(e_priv_R)]) \\ ee = ECDH(e_priv_R, e_pub_X) \\ d_2 = senc(\perp, [k_2, 0, h_3]) \\ h_4 = hash([h_3, d_2]) \\ h_5 = hash([h_4, d_3]) \\ es = ECDH(e_priv_R, s_pub_X) \\ dk_R = ith_0(HKDF(ck_3, \perp)) \\ ek_R = ith_1(HKDF(ck_3, \perp)) \end{array} \right.$

Table 12: The main process.

$P = (\vee s_priv_I, s_priv_R, h_0, ck_0)$
let $s_pub_I = pk(s_priv_I)$ **in** $\bar{c}\langle s_pub_I \rangle.$
let $s_pub_R = pk(s_priv_R)$ **in** $\bar{c}\langle s_pub_R \rangle.$
 $((!node_I(s_priv_I, s_pub_R, h_0, ck_0)) |$
 $(!node_R(s_priv_R, s_pub_I, h_0, ck_0)))$

Table 13: Messages exchanged in the channel opening protocol (lightningnetwork, 2017a).

| | | |
|----------------------|------------|--------------------|
| $node_I \rightarrow$ | $node_R :$ | $open_channel$ |
| $node_R \rightarrow$ | $node_I :$ | $accept_channel$ |
| $node_I \rightarrow$ | $node_R :$ | $funding_created$ |
| $node_R \rightarrow$ | $node_I :$ | $funding_signed$ |
| $node_I \rightarrow$ | $node_R :$ | $funding_locked$ |
| $node_R \rightarrow$ | $node_I :$ | $funding_locked$ |

tiating node is authentic to the responding node, the responding node fails to be authentic to the initiating node. This means that the initiating node can be tricked into thinking that it is running a protocol with the responding node, when in fact it is running it with the attacker.

3.3 The Channel Opening Protocol

After the users have completed the key agreement protocol, they are able to open a channel between them. To open a channel, users need to create a funding transaction and two initial commitment transactions. In the messages exchanged in the protocol, users will exchange the data needed for creating the aforementioned transactions. The channel opening protocol consists of several messages being exchanged and we can see them in table 13. Table 13 gives us a simplified overview of the protocol where we see what messages are being exchanged. Messages exchanged in this protocol will be encrypted using the session keys that the users established in the key agreement protocol.

3.3.1 Properties

The security of the protocol relies on the keys that were established beforehand. The participants will use the session keys for encrypting all of the data that is being exchanged. The protocol must preserve the session keys secrecy. Additionally, the participants were authenticated in the key agreement protocol, so the protocol must ensure that the participants authenticity still holds. We can summarize the security properties as follows:

- the encryption key used by the initiating node is secret
- the encryption key used by the responding node is secret
- the initiating node must be authentic to the responding node
- the responding node must be authentic to the initiating node

3.3.2 Protocol Description

To open a channel, we need to create three transactions: one funding transaction and two initial commitment transactions. At least one of the users need to fund the newly created channel with the funding transaction. The funding transaction sends funds to a special 2-of-2 multisig bitcoin address. The funds that are on this address can only be spent by using both of the users private keys, so that one party can not take the funds for itself. Another problem that the users want to avoid is the other party disappearing and forever locking the funds in the 2-of-2 multisig address. The initial commitment transactions are the mechanism that prevents this. They are transactions that spend all of the funds from the multisig address and dividing the funds according to how much each party has initially put in the channel. In order to create the aforementioned transactions, users will need to communicate and exchange the needed data, such as channel parameters and signatures for the initial commitment transactions. The channel parameters will not be very important for this model and they will be presented as a single blob of data that needs to stay confidential.

In this analysis we will simplify the contents of the messages in compare to the actual specification. The messages are listed and described as follows:

- *open_channel* will contain the initiating node public key (s_pub_I) and all other data will be represented as $data_I$
- *accept_channel* will contain the responding node public key (s_pub_R) and all other data will be represented as $data_R$
- *funding_created* will contain the initiating node signature, created for signing a tuple containing $data_I$ and $data_R$, using the private key s_priv_I where $s_pub_I = pk(s_priv_I)$
- *funding_signed* will contain the responding node signature, created for signing a tuple containing $data_I$ and $data_R$, using the private key s_priv_R where $s_pub_R = pk(s_priv_R)$

Table 14: The channel opening protocol message sequence diagram.

| | |
|-------------------------------|---|
| $node_I \rightarrow node_R :$ | $senc([s_pub_I, data_I], ek_I)$ |
| $node_R \rightarrow node_I :$ | $senc([s_pub_R, data_R], ek_R)$ |
| $node_I \rightarrow node_R :$ | $senc(sign([data_I, data_R], s_priv_I), ek_I)$ |
| $node_R \rightarrow node_I :$ | $senc(sign([data_I, data_R], s_priv_R), ek_R)$ |
| $node_I \rightarrow node_R :$ | $senc(\perp, ek_I)$ |
| $node_R \rightarrow node_I :$ | $senc(\perp, ek_R)$ |

- *funding_locked* will be an empty message, sent as a information that the sender has reached the end of the protocol

All messages will be encrypted using the encryption keys (ek_I, ek_R) that the participants have established in the key agreement protocol. The message sequence diagram is present in table 14. The initiating node will start the protocol with sending its public key and the additional data encrypted using ek_I to the responding node. The responding node will decrypt the message using dk_R and answer with its public key and additional data also encrypted using ek_R . The nodes will then exchange signatures for the data exchanged in the first messages, check the signatures and if they are valid, continue with the protocol. The protocol will end with the exchange of empty *funding_locked* messages. We have modeled the protocol in the applied *pi*-calculus. The initiating node process can be seen in Table 15 while the responding node process can be seen in Table 16. Finally, the main process starts the initiating and responding node processes and can be seen in Table 17.

3.3.3 Analysing the Protocol in ProVerif

The ProVerif code used for the testing whether the properties presented in 3.3.1, actually hold for the subprotocol will be omitted. The reason for this is because it looks very similar to the one used for the analysis of *the key agreement protocol* in section 3.2.3, and we will only present the results of the analysis.

```

RESULT not attacker(secret_ek_I[]) is true.
RESULT not attacker(secret_ek_R[]) is true.
RESULT inj - event(end_I(x.81)) ==>
inj - event(begin_I(x.81)) is true.
RESULT inj - event(end_R(x.82)) ==>
inj - event(begin_R(x.82)) is true.
    
```

We can see that both secrecy of the session keys and the authenticity properties hold for the protocol.

Table 15: The initiating node process.

```

(34)     nodeI(s-privI, s-pubR, ekI, dkI) =
(35)           (vdataI)c̄(encm1).
(36)     c(encm2).let m2 = sdec(encm2, dkI) in
(37)           let s-pubX = ith0(m2) in
(38)     let dataX = ith1(m2) in begin(s-pubX).
(39)           c̄(encm3).c(encm4).
(40)     let sigX = sdec(encm4, dkI) in
(41)           let t = getmess(sigX) in
(42)     let b2 = checksig(sigX, s-pubX) in
(43)           if b = true then c̄(encm5).
(44)     if s-pubX = s-pubR then end(pk(s-privI))
    
```

where

$$\left\{ \begin{array}{l} enc_{m1} = senc([pk(s_{-priv}_I), data_I], ek_I) \\ sig = sign([data_I, data_X], s_{priv}_I) \\ enc_{m3} = senc(sig, ek_I) \\ b_1 = (t = (data_I, data_X)) \\ b = b_1 \& \& b_2 \\ enc_{m5} = senc(\perp, ek_I) \end{array} \right.$$

Table 16: The responding node process.

```

(45)     nodeR(s-privR, s-pubI, ekR, dkR) =
(46)           (vdataR)c(encm1).
(47)     let m1 = sdec(encm1, dkR) in
(48)     let s-pubX = ith0(m1) in
(49)     let dataX = ith1(m1) in begin(s-pubX).
(50)           c̄(encm2).c(encm3).
(51)     let sigX = sdec(encm3, dkR) in
(52)           let t = getmess(sigX) in
(53)     let b2 = checksig(sigX, s-pubX) in
(54)           if b = true then c̄(encm4).
(55)           c(encm5).c̄(encm6).
(56)     if s-pubX = s-pubI then end(pk(s-privR))
    
```

where

$$\left\{ \begin{array}{l} enc_{m2} = senc([pk(s_{-priv}_R), data_R], ek_R) \\ b_1 = (t = [data_I, data_X]) \\ b = b_1 \& \& b_2 \\ sig = sign([data_X, data_R], s_{priv}_R) \\ enc_{m4} = senc(sig, ek_R) \\ enc_{m6} = senc(\perp, ek_R) \end{array} \right.$$

3.4 The Onion Routing Protocol

Multi-hop payments enable users to pay to a recipient they are not directly connected to (i.e. do not

Table 17: The main process.

```

P = (v s-privI, s-privR, ekI, ekR)
let s-pubI = pk(s-privI) in c̄(s-pubI).
let s-pubR = pk(s-privR) in c̄(s-pubR).
((!nodeI(s-privI, s-pubR, ekI, ekR)) |
(!nodeR(s-privR, s-pubI, ekR, ekI)))
    
```

share an open channel with). The Lightning Network uses onion routing to securely and privately route the HTLCs (Hash-Time-Locked Contracts) through the network (lightningnetwork, 2017c). The packets sent through the network are obfuscated in layers of encryption so that an attacker is not able to access their content without the right key. Moreover, the nodes that are a part of the payment route are only able to access the information they need to forward the packet to the next node in the route. Additionally, the packets are encrypted using the session keys established in the key agreement protocol described earlier.

3.4.1 Properties

The security properties of the protocol can be analysed from two perspectives: the first one from the point of view of a node being a part of the payment route and the second one, from the point of view of a node that is not a part of the route. We will analyse both perspectives. The secrecy properties that the protocol must hold are:

- secrecy of the session keys
- secrecy of the payment source private key
- secrecy of the payment intermediaries private keys
- secrecy of the payment destination private key
- secrecy of the encrypted payload of every packet

In addition to the secrecy properties, the protocol must also guarantee that the payment sender is authentic to the payment receiver. Since the protocol is based on a one-way communication (the packet is being sent from the source along the nodes in the route to the receiver, without the receiver sending a reply back) it is not possible to authenticate the payment receiver. In other words, the payment receiver need to know exactly who did send the packet with the payment information. If this property does not hold for the protocol, the payment receiver might think that it got the packet from the real payment sender, when in fact it was the attacker that send it. The payment sender authenticity property will therefore, be defined as:

Table 18: The onion routing protocol message sequence.

$$\begin{aligned}
 node_S &\rightarrow node_1: \text{ senc}([d_1, \text{ senc}(o_1, k_2)], k_1) \\
 node_1 &\rightarrow node_2: \text{ senc}([d_2, \text{ senc}(o_2, k_3)], k_2) \\
 &\dots \\
 node_{i-1} &\rightarrow node_i: \text{ senc}([d_i, \text{ senc}(o_i, k_{i+1})], k_i) \\
 &\dots \\
 node_{19} &\rightarrow node_D: \text{ senc}(d_D, k_D)
 \end{aligned}$$

- authenticity of the payment sender to the payment recipient: if the payment recipient reaches the end of the protocol with a belief that it has done so with the payment sender, then the payment sender has actually initiated a session with the payment recipient

3.4.2 Protocol Description

In this section, $node_S$ will be the node that is sending a payment through the route of intermediaries called "hops" to the recipient node we will call $node_D$. The Lightning Network uses source-routing and $node_S$ is the one constructing the route. We skip the part of the route creation and assume that $node_S$ has constructed a valid route (i.e. has selected the nodes through which the payment will be sent). In table 18 we can see the onion routing message sequence diagram, where a $node_i$ possesses a key k_i and $node_D$ possesses the key k_D .

The onion packet contains the details for each of the intermediary nodes specifying how much and where they should send funds to. In return, they receive a fee, but only if the payment is successful. To prevent the intermediary nodes of knowing the private information about the payment, the packet is wrapped in layers of encryption and every intermediary node can decrypt only one layer. The encryption algorithm used in the Lightning Network is ChaCha20 (Nir and Langley, 2018) which uses a symmetric key for encrypting and decrypting the data. Additionally, the authenticity and integrity of the data in the packet is protected by the use of Keyed-Hash Message Authentication Code (HMAC). If the packet was tampered with by one of the intermediary nodes, the HMAC included in the packet will not match the one computed over the received data and therefore the packet should be ignored and discarded. Additionally, every packet has a fixed size of 1366 bytes, meaning that packets that have to go through 1 hop and the ones that have to go through 19 hops, will have the same size which further increases the privacy of the payment.

We can describe the protocol on a simple one hop scenario, which we will use for the analysis later on:

 Table 19: $node_S$ process.

$$\begin{aligned}
 &node_S(s_priv_S, s_pub_H, ek_S) = \\
 &(\ve e_priv_H, d_H, d_D) c_1(s_pub_X). \mathbf{begin}(s_pub_X). \\
 &\overline{c_1}(\text{ senc}([pk(e_priv_H), x_H, HMAC(x_H, ss_1)], ek_S))
 \end{aligned}$$

$$\text{where } \begin{cases} b = \text{hash}([pk(e_priv_H), ss_1]) \\ e_priv_D = bk(e_priv_H, b) \\ x_H = \text{ senc}([d_H, x_D, HMAC(x_D, ss_2)], ss_1) \\ x_D = \text{ senc}([d_D, \perp], ss_2) \\ ss_1 = ECDH(e_priv_H, s_pub_H) \\ ss_2 = ECDH(e_priv_D, s_pub_X) \end{cases}$$

 Table 20: The $node_H$ process.

$$\begin{aligned}
 &node_H(s_priv_H, ek_H, dk_H) = \\
 &c_1(enc_m). \mathbf{let } m = sdec(enc_m, dk_H) \mathbf{ in} \\
 &\quad \mathbf{let } e_pub_H = ith_0(m) \mathbf{ in} \\
 &\quad \quad \mathbf{let } x_H = ith_1(m) \mathbf{ in} \\
 &\quad \quad \quad \mathbf{let } h_{prev} = ith_2(m) \mathbf{ in} \\
 &\quad \quad \quad \quad \mathbf{if } h = h_{prev} \mathbf{ then} \\
 &\quad \quad \quad \quad \mathbf{let } data = sdec(x_H, ss_1) \mathbf{ in} \\
 &\quad \quad \quad \quad \quad \mathbf{let } x_D = ith_1(data) \mathbf{ in} \\
 &\quad \quad \quad \quad \quad \quad \mathbf{let } h_{next} = ith_2(data) \mathbf{ in} \\
 &\quad \quad \quad \quad \quad \quad \overline{c_2}(\text{ senc}([bk(e_pub_H, b), x_D, h_{next}], ek_H))
 \end{aligned}$$

$$\text{where } \begin{cases} ss_1 = ECDH(s_priv_H, e_pub_H) \\ h = HMAC(x_H, ss_1) \\ b = \text{hash}(e_pub_H, ss_1) \end{cases}$$

$node_S$ sends an onion packet over a hop node $node_H$, to the destination $node_D$. $node_H$ is a common peer from $node_S$ and $node_D$ (i.e. have an open channel in-between both nodes). All of the node processes can be see in Table 19, 20 and 21. The packets will be sent along two public channels c_1 and c_2 , the first one connecting $node_S$ and $node_H$ while the second one is a connection between $node_H$ and $node_D$. In short, the packet is being encrypted using the asymmetric encryption with a key set up between the nodes in the route. The shared encryption keys are generated as a result of a Elliptic Curve Diffie Hellman (ECDH) (Diffie and Hellman, 1976) between the nodes. After it reaches the next node in the route, it is partly decrypted (i.e. one part of the packet can be decrypted), so the node knows where to forward it and then the other part (i.e. the part that couldnt be decrypted), is sent to the next node.

Table 21: The $node_D$ process.

```

(57)          $node_D(s\_priv_D, dk_D) =$ 
(58)  $c_2(enc\_m).$  let  $m = sdec(enc\_m, dk_D)$  in
(59)         let  $e\_pub_D = ith_0(m)$  in
(60)         let  $x_D = ith_1(m)$  in
(61)         let  $h_{prev} = ith_2(m)$  in
(62)         if  $h = h_{prev}$  then
(63)         let  $data = sdec(x_D, ss_2)$  in
(64)         let  $h_{next} = ith_1(data)$  in
(65)         if  $h_{next} = \perp$  then end( $pk(s\_priv_D)$ )
    
```

$$where \begin{cases} ss_2 = ECDH(s_priv_D, e_pub_D) \\ h = HMAC(x_D, ss_2) \end{cases}$$

3.4.3 Analysing the Protocol in ProVerif

The protocol analysis is divided into two scenarios; the first one with the attacker being an outside observer and the second one with the attacker being a part of the route, i.e. controlling the intermediary node.

The Outside Attacker. In the ProVerif code we have tested the secrecy of the session keys and participants private keys. Since we are sending the packet in one way, we are just using the session key that the payment sender has established with the hop node (ek_S) and the key that the hop node established with the payment receiver (ek_H). Furthermore we are testing the secrecy of the payment information intended for the hop and sender respectively as well as whether the authenticity of the sender holds.

The results that ProVerif outputs are following:

```

RESULT not attacker(ek_S[]) is true.
RESULT not attacker(ek_H[]) is true.
RESULT not attacker(s_priv_S[]) is true.
RESULT not attacker(s_priv_H[]) is true.
RESULT not attacker(s_priv_D[]) is true.
RESULT not attacker(hop_data[]) is true.
RESULT not attacker(recipient_data[]) is true.
RESULT inj-event(end(p)) ==>
    inj-event(begin(p)) is true.
    
```

The Inside Attacker. The attacker control over the intermediary node is modeled by introducing the malicious intermediary node. We will omit its actual process calculus model because we believe that the reader can imagine the model presented in Table 20 outputting all the keys and data it receives into the public channel. The ProVerif code used for analysing

the inside attack is similar to the one used in the outside attack, with the exception of testing out the secrecy of data that the intermediary node knows. The results that ProVerif outputs are:

```

RESULT not attacker(ek_S[]) is true.
RESULT not attacker(s_priv_S[]) is true.
RESULT not attacker(s_priv_D[]) is true.
RESULT not attacker(recipient_data[]) is true.
RESULT inj-event(end(p)) ==>
    inj-event(begin(p)) is true.
    
```

As we can see, the results are the same for both the inside and outside attack and therefore we can conclude that the intermediary node knowledge is limited in the sense that it does not allow the attacker to obtain the confidential data that belongs to the other participants.

4 CONCLUSION

In this paper, we have given an analysis of the Lightning network using ProVerif. The analysis shows that both the channel opening and the onion routing protocols satisfy all of the required properties. As for the key agreement protocol, it does not satisfy the authenticity property in one direction, namely that of the authenticity of the protocol responder to the initiator. This implies that in the key agreement protocol, that should provide mutual authentication, the protocol initiator cannot be sure that it is running the protocol with the responder it has in mind.

While the protocols satisfy the desired secrecy properties, other information leaks appear to be possible for the Lightning Network. Herrera-Joancomartí et al. (Herrera-Joancomartí et al., 2019) have recently presented an attack that is able to disclose the balance of a channel in the Lightning Network. This attack is based on carrying out multiple payments while ensuring that none of them will be finalized.

An analysis of the anonymity properties of the Lightning Network protocols is another topic for further work. An attacker must be prevented from learning anything about the secrets (in this case the sum of the secrets), and the support given by ProVerif for proving indistinguishability is likely to be useful in for analyzing both anonymity and the balance confidentiality property mentioned above.

REFERENCES

- Abadi, M. and Gordon, A. D. (1999). A calculus for cryptographic protocols: The spi calculus. *Information and computation*, 148(1):1–70.

- Blanchet, B. (2002). From secrecy to authenticity in security protocols. In *International Static Analysis Symposium*, pages 342–359. Springer.
- Blanchet, B. (2014). Automatic verification of security protocols in the symbolic model: the verifier ProVerif. In Aldini, A., Lopez, J., and Martinelli, F., editors, *Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures*, volume 8604 of *Lecture Notes on Computer Science*, pages 54–87. Springer Verlag.
- Blanchet, B., Smyth, B., Cheval, V., and Sylvestre, M. (2018). Proverif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial.
- Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654.
- Dolev, D. and Yao, A. C. (1981). On the security of public key protocols (extended abstract). In *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*, pages 350–357.
- Herrera-Joancomartí, J., Navarro-Arribas, G., Ranchal-Pedrosa, A., Pérez-Solà, C., and Garcia-Alfaro, J. (2019). On the difficulty of hiding the balance of lightning network channels. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, Asia CCS '19*, pages 602–612, New York, NY, USA. ACM.
- Kiayias, A. and Litos, O. S. T. (2019). A composable security treatment of the lightning network. *IACR Cryptology ePrint Archive*, 2019:778.
- Krawczyk, H. and Eronen, P. (2010). Hmac-based extract-and-expand key derivation function (hkdf).
- lightningnetwork (2017a). Github - lightning-rfc/02-peer-protocol.md.
- lightningnetwork (2017b). Github - lightningnetwork-rfc/08-transport.md.
- lightningnetwork (2017c). Github - lightningnetwork/lightning-onion: Onion routed mi- cropayments for the lightning network.
- Nir, Y. and Langley, A. (2018). Chacha20 and poly1305 for ietf protocols. Technical report.
- Perrin, T. (2016). The noise protocol framework. *Power-Point Presentation*.