The security analysis of the Lightning Network protocol

Project Report
dt1010f19

Aalborg University
Computer Science

Copyright © Aalborg University 2019

Here you can write something about which tools and software you have used for typesetting the document, running simulations and creating figures. If you do not know what to write, either leave this page blank or have a look at the colophon in some of your books.

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science
Aalborg University
http://www.aau.dk

Title:
The security analysis of the Lightning Network protocol

Theme:
The security analysis of cryptographic protocols that form the payment network called the Lightning Network using formal methods and an automatic cryptographic protocol analyser tool called ProVerif

Project Period:
Spring Semester 2019

Project Group:
dt1010f19

Participant(s):
Vilim Staroveški

Supervisor(s):
Hans Hüttel

Copies: 1

Page Numbers: 107

Date of Completion:
June 6, 2019

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.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Summary

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 the version v1.0 of the Lightning protocol was released in December 2017. The problem with the Lightning Network is that it could potentially have vulnerabilities that we still do not know of and the goal of this project is to further investigate this issue. At the time of writing, the Lightning Network consists of about 8000 nodes, with about 37000 payment channels that have about 1000 bitcoins (worth about 8 million US dollars with the current exchange rate) locked up in them. This is a big motive for an attacker to try to break the security of the protocol.

One of the limitations of the Lightning Network, is the fact that a user needs to be online when receiving a payment. This could potentially expose the user private key, which would give an attacker full access to the 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 the user authenticity. The protocol must be able to guarantee that if user A believes that is interacting with user B, then user A is indeed interacting with user B.

The goal of this report is to analyse the security of the Lightning Network by answering two main questions:

- are confidential data kept secret in the Lightning Network?
- does user authenticity hold in the Lightning Network?

In order to perform the analysis, we will consider the Lightning Network as a collection of protocols that are used depending on the user actions. We will identify and split the Lightning Network into four cryptographic sub-protocols:

1. establishing a secure and authenticated connection to another user - *the key agreement protocol*
2. opening a payment channel to another user - *the channel opening protocol*
3. making a payment to a user - *the onion routing protocol*

4. closing an existing payment channel - *the channel closing protocol*

The main focus will be on the protocols 1, 2 and 3, since in these protocols a lot of sensitive data is being exchanged that, if an attacker is able to acquire, could compromise the Lightning Network security. We will use ProVerif to verify if the protocol actually satisfies the security properties it needs to satisfy. ProVerif is a tool for automatically analyzing the security of cryptographic protocols and it is able to prove the security properties we are interested in, namely secrecy and authenticity. The ProVerif input language is a variant of the applied π-calculus and we will present models of the aformentioned sub-protocols in the applied π-calculus. ProVerif relies on the symbolic model of cryptography based on Dolev-Yao assumptions, meaning that the attacker will able to obtain every message passing through the public channel as well as modify, delete and inject messages to the channel. The results that ProVerif has produced can be seen in table 1.

Protocol	**Security property**	**ProVerif result**
The key agreement protocol	secrecy of the session key generated by the initiator	**true**
	secrecy of the session key generated by the responder	**true**
	authenticity of the initiator to the reponder	**true**
	authenticity of the responder to the initiator	**false**
The channel opening protocol	secrecy of the session key used by the initiator	**true**
	secrecy of the session key used by the responder	**true**
	authenticity of the initiator to the reponder	**true**
	authenticity of the responder to the initiator	**true**
The onion routing protocol	secrecy of the session key used by the initiator	**true**
	secrecy of the session key used by the responder	**true**
	secrecy of the private key used by the initiator	**true**
	secrecy of the private key used by the intermediary	**true**
	secrecy of the private key used by the responder	**true**
	secrecy of the packet payload	**true**
	authenticity of the initiator to the responder	**true**

Table 1: The results produced by ProVerif

From the results of the analysis, we have discovered that both *the channel opening* and *the onion routing* protocols satisfy all of the required properties. *The key agreement* protocol satisfies all but one security property: the authenticity of the protocol responder to the initiator. This finding is quite interesting and probably the most important result of this report, because it means that: *in the key agreement protocol, that should provide mutual authentication, the protocol initiator can not be sure that it is running the protocol with the responder it has in mind*.

We have presented two main questions that this report wanted to give answers to and they are:

- are confidential data kept secret in the Lightning Network?
  - *yes, all of the Lightning Network sub-protocols preserve the secrecy of confidential data*

- does user authenticity hold in the Lightning Network?
  - *for two out of three sub-protocols that were analysed (namely, the channel opening and the onion routing protocol) it does, but for the key agreement protocol it does not, because it does not provide provide a way for the protocol initiator to authenticate the responder (the protocol responder is still able to authenticate the initiator, however)*

The future work could be done on answering new questions raised by the authenticity property that does not hold in the key agreement protocol:

- how does this flaw affects the Lightning Network security?
- is there a way in which an attacker would take advantage of this flaw?
- what could be changed in the key agreement protocol so that the initiator is able to authenticate the responder?

Finally, the Lightning Network remains a fully working network which due to its immaturity can benefit from any form of the future analysis such as this report was.

Contents

Preface **xi**

1 Introduction **1**
1.1 The Lightning Network and its security issues 1
1.2 Problem statement 3
1.3 Methodology 3
1.4 Report structure 5

2 Prerequisites **7**
2.1 A process calculus for cryptographic protocols 7
2.1.1 The π-calculus 7
2.1.2 The basic definitions of the π-calculus 8
2.1.3 The applied π-calculus 10
2.2 The Dolev-Yao attacker 14
2.3 Secrecy and authenticity 15
2.4 ProVerif 17

3 The Lightning Network **23**
3.1 Introduction 23
3.2 The Lightning Network building blocks 23
3.3 The Lightning Network protocol 24

4 The Lightning Network key agreement protocol **29**
4.1 A brief introduction to the protocol 29
4.2 Properties 29
4.3 Principals and signature 30
4.4 Informal description 31
4.5 The applied π-calculus version of the protocol 37
4.6 From π to Proverif 40
4.7 Analysing the protocol in ProVerif 42
4.7.1 Attacker capabilities 42
4.7.2 Analysis of security properties 43

4.7.3 Additional analysis of the responder authentication failure . 44

5 The Lightning Network channel opening protocol 47

5.1 A brief introduction to the protocol 47
5.2 Properties 48
5.3 Principals and signature 48
5.4 Informal description 49
5.5 The applied π-calculus version of the protocol 51
5.6 From π to Proverif 52
5.7 Analysing the protocol in ProVerif 53
5.7.1 Attacker capabilities 54
5.7.2 Analysis of security properties 55

6 The Lightning Network onion routing protocol 57

6.1 A brief introduction to the protocol 57
6.2 Properties 57
6.3 Principals and signature 59
6.4 Informal description 60
6.5 The applied π-calculus version of the protocol 63
6.6 From π to Proverif 66
6.7 Analysing the protocol in ProVerif 67
6.7.1 Attacker capabilities 68
6.7.2 The outside attacker 69
6.7.3 The inside attacker 70

7 Conclusion 73

7.1 Results 73
7.2 Further work 75

Bibliography 77

A The unauthenticated Diffie-Hellman key exchange 81

B The Lightning Network key agreement protocol 85

C The unauthenticated Diffie-Hellman key exchange 93

D The Lightning Network channel opening protocol 99

E The Lightning Network onion routing protocol 103

Preface

Aalborg University, June 6, 2019

Vilim Staroveški
<vstaro17@student.aau.dk>

Chapter 1

Introduction

1.1 The Lightning Network and its security issues

The Lightning Network [28] 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 the version v1.0 of the Lightning protocol was released in December 2017. One of the big differences between a standard Bitcoin transaction and a transaction made using the Lightning Network is that whenever a user makes a standard Bitcoin transaction, it is broadcasted to the entire network, putting the bandwidth, memory, and CPU load on every fully validating node. The Lightning Network in comparison is made up of multi-hop payment channels allowing its users to make transactions between each other without recording each transaction on the blockchain.

The Lightning Network as a protocol, is a part of a fairly new space often called the "blockchain space" which is under constant attacks performed by hackers. These attacks can be successful and when they are, they result in relatively major funds being stolen. The website called "Blockchain Graveyard" [23], contains information about 66 publicly known incidents that resulted in stolen funds and/or shutdown of the product (i.e. cryptocurrency exchanges, web wallet hosting sites). On figure 1.1, we can see the estimated root causes of the breaches.

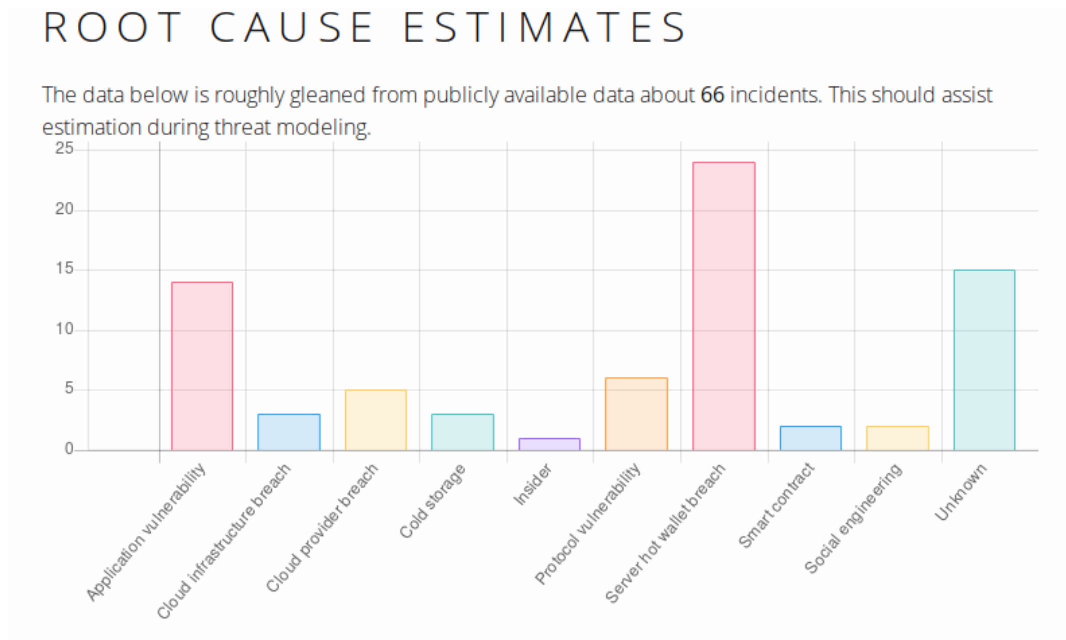

Figure 1.1: Causes to the incidents [23]

The problem with the Lightning Network is that it could potentially have vulnerabilities that we still do not know of and the goal of this project is to further investigate this issue. At the time of writing, the Lightning Network consists of about 8000 nodes, with about 37000 payment channels that have about 1000 bitcoins (worth about 8 million US dollars with the current exchange rate) locked up in them [1]. This is a big motive for an attacker to try to break the security of the protocol. The developers that are working on the Lightning Network, still do not recommend it for the use in large funds transactions and the protocol imposes a maximum payment limit that is set to 2^{24} satoshis (16777216 satoshis = 0.16777216 bitcoins) [17]. The security of the Lightning Network as a cryptographic protocol is yet to be analysed.

The main reason why the funds are being stolen in the "blockchain space", is that the user private key is leaked to the attacker [23]. The private key is a confidential piece of data, that enables the owner to make payments and therefore it is very important that it is not obtained by an attacker. The private key *secrecy* is one of the most important security properties that the Lightning Network must satisfy and it will be a part of the analysis in this report. Additionally, we will check the secrecy of other confidential payment or user data, that does not necessarily lead to the loss of funds but may compromise the payment or user privacy.

Another important security property that the Lightning Network must satisfy is the *user authenticity*. The protocol must be resistant to the e.g. the man-in-the-middle attacks, because some confidential information that is being exchanged in

the protocol could end up in the hands of an attacker. Therefore, it has to be impossible for an attacker to mislead a user into thinking that it is running a protocol with someone else.

1.2 Problem statement

The Lightning Network is still a very new technology and its security properties have not been fully tested. One of the limitations of the Lightning Network, is the fact that a user needs to be online when receiving a payment. This could potentially expose the user private key [29], which would give an attacker full access to the 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 the user authenticity. The protocol must be able to guarantee that if user A believes that is interacting with user B, then user A is indeed interacting with user B.

The goal of this report is to analyse the security of the Lightning Network specification by answering two main questions:

- are confidential data kept secret in the Lightning Network?
- does user authenticity hold in the Lightning Network?

1.3 Methodology

In order to perform the analysis, we will consider the Lightning Network as a collection of protocols that are used depending on the user actions. We will identify and split the Lightning Network into four cryptographic sub-protocols:

1. establishing a secure and authenticated connection to another user - *the key agreement protocol*
2. opening a payment channel to another user - *the channel opening protocol*
3. making a payment to a user - *the onion routing protocol*
4. closing an existing payment channel - *the channel closing protocol*

The main focus will be on the protocols 1, 2 and 3, since in these protocols a lot of sensitive data is being exchanged that, if an attacker is able to acquire, could compromise the Lightning Network security. The protocol 4 (i.e. the channel closing protocol) will not be analysed and mentioned in this report because it is typically performed by a single message exchange between the users or it can be done by a

one step action at the user end.

The analysis of the cryptographic protocols is often performed by modeling a protocol in some variant of a process calculus and using an cryptographic protocol analyser tool such as ProVerif [9], AVISPA [5] and Scyther [11]. In [10], the authors used the process calculus similar to the one we will use in this document and the ProVerif tool to analyse the SRP protocol [26] applied to the DSR routing protocol [15]. Another work [4] was done on a Single-Sign-On protocol used by Google Apps, where a flaw was discovered when analyzing the protocol using formal methods, abstracting messages by a term algebra and using the AVISPA platform [10].
In this report, we will use ProVerif to verify if the protocol actually satisfies the security properties it needs to satisfy. ProVerif is a tool for automatically analyzing the security of cryptographic protocols and it is able to prove the security properties we are interested in, namely secrecy and authenticity [9]. The ProVerif input language is a variant of the applied π-calculus. The analysis using ProVerif considers the unbounded number of sessions and the unbounded message space [9]. ProVerif relies on the symbolic model of cryptography based on Dolev-Yao assumptions [13], meaning that the attacker will able to obtain every message passing through the public channel as well as modify, delete and inject messages to the channel. Cryptography is assumed to be perfect and the attacker is able to use only the cryptographic primitives defined by the user [9].

The analysis of each protocol will have the following structure:

- a short introduction to the protocol
- the protocol security properties
- the protocol signature
- informal description of the protocol
- the applied π-calculus model of the protocol
- the protocol analysis (using ProVerif)

We will start by giving a short introduction to the protocol, define the security properties that the protocol should satisfy and then we will give a more detailed but informal description. We will then model the protocol in the applied π-calculus [6] which will give us a more formal description. The cryptographic protocols we will analyse, rely on cryptographic operations and on communication channels with security properties (authenticity and secrecy) that we want to test in this report [2]. The π-calculus is used for creating a mathematical model of a process and describe it on an abstract level. Its syntax is fairly simple but powerful

and can be used for modeling the communication channels. However, the basic π-calculus syntax can not express the cryptographic operations that we need to model a cryptographic protocol. In [2], the authors showed that it is possible to describe and analyse cryptographic protocols in the applied π-calculus and there we find a reason for using a variant of the applied π-calculus which will extend the basic π-calculus with cryptography primitives we will define for each of the three protocols. Additionally, we will extend the π-calculus with the syntax for correspondence assertions [34] that will be able to prove authenticity properties of the protocols. We will use the definition of the principal authentication as defined in [34] where the authors defined the correctness properties of the authentication protocols. The authenticity property holds if the authenticating principal can be assured that it is interacting to the principal that it has in mind.

1.4 Report structure

In chapter 2 we will go through all of the prerequisites used in the protocol analysis. Then we will give an overview of the Lightning Network specification in chapter 3. The following three chapters will contain the analysis of the three protocols that form the Lightning Network: the key agreement protocol in chapter 4, the channel opening protocol in chapter 5 and the onion routing protocol in chapter 6. The report will be completed with the conclusion in chapter 7 where we will discuss the results of the Lightning Network analysis.

Chapter 2

Prerequisites

In this chapter we will present the prerequisites for the rest of the report. In section 2.1 we introduce the process calculus that will be used for modeling the Lightning Network sub-protocols. We will then present the attacker that is considered in the analysis in section 2.2 and definitions of secrecy and authenticity that are required for describing the properties the protocols would need to satisfy in section 2.3. In section 2.4 we are going to present ProVerif, the tool that is going to be used for analysis of the protocols regarding the security properties they should provide.

2.1 A process calculus for cryptographic protocols

In this section we will describe the π-calculus. The π-calculus presented here is taken from [6], but in its simplified form since we will extend it with the additional syntax that will then form the basis of the variant of the applied π-calculus. Lastly, we will describe the applied π-calculus which will be used throughout the document in the protocols descriptions.

2.1.1 The π-calculus

The π-calculus is a model of computation that has a small and simple syntax but nevertheless it is extremely expressive [2]. It is based on the use of the communication channels that the processes use to interact between themselves where the basic computational step is the communication between two processes [6]. It is possible to model concurrent processes which can then interact between themselves by sending communication links (i.e. channels). The receiver can then use that channels to further interact with other processes [6]. The communication capabilities of a process are determined by the channels it knows about. The channels may be restricted, which means that only a certain process may use them [2]. The scope that the channel has due to the restriction can change. A process may send a re-

stricted channel to another process, which is then able to use it. The basis of the π-calculus security are in fact the scoping rules, which state that the environment of a protocol (i.e. the attacker) can not access a channel that is not explicitly given to it [2].

2.1.2 The basic definitions of the π-calculus

We consider an infinite set of *names* (channels) $\mathcal{N}$ and an infinitive 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*. *Terms* can therefore be *names* and *variables*. In Table 2.1, we can see the π-calculus syntax.

Terms	M, N	::=	x, y, z	variable
			a, b, c	name
Processes	P, Q, R	::=	0	empty process
			$\overline{a}\langle N \rangle.P$	output
			$a(x).P$	input
			$P \mid Q$	parallel composition
			$!P$	replication
			$(\nu a)P$	restriction
			if $M = N$ **then** P	match

Table 2.1: Syntax of the π-calculus [6]

We now explain the process formation rules. A process can be:

- the empty process 0 that does nothing
- an output process $\overline{a}\langle N \rangle.P$, where the term N is sent along the name a and then the process continues as P
- an input process $a(x).P$, where the process is ready to receive a term on the channel a and the process continues as P with x bound to the received term
- a parallel composition $P \mid Q$ represents a process that behaves as both P and Q in parallel; P and Q can act independently but they can interact as well
- a replication $!\,P$ that behaves as an infinite number of copies of P running in parallel
- a restriction $(\nu a)P$ where the process acts like P but with a name a being bound in P (i.e. a is a newly created name in P)

- a match **if** $M = N$ **then** P is used for a process that will behave as P if the terms M and N are the same; otherwise, it will do nothing

We say that in the input process $a(x).P$, *binds* x in P and the occurrences of x in P are called *bound* [6]. The restriction operator $(\nu a)P$ also binds a in P. A name is *free* if it is not bound by an input prefix (i.e. $a(x).P$) or restriction (i.e. $(\nu a)P$). We use $\text{bn}(P)$ and $\text{bv}(P)$ for the set of *bound* names and variables in P, respectively, and we use $\text{fn}(P)$ and $\text{fv}(P)$ for the set of *free* names and variables in P, respectively. A process is *closed* if it has no free variables, but it can have free names. We will write $\{M_1/x_1, ..., M_n/x_n\}$ for the *substitution* that replaces a variable x_i with a term M_i for every $i \in \{1, ..., n\}$ [8].

We also introduce the *structural congruence* which is used to identify the processes that are identical up to structure [6]. For example, a process $a(x).\bar{b}\langle x\rangle$ and a process $a(y).\bar{b}\langle y\rangle$ are the variants of an alpha-conversion, differing only in the choice of the bound name. Both of the processes will receive some term on the channel a and send it out on the channel b. Another example are the processes $P \mid Q$ and $Q \mid P$. We will use $\equiv$ as a symbol for the *structural congruence* while its definition will be a collection of rules applied on a closed process and it is presented in Table 2.2.

(Struct Refl)	$P \equiv P$
(Struct Nil)	$P \mid 0 \equiv P$
(Struct Comm)	$P \mid Q \equiv Q \mid P$
(Struct Assoc)	$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
(Struct Switch)	$(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$
(Struct Drop)	$(\nu a)0 \equiv 0$
(Struct Extrusion)	$(\nu a)(P \mid Q) \equiv P \mid (\nu a)Q \quad \text{if } a \notin \text{fn}(P)$

(Struct Red) (Struct Symm) (Struct Trans)

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

(Struct Par) (Struc Res)

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

Table 2.2: Structural congruence rules [2]

We will define a *reduction* relation, $\rightarrow$, by a collection of reduction rules on closed processes, presented in Table 2.3. Additionally, we define a relation, $\nrightarrow$, that specifies that a process on the left side can not be reduced to the process on the

right side of the symbol.

(Red Interact)	(Red Par)	(Red Repl)
$\overline{a}\langle M\rangle.P \mid a(x).Q \rightarrow P \mid Q\{M/x\}$	$\dfrac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R}$	$!\,P \rightarrow P\ \mid !\,P$
(Red Struct)	**(Red Res)**	**(Red Match)**
$\dfrac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$	$\dfrac{P \rightarrow Q}{(\nu a)P \rightarrow (\nu a)Q}$	**if** $M = M$ **then** $P \rightarrow P$

Table 2.3: Reduction rules [2]

2.1.3 The applied π-calculus

The applied π-calculus is the process calculus that we will use in modeling the cryptographic protocols in this document. It is based on the π-calculus described in section 2.1.1 and inspired by the process calculi defined in [8] and [10]. We extend the π-calculus syntax with a finite set of function symbols Σ, representing constructors and destructors. The finite set Σ will be defined as a part of a protocol signature for every protocol independently. The constructors are used for building terms, while the destructors are used for manipulating the terms in processes.

Terms	M, N	::=	x, y, z	variable
			a, b, c	name
			$f(M_1, ..., M_n)$	constructor application
Processes	P, Q	::=	0	nil
			$\overline{a}\langle N \rangle.P$	output
			$a(x).P$	input
			$P \mid Q$	parallel composition
			$!P$	replication
			$(\nu a)P$	restriction
			if $M = N$ **then** P	match
			let $x = g(M_1, ..., M_n)$ **in** P	destructor application
			begin$(M).P$	begin event
			end$(M).P$	end event
			begin_ex$(M).P$	executed begin event
			end_ex$(M).P$	executed end event

Table 2.4: Syntax of the applied π-calculus

In Table 2.4, f represents a constructor, while g represents a destructor ($f, g \in \Sigma$). Furthermore, destructors are defined by the reduction rules which will be extended for each of the protocols to include all of the protocol-specific destructors. The π-calculus reduction system will be used for the applied π-calculus with the additions from Table 2.5. (Red Dest 1) and (Red Dest 2) are the destructor applications, where if there exists a reduction rule $g(M_1, ..., M_n) \rightarrow M'$, the process continues as $P\{M'/x\}$, otherwise, it becomes an empty process.

$$\text{(Red Begin)} \quad \textbf{begin}(M).P \rightarrow \textbf{begin_ex}(M) \mid P$$
$$\text{(Red End)} \quad \textbf{end}(M).P \rightarrow \textbf{end_ex}(M) \mid P$$

(Red Dest 1)

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

(Red Dest 2)

$$\frac{g(M_1, ..., M_n) \not\rightarrow}{\textbf{let } x = g(M_1, ..., M_n) \textbf{ in } P \rightarrow 0}$$

Table 2.5: The additions to the π-calculus reduction system

In the following example we can see how we can extend the applied π-calculus syntax for a simple cryptographic protocol.

Example 2.1.1. We consider the unauthenticated Diffie-Hellman key exchange [12, 27]. The protocol has the following message sequence diagram:

Message 1	$node_I$	$\rightarrow$	$node_R$	:	s_pub_I
Message 2	$node_R$	$\rightarrow$	$node_I$	:	s_pub_R
Message 3	$node_I$	$\rightarrow$	$node_R$	:	$senc(m, ECDH(s_priv_I, s_pub_R))$

Table 2.6: The unauthenticated Diffie-Hellman key agreement protocol message sequence

The initiator node sends the first message containing its public key. The responding node responds with its own public key. The initiating node can then send a message m encrypted using a key that was made by combining his own private and responding node public key. The responding node can decrypt the message m because it can construct the same key locally, by combining his own private and initiating node public key.
This protocol will have the signature: $(\mathcal{S}, \Sigma)$, 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 encrypted data and messages exchanged between the nodes, while the sort *key* will be used for all types of keys used (e.g. public and private keys, encryption key, etc).
In the signature $(\mathcal{S}, \Sigma)$, the function symbols are: $\Sigma = \{pk, ECDH, senc, sdec\}$ and their definitions are given in the table 2.7.

$$
\begin{aligned}
pk &: key \rightarrow key \\
ECDH &: key \times key \rightarrow key \\
senc &: term \times key \rightarrow term \\
sdec &: term \times key \rightarrow term
\end{aligned}
$$

Table 2.7: Cryptographic primitives

Where:

- $pk(k)$ is a public key generation function that takes a private key k and returns its public key pair
- $ECDH(k_1, k_2)$ is an abstraction of the Elliptic Curve Diffie-Hellman[12] 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 data d with a symmetric key k and produces its plaintext version

The applied π-calculus syntax is extended with the cryptographic primitives from table 2.7. In this example the constructors are *pk*, *ECDH* and *senc*, while the only destructor is *sdec*. Constructors can be directly applied on terms, but for the destructor *sdec* we need to define additional reduction rule:

$$\text{(Red Decrypt)} \quad sdec(senc(M, N), N) \rightarrow M$$

The reduction rule (Red Decrypt) specifies that with the correct key, decryption reduces to a term M that was the previously encrypted plaintext.

Another addition to the syntax of the forms, a process can take are:

- **begin**$(M).P$, where the process emits event **begin**(M) and then continues as P
- **end**$(M).P$, where the process emits event **end**(M) and then continues as P
- **begin_ex**(M) and **end_ex**(M) that are used to remember that the events **begin**(M) and **end**(M) respectively, have been executed; they will not appear in the beginning of the processes but in the process reductions

The events (**begin**(M), **end**(M), **begin_ex**(M), **end_ex**(M)) are the correspondence assertions which we will use for proving the authenticity. We will also say that a process that does not contain executed events (**begin_ex**(M) and **end_ex**(M)) is called a *starting process*.

Example 2.1.2. We can now present the full model of the protocol from example 2.1.1 in the applied π-calculus. The model consists of the processes describing the behaviour of the participants and a main process that initializes and starts them. The participants processes will be $node_I$ for the protocol initiator and $node_R$ for the responding participant. $node_I$ is modeled as follows:

$$\begin{aligned} node_I(s_priv_I, s_pub_R, m) = & \bar{c}\langle pk(s_priv_I)\rangle.c(s_pub_X).\mathbf{begin}(s_pub_X) \\ & \mathbf{let}\ k = ECDH(s_priv_I, s_pub_X)\ \mathbf{in} \\ & \mathbf{if}\ s_pub_X = s_pub_R\ \mathbf{then} \\ & \mathbf{end}(pk(s_priv_I)).\bar{c}\langle senc(m, k)\rangle \end{aligned}$$

The responding node process, $node_R$ is:

$$\begin{aligned} node_R(s_priv_R, s_pub_I, m) = & c(s_pub_X).\mathbf{begin}(s_pub_X).\bar{c}\langle pk(s_priv_R)\rangle. \\ & c(enc_m').\mathbf{let}\ k = ECDH(s_priv_R, s_pub_X)\ \mathbf{in} \\ & \mathbf{let}\ m' = sdec(enc_m', k)\ \mathbf{in} \\ & \mathbf{if}\ m' = m\ \mathbf{then} \\ & \mathbf{if}\ s_pub_X = s_pub_I\ \mathbf{then}\ \mathbf{end}(pk(s_priv_R)) \end{aligned}$$

And finally, the main process that starts the participants processes is:

$$
\begin{aligned}
P = &(\nu s_priv_I, s_priv_R, m) \\
&\quad \textbf{let } s_pub_I = pk(s_priv_I) \textbf{ in } \bar{c}\langle s_pub_I \rangle. \\
&\quad \textbf{let } s_pub_R = pk(s_priv_R) \textbf{ in } \bar{c}\langle s_pub_R \rangle. \\
&\quad ((!node_I(s_priv_I, s_pub_R, m))|(!node_R(s_priv_R, s_pub_I, m)))
\end{aligned}
$$

The main process initializes the protocol participants with assigning them their private keys and public key from the other party. Additionally, the main process outputs the public keys of the participants to the public channel to make them available to the attacker. In the last line, an unbounded number of participants processes are run in parallel.

2.2 The Dolev-Yao attacker

The Dolev-Yao assumptions [13] are in the heart of the symbolic analysis we are performing in this report. The messages exchanged in the protocol will be sent through the public channels. These channels are assumed to be controlled by an attacker with the Dolev-Yao capabilities [9]. In other words, we are considering an active attacker which:

- 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 [13]

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 cryptoanalysis 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).
Modeling an attacker that can perform all of the aformentioned actions can be difficult and lead to errors. In [2], the authors are modeling the attacker as an arbitrary process, which will be able to interact with the protocol. We will follow that approach as well.

The capabilities of an attacker will be modeled using an inference system. The inference system is a finite set of rules of the form $\frac{u_1 ... u_n}{u}$ where $u, u_1, ..., u_n$ are of

sort *term*. The attacker can deduce a term u, if there exists a sequence of inference system rules that can be applied on a term that attacker already knows and that results in u. The inference system, will be determined by the signature of the protocol that is being analysed, so it will be defined for each protocol independently.

Example 2.2.1. We can consider the following inference system for the protocol presented in example 2.1.1:

(Inf ECDH)	(Inf Pk)
$\dfrac{M\ N}{ECDH(M,N)}$	$\dfrac{ith_i[M_1,...,M_n]}{M_1}$
(Inf Enc)	(Inf Dec)
$\dfrac{M\ N}{senc(M,N)}$	$\dfrac{sdec(senc(M,N),N)}{M}$

Table 2.8: The inference system for the unauthenticated Diffie-Hellman protocol

The rules of the inference system enables the attacker to:

- perform a Elliptic Curve Diffie-Hellman (Inf ECDH)
- compute a public key from a private key (Inf Pk)
- encrypt a term (Inf Enc)
- decrypt a term, if it knows the correct encryption key (Inf Dec)

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 in section 2.2. As defined in [8] "an adversary is any process that has a set of public names $\mathcal{S}$ in its initial knowledge":

Definition 1. *Let $\mathcal{S}$ be a finite set of names. The closed process Q is an $\mathcal{S}$-adversary if and only if* $\mathrm{fn}(Q) \subseteq S$ *and Q does not contain begin, end, begin_ex or end_ex events [8].*

Informally, we can say that secrecy of a piece of data is preserved in the protocol, if the piece of data is never output to the public channels and therefore obtained by the attacker. The more formal definition of secrecy will also be taken from [8]:

Definition 2. *We say that the closed process P outputs M on c if and only if $P \rightarrow^* \overline{a}\langle M\rangle.R \mid R'$ for some process R and R'. The closed process P preserves the secrecy of M from $\mathcal{S}$ if and only if $P \mid Q$ does not output M on c for any $\mathcal{S}$-adversary Q and any $c \in \mathcal{S}$ [8].*

In other words, the process P preserves the secrecy of a term M if there does not exists a sequence of reduction rules (i.e. $\rightarrow^*$), that would make P evolve to a process $\overline{a}\langle M\rangle.R \mid R'$.

We will say that principal A is authentic to principal B, if it is the case that B is correct in assuming that she is being talked to by A. In the case of a protocol run, we will say that A is authentic to B if it is true that, when B has completed a protocol run with believing that it has run it with A as the initiator, then in fact A did initiated that protocol run; similarly, B is authentic to A if it is true that, when A has completed a protocol run with believing that it has run it B as the responder, then in fact B did respond to that protocol run. For describing the aformentioned authenticity, we will use correspondence assertions [34], that can model relationships between events in form: "if an event **end**(M) has been executed, then event **begin**(M) has been previously executed" [9]. Moreover, the non-injective and injective agreements will be modeled using the correspondence assertions. Informally, non-injective agreement is used in situations where we just want to express that an event **end**(M) happened after the event **begin**(M) at least once. Non-injective agreement is defined as follows:

Definition 3. *The closed starting process P satisfies non-injective agreement with respect to $\mathcal{S}$-adversaries if and only if for any $\mathcal{S}$-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.)[8]*

The injective agreement will be used in situations where we want to express that the event **end**(M) happened after the event **begin**(M) and at most as many times as event **begin**(M). It is defined as follows:

Definition 4. *The closed starting process P satisfies injective agreement with respect to $\mathcal{S}$-adversaries if and only if for any $\mathcal{S}$-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.)[8]*

This definition corresponds to the Lowe's definition of the agreement [22] and it will be used for proving authenticity properties of the analysed protocol.

Example 2.3.1. In the example 2.1.1 we have presented the unauthenticated Diffie-Hellman key exchange. The security property that this protocol provides is the secrecy of the shared secret key. Although we already know that the protocol does not provide the participant authenticity, we will show how we could test out the authenticity of the participants. In the applied π-calculus models of the participants (from example 2.1.2) we used event constructs (i.e. $\mathbf{begin}(M)$ and $\mathbf{end}(M)$). They are indicating the following events:

- $\mathbf{begin}(s_pub_X)$ (in $node_I$) is used for recording that $node_I$ has started a protocol run with a interlocutor whose public key is s_pub_X
- $\mathbf{begin}(s_pub_X)$ (in $node_R$) is used for recording that $node_R$ has started a protocol run with a interlocutor whose public key is s_pub_X
- $\mathbf{end}(pk(s_priv_I))$ (in $node_I$) is used to record $node_I$ belief that it has ended a protocol run with the actual $node_R$; this event is emitted only if the s_pub_X from the protocol start, is in fact $node_R$ public key
- $\mathbf{end}(pk(s_priv_R))$ (in $node_R$) is used to record $node_R$ belief that it has ended a protocol run with the actual $node_I$; this event is emitted only if the s_pub_X from the protocol start, is in fact $node_I$ public key

To prove the authenticity of the participants we need to test out whether the injective agreement holds in both ways. Namely, we will be testing if $node_I$ is authentic to $node_R$, which holds if when $node_R$ has finished the protocol run with $node_I$ (which emits the event $\mathbf{end}(pk(s_priv_R))$ (in $node_R$)), then earlier $node_I$ has actually started the protocol run with the $node_R$ (which emits the event $\mathbf{begin}(s_pub_X)$ (in $node_I$)), where $pk(s_priv_R) = s_pub_X$. This will hold only if the number of times that the event $\mathbf{begin}(s_pub_X)$ (in $node_I$) has happened, is the same or less then the number of times the event $\mathbf{end}(pk(s_priv_R))$ (in $node_R$) has happened. Similarly, we will test if $node_R$ is authentic to $node_I$.

2.4 ProVerif

ProVerif is a tool for automatically analyzing the security of cryptographic protocols [9]. It is able to prove secrecy, authenticity and observational equivalence properties of the secrurity protocols [7]. Its analysis is considered on a unbounded number of sessions and unbounded message space and performed on a symbolic model of a protocol [9]. ProVerif automatically translates a protocol into an abstract model using Horn clauses [14] and determines whether the required security properties hold based on the resolution of these clauses.
The environment in which a protocol is analysed by ProVerif is assumed to be controlled by an attacker with the Dolev-Yao capabilities [13] [9]. In the ProVerif

analysis, the attacker captures the behaviour of a dishonest participant, so we only need to model the honest participants. ProVerif uses a variant of the applied π-calculus as a input language which allows us to encode cryptographic primitives and security objectives a cryptographic protocol has. Cryptography is assumed to be perfect and the attacker can successfully perform operations such as decryption only if it is in possession of the required keys [9]. The ProVerif syntax and its usage will be demonstrated in the example 2.4.1.

Example 2.4.1. We will show an example of a ProVerif program that performs the analysis of the unauthenticated Diffie-Hellman key exchange introduced in example 2.1.1. We can divide a typical ProVerif program into three parts:

- the *declarations* part, where we introduce data types and cryptographic primitives as constructors and destructors
- the *process macros* part, where we introduce the protocol sub-processes (such as the protocol participant processes)
- the *main process* part, where we define the main process that runs the protocol [9]

The declaration part for the protocol in this example starts with definition of free names used in the protocol. Firstly, we define a free name *c* of type *channel*, that will be a public channel which will be used for messages exchange:

```
0 free c:channel.
```

By default, free names in ProVerif are *public*, meaning that they are known to the attacker. If we want to define a free name that is not known by the attacker, we need to declare it as *private*. The channel *c* is assumed to be controlled by an attacker which can intercept all messages, read, modify, delete and inject new ones in the channel, so it will be declared as *public*. On the other hand, we define another free name that will be declared *private* and it will represent a secret message *m* that the protocol initiator wants to sent to the other participant:

```
0 free m:bitstring [private].
```

The type of the message *m* is bitstring, which can be considered as a random piece of data. The language used in ProVerif is strongly typed and the user can use the language native types for names (i.e. *channel* and *bitstring*), but also types that the user defines [9]. For expressing the use of the public key cryptography and symmetric encryption, we will declare new types as:

```
0 type private_key.
1 type public_key.
2 type enc_key.
```

With free names and types being declared, we can declare the cryptographic primitives used by the protocol:

```
0 fun pk(private_key): public_key.
1 fun to_enc_key(public_key): enc_key [typeConverter].
2 fun ECDH(private_key, public_key): public_key.
3 fun senc(bitstring, enc_key): bitstring.
4 reduc forall x:bitstring, k:enc_key; sdec(senc(x, k), k) = x.
```

Where:

- *pk* is a constructor that derives a public key from a private key
- *to_enc_key* is a type converter that converts a *public_key* to a *enc_key*
- *ECDH* is a constructor that combines two keys into a new public key representing a key combination as in Elliptic-Curve Diffie-Hellman key exchange
- *senc* is a constructor that represents a symmetric encryption
- *sdec* is a destructor that represents a symmetric decryption

In addition to the *ECDH* constructor, we will have to define the fundamental equality for Diffie-Hellman key exchanges [12], which is described using the following equation form:

```
0 equation forall k_1:private_key, k_2:private_key; ECDH(k_1,
    pk(k_2)) = ECDH(k_2, pk(k_1)).
```

Lastly, we will declare the event constructs that we will use in the testing out the authenticity properties of the participants:

```
0 event begin_I(public_key).
1 event end_I(public_key).
2 event begin_R(public_key).
3 event end_R(public_key).
```

In difference to the applied π-calculus syntax, we assign each event its own name:

- **begin**(s_pub_X) (in $node_I$) is $begin_R(public_key)$
- **begin**(s_pub_X) (in $node_R$) is $begin_I(public_key)$
- **end**$(pk(s_priv_I))$ (in $node_I$) is $end_I(public_key)$
- **end**$(pk(s_priv_R))$ (in $node_R$) is $end_R(public_key)$

The process macros part of the program, will contain the protocol participant processes. Process *node_I* will be the protocol initiator and *node_R* will be the responding node:

```
0 let node_I(s_priv_I:private_key, s_pub_R:public_key) =
1   out(c, pk(s_priv_I));
2   in(c, s_pub_X:public_key);
3   event begin_R(s_pub_X);
4   let k = ECDH(s_priv_I, s_pub_X) in
5   if s_pub_X = s_pub_R then
6   event end_I(pk(s_priv_I));
7   out(c, senc(m, to_enc_key(k))).
8
9 let node_R(s_priv_R:private_key, s_pub_I:public_key) =
10   in(c, s_pub_X:public_key);
11   event begin_I(s_pub_X);
12   out(c, pk(s_priv_R));
13   in(c, enc_m:bitstring);
14   let k = ECDH(s_priv_R, s_pub_X) in
15   let m' = sdec(enc_m, to_enc_key(k)) in
16   if (m = m' && s_pub_X = s_pub_I) then
17   event end_R(pk(s_priv_R)).
```

The participants models have already been presented in example 2.1.2 with a slightly different syntax, so we will not further explain them. The main process part is as follows:

```
0 process
1   new s_priv_I:private_key;
2   new s_priv_R:private_key;
3   let s_pub_I = pk(s_priv_I) in out(c, s_pub_I);
4   let s_pub_R = pk(s_priv_R) in out(c, s_pub_R);
5   ( (!node_I(s_priv_I, s_pub_R)) | (!node_R(s_priv_R,
     s_pub_I)) )
```

Similarly, as for the participants sub-processes, we have already explained the main process in example 2.1.1.
Having the protocol model presented in the ProVerif input language syntax, we can test out its security properties. We have described them already in example 2.3.1.

Firstly, we will test out the secrecy of the shared secret key. We can do that by testing out whether the attacker can obtain the message m that is encrypted with the shared secret key and exchanged in the final step of the protocol. The ProVerif code that tests out the message m secrecy is:

```
0 query attacker(m).
```

In this line we are testing whether the attacker can obtain the message m. The result of that query is:

```
0 RESULT not attacker(m[]) is true.
```

This result tells us that the attacker is not able to obtain the message m. We have successfully proven that the shared key secrecy holds.

Now we will test out the authenticity property of the protocol. As described in example 2.3.1, we need to test out whether the injective agreement between the participants hold in both ways. The ProVerif code for testing these properties is:

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

The first query is testing whether the event $end_I(x)$ has happened after and at most as many times as did the event $begin_I(x)$, where x is the $node_I$ public key. In other words we are testing whether $node_I$ has finished a protocol run with $node_R$ at most as many times as did $node_R$ respond to the protocol run initiated by $node_I$. If the result of the query is *true*, then $node_R$ is authentic to $node_I$. It is OK if $node_R$ has responded to a protocol run initiated by the $node_I$ (emitting the $begin_I(x)$ event), but because of some reason, the protocol run was never finished (the event $end_I(x)$ would then not be emitted). But if it happens that $node_I$ has finished a protocol run more times than $node_R$ has responded, it would mean that $node_I$ has done some of these protocol runs with an attacker, which would mean that $node_R$ was never authenticated by $node_I$. Similarly, the second query is testing whether the $end_R(x)$ has happened at most as many times as did the event $begin_R(x)$, which would indicate that $node_R$ has finished the protocol run with $node_I$ at most as many times as $node_I$ has started a protocol run with $node_R$. If this holds, then $node_I$ is authentic to $node_R$.

These queries give the following results:

```
0 RESULT inj-event(end_I(x_46)) ==> inj-event(begin_I(x_46))
      is false.
1 RESULT inj-event(end_R(x_47)) ==> inj-event(begin_R(x_47))
      is false.
```

The results show us that the authenticity property does not hold for any of the participants, and therefore we have proven the starting statement about the protocol being unauthenticated. Full ProVerif code used for analysing the protocol is given in appendix A.

Chapter 3

The Lightning Network

In this chapter we will present a short overview to the Lightning Network [28]. First we are going to give a short introduction in section 3.1, followed by the presentation of the building blocks that make the Lightning Network possible in section 3.2. Finally, we will present an overview of the Lightning Network given by its use cases in section 3.3.

3.1 Introduction

The Lightning Network is a second layer protocol that works on top of the Bitcoin protocol [24]. Due to its design (i.e. every transaction needs to be validated by every node in the network), the Bitcoin network can currently process about 21 transaction per second [33], while other payment networks, such as Visa are capable of handling 65000 transactions per second [31]. The main purpose of the Lightning Network is to increase the transaction throughput of the Bitcoin network. Additionally, the Lightning Network enables the users to make micro-transactions that typically could not be made on the Bitcoin network. The idea behind the Lightning Network is to use payment channels in which some of the transactions would happen offline (i.e. off-chain) and the network would only be validating settlement transactions. The Lightning Network can enable near-instant transactions, at a rate of thousands to millions per second with a fee that is a fraction of a cent [30].

3.2 The Lightning Network building blocks

To be able to understand how the Lightning Network works, we will need to introduce the concepts and technologies that make it possible. First of, we will introduce the special type of a digital signature, called *multisignature* [32]. Normally, in the digital signatures used for signing Bitcoin transactions, one person authorizes the

transfer of funds by signing the transaction using its private key. In a *multisignature*, more parties can be involved to authorize an transaction. For example, a signature for a transaction in which 4 out of 7 parties need to approve it make it valid we will call *4-of-7 multisignature*.
The second building block is the use of time locks in the Bitcoin transactions. Time locks can be used for postdating a transaction to a future point in time [3]. There are two types of time locks: CheckLockTimeVerify (CLTV) is a time lock that permits the transaction to be validated before the time specified in it; CheckSequenceVerify (CSV) is a time lock that permits the spending of funds that the transaction has transferred.
The third building block is the HashTimeLockedContract (HTLC) which is a conditional payment between two parties. The recipient can redeem funds from a HTLC if it presents its signature and a special secret value, called the payment preimage. Otherwise, the sender can cancel the HTLC after its time lock has passed.

3.3 The Lightning Network protocol

The Lightning Network protocol is in fact a collection sub-protocols and in section 1.3 we have split it into four sub-protcols. We will explain how they are being used in the Lightning Network as a whole. In a normal use of the Lightning Network, some user *A* connects to another user *B* and opens a channel to it. To connect to user *B*, user *A* needs to initiate *the key agreement* sub-protocol with user *B*. This sub-protocol will produce the session keys that will be used for encrypting all of the future communication between users *A* and *B*. To open a channel, one or both of the participants need to fund the channel. This is done with the *funding transaction*, that sends funds from a user (or from both of them) to a new address. When these funds are transferred, they are locked by a 2-of-2 multisignature, so that way both of the users have to approve the future transactions from that address. The future transactions that will be made, are called *commitment transactions* which redistributes the channel funds among the participants. Before user *A* broadcasts the *funding transaction* to the network, it needs to create the initial *commitment transaction* which will redistribute the channel funds according to how much each party intends to lock in the channel. The reason for creating the initial *commitment transaction* before broadcasting the *funding transaction*, is to make sure that if one party decides not to cooperate anymore, the other party can still redeem the funds that it has put in the channel initially. To open a channel to user *B*, user *A* will have to initiate *the channel opening* sub-protocol. In this sub-protocol they will exchange the necessary data for creating the *funding transaction* and the initial *commitment transaction*. The channel is considered open once the funding transaction is confirmed by the network.

Example 3.3.1. For the purpose of the example, we will say that user B is a coffee shop owner and user A is its customer. User A buys a cup that costs 0.001 bitcoins on a daily basis. To be able to transact fast and to avoid high transaction fees user A decides to use the Lightning Network, opens a channel to user B and funds it with 0.01 bitcoin. User A needs to connect to user B and initiate the sub-protocol for opening a channel. After the channel is open it can buy a coffee by updating the channel: user A needs to create a new commitment transaction that splits the funds in way that user A gets 0.009 bitcoins and user B gets 0.001 bitcoins. This can be done as long as there are enough funds on user A side of the channel.

This protocol seems to be working well, but we still have to rely on users being honest in a sense that none of them will not try to cheat the other party. The way the commitment transactions are made, they can be broadcast any time by any party. We could imagine a scenario where one party would try to broadcast the older commitment transaction that gets that party more funds than it should receive by the most recent commitment transaction. To prevent this, each party creates its version of the commitment transaction. User A's commitment transaction will have two outputs: the first one containing the amount user A is supposed to receive according to the last update and the second one containing the funds that user B is supposed to receive. The first output can be redeemed by user A but only after a certain time lock has passed or by user B if it provides the secret value. The second output can be redeemed by user B and it can be done immediately. User B's commitment transaction will be mirrored, one of the outputs user B can redeem after a certain time lock or user A can redeem if it provides a secret value; the other is immediately redeemable by user A. Now we have a situation where whoever broadcasts its own commitment transaction, will have to wait a certain time lock to pass to be able to redeem the funds, while the other party will have their funds immediately. Additionally, the other party has a chance to redeem the all of the channel funds if it knows the secret value. This secret value will be exchanged between the parties before every channel update and in that way potentially punish the party that tries to cheat.

Example 3.3.2. We will continue example 3.3.1 with user A being a customer and user B a coffee shop owner. After user A has purchased 7 coffees, the current channel distribution is as follows: user A is supposed to receive 0.003 bitcoins and user B is supposed to receive 0.007 bitcoins. User A now wants to cheat user B and broadcasts one of the old commitment transactions in which user A would get 0.008 bitcoins and user B would get 0.002 bitcoins. The old commitment transaction has the following outputs:

- 0.008 bitcoins:
  - redeemable by user A providing its signature but only after lock time has passed

    - redeemable by user B providing its signature and a secret value

- 0.002 bitcoins:
    - can redeemable by user B providing its signature

When user B sees that user A has broadcast the old commitment transaction, it immediately redeems its 0.002 bitcoins and the additional 0.008 bitcoins because the needed secret value was given to it, when this old commitment transaction was being revoked by a new commitment transaction. User B has now received the whole channel balance and user A was left without anything.

Besides the direct payments, the Lightning Network enables users to make payments over an intermediary nodes. Since user A is already connected and have an open channel to user B, it can send payments to user B peers (i.e. users that are connected to user B) and to peers of user B peers and so on. User B receives a fee, when acting as an intermediary in a payment that user A makes through it. The technology that makes intermediary payments possible is the HashedTimeLocked-Contract (HTLC). We can show the way they work on example 3.3.3.

Example 3.3.3. Let's say that user A has still 0.003 bitcoins left in a channel with user B and it wants to send 0.001 to user C who is user B's peer. The way it would do it is to send 0.001 bitcoin to user B, who would then send 0.001 bitcoin to user C. But here we have to rely on other users being honest, which does not has to be the case. Instead, user A will create a HTLC which will hold 0.001 bitcoin that can be redeemable in two ways: the first, user A can cancel the HTLC and take 0.001 bitcoin, but only after a specified time lock has passed; and second, user B is able to redeem 0.001 bitcoin if it knows a secret value x that only user A knows. User A will send this HTLC to user B. User B will then do the same and create its own HTLC that will also hold 0.001 bitcoin and it will be redeemable in two ways: the first, user B can cancel the HTLC and take 0.001 bitcoin, but only after a specified time lock has passed; and second, user C is able to redeem 0.001 bitcoin if it knows a secret value x. User B will send this HTLC to user C. Only thing left is for user A to send the secret x to user C who will then be able to redeem funds from user B, who will then be able to redeem funds from user A.
Additionally, to incentivize user B to go through the effort of intermediating in the payment, user A can offer him an HTCL with 0.0011 bitcoins locked in. So when user C redeems 0.001 bitcoin from the HTLC user B has offered, user B will take 0.0011 bitcoin from user A's HTLC and make a profit of 0.0001 bitcoin.

In practice, HTLCs are sent using *the onion routing* sub-protocol. The payment sender creates a payment route. The payment route is created using the local routing table of the payment sender. Every node keeps track of the messages exchanged in the network, such as a new node or channel announcement and maintains its

local view of the network's topology [19]. When the route is found, the payment sender can construct the onion packet which will be constructed in such a way that every node in the route can only know who was the previous node that sent it and where should the packet go next. Each node along the payment route will be able to decrypt the outer layer of the packet and obtain the payment information. This information is necessary for their own HTLC construction. When the packet reaches the destination, every node along the route can redeem the funds from the HTLC that is offered by the previous node in the route.
Finally, user A can initiate *the close channel* sub-protocol to close the channel and redeem its funds whenever it wants. It can do so, by broadcasting the most recent commitment transaction. But in that case it will have to wait for a time lock to pass to redeem its funds. Alternative is to cooperate with user B to mutually close the channel. Since the channel funds are locked on a multisignature address on which they both need to approve the transfer of funds, they can create a new closing transaction which distributes funds according to the latest commitment transaction, but without any time locks. The closing transaction can then be broadcast to the network and both parties can redeem their respective funds immediately.

Chapter 4

The Lightning Network key agreement protocol

In this chapter we are presenting the first sub-protocol of the Lightning Network: *the key agreement protocol*. We will describe the security properties that the protocol needs to satisfy in section 4.2 followed by the protocol signature and informal description in section 4.3 and 4.4 respectively. In section 4.5 we will present a model of the protocol using the applied π-calculus, which we will use to write the ProVerif compatible code in section 4.6. Finally, we will analyse and discuss the results produced by ProVerif in section 4.7.

4.1 A brief introduction to the protocol

The Lightning Network key agreement protocol is intended to provide a mutual authentication between two participants (i.e. nodes) and to create secret session keys (asymmetric encryption key-pair: ek_I and dk_I for the protocol initiator and ek_R and dk_R for the protocol responder). The newly generated session keys will be used for encrypting all future messages between the participants and authenticating any information advertised on behalf of a participant.

4.2 Properties

We can summarize the security properties that the protocol must satisfy into a following list:

- the initiating node must be authentic to the responding node
- the responding node must be authentic to the initiating node
- the secrecy of the generated secret session keys ek_I and dk_I must hold

- the secrecy of the generated secret session keys ek_R and dk_R must hold

The authentication property of the initiating node to the responding node means that, if the responding node reaches the end of the protocol with a belief that it has done so with a initiating node, then the initiating node has actually engaged a session with the responding node. Similarly for the authentication of the responding node to the initiating node. The secrecy of a secret session key ek_I and dk_I implies that if the initiating node has reached the end of the protocol with the responding node, then the keys ek_I and dk_I, generated at the initiating node side as a result of the protocol, are secret and can be used for encrypting and decrypting future communication with the responding node. Similarly for the secrecy of the secret session keys ek_R and dk_R that is generated at the responding node side.

4.3 Principals and signature

The protocol will have the following *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 key, etc).

In the signature ($\mathcal{S}$, Σ), the function symbols are: $\Sigma = \{\bot$, pk, $hash$, $ECDH$, $HKDF$, $senc$, $sdec$, $[\,]$, $ith_i\}$ and their definitions are given in the table 4.1.

$\bot$	:	$\rightarrow term$	$HKDF$	:	$key \times term \rightarrow term$
$ECDH$	:	$key \times key \rightarrow key$	$hash$	:	$term \rightarrow term$
$senc$	:	$term \times key \rightarrow term$	$[\,]$	:	$term \times term \rightarrow term$
$sdec$	:	$term \times key \rightarrow term$	ith_i	:	$term \rightarrow term$
pk	:	$key \rightarrow key$			

Table 4.1: Cryptographic primitives

Where:

- $\bot$ represents a null or empty data
- $ECDH(k_1, k_2)$ is an abstraction of the Elliptic Curve Diffie-Hellman[12] 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 data d with a symmetric key k and produces its plaintext version

- $pk(k)$ is a public key generation function that takes a private key k and returns its public key pair
- $HKDF(k, x)$ represents a Hashed Message Authentication Code (HMAC)-based key derivation function [16], 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 $[x_1, x_2, ..., x_{n-1}, x_n]$ for the expression $[x_1, [x_2, [..., [x_{n-1}, x_n]]]]$
- $ith_i(M)$ function that returns i-th element of a tuple M

We will also extend the applied π-calculus reduction system with additional reduction rules for the destructors we will use:

$$\begin{array}{rl} \text{(Red Decrypt)} & sdec(senc(M, N), N) \rightarrow M \\ \text{(Red Ith}_\text{i}\text{)} & ith_i([M_1, ..., M_n]) \rightarrow M_i \end{array}$$

4.4 Informal description

The handshake chosen for the authenticated key exchange in the Lightning Network is Noise_XK [20]. Noise_XK is one of twelve fundamental interactive handshake patterns of the Noise Protocol Framework [27]. The full name of the protocol is Noise_XK_secp256k1_ChaChaPoly_SHA256 which specifies what elliptic curve is being used (secp256k1), what Authenticated Encryption with Associated Data (AEAD) algorithm is being used (ChaCha with Poly1305 [25]) and which hashing function is being used (SHA256).
Throughout the handshake process, each side maintains the following variables:

- *e*: a freshly generated ephemeral key-pair; *e_priv* is the private and *e_pub* is the public component
- *s*: a static key-pair; *s_priv* is the private and *s_pub* is the public component
- *ck*: the chaining key which is an accumulated hash of all previous ECDH outputs and is used to derive the encryption keys at the end of the protocol
- *h*: the handshake hash which is an accumulated hash of all handshake data sent and received; it is never transmitted, but used as the Associated Data in the AEAD messages

- $temp_k_1$, $temp_k_2$, $temp_k_3$: the intermediate keys which are used for encrypting messages during the handshake [20].

We will consider the handshake state as a tuple of variables that each side maintains:

- initiating node: $[e_priv_I, e_pub_I, s_priv_I, s_pub_I, ck_I, h_I, temp_k_{1I}, temp_k_{2I}, temp_k3I]$
- responding node: $[e_priv_R, e_pub_R, s_priv_R, s_pub_R, ck_R, h_R, temp_k_{1R}, temp_k_{2R}, temp_k3R]$

In table 4.2 we can see the message sequence of the protocol.

Message 0	$node_R$	$\rightarrow$	$node_I$	:	s_pub_R
Message 1	$node_I$	$\rightarrow$	$node_R$	:	$[e_pub_I, senc(\perp, [temp_k_1, 0, h_1])]$
Message 2	$node_R$	$\rightarrow$	$node_I$	:	$[e_pub_R, senc(\perp, [temp_k_2, 0, h_2])]$
Message 3	$node_I$	$\rightarrow$	$node_R$	:	$[senc(s_pub_I, [temp_k_2, 1, h_3]), senc(\perp, [temp_k_3, 0, h_4])]$

Table 4.2: The key agreement protocol message sequence

The protocol has three phases but before the phase one, both parties initialize their handshake state, namely, *ck* and *h*:

$$ck_I = hash(protocol_name)$$

$$h_I = hash([protocol_name, prologue])$$

$$ck_R = hash(protocol_name)$$

$$h_R = hash([protocol_name, prologue])$$

where *protocol_name* is "Noise_XK_secp256k1_ChaChaPoly_SHA256" encoded as an ASCII string, *prologue* is "lightning" also encoded as an ASCII string. The initiating node acquires the responding node static public key (s_pub_R) and updates the handshake hash:

$$h_I = hash([h, s_pub_R])$$

The responding node does the same and updates the handshake hash with its local static public key:

$$h_R = hash([h, s_pub_R])$$

Act One

In the first phase, a message is sent from the initiating node to the responding node. The initiating node (sender) actions:

1. generates a new ephemeral key pair (e_priv_I and e_pub_I)
2. updates the handshake hash with previously generated ephemeral public key:

$$h_I = hash([h_I, e_pub_I])$$

3. performs Elliptic Curve Diffie Hellman with ephemeral private key and the responding nodes static public key

$$es = ECDH(e_priv_I, s_pub_I)$$

4. set $temp_k_{1I}$ to the first and new ck_I to the second output of the HKDF function:

$$temp_k_{1I} = ith_0(HKDF(ck_I, es))$$
$$ck_I = ith_1(HKDF(ck_I, es))$$

5. encrypt an empty text with the current handshake state as the key:

$$enc_d1 = senc(\perp, [temp_k_{1I}, 0, h])$$

6. update the handshake hash:

$$h_I = hash([h_I, enc_d1])$$

7. send m to the responding node:

$$m = [e_pub_I, enc_d1]$$

The responding node (receiver) actions:

1. receive m and extract e_pub_I and enc_d1 from it
2. update the handshake hash with the received ephemeral public key:

$$h_R = hash([h_R, e_pub_I])$$

3. perform ECDH with the received ephemeral public key and its own static private key (due to the ECDH, the resulting value will be equal to the one computed by the initiating node in its own step 3)

$$es = ECDH(e_pub_I, s_priv_R)$$

4. set $temp_k_{1R}$ to the first and new ck_R to the second output of the HKDF function:

$$temp_k_{1R} = ith_0(HKDF(ck_R, es))$$
$$ck_R = ith_1(HKDF(ck_R, es))$$

5. decrypt the received encrypted payload:

$$d1 = sdec(enc_d1, [temp_k_{1R}, 0, h_R])$$

6. if d1 is in fact empty text, then proceed
7. update the handshake hash with the received encrypted payload:

$$h_R = hash([h_R, enc_d1])$$

Act Two

In the second phase, a message is sent from the responding node to the initiating node. The responding node (sender) actions:

1. generates a new ephemeral key pair (e_priv_R and e_pub_R)
2. updates the handshake hash with previously generated ephemeral public key:

$$h_R = hash([h_R, e_pub_R])$$

3. performs Elliptic Curve Diffie Hellman with ephemeral private key and the received ephemeral static public key from the Act One

$$ee = ECDH(e_priv_R, e_pub_I)$$

4. set $temp_k_{2R}$ to the first and new ck_R to the second output of the HKDF function:

$$temp_k_{2R} = ith_0(HKDF(ck_R, ee))$$

$$ck_R = ith_1(HKDF(ck_R, ee))$$

5. encrypt an empty text with the current handshake state as the key:

$$enc_d2 = senc(\perp, [temp_k_{2R}, 0, h_R])$$

6. update the handshake hash:

$$h_R = hash([h_R, enc_d2])$$

7. send $m2$ to the initiating node:

$$m2 = [e_pub_R, enc_d2]$$

The initiating node (receiver) actions:

1. receive $m2$ and extract e_pub_R and enc_d2 from it
2. update the handshake hash with the received ephemeral public key:

$$h_I = hash([h_I, e_pub_R])$$

3. perform ECDH with the received ephemeral public key and its own ephemeral public key

$$ee = ECDH(e_pub_R, e_priv_I)$$

4. set $temp_k_{2I}$ to the first and new ck_I to the second output of the HKDF function:

$$temp_k_{2I} = ith_0(HKDF(ck_I, ee))$$

$$ck_I = ith_1(HKDF(ck_I, ee))$$

5. decrypt the received encrypted payload:

$$d2 = sdec(enc_d2, [temp_k_{2I}, 0, h_I])$$

6. if d2 is in fact empty text, then proceed

7. update the handshake hash with the received encrypted payload:

$$h_I = hash([h_I, enc_d2])$$

Act Three

In the third and final phase, a message is sent from the initiating node to the responding node as a concluding step. The initiating node (sender) actions:

1. encrypt its static public key:

$$enc_d3 = senc(s_pub_I, [temp_k_{2I}, 1, h_I)$$

2. update the handshake hash:

$$h_I = hash([h_I, enc_d3])$$

3. perform ECDH with its own static private key and previously received ephemeral public key of the responder

$$se = ECDH(s_priv_I, e_pub_R)$$

4. set $temp_k_{3I}$ to the first and new ck_I to the second output of the HKDF function:

$$temp_k_{3I} = ith_0(HKDF(ck_I, se))$$

$$ck_I = ith_1(HKDF(ck_I, se))$$

5. encrypt an empty text with the current handshake state as a key:

$$enc_d4 = senc(\perp, [temp_k_{3I}, 0, h_I])$$

6. derive the session keys:

$$ek_I = ith_0(HKDF(ck_I, \perp))$$

$$dk_I = ith_1(HKDF(ck_I, \perp))$$

7. send $m3$ to the responding node:

$$m3 = [enc_d3, enc_d4]$$

The responding node (receiver) actions:

1. receive $m3$ and extract enc_d3 and enc_d4:
2. decrypt enc_d3 to obtain the initiating node public static key:

$$s_pub_I = sdec(enc_d3, [temp_k_{2R}, 1, h_R])$$

3. update the handshake state:

$$h_R = hash([h_R, enc_d3])$$

4. perform ECDH with its own ephemeral private key and the received public key

$$se = ECDH(e_priv_R, s_pub_I)$$

5. set $temp_k_{3R}$ to the first and new ck_R to the second output of the HKDF function:

$$temp_k_{3R} = ith_0(HKDF(ck_R, se))$$
$$ck_R = ith_1(HKDF(ck_R, se))$$

6. decrypt enc_d4:

$$d4 = sdec(enc_d4, [temp_k_{3R}, 0, h_R])$$

7. if $d4$ is in fact empty text, then proceed
8. derive the session keys:

$$ek_R = ith_0(HKDF(ck_R, \perp))$$

$$dk_R = ith_1(HKDF(ck_R, \perp))$$

At the conclusion of the Act Three, both sides have derived the session keys and they will use these keys for encrypting all of the future communication [20].

4.5 The applied π-calculus version of the protocol

To further explain the protocol, we will model the protocol in the applied π-calculus in this section. $node_I$ will be the initiating node and $node_R$ will be responding node. We will skip the initialization of the handshake state (namely, the handshake hash and the chaining key). The messages will be exchanged through a public channel c. $node_I$'s process is:

$$
\begin{aligned}
node_I(s_priv_I, s_pub_R, h_0, ck_0) = & (\nu e_priv_I)c(s_pub_X).\mathbf{begin}(s_pub_X). \\
& \mathbf{let}\ temp_k_1 = ith_0(HKDF(ck_0, es))\ \mathbf{in} \\
& \mathbf{let}\ ck_1 = ith_1(HKDF(ck_0, es))\ \mathbf{in} \\
& \bar{c}\langle [pk(e_priv_I), d_1] \rangle. \\
& c(m_2).\mathbf{let}\ e_pub_R = ith_0(m_2)\ \mathbf{in} \\
& \mathbf{let}\ temp_k_2 = ith_0(HKDF(ck_1, ee))\ \mathbf{in} \\
& \mathbf{let}\ ck_2 = ith_1(HKDF(ck_1, ee))\ \mathbf{in} \\
& \mathbf{let}\ d_2 = ith_1(m_2)\ \mathbf{in} \\
& \mathbf{let}\ temp_k_3 = ith_0(HKDF(ck_2, se))\ \mathbf{in} \\
& \mathbf{let}\ ck_3 = ith_1(HKDF(ck_2, se))\ \mathbf{in} \\
& \mathbf{let}\ dd_2 = sdec(d_2, [temp_k_2, 0, h_3])\ \mathbf{in} \\
& \mathbf{if}\ dd_2 = \perp\ \mathbf{then}\ \bar{c}\langle m_3 \rangle. \\
& \mathbf{if}\ s_pub_X = s_pub_R\ \mathbf{then}\ \mathbf{end}(pk(s_priv_I))
\end{aligned}
$$

$$
\text{where}
\begin{cases}
h_1 = hash([h_0, pk(e_priv_I)]) \\
es = ECDH(e_priv_I, s_pub_X) \\
d_1 = senc(\perp, [temp_k_1, 0, h_1]) \\
h_2 = hash([h_1, d_1]) \\
h_3 = hash([h_2, e_pub_R]) \\
ee = ECDH(e_priv_I, e_pub_R) \\
h_4 = hash([h_3, d_2]) \\
d_3 = senc(pk(s_priv_I), [temp_k_2, 1, h_4]) \\
h_5 = hash([h_4, d_3]) \\
se = ECDH(s_priv_I, e_pub_R) \\
ek_I, dk_I = HKDF(ck_4, \perp) \\
m_3 = [d_3, d_4]
\end{cases}
$$

$node_I$'s process follows the steps described in section 4.4. At the start of the process, the environment provides the public key (s_pub_X) of the responding node. This is important because we want to enable the process to start a protocol run with even with the attacker and it will be used in proving the authenticity property. The events used in the initiating node process are:

- **begin**(s_pub_X) which marks the event of $node_I$ starting a protocol run with an interlocutor whose public key is s_pub_X
- **end**(s_pub_I) which marks the $node_I$'s belief that it has successfully completed a protocol run with $node_R$ (the actual responding node) and it is emitted only if the s_pub_X from the beginning is equal to the $node_R$'s static public key s_pub_R

The responding node ($node_R$) is:

$$
\begin{aligned}
node_R(s_priv_R, s_pub_I, h_0, ck_0) = &(\nu e_priv_R) c(m_1).\\
&\textbf{let } e_pub_X = ith_0(m_1) \textbf{ in}\\
&\textbf{let } d_1 = ith_1(m_2) \textbf{ in}\\
&\textbf{let } temp_k_1 = ith_0(HKDF(ck_0, se)) \textbf{ in}\\
&\textbf{let } ck_1 = ith_1(HKDF(ck_0, se)) \textbf{ in}\\
&\textbf{let } dd_1 = sdec(d_1, [temp_k_1, 0, h_1]) \textbf{ in}\\
&\textbf{if } dd_1 = \bot \textbf{ then}\\
&\textbf{let } temp_k_2 = ith_0(HKDF(ck_1, ee)) \textbf{ in}\\
&\textbf{let } ck_2 = ith_1(HKDF(ck_1, ee)) \textbf{ in}\\
&\overline{c}\langle [pk(e_priv_R), d_2] \rangle . c(m_3).\\
&\textbf{let } d_3 = ith_0(m_3) \textbf{ in}\\
&\textbf{let } d_4 = ith_1(m_3) \textbf{ in}\\
&\textbf{let } s_pub_X = sdec(d_3, [temp_k_2, 1, h_4]) \textbf{ in begin}(s_pub_X).\\
&\textbf{let } temp_k_3 = ith_0(HKDF(ck_2, es)) \textbf{ in}\\
&\textbf{let } ck_3 = ith_1(HKDF(ck_2, es)) \textbf{ in}\\
&\textbf{let } dd_4 = sdec(d_4, [temp_k_3, 0, h_5]) \textbf{ in}\\
&\textbf{if } dd_4 = \bot \textbf{ then}\\
&\textbf{if } s_pub_X = s_pub_I \textbf{ then end}(pk(s_priv_R))
\end{aligned}
$$

$$
\text{where} \begin{cases}
h_1 = hash([h_0, e_pub_X])\\
se = ECDH(s_priv_R, e_pub_X)\\
temp_k_1, ck_1 = HKDF(ck_0, se)\\
h_2 = hash([h_1, d_1])\\
h_3 = hash([h_2, pk(e_priv_R)])\\
ee = ECDH(e_priv_R, e_pub_X)\\
temp_k_2, ck_2 = HKDF(ck_1, ee)\\
d_2 = senc(\bot, [temp_k_2, 0, h_3])\\
h_4 = hash([h_3, d_2])\\
h_5 = hash([h_4, d_3])\\
es = ECDH(e_priv_R, s_pub_X)\\
temp_k_3, ck_3 = HKDF(ck_2, es)\\
dk_R, ek_R = HKDF(ck_3, \bot)
\end{cases}
$$

$node_R$'s process is following the steps described in section 4.4. Similarly, as for the initiating process, we use events:

- **begin**(s_pub_X) which marks the event of $node_R$ receiving s_pub_X from some interlocutor that initiated a protocol run
- **end**(s_pub_R) which marks the $node_R$'s belief that it has successfully completed a protocol run with $node_I$ (the actual initiating node) and it is emitted only if the received s_pub_X is equal to the $node_I$'s static public key s_pub_I

The main process that will start the initiating and responding nodes processes is:

$$\begin{aligned} P =& (\nu s_priv_I, s_priv_R, h_0, ck_0) \\ & \quad \textbf{let } s_pub_I = pk(s_priv_I) \textbf{ in } \bar{c}\langle s_pub_I \rangle. \\ & \quad \textbf{let } s_pub_R = pk(s_priv_R) \textbf{ in } \bar{c}\langle s_pub_R \rangle. \\ & \quad ((!node_I(s_priv_I, s_pub_R, h_0, ck_0)) | (!node_R(s_priv_R, s_pub_I, h_0, ck_0))) \end{aligned}$$

The participants static public keys are outputted to the public channel to make sure that the attacker receives them.

4.6 From π to Proverif

In this section we will present the model in a version of the applied π-calculus that ProVerif uses. Additionally, the protocol will be generalized by allowing the environment to decide which process ($node_I$ and $node_R$) will be the initiator and the responder. We are also introducing two new subtypes of a key to make the code more understandable. *public_key* represent the node ID and *private_key* is its private pair. The ephemeral keys will also be represented with *public_key* and *private_key* types. *bitstring* and *bool* are language standard types and they will be used for representing different sorts of data and boolean algebra values respectively. We will also specify the protocol public constants:

- *prologue* representing an ASCII-encoded text "lightning"
- *protocol_name* representing an ASCII-encoded text "Noise_XK_secp256k1_ChaChaPoly_SHA256"
- *empty* representing an empty text
- *int_zero* and *int_one* representing integers 0 and 1

Cryptographic primitives and functions used in the model are presented in the table 4.3.

$$\begin{aligned}
pk &: private_key \rightarrow public_key \\
ECDH &: private_key \times public_key \rightarrow public_key \\
senc &: bitstring \times bitstring \rightarrow bitstring \\
sdec &: bitstring \times bitstring \rightarrow bitstring \\
hash &: bitstring \rightarrow bitstring \\
HKDF_o1 &: bitstring \times bitstring \rightarrow bitstring \\
HKDF_o2 &: bitstring \times bitstring \rightarrow bitstring \\
eq &: bitstring \times bitstring \rightarrow bool
\end{aligned}$$

Table 4.3: Cryptographic primitives and functions

Where:

- $pk(k)$ computes a public from a private key
- $ECDH(k_1, k_2)$ computes a new public key combining a private key k_1 and a public key k_2
- $senc(d, k)$ encrypts data d using the key k
- $sdec(d, k)$ decrypts data d using the key k
- $hash(x)$ computes a hash value of x
- $HKDF_o1(ck, k)$ returns the first key that is derived from combining two *bitstring*s (typically it will be a chaining key and a public key)
- $HKDF_o1(ck, k)$ similarly as the above but returns the second key that is derived
- $eq(x, y)$ returns *true* if x is equal to y

We did not created a new type for encryption keys, but rather allowed that a key can be any *bitstring*. The reason for this is to enable use of tuples as keys, since we will need to combine a key with the current handshake state in encryption.
The rest of the ProVerif implementation are the subprocesses and the main processes definitions. The only difference to the model introduced in section 4.5, is that we are letting the environment to decide on which static public keys the initiating and responding node processes are going to use. This is done for making the analysis more general by removing an undesirable restriction on the participants, namely that the initating node must use s_pub_I and the responding node must use s_pub_R as their static public keys. The full version of the ProVerif code can be found in Appendix B.

4.7 Analysing the protocol in ProVerif

In section 4.2 we have presented the security properties the protocol must have. To recall, the protocol must provide:

- that the initiating node is authentic to the responding node
- that the responding node is authentic to the initiating node
- the secrecy of the generated secret session keys ek_I and dk_I
- the secrecy of the generated secret session keys ek_R and dk_R

In this section we will test whether these properties actually hold for the protocol. But before we test out the properties, we will present the attacker capabilities.

4.7.1 Attacker capabilities

We will consider a Dolev-Yao attacker as described in section 2.2. The attacker capabilities are expressed in the inference system from table 4.4.

(Inf Null)	(Inf ECDH)	(Inf Senc)
$\frac{}{\perp}$	$\frac{M \ N}{ECDH(M,N)}$	$\frac{M \ N}{senc(M,N)}$
(Inf Sdec)	(Inf Hash)	(Inf Pk)
$\frac{sdec(senc(M,N),N)}{M}$	$\frac{M}{hash(M)}$	$\frac{M}{pk(M)}$
(Inf HKDF)	(Inf Tup)	(Inf Elem)
$\frac{M \ N}{HKDF(M,N)}$	$\frac{M_1 \ ... \ M_n}{[M_1,...,M_n]}$	$\frac{ith_i[M_1,...,M_n]}{M_1}$

Table 4.4: The inference system

The rules of the inference system enables the attacker to:

- build and use the null ($\perp$) data (Inf Null)

- perform an Elliptic Curve Diffie Hellman between two keys (Inf ECDH)
- encrypt a term (Inf Senc)
- decrypt a term, if it knows the correct encryption key (Inf Sdec)
- compute a hash from a term (Inf Hash)
- derive a public key from a private key (Inf Pk)
- compute HKDF output (Inf HKDF)
- build a tuple of an arbitrary size (Inf Tup)
- retrieve elements of a tuple (Inf Elem)

Additionally, the attacker will know all of the public constants that we use in the protocol as well as, the participants public keys.

4.7.2 Analysis of security properties

The part of the code that will test whether the aformentioned properties hold is:

```
0 query attacker(secret_ek_I);
1      attacker(secret_dk_I);
2      attacker(secret_ek_R);
3      attacker(secret_dk_R).
4
5 query x:public_key, y:public_key; inj-event(end_I(x,y)) ==>
    inj-event(begin_I(x,y)).
6 query x:public_key, y:public_key; inj-event(end_R(x,y)) ==>
    inj-event(begin_R(x,y)).
```

secret_ek_I, *secret_dk_I*, *secret_ek_R* and *secret_dk_R* are free names, not available to the attacker which will be 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. The queries for testing whether the injective agreement holds in both way, work as described in section 2.3 where the first one is testing is the responding node authentic to the initiating node, while the second one is testing whether the initiating node is authentic to the responding node.
The results that ProVerif outputs are following:

```
0 RESULT not attacker(secret_ek_I[]) is true.
1 RESULT not attacker(secret_dk_I[]) is true.
2 RESULT not attacker(secret_ek_R[]) is true.
3 RESULT not attacker(secret_dk_R[]) is true.
```

```
4 RESULT inj-event(end_I(x_229,y_230)) ==>
    inj-event(begin_I(x_229,y_230)) is false.
5 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, we see that the responding node is not 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. The second result proves that the initiating node is authentic to the responding node.

4.7.3 Additional analysis of the responder authentication failure

ProVerif provides us with steps that an attacker needs to take to make the responder authentication fail. The part of the ProVerif output that is analysing this attack is given in appendix C, and here we are going to give a short summary of it. We are going to reference the lines of the ProVerif code that represents the model of the protocol which is listed in appendix B. The attack is performed as follows:

1. the attacker obtains $pk(real_s_priv_I)$ and $pk(real_s_priv_R)$ from the public channel, that are output (lines 194 and 195)
2. from the obtained public keys, the attacker creates a tuple: $[pk(real_s_priv_I), pk(real_s_priv_R)]$
3. the attacker sends a message to the initiating node containing the tuple from step 2 (line 75)
4. the initiating node is sending back a message $m1$ containing its ephemeral public key together with the message MAC(Message Authentication Code), denoted as enc_d1: $m1 = [pk(e_priv_I), enc_d1]$ (line 97)
5. the attacker receives the message $m1$ and extracts its elements to obtain $pk(e_priv_I)$ and enc_d1
6. the attacker now sends to the responding node, the static public key that belongs to the initiating node which was obtained in step 1, which the responding node will use as its own (this is due to the generalization described in section 4.6) (line 134)
7. the attacker now sends the ephemeral public key that the initiating node uses which was obtained in step 5, to the responding node (line 147)

8. the responding node sends the message $m2$ to the attacker containing $pk(e_priv_R)$ and enc_d2 (where enc_d2 is the MAC of the message) (line 166)

9. the attacker receives the message $m2$ and extracts $pk(e_priv_R)$ and enc_d2 from it

10. the attacker sends a message to the initiating node containing a tuple: $[pk(e_priv_R), enc_d2]$

11. the initiating node receives the message (line 100), finishes the rest of the protocol and finally sends a message $m3$ which contains its static public key and emits the event $end_I(s_pub_I, s_pub_Y)$ where s_pub_Y is the other party static public key (line 121)

12. the attacker can intercept the message $m3$ and the responding node would never finish the protocol run (while on the other side, the initiating node has finished a protocol run); this would also mean that due to the fact that the responding node has never received the static public key that the initiating node is using, it would never emit $begin_I(s_pub_X, s_pub_R)$ where s_pub_X is the other party static public key (line 172)

The definition of the injective agreement states that the event **end**(M) must have happened after the event **begin**(M) and at most as many times as event **begin**(M). In the described scenario, the event **end**(M) happens, while the event **begin**(M) does not happen at all, which not only breaks the injective agreement, but also it breaks the non-injective agreement. The initiating node was tricked into communicating and finishing a protocol run with the attacker while thinking that it is communicating with the responding node it wanted to run the protocol with. Although it seems like the attacker has taken advantage of the fact that it can assign the static public keys to the participants, and in step 6 it has assigned the responding node with a static public key that the initiating node is using. But the result would not change even if we would restrict the participants to use different static public keys: the attacker would still be able to receive the message $m2$ that the responding node is sending in line 165. We have implemented a less general version of the protocol in which have checked if we would get the same results and we did, but we will not reference that implementation, since it was just a sanity check.

Chapter 5

The Lightning Network channel opening protocol

In this chapter we are presenting the second sub-protocol of the Lightning Network: *the channel opening protocol*. We will describe the security properties that the protocol needs to satisfy in section 5.2 followed by the protocol signature and informal description in section 5.3 and 5.4 respectively. In section 5.5 we will present a model of the protocol using the applied π-calculus, which we will use to write the ProVerif compatible code in section 5.6. Finally, we will analyse and discuss the results produced by ProVerif in section 5.7.

5.1 A brief introduction to the 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 aformentioned transactions. The channel opening protocol [17], consists of several messages being exchanged and we can see them in table 5.1. Table 5.1 gives us a simplified overview of the protocol where we see what messages are being exchanged. More detailed message sequence diagram will be presented in section 5.4.

Message 1	$node_I$	$\rightarrow$	$node_R$	:	$open_channel$
Message 2	$node_R$	$\rightarrow$	$node_I$	:	$accept_channel$
Message 3	$node_I$	$\rightarrow$	$node_R$	:	$funding_created$
Message 4	$node_R$	$\rightarrow$	$node_I$	:	$funding_signed$
Message 5	$node_I$	$\rightarrow$	$node_R$	:	$funding_locked$
Message 6	$node_R$	$\rightarrow$	$node_I$	:	$funding_locked$

Table 5.1: Messages exchanged in the channel opening protocol [17]

Messages exchanged in this protocol will be encrypted using the session keys that the users established in the key agreement protocol.

5.2 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

5.3 Principals and signature

The protocol will have the following *signature* ($\mathcal{S}$, Σ). $\mathcal{S}$ contains two sorts: $\mathcal{S} = \{term, key\}$. The sort *term* will be used for all of the exchanged data (e.g. messages, encrypted data, signatures, etc), while the sort *key* will be used for all types of keys used (e.g. public and private keys, encryption key, etc).

In the signature ($\mathcal{S}$, Σ), the function symbols are: $\Sigma = \{\perp$, *pk*, *sign*, *getmess*, *checksig*, *senc*, *sdec*, $[\,]$, $ith_i\}$ and their definitions are given in the table 5.2.

$\perp$	:	$\rightarrow term$	$getmess$	:	$key \times term \rightarrow term$
pk	:	$key \times key \rightarrow key$	$checksig$	:	$term \rightarrow term$
$senc$	:	$term \times key \rightarrow term$	$[\,]$	:	$term \times term \rightarrow term$
$sdec$	:	$term \times key \rightarrow term$	ith_i	:	$term \rightarrow term$
$sign$	:	$key \rightarrow key$			

Table 5.2: Cryptographic primitives

- $\perp$ represents a null or empty data
- $pk(k)$ is a public key generation function that takes a private key k and returns its public key pair
- $senc(d, k)$ encrypts a data d with a symmetric key k and produces a ciphertext
- $sdec(d, k)$ decrypts a data d with a symmetric key k and produces its plaintext version
- $sign(d, k)$ signs a data d with a key k
- $getmess(s)$ returns the data that is being signed with the signature s
- $checksig(s, k)$ returns true, if the signature s was made using a private key that corresponds to the public key k
- $[x, y]$ is a tuple creation function in which we also consider the short form $[x_1, x_2, ..., x_{n-1}, x_n]$ for the expression $[x_1, [x_2, [..., [x_{n-1}, x_n]]]]$
- $ith_i(M)$ function that returns i-th element of a tuple M

We will also extend the applied π-calculus reduction system with additional reduction rules for the destructors we will use:

(Red Decrypt)	$sdec(senc(M, N), N) \rightarrow M$
(Red Getmess)	$getmess(sign(M, N), N) \rightarrow M$
(Red Checksig)	$checksig(sign(M, N), pk(N)) \rightarrow true$
(Red Ith_i)	$ith_i([M_1, ..., M_n]) \rightarrow M_i$

5.4 Informal 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. The more detailed description of this mechanism was already presented in chapter 3.
In order to create the aformentioned 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 of high importance 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 $s_pub_R = pk(s_priv_R)$
- $funding_locked$ 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 5.3.

$open_channel$	$node_I$	$\rightarrow$	$node_R$	:	$senc([s_pub_I, data_I], ek_I)$
$accept_channel$	$node_R$	$\rightarrow$	$node_I$	:	$senc([s_pub_R, data_R], ek_R)$
$funding_created$	$node_I$	$\rightarrow$	$node_R$	:	$senc(sign([data_I, data_R], s_priv_I), ek_I)$
$funding_signed$	$node_R$	$\rightarrow$	$node_I$	:	$senc(sign([data_I, data_R], s_priv_R), ek_R)$
$funding_locked$	$node_I$	$\rightarrow$	$node_R$	:	$senc(\perp, ek_I)$
$funding_locked$	$node_R$	$\rightarrow$	$node_I$	:	$senc(\perp, ek_R)$

Table 5.3: The channel opening protocol message sequence diagram

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.

5.5 The applied π-calculus version of the protocol

We will now model the protocol in the applied π-caculus. $node_I$ will be the initiating node and $node_R$ will be responding node. The initiating node process is:

$$
\begin{aligned}
node_I(s_priv_I, s_pub_R, ek_I, dk_I) = &(\nu data_I)\bar{c}\langle enc_m_1\rangle.\\
&c(enc_m_2).\mathbf{let}\ m_2 = sdec(enc_m_2, dk_I)\ \mathbf{in}\\
&\mathbf{let}\ s_pub_X = ith_0(m_2)\ \mathbf{in}\\
&\mathbf{let}\ data_X = ith_1(m_2)\ \mathbf{in}\ \mathbf{begin}(s_pub_X).\\
&\bar{c}\langle enc_m_3\rangle.c(enc_m_4).\\
&\mathbf{let}\ sig_X = sdec(enc_m_4, dk_I)\ \mathbf{in}\\
&\mathbf{let}\ t = getmess(sig_X)\ \mathbf{in}\\
&\mathbf{let}\ b_2 = checksig(sig_X, s_pub_X)\ \mathbf{in}\\
&\mathbf{if}\ b = true\ \mathbf{then}\ \bar{c}\langle enc_m_5\rangle.\\
&\mathbf{if}\ s_pub_X = s_pub_R\ \mathbf{then}\ \mathbf{end}(pk(s_priv_I))
\end{aligned}
$$

$$
\text{where}
\begin{cases}
enc_m_1 = senc([pk(s_priv_I), data_I], ek_I)\\
sig = sign([data_I, data_X], s_p riv_I)\\
enc_m_3 = senc(sig, ek_I)\\
b_1 = (t = (data_I, data_X))b = b_1 \&\& b_2\\
enc_m_5 = senc(\perp, ek_I)
\end{cases}
$$

The initiating node process follows the protocol description from section 5.4. The data received from the responding node is labeled with the letter X, due to the fact that the attacker can communicate with the node. In the last line, we will test if the interlocutor was in fact the responding node $node_R$ and if it was, then we will emit the event marking the end of the protocol run with the right node.
The responding node process is:

$$
\begin{aligned}
node_R(s_priv_R, s_pub_I, ek_R, dk_R) =& (\nu data_R) c(enc_m_1).\\
& \textbf{let } m_1 = sdec(enc_m_1, dk_R) \textbf{ in}\\
& \textbf{let } s_pub_X = ith_0(m_1) \textbf{ in}\\
& \textbf{let } data_X = ith_1(m_1) \textbf{ in } \textbf{begin}(s_pub_X).\\
& \bar{c}\langle enc_m_2 \rangle.c(enc_m_3).\\
& \textbf{let } sig_X = sdec(enc_m_3, dk_R) \textbf{ in}\\
& \textbf{let } t = getmess(sig_X) \textbf{ in}\\
& \textbf{let } b_2 = checksig(sig_X, s_pub_X) \textbf{ in}\\
& \textbf{if } b = true \textbf{ then } \bar{c}\langle enc_m_4 \rangle.\\
& c(enc_m_5).\bar{c}\langle enc_m_6 \rangle.\\
& \textbf{if } s_pub_X = s_pub_I \textbf{ then } \textbf{end}(pk(s_priv_R))
\end{aligned}
$$

$$
\text{where} \begin{cases}
enc_m_2 = senc([pk(s_priv_R), data_R], ek_R)\\
b_1 = (t = [data_I, data_X])\\
b = b_1 \&\& b_2\\
sig = sign([data_X, data_R], s_priv_R)\\
enc_m_4 = senc(sig, ek_R)\\
enc_m_6 = senc(\bot, ek_R)
\end{cases}
$$

Finally, the main process that runs $node_I$ and $node_R$ processes is:

$$
\begin{aligned}
P =& (\nu s_priv_I, s_priv_R, ek_I, ek_R)\\
& \textbf{let } s_pub_I = pk(s_priv_I) \textbf{ in } \bar{c}\langle s_pub_I \rangle.\\
& \textbf{let } s_pub_R = pk(s_priv_R) \textbf{ in } \bar{c}\langle s_pub_R \rangle.\\
& ((!node_I(s_priv_I, s_pub_R, ek_I, ek_R)) | (!node_R(s_priv_R, s_pub_I, ek_R, ek_I)))
\end{aligned}
$$

The participants static public keys are outputted to the public channel to make sure that the attacker receives them.

5.6 From π to Proverif

In representing the protocol in the applied π-calculus that ProVerif uses we will introduce some changes to the model presented in the section 5.5. To start with, we

are introducing two new types: *public_key* and *private_key* which will be subtypes of the type *key* used in the model in section 5.5. These types of keys will be used the same as the *key* type was used for encryption keys and participants public and private keys. Additionally, the data exchanged in the protocol will be of type *bitstring* that is a part of the ProVerif syntax. We will also use a constant *empty* for representing the empty data ($\perp$).

Cryptographic primitives and functions used in the model are presented in table 5.4.

$$
\begin{aligned}
pk &: private_key \to public_key \\
sign &: bitstring \times private_key \to bitstring \\
getmess &: bitstring \to bitstring \\
checksig &: bitstring \times public_key \to bool \\
senc &: bitstring \times bitstring \to bitstring \\
sdec &: bitstring \times bitstring \to bitstring \\
eq &: bitstring \times bitstring \to bool
\end{aligned}
$$

Table 5.4: Cryptographic primitives and functions

Where:

- $pk(k)$ computes a public from a private key
- $sign(x, k)$ signs a data x with a private key k
- $getmess(s)$ returns the message that was signed with the signature s
- $checksig(s, k)$ returns *true* if the signature was created using the same private key which was used to derive the public key k
- $senc(d, k)$ encrypts data d using the key k
- $sdec(d, k)$ decrypts data d using the key k
- $eq(x, y)$ returns *true* if x is equal to y

The rest of the ProVerif implementation are the subprocesses and the main processes definitions. Since these processes were already described in section 5.5, we will not show the code here, but reference the full version of it in Appendix D.

5.7 Analysing the protocol in ProVerif

In this section we will test whether the security properties hold for the protocol. To recall, these properties are:

- the initiating node encryption key secrecy must hold
- the responding node encryption key secrecy must hold
- the initiating node must be authentic to the responding node
- the responding node must be authentic to the initiating node

Before we can analyse the protocol properties, we will introduce the capabilities the attacker has in this protocol.

5.7.1 Attacker capabilities

We will consider a Dolev-Yao attacker as described in section 2.2. The attacker capabilities are expressed in the inference system from table 5.5.

$$\text{(Inf Null)}\ \frac{}{\perp} \qquad \text{(Inf Pk)}\ \frac{M}{pk(M)} \qquad \text{(Inf Senc)}\ \frac{M\ N}{senc(M,N)}$$

$$\text{(Inf Sdec)}\ \frac{sdec(senc(M,N),N)}{M} \qquad \text{(Inf Sign)}\ \frac{M\ N}{sign(M,N)} \qquad \text{(Inf Getmess)}\ \frac{M}{getmess(M)}$$

$$\text{(Inf Checksig)}\ \frac{M\ N}{checksig(M,N)} \qquad \text{(Inf Tup)}\ \frac{M_1\ ...\ M_n}{[M_1,...,M_n]} \qquad \text{(Inf Elem)}\ \frac{ith_i[M_1,...,M_n]}{M_1}$$

Table 5.5: The inference system

The rules of the inference system enables the attacker to:

- build and use the null ($\perp$) data (Inf Null)
- derive a public key from a private key (Inf Pk)
- encrypt a term (Inf Senc)
- decrypt a term, if it knows the correct encryption key (Inf Sdec)

- compute a signature for a term (Inf Sign)
- retrieve a message from a signature (Inf Getmess)
- check the validity of a signature (Inf Checksig)
- build a tuple of an arbitrary size (Inf Tup)
- retrieve elements of a tuple (Inf Elem)

Additionally, the attacker will know all of the participants public keys.

5.7.2 Analysis of security properties

The part of the code that will test whether the aformentioned properties hold is:

```
0 query attacker(secret_ek_I);
1       attacker(secret_ek_R).
2
3 query x:public_key; inj-event(end_init(x)) ==>
     inj-event(begin_init(x)).
4 query x:public_key; inj-event(end_resp(x)) ==>
     inj-event(begin_resp(x)).
```

secret_ek_I and *secret_ek_R* are free names which will be encrypted with the session keys and given to the attacker to try to decrypt them. If it succeeds, then some of the session keys has been leaked. The queries for testing whether the injective agreement holds in both way, work as described in section 2.3.
The queries results are following:

```
0 RESULT not attacker(secret_ek_I[]) is true.
1 RESULT not attacker(secret_ek_R[]) is true.
2 RESULT inj-event(end_I(x_81)) ==> inj-event(begin_I(x_81))
     is true.
3 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.

Chapter 6

The Lightning Network onion routing protocol

In this chapter we are presenting the third sub-protocol of the Lightning Network: *the onion routing protocol*. We will describe the security properties that the protocol needs to satisfy in section 6.2 followed by the protocol signature and informal description in section 6.2 and 5.4 respectively. In section 6.5 we will present a model of the protocol using the applied π-calculus, which we will use to write the ProVerif compatible code in section 6.6. Finally, we will analyse and discuss the results produced by ProVerif in section 6.7.

6.1 A brief introduction to the protocol

Multi-hop payments enable users to pay to a recipient they are not directly connected to (i.e. do not 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 [21]. 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 in chapter 4.

6.2 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.
The node that is a part of the route must not be able to deduce:

- its position in the route
- which node is the source of the payment
- which node is the destination of the payment
- how long the payment route is
- the exact payment amount

The exact payment amount in the last point is the amount the payment destination is receiving after all the intermediary fees are subtracted. Since the node is an intermediary, it can always know the maximum amount the payment destination can receive.
The outside observer i.e. the node that is not a part of the route must not be able to:

- deduce which node is the source of the payment
- deduce which node is the destination of the payment
- deduce the payment amount
- correlate two packets of the same route together

For the protocol to be able to satisfy all of the requirements, we need to be sure that the participant session keys and private keys are not leaked to the attacker. Moreover, the encrypted payload of the packet must also be confidential. Hence, we are stating the secrecy properties that the protocol must hold as:

- 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:

- 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

6.3 Principals and signature

The protocol will have the following *signature* $(\mathcal{S}, \Sigma)$, where $\mathcal{S}$ is a set of sorts and contains two different sorts: $\mathcal{S} = \{term, key\}$ and where Σ is a set of function symbols with following symbols: $\Sigma = \{\perp, ECDH, senc, sdec, pk, bk, HMAC, [\,], ith_i\}$. All data exchanged between the nodes, as well as its encrypted form and its hash values will be of sort *term* while the keys (e.g. shared secret keys, encryption keys, etc.) used in the protocol will be of sort *key*. The function symbols are given in the table 6.1.

$\perp$	:	$\to term$	pk	:	$key \to key$
$ECDH$	:	$key \times key \to key$	bk	:	$key \times key \to key$
$senc$	:	$term \times key \to term$	$HMAC$	:	$term \times key \to term$
$sdec$	:	$term \times key \to term$	$[\,]$	:	$term \times term \to term$
$hash$	:	$term \to term$	ith_i	:	$term \to term$

Table 6.1: Cryptographic primitives

- $\perp$ represents a null or empty data
- $ECDH(k_1, k_2)$ is an abstraction of the Elliptic Curve Diffie-Hellman key exchange where we combine two keys k_1 and k_2 into a resulting shared secret key
- $senc(d, k)$ represents a symmetric encryption using k as a encryption key while d is the data being encrypted
- $sdec(d, k)$ represents a inverse function to encrypting; it takes the encrypted data d and decrypts it back to the readable form using the decryption key k
- $hash(d)$ represents a hashing function

- $pk(k)$ is a public key generation function that takes a private key k and returns its public key pair
- $bk(k, b)$, 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
- $[x, y]$ is a tuple creation function in which we also consider the short form $[x_1, x_2, ..., x_{n-1}, x_n]$ for the expression $[x_1, [x_2, [..., [x_{n-1}, x_n]]]]$
- $ith_i(M)$ function that returns i-th element of a tuple M

We will also extend the applied π-calculus reduction system with additional reduction rules for the destructors we will use:

$$\begin{array}{cc} \text{(Red Decrypt)} & sdec(senc(M, N), N) \rightarrow M \\ \text{(Red Ith}_\text{i}\text{)} & ith_i([M_1, ..., M_n]) \rightarrow M_i \end{array}$$

6.4 Informal 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 $node_D$. Currently, the payments are limited to a maximum 20 hops (19 intermediaries and 1 recipient node) [18]. The Lightning Network uses source-routing and $node_S$ is the one constructing the route. It is very important to give the senders full control over the routing, because they can then decide on the number of hops, total amount of the fees they will have to pay and finally the worst-case time-lock period inside which their funds could be locked in a pending payment [18]. We skip the part of the route creation and assume that $node_S$ has constructed a valid route (as in has selected the nodes through which the payment will be sent). In table 6.2 we can see the onion routing message sequence diagram, where a $node_{Hi}$ possesses a key k_i and $node_D$ possesses the key k_D.

Message 1	$node_S$	$\rightarrow$	$node_{H1}$	:	$senc([d_1, senc(o_1, k_2)], k_1)$
Message 2	$node_{H1}$	$\rightarrow$	$node_{H2}$	:	$senc([d_2, senc(o_2, k_3)], k_2)$
...					
Message i	$node_{Hi-1}$	$\rightarrow$	$node_{Hi}$	:	$senc([d_i, senc(o_i, k_{i+1})], k_i)$
...					
Message 20	$node_{H19}$	$\rightarrow$	$node_D$	:	$senc(d_D, k_D)$

Table 6.2: The onion routing protocol message sequence

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 [25] 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. In the following section, we will describe the protocol workings in a multi-hop payment scenario.

Assume that $node_S$ has constructed a route: $route = \{n_0, n_1, ..., n_{r-1}, n_r\}$, where n_0 is the $node_S$, n_1 to n_{r-1} are the hops and n_r is the $node_D$ and where all n_i and n_{i+1} are in fact peers (i.e. have an open channel in-between). In order to create the onion packet routed over r hops, the sender needs to create r shared secrets, r ephemeral public keys, r blinding factors and r payment forwarding details. Firstly, $node_S$ collects the public keys from every node along the route: $pub_keys = \{pub_key_1, ..., pub_key_{r-1}, pub_key_r\}$ where pub_key_i is the n_i's node ID (node n_i possesses the private key for its pub_key_i). $node_S$ then generates a random 32-byte key called *session_key* and initializes the ephemeral private key for the first hop to the *session_key* and derives its ephemeral public key pair:

$$eph_priv_key_1 = session_key$$

$$eph_pub_key_1 = pk(eph_priv_key_1)$$

With the first ephemeral key pair, $node_S$ can now, for each of the r (starting at 1 as the first hop) hops along the route do the following:

1. Elliptic Curve Diffie-Hellman (ECDH) [12] shared secret generation; $node_S$ takes the hop's public key (pub_key_i) and i-th ephemeral private key($eph_priv_key_i$), and combines them to produce the shared secret(sh_secret_i):

$$sh_secret_i = \text{ECDH}(pub_key_i, eph_priv_key_i)$$

2. Compute rho_key_i and mu_key_i; both of the keys are in fact the HMAC of some fixed data (rho=`0x72686F`, mu=`0x6d75`) using the shared secret as the

HMAC key:

$$rho_key_i = \text{HMAC}(\texttt{0x72686F}, sh_secret_i)$$

$$mu_key_i = \text{HMAC}(\texttt{0x6d75}, sh_secret_i)$$

3. Compute the blinding factor; by hashing the n_i's ephemeral public key and n_i's shared secret:

$$b_i = hash([eph_pub_key_i, sh_secret_i])$$

4. Compute the ephemeral key pair for the next hop (n_{i+1}); by blinding the current ephemeral key pair with the previously computed blinding factor:

$$eph_priv_key_{i+1} = bk(eph_priv_key_i, b_i)$$

$$eph_pub_key_{i+1} = pk(eph_priv_key_{i+1})$$

The ephemeral keys will be used by the intermediary nodes to create their own copy of the shared secret. When encrypting and computing the packet HMAC $node_S$ has to compute two special keys for each hop: *rho_key* is used as the encryption key and *mu_key* used as the HMAC key used for the packet HMAC computation. The blinding factor computed in the third step is needed to be sure that no one can link two packets from the same route by looking at their ephemeral public keys. When all the keys and shared secrets are computed, $node_S$ can construct the packet. It firstly initializes 1366 `0x00`-bytes (let's call it m), and then iteratively, for each hop in the route in the descending order, starting with the $node_D$:

1. computes the HMAC of m (using rho_key_i)

2. creates the payment data d_i

3. copies the payment data d_i to m

4. encrypts m (using mu_key_i)

After this loop is done, we end up with m containing all the payment data for each of the intermediary nodes, encrypted in layers and protected by HMAC values. m is then included in the onion packet which contains the first ephemeral public key ($eph_pub_key_1$), m and the last m's HMAC. The packet is then encrypted once again with the session key the node has established with the first hop in the route in the key agreement protocol. Finally, the packet is sent to the first hop in the route (n_1).

n_1 gets the packet, decrypts it using the session key it has with the sender and computes the shared secret using his node ID private key ($priv_key_1$) and

ephemeral public key ($eph_pub_key_1$) from the packet and doing so, the ECDH key exchange is completed: $sh_secret_1 = \text{ECDH}(priv_key_1, eph_pub_key_1)$. It is important to notice that, the receiving node does not know who is the $node_S$ that initiated the key exchange, which is a big plus for protocol privacy. Once the receiving node has computed the shared secret, it can compute *mu_key* and *rho_key* in order to check the HMAC validity of the packet and if valid, decrypt the encrypted part of the packet.

After getting the payment information, n_1 can construct the outgoing packet. Firstly, n_1 computes the next ephemeral public key (*eph_pub_key_2*) to put in the packet. It starts by computing the blinding factor and then blinds the ephemeral public key from the packet to create the next ephemeral public key:

$$b_1 = \text{hash}([eph_pub_key_1, sh_secret_1])$$

$$eph_pub_key_2 = bk(eph_pub_key_1, b_1)$$

This computation results in the same $eph_pub_key_2$ as $node_S$ did compute when it was constructing the packet. The node includes the newly computed $eph_pub_key_2$, together with the encrypted part of the previous onion and its HMAC, into the outgoing packet. The outgoing packet is again encrypted using the session key the node has with the next hop in the route and it forwards it along the route.

6.5 The applied π-calculus version of the protocol

The protocol described in the previous section can be modeled using the applied π-calculus described in section 2.1.3. Since this model is a symbolic model of the actual onion routing protocol, we are considering some assumptions and abstracting away the computational parts of it. Firstly, we assume that the payment source (i.e. $node_S$) has constructed a valid route to the payment recipient and knows the node IDs (i.e. public keys) of all of the nodes in the route including the recipient. Secondly, the model is presented on a shortest multi-hop payment scenario where there is only one intermediate node, so the protocol participants are: $node_S$, $node_H$ and $node_D$. Furthermore, the shared secret two processes have established is going to be used as the HMAC-key and the symmetric encryption key. This simplification is possible, due to the fact that in the actual protocol, the HMAC-key and the symmetric encryption key are derived from the shared secret and some additional public data.
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 $node_D$. To construct an onion packet and securely send it to the first hop in the route, $node_S$ has to establish a shared secret between all the nodes in the route

including the recipient node. With the shared secrets established, it can continue with creating other keys needed (i.e. ephemeral public-private key pairs). Finally, it will encrypt the data in layers of encryption and each hop in the route will be able to decrypt only one layer of the packet. Additionally, the layers are going to be protected by their own HMAC value computed by the $node_S$. The $node_S$ process is:

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

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

Where:

- s_pub_H, s_pub_D are hop and destination static public key
- s_pub_X is a public key that the environment provides
- e_priv_H, e_priv_D are hop and destination ephemeral private key
- d_H, d_D are hop and destination payment information; it contains information about where the packet should be forwarded to
- *ek_S* is the session key that is used for encrypting the communication between $node_S$ and $node_H$

The public key s_pub_X is given by the environment and it will be used for testing the authenticity property. The last (innermost) HMAC does not have any further data to protect since the route stops with the destination node and it is set to null ($\perp$). If a node receives a packet with the inner HMAC set to null, it can be sure that the receiving node is in fact the packet final destination. We are also emitting a **begin**(M) event where M is going to be the public key of the payment destination, to mark the start of the protocol run.

The $node_H$ will receive the onion packet, decrypt it with the session key and check its HMAC and depending on its validity it will continue processing the

packet. Lastly, it will extract the payment data needed for the outgoing packet construction and send out the outgoing packet to the next node:

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

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

The packet is again encrypted and forwarded to $node_D$ which validates the HMAC and decrypts the last layer of the onion. If the inner HMAC is null ($\perp$), the packet is valid and the node will emit event $\mathbf{end}(M)$ where M is going to be its public key:

$$
\begin{aligned}
node_D(s_priv_D, dk_D) = & c_2(enc_m).\mathbf{let}\ m = sdec(enc_m, dk_D)\ \mathbf{in} \\
\mathbf{let}\ e_pub_D = ith_0(m)\ \mathbf{in} & \\
& \mathbf{let}\ x_D = ith_1(m)\ \mathbf{in} \\
& \mathbf{let}\ h_{prev} = ith_2(m)\ \mathbf{in} \\
& \mathbf{if}\ h = h_{prev}\ \mathbf{then} \\
& \mathbf{let}\ data = sdec(x_D, ss_2)\ \mathbf{in} \\
& \mathbf{let}\ h_{nxt} = ith_1(data)\ \mathbf{in} \\
& \mathbf{if}\ h_{nxt} = \perp\ \mathbf{then}\ \mathbf{end}(pk(s_priv_D))
\end{aligned}
$$

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

By emitting the $\mathbf{end}(M)$ event we will be able to test the destination authenticity

property.
The main process that will run the aformentioned participant processes is:

$$
\begin{aligned}
P = &(\nu s_priv_S, s_priv_H, s_priv_D, ek_S, ek_H)\\
&\textbf{let } s_pub_S = pk(s_priv_S) \textbf{ in } \overline{c1}\langle s_pub_S\rangle.\\
&\textbf{let } s_pub_H = pk(s_priv_H) \textbf{ in } \overline{c1}\langle s_pub_H\rangle.\\
&\textbf{let } s_pub_D = pk(s_priv_D) \textbf{ in } \overline{c1}\langle s_pub_D\rangle.\\
&((!node_S(s_priv_S, s_pub_H, ek_S))|(!node_H(s_priv_H, ek_H, ek_S))|(!node_D(s_priv_D, ek_H)))
\end{aligned}
$$

The session keys are assigned so that the nodes can encrypt communication between themselves. The participants static public keys are outputted to the public channel to make sure that the attacker receives them.

6.6 From π to Proverif

Having the model represented in the applied π-calculus notation, we can test out its security properties using ProVerif [7]. We need to translate the model into the ProVerif's input language, which is a slightly different variant of applied π-calculus [9]. We will model the protocol the same way we did in section 6.5, so the protocol will be demonstrated on a three users scenario, where the $node_S$ (the payment origin) makes a payment to the recipient node ($node_D$) over one intermediary hop node ($node_H$). Typically, the ProVerif program starts with the types that are going to be used.

In comparison with the applied π-calculus model, we are introducing more types to the model. The reason for that is to make the code more understandable. *public_key* represent the node ID and *private_key* is its private pair. The ephemeral keys will also be represented with *public_key* and *private_key* types. *shared_secret* will be the type used for modeling the shared secret key between the participants and the $blinding_factor$ will represent the key for key blinding. *bitstring* and *bool* are language standard types and they will be used for representing different sorts of data and boolean algebra values respectively. The communication between the participants in the protocol will be done over the public channels which are assumed to be controlled by an attacker with Dolev-Yao capabilities [13] as described in section 2.2. An attacker may read, modify, delete and inject messages exchanged over the public channels [9]. As we did in the previous section, we will use the two public channels c_1 and c_2. Cryptographic primitives and functions used in the model are presented in the table 6.3.

$$
\begin{array}{rcl}
pk & : & private_key \rightarrow public_key \\
ECDH & : & private_key \times public_key \rightarrow shared_secret \\
HMAC & : & bitstring \times shared_secret \rightarrow bitstring \\
to_bitstring & : & shared_secret \rightarrow bitstring \\
senc & : & bitstring \times bitstring \rightarrow bitstring \\
sdec & : & bitstring \times bitstring \rightarrow bitstring \\
calc_blinding_factor & : & eph_public_key \times shared_secret \rightarrow blinding_factor \\
blind_priv & : & private_key \times blinding_factor \rightarrow private_key \\
blind_pub & : & public_key \times blinding_factor \rightarrow public_key \\
onion & : & public_key \times bitstring \times bitstring \rightarrow bitstring \\
eq & : & bitstring \times bitstring \rightarrow bool
\end{array}
$$

Table 6.3: Cryptographic primitives and functions

The constructor *pk* is used to create public keys and they abstract away the elliptic curve point multiplication which the onion routing protocol uses in practise. The shared secret is created by using *calc_secret* constructor which is an abstraction of the Elliptic Curve Diffie-Hellman key exchange that happens between two parties (or in this protocol, between $node_S$ and every other node along the payment route). *HMAC* is the abstraction of a standard HMAC computation while *senc* and *sdec* are abstractions of a standard symmetric encryption. $to_bitstring$ is used for converting *shared_secret* to *bitstring*, so it can be used for encryption. *calc_blinding_factor* creates a blinding factor by combining the *public_key* and the *shared_secret*, which is then used in blinding the private (*blind_priv*) and public key (*blind_pub*). Finally, we define a constructor for the onion packet that takes three arguments a *public_key* key, a *bitstring* representing the payment data for the node and another *bitstring* representing the packet's HMAC.
The second part of the ProVerif implementation are the subprocesses and the main processes definitions. Since these processes were already described in section **??**, we will not show the code here, but reference the full version of it in Appendix E.

6.7 Analysing the protocol in ProVerif

In section 6.2 we have introduced the security properties that should hold for the onion routing protocol. In this section, we will analyse the secrecy of the protocol data and the authenticity of the payment destination, using the model of the protocol described in section 6.6, and using the code in Appendix E. To recall, the properties the protocol should satisfy are:

- secrecy of the session keys
- secrecy of the payment source private key

- secrecy of the payment intermediary private key
- secrecy of the payment destination private key
- secrecy of the encrypted payload of every packet
- the payment sender must be authentic to the recipient

The analysis is divided into two scenarios; the first one with the attacker being an outside observer (section 6.7.2) and the second one with the attacker being a part of the route, i.e. controlling the intermediary node (section 6.7.3). But firstly, we will introduce the capabilities and the initial knowledge the attacker has in both of the aforementioned scenarios.

6.7.1 Attacker capabilities

We will consider a Dolev-Yao attacker as described in section 2.2. The attacker capabilities are expressed in the inference system from table 6.4.

(Inf Null)	(Inf ECDH)	(Inf Senc)	(Inf Sdec)
$\dfrac{}{\bot}$	$\dfrac{M\ N}{ECDH(M,N)}$	$\dfrac{M\ N}{senc(M,N)}$	$\dfrac{sdec(senc(M,N),N)}{M}$
(Inf Hash)	(Inf Pk)	(Inf Bk)	(Inf HMAC)
$\dfrac{M}{hash(M)}$	$\dfrac{M}{pk(M)}$	$\dfrac{M}{bk(M)}$	$\dfrac{M\ N}{HMAC(M,N)}$
(Inf Tup)	(Inf Elem)		
$\dfrac{M_1\ ...\ M_n}{[M_1,...,M_n]}$	$\dfrac{ith_i[M_1,...,M_n]}{M_1}$		

Table 6.4: The inference system

The rules of the inference system enables the attacker to:

- build and use the null ($\bot$) data (Inf Null)
- perform an Elliptic Curve Diffie Hellman between two keys (Inf ECDH)

- encrypt a term (Inf Senc)
- decrypt a term, if it knows the correct encryption key (Inf Sdec)
- compute a hash from a term (Inf Hash)
- derive a public key from a private key (Inf Pk)
- blind a key (Inf Bk)
- compute HMAC of a term using a key (Inf HMAC)
- build a tuple of an arbitrary size (Inf Tup)
- retrieve elements of a tuple (Inf Elem)

Since the packets are being transferred through the public channel, we assume that every packet will be intercepted by the outside attacker. If the attacker has the control over a intermediary node, it can remove the outer encryption protected by the session keys and it can extract the ephemeral public key, the encrypted payload and the packet HMAC. The participants public keys are also known to the attacker.

6.7.2 The outside attacker

We will use the following code to test out the security properties in ProVerif:

```
0  query attacker(ek_S).
1  query attacker(ek_H).
2
3  query attacker(s_priv_S).
4  query attacker(s_priv_H).
5  query attacker(s_priv_D).
6
7  query attacker(hop_data).
8  query attacker(recipient_data).
9
10 query p:public_key;
11     inj-event(end(p)) ==> inj-event(begin(p)).
```

The lines 0 and 1 are testing the secrecy of the session 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). The lines 3-5 are testing the secrecy of the participants private keys. The lines 7 and 8 are testing the secrecy of the payment information intended for the hop and sender respectively. The last line is testing whether the authenticity of the sender holds.
The results that ProVerif outputs are following:

```
0 RESULT not attacker(ek_S[]) is true.
1 RESULT not attacker(ek_H[]) is true.
2 RESULT not attacker(s_priv_S[]) is true.
3 RESULT not attacker(s_priv_H[]) is true.
4 RESULT not attacker(s_priv_D[]) is true.
5 RESULT not attacker(hop_data[]) is true.
6 RESULT not attacker(recipient_data[]) is true.
7 RESULT inj-event(end(p)) ==> inj-event(begin(p)) is true.
```

All of the properties are satisfied: the attacker is not able to obtain any of the keys or the payment data and additionally, the sender is authentic to the receiver.

6.7.3 The inside attacker

The attacker control over the intermediary node is modeled by the intermediary node leaking everything it knows into the public channel. In the following process calculus model we can see the malicious version of the intermediary node:

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

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

The only difference to the non-malicious intermediary node, is in the fact that the malicious node outputs all data it knows to the public channel. In this analysis, the attacker already knows the confidential data that the hop node knows. So the properties that the protocol needs to satisfy are as follows:

- secrecy of the session key of the sender
- secrecy of the payment source private key
- secrecy of the payment destination private key
- secrecy of the encrypted payload with the payment information intended for the destination
- the payment sender must be authentic to the recipient

We will use the following code to test out the aformentioned security properties in ProVerif:

```
0 query attacker(ek_S).
1
2 query attacker(s_priv_S).
3 query attacker(s_priv_D).
4
5 query attacker(recipient_data).
6
7 query p:public_key;
8     inj-event(end(p)) ==> inj-event(begin(p)).
```

The code 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:

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

As we can see, the results are the same as the ones in section 6.7.2 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.

Chapter 7

Conclusion

In this report, we have addressed the problem of the potentially insecure payment network, called the Lightning Network. The Lightning Network is a decentralized network of payment channels that works on top of the cryptocurrency called Bitcoin. We found the importance of this report in the fact that the Lightning Network is already being used by a significant user base and with at least 8 million US dollars worth of funds locked in it and any potential vulnerability found by the work like this, could prevent possible thefts of the user funds or private data. Similarly, if this work would confirm the security properties that the Lightning Network should have, it would further increase the user trust in the network.

7.1 Results

The Lightning Network is a complicated protocol and to make the analysis less difficult, we have identified and split the Lightning Network into four sub-protocols:

- *the key agreement protocol* which is used for mutual authentication of the participants and for creation of a secret key that will be used for encrypting any future communication
- *the channel opening protocol* which is used for establishing a new payment channel between two users
- *the onion routing protocol* which is used for routing the payment data through the network of users
- *the channel closing protocol* which is used for closing a channel

The channel closing protocol was not analysed due to its relatively small importance to the Lightning Network security. We have defined *secrecy* (of confidential protocol data) and *authenticity* (of the protocol participants) as the security properties

that sub-protocols need to satisfy. Each of the sub-protocols we described informally and then created a model using the applied π-calculus after which we have performed the analysis using an automatic cryptographic protocol analyser tool called ProVerif. The analysis that was performed was a symbolic analysis that considered an attacker with the Dolev-Yao capabilities. The results that ProVerif has produced can be seen in table 7.1.

Protocol	Security property	ProVerif result
The key agreement protocol	secrecy of the session key generated by the initiator	**true**
	secrecy of the session key generated by the responder	**true**
	authenticity of the initiator to the reponder	**true**
	authenticity of the responder to the initiator	**false**
The channel opening protocol	secrecy of the session key used by the initiator	**true**
	secrecy of the session key used by the responder	**true**
	authenticity of the initiator to the reponder	**true**
	authenticity of the responder to the initiator	**true**
The onion routing protocol	secrecy of the session key used by the initiator	**true**
	secrecy of the session key used by the responder	**true**
	secrecy of the private key used by the initiator	**true**
	secrecy of the private key used by the intermediary	**true**
	secrecy of the private key used by the responder	**true**
	secrecy of the packet payload	**true**
	authenticity of the initiator to the responder	**true**

Table 7.1: The results produced by ProVerif

From the results of the analysis, we have discovered that both *the channel opening* and *the onion routing* protocols satisfy all of the required properties. *The key agreement* protocol satisfies all but one security property: the authenticity of the protocol responder to the initiator. This finding is quite interesting and probably the most important result of this report, because it means that: *in the key agreement protocol, that should provide mutual authentication, the protocol initiator can not be sure that it is running the protocol with the responder it has in mind*.

In section 1.2 we have presented two main questions that this report wanted to give answers to and they are:

- are confidential data kept secret in the Lightning Network?
  - *yes, all of the Lightning Network sub-protocols preserve the secrecy of confidential data*
- does user authenticity hold in the Lightning Network?
  - *for two out of three sub-protocols that were analysed (namely, the channel opening and the onion routing protocol) it does, but for the key agreement protocol it does not, because it does not provide provide a way for the protocol initiator to authenticate the responder (the protocol responder is still able to authenticate the initiator, however)*

7.2 Further work

The authentication flaw in *the key agreement* protocol, raises new questions:

- how does this flaw affects the Lightning Network security?
- is there a way in which an attacker would take advantage of this flaw?
- what could be changed in the key agreement protocol so that the initiator is able to authenticate the responder?

These questions remain unanswered and will be left out for the future work. Since the analysis done in this report was a symbolic analysis, where we have abstracted away the computational parts of the protocol, an additional future work could be done on the computational analysis of the Lightning Network. Finally, the Lightning Network remains a fully working network which due to its immaturity can benefit from any form of the future analysis such as this report was.

Bibliography

[1] 1ML.com. *Lightining Network Statistics*. 2019. URL: https://1ml.com/statistics.

[2] Martın Abadi and Andrew D Gordon. "A calculus for cryptographic protocols: The spi calculus". In: *Information and computation* 148.1 (1999), pp. 1–70.

[3] Andreas M. Antonopoulos. *Mastering Bitcoin. Programming the open blockchain*. O'Reilly Media, 2017.

[4] Alessandro Armando et al. "Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps". In: *Proceedings of the 6th ACM workshop on Formal methods in security engineering*. ACM. 2008, pp. 1–10.

[5] Alessandro Armando et al. "The AVISPA tool for the automated validation of internet security protocols and applications". In: *International conference on computer aided verification*. Springer. 2005, pp. 281–285.

[6] Jan A Bergstra, Alban Ponse, and Scott A Smolka. *Handbook of process algebra*. Elsevier, 2001, pp. 479–545.

[7] Bruno Blanchet. "Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif". In: *Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures*. Ed. by Alessandro Aldini, Javier Lopez, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Springer, 2014, pp. 54–87.

[8] Bruno Blanchet. "From secrecy to authenticity in security protocols". In: *International Static Analysis Symposium*. Springer. 2002, pp. 342–359.

[9] Bruno Blanchet et al. *ProVerif 2.00: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. 2018.

[10] Véronique Cortier, Jan Degrieck, and Stéphanie Delaune. "Analysing routing protocols: four nodes topologies are sufficient". In: *International Conference on Principles of Security and Trust*. Springer. 2012, pp. 30–50.

[11] Cas JF Cremers. "The scyther tool: Verification, falsification, and analysis of security protocols". In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 414–418.

[12] Whitfield Diffie and Martin Hellman. "New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654.

[13] Danny Dolev and Andrew Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[14] Alfred Horn. "On sentences which are true of direct unions of algebras". In: *The Journal of Symbolic Logic* 16.1 (1951), pp. 14–21.

[15] David B Johnson, David A Maltz, Josh Broch, et al. "DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks". In: *Ad hoc networking* 5 (2001), pp. 139–172.

[16] Hugo Krawczyk and Pasi Eronen. "Hmac-based extract-and-expand key derivation function (hkdf)". In: (2010).

[17] lightningnetwork. *GitHub - lightning-rfc/02-peer-protocol.md*. 2017. URL: https://github.com/lightningnetwork/lightning-rfc/blob/master/02-peer-protocol.md.

[18] lightningnetwork. *GitHub - lightningnetwork-rfc/04-onion-routing.md*. 2017. URL: https://github.com/lightningnetwork/lightning-rfc/blob/master/04-onion-routing.md.

[19] lightningnetwork. *GitHub - lightningnetwork-rfc/07-routing-gossip.md*. 2017. URL: https://github.com/lightningnetwork/lightning-rfc/blob/master/07-routing-gossip.md.

[20] lightningnetwork. *GitHub - lightningnetwork-rfc/08-transport.md*. 2017. URL: https://github.com/lightningnetwork/lightning-rfc/blob/master/08-transport.md.

[21] lightningnetwork. *GitHub - lightningnetwork/lightning-onion: Onion Routed Micropayments for the Lightning Network*. 2017. URL: https://github.com/LightningNetwork/lightning-onion.

[22] Gavin Lowe. "A hierarchy of authentication specifications". In: *Proceedings 10th Computer Security Foundations Workshop*. IEEE. 1997, pp. 31–43.

[23] magoo. *Blockchain Graveyard*. 2019. URL: https://magoo.github.io/Blockchain-Graveyard/.

[24] Satoshi Nakamoto et al. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[25] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. Tech. rep. 2018.

[26] Panagiotis Papadimitratos and Zygmunt Haas. "Secure routing for mobile ad hoc networks". In: *Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002)*. CONF. SCS. 2002.

[27] Trevor Perrin. "The noise protocol framework". In: *PowerPoint Presentation* (2016).

[28] Joseph Poon and Thaddeus Dryja. *The bitcoin lightning network: Scalable off-chain instant payments*. 2016.

[29] BitMEX Research. *The Lightning Network*. 2018. URL: https://blog.bitmex.com/the-lightning-network/.

[30] Elizabeth Stark. *What is the Lightning Network and how can it help Bitcoin scale?* URL: https://coincenter.org/entry/what-is-the-lightning-network.

[31] *VISA Fact Sheet*. URL: https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf.

[32] Aaron van Wirdum. *Understanding the Lightning Network*. 2016. URL: https://bitcoinmagazine.com/articles/understanding-the-lightning-network-part-building-a-bidirectional-payment-channel-1464710791/.

[33] *With 100% segwit transactions, what would be the max number of transaction confirmation possible on a block?* URL: https://bitcoin.stackexchange.com/questions/59408/with-100-segwit-transactions-what-would-be-the-max-number-of-transaction-confi.

[34] Thomas YC Woo and Simon S Lam. "A semantic model for authentication protocols". In: *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE. 1993, pp. 178–194.

Appendix A

The unauthenticated Diffie-Hellman key exchange

```
0  (* Public communication channel *)
1  free c:channel.
2
3  (* Message *)
4  free m: bitstring [private].
5
6  (* Public key crypto *)
7  type private_key.
8  type public_key.
9  type enc_key.
10 fun pk(private_key): public_key.
11 fun to_enc_key(public_key): enc_key [typeConverter].
12
13 (* ECDH *)
14 fun ECDH(private_key, public_key): public_key.
15 equation forall k_1:private_key, k_2:private_key;
16   ECDH(k_1, pk(k_2)) = ECDH(k_2, pk(k_1)).
17
18 (* Symmetric encryption *)
19 fun senc(bitstring, enc_key): bitstring.
20 reduc forall x:bitstring, k:enc_key; sdec(senc(x, k), k) = x.
21
22 (* Secrecy of m *)
23 query attacker(m).
24
```

```
25 (* Events *)
26 event begin_I(public_key).
27 event end_I(public_key).
28 event begin_R(public_key).
29 event end_R(public_key).
30
31 query x:public_key; inj-event(end_I(x)) ==>
     inj-event(begin_I(x)).
32 query x:public_key; inj-event(end_R(x)) ==>
     inj-event(begin_R(x)).
33
34
35 let node_I(s_priv_I:private_key, s_pub_R:public_key) =
36   out(c, pk(s_priv_I));
37
38   in(c, s_pub_X:public_key);
39   event begin_R(s_pub_X);
40
41   let k = ECDH(s_priv_I, s_pub_X) in
42   if s_pub_X = s_pub_R then
43   event end_I(pk(s_priv_I));
44   out(c, senc(m, to_enc_key(k))).
45
46 let node_R(s_priv_R:private_key, s_pub_I:public_key) =
47   in(c, s_pub_X:public_key);
48   event begin_I(s_pub_X);
49
50   out(c, pk(s_priv_R));
51
52   in(c, enc_m:bitstring);
53   let k = ECDH(s_priv_R, s_pub_X) in
54   let m' = sdec(enc_m, to_enc_key(k)) in
55   if (m = m' && s_pub_X = s_pub_I) then
56   event end_R(pk(s_priv_R)).
57
58 process
59   new s_priv_I:private_key;
60   new s_priv_R:private_key;
61   let s_pub_I = pk(s_priv_I) in out(c, s_pub_I);
62   let s_pub_R = pk(s_priv_R) in out(c, s_pub_R);
63   ( (!node_I(s_priv_I, s_pub_R)) | (!node_R(s_priv_R,
```

```
s_pub_I)) )
```

Appendix B

The Lightning Network key agreement protocol

```
0 set ignoreTypes = false.
1
2 (* Public communication channel *)
3 free c:channel.
4
5 (* Public key crypto *)
6 type private_key.
7 type public_key.
8 fun pk(private_key): public_key.
9 fun to_bitstring(public_key): bitstring [typeConverter].
10
11 (* ECDH *)
12 fun ECDH(private_key, public_key): public_key.
13 equation forall priv_1:private_key, priv_2:private_key;
14   ECDH(priv_1, pk(priv_2)) = ECDH(priv_2, pk(priv_1)).
15
16 (* Hashing function *)
17 fun hash(bitstring): bitstring.
18
19 (* Key derviation function *)
20 fun hkdf_o1(bitstring, bitstring): bitstring.
21 fun hkdf_o2(bitstring, bitstring): bitstring.
22
23 (* Symmetric encryption *)
24 fun senc(bitstring, bitstring): bitstring.
```

```
25 fun sdec(bitstring, bitstring): bitstring.
26 equation forall m:bitstring, s:bitstring; sdec(senc(m,s), s)
     = m.
27 equation forall m:bitstring, s:bitstring; senc(sdec(m,s), s)
     = m.
28
29 (* Bitstring equality *)
30 fun eq(bitstring, bitstring): bool
31   reduc
32     forall x:bitstring; eq(x,x) = true
33     otherwise forall x:bitstring, y:bitstring; eq(x,y) =
     false.
34
35 (* Constants *)
36 const protocol_name, prologue, empty, int_zero, int_one:
     bitstring.
37
38
39 (* Secrecy assumptions *)
40 not attacker(new real_s_priv_I).
41 not attacker(new real_s_priv_R).
42
43
44 (* Secrecy queries *)
45 free secret_ek_I, secret_dk_I, secret_ek_R, secret_dk_R:
     bitstring [private].
46 query attacker(secret_ek_I);
47     attacker(secret_dk_I);
48     attacker(secret_ek_R);
49     attacker(secret_dk_R).
50
51 (* Correspondence assertion queries *)
52 event begin_I(public_key, public_key).
53 event end_I(public_key, public_key).
54 event begin_R(public_key, public_key).
55 event end_R(public_key, public_key).
56
57 query x:public_key, y:public_key; inj-event(end_I(x,y)) ==>
     inj-event(begin_I(x,y)).
58 query x:public_key, y:public_key; inj-event(end_R(x,y)) ==>
     inj-event(begin_R(x,y)).
```

```
59
60 (* Ephmeral keys included
61 event begin_I_full(public_key, public_key, public_key,
    public_key).
62 event end_I_full(public_key, public_key, public_key,
    public_key).
63 event begin_R_full(public_key, public_key, public_key,
    public_key).
64 event end_R_full(public_key, public_key, public_key,
    public_key).
65
66 query p1:public_key, p2:public_key, e1:public_key,
    e2:public_key; inj-event(end_I_full(p1, p2, e1, e2)) ==>
    inj-event(begin_I_full(p1, p2, e1, e2)).
67 query p1:public_key, p2:public_key, e1:public_key,
    e2:public_key; inj-event(end_R_full(p1, p2, e1, e2)) ==>
    inj-event(begin_R_full(p1, p2, e1, e2)).
68 *)
69
70 (* Role of the initiator that owns public key s_pub_I and
    initiates a protocol run with someone who owns s_pub_Y *)
71 let initiating_node(real_s_priv_I:private_key,
    real_s_priv_R:private_key) =
72
73   (* Let environment select your own public key (x) and the
    key of the responder (s_pub_Y) *)
74   (* First set up the initiator *)
75   in(c, (x:public_key, s_pub_Y:public_key));
76   if(x = pk(real_s_priv_I) || x = pk(real_s_priv_R)) then
77   let s_priv_I = if x = pk(real_s_priv_I) then real_s_priv_I
    else real_s_priv_R in
78   let s_pub_I = pk(s_priv_I) in
79   (* Real start of the role *)
80   event begin_R(s_pub_I, s_pub_Y);
81
82   (* Initialization *)
83   let h0 = hash(protocol_name) in
84   let ck0 = h0 in
85   let h1 = hash((h0, prologue)) in
86   let h2 = hash((h1, s_pub_Y)) in
87
```

```
88  (* Act One *)
89  new e_priv_I:private_key;
90  let h3 = hash((h2, pk(e_priv_I))) in
91  let es = ECDH(e_priv_I, s_pub_Y) in
92  let temp_k1 = hkdf_o1(ck0, to_bitstring(es)) in
93  let ck1 = hkdf_o2(ck0, to_bitstring(es)) in
94  let enc_d1 = senc(empty, (temp_k1, int_zero, h3)) in
95  let h4 = hash((h3, enc_d1)) in
96  let m1 = (pk(e_priv_I), enc_d1) in
97  out(c, m1);
98
99  (* Act Two *)
100 in(c, m2:bitstring);
101 let (e_pub_Y:public_key, enc_d2:bitstring) = m2 in
102 let h5 = hash((h4, e_pub_Y)) in
103 let ee = ECDH(e_priv_I, e_pub_Y) in
104 let temp_k2 = hkdf_o1(ck1, to_bitstring(ee)) in
105 let ck2 = hkdf_o2(ck1, to_bitstring(ee)) in
106 let d2 = sdec(enc_d2, (temp_k2, int_zero, h5)) in
107 if(eq(d2, empty)) then (
108   let h6 = hash((h5, enc_d2)) in
109
110   (* Act Three *)
111   let enc_d3 = senc(to_bitstring(s_pub_I), (temp_k2,
     int_one, h6)) in
112   let h7 = hash((h6, enc_d3)) in
113   let se = ECDH(s_priv_I, e_pub_Y) in
114   let temp_k3 = hkdf_o1(ck2, to_bitstring(se)) in
115   let ck3 = hkdf_o2(ck2, to_bitstring(se)) in
116   let enc_d4 = senc(empty, (temp_k3, int_zero, h7)) in
117   (* Derive the session keys *)
118   let ek_I = hkdf_o1(ck3, empty) in
119   let dk_I = hkdf_o2(ck3, empty) in
120   if(s_pub_Y = pk(real_s_priv_R) || s_pub_Y =
     pk(real_s_priv_I)) then
121     event end_I(s_pub_I, s_pub_Y);
122
123   (* Test out the secrecy of the session keys *)
124   out(c, senc(secret_ek_I, ek_I));
125   out(c, senc(secret_dk_I, dk_I));
126   let m3 = (enc_d3, enc_d4) in
```

```
127     out(c, m3)
128   ).
129
130 (* Role of the responder that owns public key s_pub_R and
      responds to someone who owns s_pub_X *)
131 let responding_node(real_s_priv_I:private_key,
      real_s_priv_R:private_key) =
132   (* Let environment select your own public key (y) *)
133   (* First set up the responder *)
134   in(c, y:public_key);
135   if(y = pk(real_s_priv_I) || y = pk(real_s_priv_R)) then
136   let s_priv_R = if y = pk(real_s_priv_I) then real_s_priv_I
      else real_s_priv_R in
137   let s_pub_R = pk(s_priv_R) in
138   (* Real start of the role *)
139
140   (* Initialization *)
141   let h0 = hash(protocol_name) in
142   let ck0 = h0 in
143   let h1 = hash((h0, prologue)) in
144   let h2 = hash((h1, s_pub_R)) in
145
146   (* Act One *)
147   in(c, m1:bitstring);
148   let (e_pub_X:public_key, enc_d1:bitstring) = m1 in
149   let h3 = hash((h2, e_pub_X)) in
150   let es = ECDH(s_priv_R, e_pub_X) in
151   let temp_k1 = hkdf_o1(ck0, to_bitstring(es)) in
152   let ck1 = hkdf_o2(ck0, to_bitstring(es)) in
153   let d1 = sdec(enc_d1, (temp_k1, int_zero, h3)) in
154   if(eq(d1, empty)) then (
155     let h4 = hash((h3, enc_d1)) in
156
157     (* Act Two *)
158     new e_priv_R:private_key;
159     let h5 = hash((h4, pk(e_priv_R))) in
160     let ee = ECDH(e_priv_R, e_pub_X) in
161     let temp_k2 = hkdf_o1(ck1, to_bitstring(ee)) in
162     let ck2 = hkdf_o2(ck1, to_bitstring(ee)) in
163     let enc_d2 = senc(empty, (temp_k2, int_zero, h5)) in
164     let h6 = hash((h5, enc_d2)) in
```

```
165     let m2 = (pk(e_priv_R), enc_d2) in
166     out(c, m2);
167
168     (* Act Three *)
169     in(c, m3:bitstring);
170     let (enc_d3:bitstring, enc_d4:bitstring) = m3 in
171     let to_bitstring(s_pub_X) = sdec(enc_d3, (temp_k2,
      int_one, h6)) in
172     event begin_I(s_pub_X, s_pub_R);
173     let h7 = hash((h6, enc_d3)) in
174     let se = ECDH(e_priv_R, s_pub_X) in
175     let temp_k3 = hkdf_o1(ck2, to_bitstring(se)) in
176     let ck3 = hkdf_o2(ck2, to_bitstring(se)) in
177     let d4 = sdec(enc_d4, (temp_k3, int_zero, h7)) in
178     if(eq(d4, empty)) then (
179       (* Derive the session keys *)
180       let dk_R = hkdf_o1(ck3, empty) in
181       let ek_R = hkdf_o2(ck3, empty) in
182
183       if s_pub_X = pk(real_s_priv_I) then
184         event end_R(s_pub_X, s_pub_R);
185       out(c, senc(secret_ek_R, ek_R));
186       out(c, senc(secret_dk_R, dk_R))
187     )
188   ).
189
190
191 process
192   new real_s_priv_I:private_key;
193   new real_s_priv_R:private_key;
194   let real_s_pub_I = pk(real_s_priv_I) in out(c,
      real_s_pub_I);
195   let real_s_pub_R = pk(real_s_priv_R) in out(c,
      real_s_pub_R);
196   ( (!initiating_node(real_s_priv_I, real_s_priv_R)) |
      (!responding_node(real_s_priv_I, real_s_priv_R)) )
197
198 (* Correspondence assertions reminder:
199   begin_I(pubX, pubR)
200     -> executed at the start of the process R
201     -> pubX is the public key of someone who initiated the
```

```
        protocol with R
202       -> R: "I am now running a protocol with the owner of
        pubX!"
203     end_I(pubI, pubX)
204       -> needs a check: pubX == pubR so that we are sure that
        we are actually talking to R
205       -> executed at the end of the process I
206       -> I: "I have successfully ended a protocol run with R"
207     begin_R(pubI, pubX)
208       -> executed at the start of the process I
209       -> pubX is the public key of someone who was selected by
        the environment and I initiated a protocol run with it
210       -> I: "I am now running a protocol with the owner of
        pubX!"
211     end_R(pubX, pubR)
212       -> needs a check: pubX == pubI so that we are sure that
        we are actually talking to I
213       -> executed at the end of the process R
214       -> R: "I have successfully ended a protocol run with I"
215
216
217     query x:public_key; inj-event(end_I(x)) ==>
        inj-event(begin_I(x)).
218     query x:public_key; inj-event(end_R(x)) ==>
        inj-event(begin_R(x)).
219   *)
```

Appendix C

The unauthenticated Diffie-Hellman key exchange

```
0 Starting query inj-event(end_I(x_229,y_230)) ==>
    inj-event(begin_I(x_229,y_230))
1 goal reachable:
    end(endsid_105550,end_I(pk(real_s_priv_R[]),pk(real_s_priv_I[])))
2 Abbreviations:
3 e_priv_I_105602 = e_priv_I_36[s_pub_Y =
    pk(real_s_priv_I[]),x_28 = pk(real_s_priv_R[]),!1 =
    endsid_105599]
4 e_priv = e_priv_R[m1_100 =
    (pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_name),to_bitstring(EC
    = pk(real_s_priv_I[]),!1 = @sid_105590]
5
6 1. The message pk(real_s_priv_I[]) may be sent to the
    attacker at output {4}.
7 attacker_public_key(pk(real_s_priv_I[])).
8
9 2. The message pk(real_s_priv_R[]) may be sent to the
    attacker at output {6}.
10 attacker_public_key(pk(real_s_priv_R[])).
11
12 3. By 2, the attacker may know pk(real_s_priv_R[]).
13 By 1, the attacker may know pk(real_s_priv_I[]).
14 Using the function public_key-public_key-tuple the attacker
    may obtain (pk(real_s_priv_R[]),pk(real_s_priv_I[])).
15 attacker_bitstring((pk(real_s_priv_R[]),pk(real_s_priv_I[]))).
```

```
16
17 4. The message (pk(real_s_priv_R[]),pk(real_s_priv_I[]))
      that the attacker may have by 3 may be received at input
      {8}.
18 So the message
      (pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_name),to_bitstri
      may be sent to the attacker at output {66}.
19 attacker_bitstring((pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_
20
21 5. By 4, the attacker may know
      (pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_name),to_bitstri
22 Using the function 2-proj-public_key-bitstring-tuple the
      attacker may obtain
      senc(empty,(hkdf_o1(hash(protocol_name),to_bitstring(ECDH(e_priv_I_1056
23 attacker_bitstring(senc(empty,(hkdf_o1(hash(protocol_name),to_bitstring(EC
24
25 6. By 4, the attacker may know
      (pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_name),to_bitstri
26 Using the function 1-proj-public_key-bitstring-tuple the
      attacker may obtain pk(e_priv_I_105602).
27 attacker_public_key(pk(e_priv_I_105602)).
28
29 7. By 6, the attacker may know pk(e_priv_I_105602).
30 By 5, the attacker may know
      senc(empty,(hkdf_o1(hash(protocol_name),to_bitstring(ECDH(e_priv_I_1056
31 Using the function public_key-bitstring-tuple the attacker
      may obtain
      (pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_name),to_bitstri
32 attacker_bitstring((pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_
33
34 8. The message pk(real_s_priv_I[]) that the attacker may
      have by 1 may be received at input {130}.
35 The message
      (pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_name),to_bitstri
      that the attacker may have by 7 may be received at input
      {140}.
36 So the message
      (pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstrin
      may be sent to the attacker at output {157}.
37 attacker_bitstring((pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_n
38
```

```
39 9. By 8, the attacker may know
     (pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECD
40 Using the function 2-proj-public_key-bitstring-tuple the
     attacker may obtain
     senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECDH(e_priv_I_1
41 attacker_bitstring(senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring
42
43 10. The message pk(real_s_priv_I[]) that the attacker may
     have by 1 may be received at input {130}.
44 The message
     (pk(e_priv_I_105602),senc(empty,(hkdf_o1(hash(protocol_name),to_bitstring(EC
     that the attacker may have by 7 may be received at input
     {140}.
45 So the message
     (pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECD
     may be sent to the attacker at output {157}.
46 attacker_bitstring((pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),
47
48 11. By 10, the attacker may know
     (pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECD
49 Using the function 1-proj-public_key-bitstring-tuple the
     attacker may obtain pk(e_priv).
50 attacker_public_key(pk(e_priv)).
51
52 12. By 11, the attacker may know pk(e_priv).
53 By 9, the attacker may know
     senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECDH(e_priv_I_1
54 Using the function public_key-bitstring-tuple the attacker
     may obtain
     (pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECD
55 attacker_bitstring((pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),
56
57 13. The message (pk(real_s_priv_R[]),pk(real_s_priv_I[]))
     that the attacker may have by 3 may be received at input
     {8}.
58 The message
     (pk(e_priv),senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECD
     that the attacker may have by 12 may be received at input
     {67}.
59 So event end_I(pk(real_s_priv_R[]),pk(real_s_priv_I[])) may
     be executed at {85} in session endsid_105599.
```

```
60 end(endsid_105599,end_I(pk(real_s_priv_R[]),pk(real_s_priv_I[]))).
61
62
63 A more detailed output of the traces is available with
64   set traceDisplay = long.
65
66 new real_s_priv_I: private_key creating real_s_priv_I_105609
     at {1}
67
68 new real_s_priv_R: private_key creating real_s_priv_R_105608
     at {2}
69
70 out(c, ~M_105820) with ~M_105820 = pk(real_s_priv_I_105609)
     at {4}
71
72 out(c, ~M_106019) with ~M_106019 = pk(real_s_priv_R_105608)
     at {6}
73
74 in(c, (~M_106019,~M_105820)) with ~M_106019 =
     pk(real_s_priv_R_105608), ~M_105820 =
     pk(real_s_priv_I_105609) at {8} in copy a_105606
75
76 event
     begin_R(pk(real_s_priv_R_105608),pk(real_s_priv_I_105609))
     at {53} in copy a_105606
77
78 new e_priv_I_36: private_key creating e_priv_I_105610 at
     {58} in copy a_105606
79
80 out(c, (~M_106705,~M_106706)) with ~M_106705 =
     pk(e_priv_I_105610), ~M_106706 =
     senc(empty,(hkdf_o1(hash(protocol_name),to_bitstring(ECDH(e_priv_I_1056
     at {66} in copy a_105606
81
82 in(c, ~M_105820) with ~M_105820 = pk(real_s_priv_I_105609)
     at {130} in copy a_105607
83
84 in(c, (~M_106705,~M_106706)) with ~M_106705 =
     pk(e_priv_I_105610), ~M_106706 =
     senc(empty,(hkdf_o1(hash(protocol_name),to_bitstring(ECDH(e_priv_I_1056
     at {140} in copy a_105607
```

```
85
86 new e_priv_R: private_key creating e_priv_R_105611 at {149}
     in copy a_105607
87
88 out(c, (~M_107345,~M_107346)) with ~M_107345 =
     pk(e_priv_R_105611), ~M_107346 =
     senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECDH(real_s_pri
     at {157} in copy a_105607
89
90 in(c, (~M_107345,~M_107346)) with ~M_107345 =
     pk(e_priv_R_105611), ~M_107346 =
     senc(empty,(hkdf_o1(hkdf_o2(hash(protocol_name),to_bitstring(ECDH(e_priv_I_1
     at {67} in copy a_105606
91
92 event
     end_I(pk(real_s_priv_R_105608),pk(real_s_priv_I_105609))
     at {85} in copy a_105606 (goal)
93
94 The event
     end_I(pk(real_s_priv_R_105608),pk(real_s_priv_I_105609))
     is executed in session a_105606.
95 A trace has been found.
96 RESULT inj-event(end_I(x_229,y_230)) ==>
     inj-event(begin_I(x_229,y_230)) is false.
97 RESULT (even event(end_I(x_105554,y_105555)) ==>
     event(begin_I(x_105554,y_105555)) is false.)
```

Appendix D

The Lightning Network channel opening protocol

```
0 (* Public communication channel *)
1 free c:channel.
2
3 (* Public key crypto *)
4 type private_key.
5 type public_key.
6 fun pk(private_key): public_key.
7
8 (* Signatures *)
9 fun sign(bitstring, private_key): bitstring.
10 reduc forall x:bitstring, k:private_key; getmess(sign(x, k))
     = x.
11 reduc forall x:bitstring, k:private_key; checksig(sign(x,k),
     pk(k)) = true.
12
13 (* Symmetric encryption *)
14 fun senc(bitstring, private_key): bitstring.
15 fun sdec(bitstring, private_key): bitstring.
16 equation forall m:bitstring, k:private_key; sdec(senc(m,k),
     k) = m.
17 equation forall m:bitstring, k:private_key; senc(sdec(m,k),
     k) = m.
18
19 (* Bitstring equality *)
20 fun eq(bitstring, bitstring): bool
```

```
21 reduc
22   forall x:bitstring; eq(x,x) = true
23   otherwise forall x:bitstring, y:bitstring; eq(x,y) = false.
24
25
26 (* Secrecy assumptions *)
27 not attacker(new s_priv_I).
28 not attacker(new s_priv_R).
29
30 (* Constants *)
31 const empty: bitstring.
32
33 (* Secrecy queries *)
34 free secret_ek_I, secret_ek_R: bitstring [private].
35 query attacker(secret_ek_I);
36     attacker(secret_ek_R).
37
38
39 (* Events *)
40 event begin_I(public_key).
41 event end_I(public_key).
42 event begin_R(public_key).
43 event end_R(public_key).
44
45 query x:public_key; inj-event(end_I(x)) ==>
     inj-event(begin_I(x)).
46 query x:public_key; inj-event(end_R(x)) ==>
     inj-event(begin_R(x)).
47
48
49
50 let initiating_node(s_priv_I:private_key,
     s_pub_R:public_key, ek_I:private_key, dk_I:private_key) =
51
52   (* Send 'open_channel' *)
53   new data_I:bitstring;
54   let enc_m1 = senc((pk(s_priv_I), data_I), ek_I) in
55   out(c, enc_m1);
56
57   (* Receive 'accept_channel' *)
58   in(c, enc_m2:bitstring);
```

```
59   let (s_pub_X:public_key, data_X:bitstring) = sdec(enc_m2,
     dk_I) in
60   event begin_R(s_pub_X);
61
62   (* Senc 'funding_created' *)
63   let sig_I = sign((data_I, data_X), s_priv_I) in
64   let enc_m3 = senc(sig_I, ek_I) in
65   out(c, enc_m3);
66
67   (* Receive 'funding_signed' *)
68   in(c, enc_m4:bitstring);
69   let sig_X = sdec(enc_m4, dk_I) in
70   (* Check the message content and its sig *)
71   if(eq((data_I, data_X), getmess(sig_X)) && checksig(sig_X,
     s_pub_X)) then (
72
73     (* Send 'funding_locked' *)
74     let enc_m5 = senc(empty, ek_I) in
75     out(c, enc_m5);
76
77     if s_pub_X = s_pub_R then
78     event end_I(pk(s_priv_I));
79     out(c, senc(secret_ek_I, ek_I))
80   ).
81
82 let responding_node(s_priv_R:private_key,
     s_pub_I:public_key, ek_R:private_key, dk_R:private_key) =
83
84   (* Receive 'open_channel' *)
85   in(c, enc_m1:bitstring);
86   let (s_pub_Y:public_key, data_Y:bitstring) = sdec(enc_m1,
     dk_R) in
87   event begin_I(s_pub_Y);
88
89   (* Send 'accept_channel' *)
90   new data_R:bitstring;
91   let enc_m2 = senc((pk(s_priv_R), data_R), ek_R) in
92   out(c, enc_m2);
93
94   (* Receive 'funding_created' *)
95   in(c, enc_m3:bitstring);
```

```
96    let sig_Y = sdec(enc_m3, dk_R) in
97    (* Check the message content and its sig *)
98    if(eq((data_Y, data_R), getmess(sig_Y)) && checksig(sig_Y,
       s_pub_Y)) then (
99
100     (* Send 'funding_signed' *)
101     let sig_R = sign((data_Y, data_R), s_priv_R) in
102     let enc_m4 = senc(sig_R, ek_R) in
103     out(c, enc_m4);
104
105     (* Receive 'funding_locked'*)
106     in(c, enc_m5:bitstring);
107
108     (* Send 'funding_locked' *)
109     let enc_m6 = senc(empty, ek_R) in
110     out(c, enc_m6);
111
112     if s_pub_Y = s_pub_I then
113     event end_R(pk(s_priv_R));
114     out(c, senc(secret_ek_R, ek_R))
115   ).
116
117 process
118   new s_priv_I:private_key;
119   new s_priv_R:private_key;
120   new k1:private_key;
121   new k2:private_key;
122   let s_pub_I = pk(s_priv_I) in out(c, s_pub_I);
123   let s_pub_R = pk(s_priv_R) in out(c, s_pub_R);
124   ( (!initiating_node(s_priv_I, s_pub_R, k1, k2)) |
      (!responding_node(s_priv_R, s_pub_I, k2, k1)) )
```

Appendix E

The Lightning Network onion routing protocol

```
0 (* Public channel between the source and the hop node *)
1 free c1:channel.
2 (* Public channel between the hop and the recipient node *)
3 free c2:channel.
4
5 (* Public key cryptography *)
6 type private_key.
7 type public_key.
8 fun pk(private_key): public_key.
9
10 (* Shared secret between the node_S and a node_H *)
11 type shared_secret.
12 fun ECDH(private_key, public_key): shared_secret.
13 fun to_bitstring(shared_secret): bitstring [typeConverter].
14
15 (* ECDH *)
16 equation forall priv1:private_key, priv2:private_key;
17   ECDH(priv1, pk(priv2)) = ECDH(priv2, pk(priv1)).
18
19 (* ChaCha20 encryption - symetric (bijections) - using
     shared secret as rho-key *)
20 fun senc(bitstring, bitstring): bitstring.
21 fun sdec(bitstring, bitstring): bitstring.
22 equation forall m:bitstring, k:bitstring; sdec(senc(m,k), k)
     = m.
```

```
23 equation forall m:bitstring, k:bitstring; senc(sdec(m,k), k)
   = m.
24
25 (* HMAC *)
26 fun HMAC(bitstring, shared_secret): bitstring.
27
28 (* Packet structure: public_key, enc_next_hop_info, HMAC *)
29 fun onion(public_key, bitstring, bitstring): bitstring
   [data].
30
31 (* Blinding factor generation *)
32 type blinding_factor.
33 fun calc_blinding_factor(public_key, shared_secret):
   blinding_factor.
34
35 (* Key blinding *)
36 fun blind_priv(private_key, blinding_factor): private_key.
37 fun blind_pub(public_key, blinding_factor): public_key.
38
39 (* Bitstring equality *)
40 fun eq(bitstring, bitstring): bool
41   reduc
42     forall x:bitstring; eq(x,x) = true
43     otherwise forall x:bitstring, y:bitstring; eq(x,y) =
   false.
44
45 (* Events *)
46 event begin(public_key).
47 event end(public_key).
48
49 (* Empty data *)
50 free null_data:bitstring.
51
52 (* Participants private keys *)
53 free s_priv_S:private_key [private].
54 free s_priv_H:private_key [private].
55 free s_priv_D:private_key [private].
56
57 free ek_S:bitstring [private].
58 free ek_H:bitstring [private].
59 free ek_D:bitstring [private].
```

```
60
61 (* Data for the hop node *)
62 free hop_data:bitstring [private].
63 (* Data for the recipient node *)
64 free recipient_data:bitstring [private].
65
66
67 (* The node_S subprocess *)
68 let node_S(s_priv_S:private_key, s_pub_H:public_key,
     ek_S:bitstring) =
69
70   (* Letting the environment decide on the recipient *)
71   in(c1, pub_x:public_key);
72
73   (* Create shared secrets and ephemeral keys *)
74   new e_priv_1:private_key;
75   let s1 = ECDH(e_priv_1, s_pub_H) in
76   let b = calc_blinding_factor(pk(e_priv_1), s1) in
77   let e_priv_2 = blind_priv(e_priv_1, b) in
78   let s2 = ECDH(e_priv_2, pub_x) in
79   event begin(pub_x);
80
81   (* Encrypting the onion and calculating the HMACs (last
     HMAC is set to null) *)
82   let enc_recipient_data = senc((recipient_data, null_data),
      to_bitstring(s2)) in
83   let enc_hop_data = senc((hop_data, enc_recipient_data,
     HMAC(enc_recipient_data, s2)), to_bitstring(s1)) in
84
85   (* Send the onion to the first hop *)
86   let o = onion(pk(e_priv_1), enc_hop_data,
     HMAC(enc_hop_data, s1)) in
87   out(c1, senc(o, ek_S)).
88
89
90 (* The hop node subprocess *)
91 let node_H(s_priv_H:private_key, ek_H:bitstring,
     dk_H:bitstring) =
92
93   (* Receive and process the onion *)
94   in(c1, enc_x:bitstring);
```

```
95     let x = sdec(enc_x, dk_H) in
96     let onion(e_pub_1, enc_hop_data, h) = x in
97     let s1 = ECDH(s_priv_H, e_pub_1) in
98     (* If attacker controls the node *)
99     (* out(c1, s_priv_H); *)
100    (* out(c1, e_pub_1); *)
101    let (forward_data:bitstring, enc_recipient_data:bitstring,
        h_next:bitstring) = sdec(enc_hop_data, to_bitstring(s1))
        in
102
103    (* Check the HMAC and the load *)
104    if(eq(HMAC(enc_hop_data, s1), h)) then (
105      if(eq(forward_data, hop_data)) then (
106
107        (* Send the outgoing packet *)
108        let b = calc_blinding_factor(e_pub_1, s1) in
109        let out_o = onion(blind_pub(e_pub_1, b),
        enc_recipient_data, h_next) in
110        out(c2, senc(out_o, ek_H))
111      )
112    ).
113
114
115 (* The recipient node subprocess *)
116 let node_D(s_priv_D:private_key, dk_D:bitstring) =
117
118    (* Receive and process the onion *)
119    in(c2, enc_x:bitstring);
120    let x = sdec(enc_x, dk_D) in
121    let onion(e_pub_2, enc_recipient_data, h) =  x in
122    let s2 = ECDH(s_priv_D, e_pub_2) in
123
124    (* Check the HMAC *)
125    if(eq(HMAC(enc_recipient_data, s2), h)) then
126    (
127      let (data:bitstring, h_next:bitstring) =
        sdec(enc_recipient_data, to_bitstring(s2)) in
128      if(eq(data, recipient_data) && eq(h_next, null_data))
        then (
129        event end(pk(s_priv_D))
130      )
```

```
131    ).
132
133 (* Keys and private data secrecy testing *)
134 query attacker(hop_data).
135 query attacker(recipient_data).
136 query attacker(s_priv_S).
137 query attacker(s_priv_H).
138 query attacker(s_priv_D).
139 query attacker(ek_S).
140 query attacker(ek_H).
141
142 (* Authenticity testing *)
143 query p:public_key; inj-event(end(p)) ==>
      inj-event(begin(p)).
144
145 (* Main process *)
146 process
147   (* Make participants public keys 'public' by outputing
      them in the channel *)
148   let s_pub_S = pk(s_priv_S) in out(c1, s_pub_S);
149   let s_pub_H = pk(s_priv_H) in out(c1, s_pub_H);
150   let s_pub_D = pk(s_priv_D) in out(c1, s_pub_D);
151
152   (* Start the participants subprocesses in parallel *)
153   ( (!node_S(s_priv_S, s_pub_H, ek_S)) | (!node_H(s_priv_H,
      ek_H, ek_S)) | (!node_D(s_priv_D, ek_H)))
```