UNIVERSITY
OF UDINE

Department of
Mathematics, Computer Science and Physics

BACHELOR THESIS IN
INFORMATICA

Formal Verification of the Session Protocol in the Symbolic Model

CANDIDATE
D'Ambrosi Denis

SUPERVISOR
Prof. Miculan Marino

Academic Year 2021-2022

Institute Contacts
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

to mom and dad,
who supported me all the way until the end

Acknowledgements

I would love to dedicate some lines to the people who helped me achieving this milestone.

First of all, I want to thank my supervisor, Prof. Marino Miculan, for his patience and expertise without which this dissertation would not have been possible. I would also like to thank Matteo Paier and Alessandro Zanatta for being so nice and available in helping me overcome some hardships during the experimentation phase of this research.

Then, I really need to thank my roommates Zano, Pippo and Nic, my study (and laughs) companions Baz and Roby, my priceless life partner Federica and all the other friends I had the luck to meet in Udine for making this last year one of the best in my life so far, both academically and otherwise.

Finally, I want to dedicate a special thanks to my family, for their constant encouragement and support, and my friends Leo, Rigna, Megna, Ema and Samu, for always standing by my side both in the best and in the hardest times of these last years.

Contents

1

Introduction

In the last few decades we firsthandedly witnessed a rapid evolution of our way of life into a progressively more-virtual, interconnected society. While this transition has brought many benefits to our everyday life, such as direct access to information, instantaneous communication and less burdensome bureaucracy, as a consequence it entailed an intense effort focused on providing the same security guarantees of the real world within this new digital infrastructure. In particular, an active field of cybersecurity research revolves around the creation, verification and implementation of encryption protocols aimed at defending online connections from attackers' reach. To prove that a given encryption scheme is compliant to its security specification, researchers verify its safety features by formalizing and analysing its dynamic within a formal model. This verification process assures us that, within the assumptions taken by the model itself, the protocol is safe against entire classes of attacks. These security schemes are then applied to virtual exchanges of any purpose and scale (e-voting procedures, chat apps and payment systems to name a few) to protect their data and users from cybercriminals, governments and other external agents.

Among all possible contexts in which strong security protocols are considered indispensable, messaging platforms are an interesting example of intrinsic balance between ease of use and security. Each application prioritizes its goals based either on popularity or criticality of its data, thus inevitably skimping on the less important one. A compelling attempt to merge both worlds was made by the Oxen Privacy Tech Foundation, which decided to create a user-friendly, but yet security-centric messaging cross-platform application: Session. This open-source app promises to provide E2E encrypted anonymous conversations through the use of a distributed and dynamic architecture. At first sight this set of features may represent the new industry standard for ordinary communication, but at the best of our knowledge these bolds claims have never been proved within a formal system, thus we decided to undertake the symbolic verification of Session's protocol.

In this thesis we are going to demonstrate that the aforementioned app conforms to its security specification for peer-to-peer chats by formalizing all of its sub-protocols and privacy features using the Tamarin prover. This automatic tool will allow us to show that, within reasonable assumptions and restrictions, the platform can be considered secure from a symbolic standpoint. Moreover, since Session's publicly available project documentation is messy and contradictory at times, this dissertation currently represents the only complete specification of the underlying protocol in all of its phases. Finally, we will also elaborate on the need of an alternative paradigm to analyse onion routing schemes by verifying an

anonimous routing protocol within a restricted version of the Dolev Yao model.

Similarly to the works of N. Kobeissi et al. [62] and M. Miculan and N. Vitacolonna [60], we will undertake the automated symbolic verification of a popular chatting platform. However, since (unlike Telegram and Signal) Session was engineered within a distributed architecture, we are going to formalize and check some innovative solutions developed to obtain anonymous communication through a decentralized context.

We will firstly introduce the symbolic model along with Tamarin prover, providing an overview of its features and internal functioning from a user's perspective. Then, in chapter 3, we will focus on Session and its major engineering choices in order to give a brief description of its network architecture. Chapters 4 and 5 present the actual experimentation for this dissertation: each one deals with the reverse-engineering, summarization, formalization and verification of a part of the entire protocol (in chapter 4 we analyze the E2E protocol treating onion routing as an abstraction, while in chapter 5 we study more deeply the routing procedure). A brief summary of the proofs results and Tamarin's performance is presented at the end of each chapter. Finally, we will finish this thesis by presenting the conclusions of our research. Note that, since the E2E protocol relies heavily on an external library function to provide peer-to-peer encryption, we analyze the API called in the additional appendix A.

2

Automated Symbolic Verification of security protocols

Cryptographic protocols are becoming more ubiquitous and complicated by the day, thus increasing the probability of possible wrong engineering choices that may lead to inconsistencies and vulnerabilities. To ensure that a certain protocol is secure, two main formal checking models have been proposed: the *standard model* (also known as the *computational model*) and the *symbolic model*. In this thesis we'll undertake the symbolic verification of a messaging protocol, so now we'll focus on the latter.

2.1 The symbolic model

Known also as the *Dolev Yao* model, the symbolic model abstracts from cryptography by substituting real-world cryptographic operations with term-algebras [45]. Encryption schemes are formalized through binary isomorphisms within $\{0,1\}*$: given a symmetrical key k and the relevant pair of cryptographic encryption and decryption primitives senc_k and sdec_k, symmetrical cryptography is defined through the following identity:

$$\text{sdec}_k \ \text{senc}_k = 1 \tag{2.1}$$

Only the entities that are in possess of k are able to encrypt or decrypt any message with it [36]: this assumption is known as perfect cryptography. Furthermore, given a message M and its image $\text{senc}_k \ M$, we assume that it is impossible for an attacker who does not know k to:

- guess or bruteforce k;
- manipulate $\text{senc}_k \ M$
- infere any information about M from $\text{senc}_k \ M$.

Similarly to equation 2.1, asymmetric encryption and signing are modeled through the following identity:

$$\text{adec}_{pr} \ \text{aenc}_{pub} = \text{adec}_{pub} \ \text{aenc}_{pr} = 1 \tag{2.2}$$

In this case, we do not consider a single key used for both encryption and decryption k, but instead an entangled pair of keys pr and pub. The assumptions for cryptography are the same as above.

A protocol is modeled as a series of formal terms (messages) exchanged by abstract machines (clients) through an attacker-controlled network. We assume that any connection can be eavesdropped by a malevolent agent that is also able to intercept, modify, forge and drop messages on the fly, with the only exception of the encryption constraints previously described. After formalizing a protocol, its security goals can be expressed as [56]:

- Trace properties: invariants that should hold for each possible execution of the protocol;
- Observational equivalence properties: properties that an attacker should not be able to distinguish between two different runs of the protocol.

The use of algebraic terms within the Dolev Yao model allows for simple abstractions during formal checking. As opposed to the aforementioned standard model, where bitstrings take the place of terms and perfect cryptography is not an assumption, the symbolic model is less realistic since it is able to guarantee only high-level security goals on the conjecture of flawless implementations of the cryptographic primitives actually used.

2.2 Limits of the symbolic model

In order to streamline the proof of security properties belonging to a certain protocol, multiple automatic tools were proposed [56]. These softwares, in order to prove security goals, have to compute the set of terms that an attacker is able to deduce while the protocol is executing, although since this set can possibly be infinite, the general problem suffers of both undecidability and infinite state space. To convince ourselves that this is the case, we just need to think about the fact that we can have an infinite number of sessions for each protocol, each one with different nonces and messages of unlimited size. Without bounding at least two of these sources of infinity, termination can not be guaranteed by any tool in this category.

In particular, as shown by N. Durgin et al [61], by restricting different sources we can ensure to reduce insecurity to different complexity classes:

- if we limit the number of nonces and messages, we end up with a DEXP-complete problem;
- if we restrict the number of sessions (and thus the number of nonces and messages), the problem will still be affected from infinite state space, but its solution will be NP-complete; this workaround was adopted by a variety of tools such as Cl-AtSe [72], OFMC [42] and SATMC [30].

Alternatively to bounding the sources of infinity, some software address undecidability by requiring human input (for example, Tamarin Prover [65] features an interactive mode that allows the user to decide which security goals prioritize based on intermediate constraint systems), returning inconclusive results (for example, ProverIf [35] may return an invalid attack trace) or even allowing non-termination.

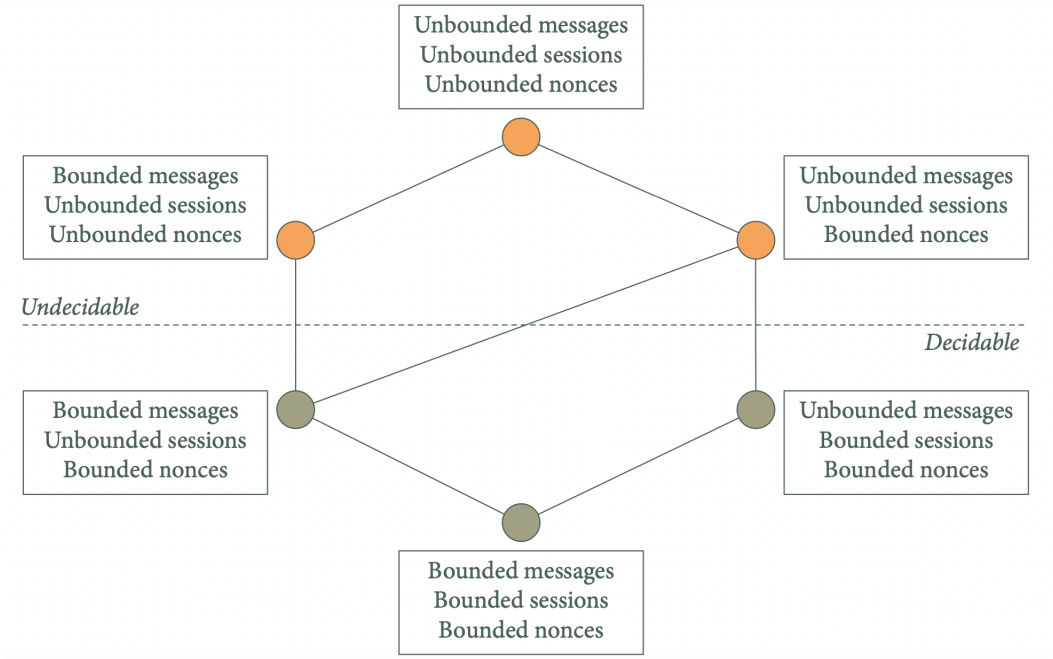

Figure 2.1: Decidability in symbolic verification. Image taken by N. Vitacolonna's presentation [75]

2.3 Tamarin Prover Overview

The automatic tool we used to complete the sperimentation for this thesis is Tamarin Prover (which from now on we will call Tamarin for the sake of semplicity). We will now do a quick overview of its syntax, semantics and internal functioning; please refer to S. Meier's [59] and B. Schmidt's [67] PhD thesis and [32] for further information about the theoretical foundations regarding this tool, as long as its official manual [71] to resolve any doubts about its use.

2.3.1 Basic definitions

Tamarin allows protocol (and adversary) modeling through multiset rewriting rules.

Definition 2.3.1 (Multiset rewriting rule)**.** Given a multiset $\Gamma_t = \{F_0, ..., F_n\}$ and a sequence of multisets $trace_t = \langle a_0, ..., a_{t-1} \rangle$ at a time t, we can define a rewrite rule as a triple of multisets $RR = \langle L, A, R \rangle$ (written as $RR = L - [A] \to R$) such that:

- we can apply RR to Γ_t if there is at least one ground instance (i.e. an instance with no variables) $rr = l - [a] \to r$ of RR so that $l \subseteq^{\#} \Gamma_t$
- applying rr to Γ_t yields to a new state Γ_{t+1} and an increased trace $trace_{t+1}$ obtained as

$$\Gamma_{t+1} = \Gamma_t \setminus^{\#} l \cup^{\#} r \tag{2.3}$$

$$trace_{t+1} = \langle a_0, ..., a_{t-1}, a_t \rangle \tag{2.4}$$

in which $\setminus^{\#}$ and $\cup^{\#}$ are the multiset equivalent operations for set difference and union. From now

on, we will refer to L, R and A as the multisets of *premises*, *conclusions* and *action facts* of a rule (in this order).

Note that within the trace $trace_{t+1}$, a is associated to the time istant t: this will be useful later on since security properties will be specified through first order logic guarded formulas. To be precise, a *trace* is the sequence of sets of ground (action) facts that happened during the execution of a series of rewriting rules.

More formally, given a set of rules P, all the possible traces generated by executions of P are

$$traces(P) = \{\langle A_1, ..., A_n\rangle | \exists S_1, ..., S_n . \varnothing^{\#} \xrightarrow{A_1} S_1 \xrightarrow{A_2} ... \xrightarrow{A_n} S_n \text{ and no ground istance of Fresh() is used twice } \}$$

For the purpose of this thesis is also worth mentioning that since in a rewriting rule $L - [A] \rightarrow R$ any of the three multisets could be empty, a trace in which all istances of $\varnothing^{\#}$ are removed is called an *observable trace*.

Within Tamarin, all multisets are composed by facts. Facts feature a precise arity, are named with a starting capital letter and can be either *linear* or *persistent*:

- linear facts can be consumed only once and are useful to model state transitions and ephemeral messages;
- persistent facts can be consumed unlimited times and are optimal to model enduring knowledge (and are prefixed by an exclamation mark). Note that the introduction of persistent facts requires a slight change to equation 2.3: whereas before all elements of l were removed from Γ_t, now only the linear facts beloging to l will be subtracted, while persistent facts will never be consumed.

In turn, facts are made up of terms, which can be public (prefixed by a \$), freshly created (prefixed by a ~) or normal messages (note that public and fresh terms are sub-categories of normal messages).

2.3.2 Term algebra

In order to continue explaining Tamarin's functioning, we have to take a look at the theoretical foundations behind the algebraic terms introduced in 2.1.

Definition 2.3.2 (Signature)**.** A signature Σ is a finite set of functions of defined arity.

Definition 2.3.3 (Σ-terms)**.** Given a signature Σ and a set of variables χ, with $\Sigma \cap \chi = \varnothing$ we can define the set of Σ-terms $\mathcal{T}_\Sigma(\chi)$ as the minimal set such as

- $\chi \subseteq \mathcal{T}_\Sigma(\chi)$
- $t_1, ..., t_n \in \mathcal{T}_\Sigma(\chi) \wedge f/n \in \Sigma \implies f(t_1, ..., t_n) \in \mathcal{T}_\Sigma(\chi)$

Note that f/n indicates a function f of arity n.

Definition 2.3.4 (Substitution)**.** Given a signature Σ and a set of variables χ, with $\Sigma \cap \chi = \varnothing$, a substitution is a function $\sigma : \chi \to \mathcal{T}_\Sigma(\chi)$.

Similarly, given a function $f/n \in \Sigma$, a substitution can be extended to a mapping $\sigma' : \mathcal{T}_\Sigma(\chi) \to \mathcal{T}_\Sigma(\chi)$ such that

$$f(t_1, ..., t_n)\sigma' = f(t_1\sigma, ..., t_n\sigma)$$

Definition 2.3.5 (Equation over Σ)**.** Given a signature Σ, a set of variables χ, with $\Sigma \cap \chi = \varnothing$, an equation over Σ is a pair of terms (t, u) with $t, u \in \mathcal{T}_\Sigma(\chi)$. Note that in this case, the equation would be written $t \simeq u$. In order to break loops while simplyfing terms, equations can be oriented (as $t \to u$).

By introducing a set of equations E we can create a congruence relation $=_E$ on terms t and, consequently, equivalence classes $[t]_E$. The congruence relationship is known as an *equational theory*. Introducing equational theories allows us to unify terms based on the quotient algebra $\mathcal{T}_\Sigma(\chi)/_{=_E}$: two terms $t, u \in \chi \cup \mathcal{T}_\Sigma(\chi)$ are equal in modulo E if and only if they belong to the same class:

$$t =_E u \iff [t]_E = [u]_E$$

Definition 2.3.6 ((Σ, E)-Unification)**.** Given a signature Σ, a set of variables χ, with $\Sigma \cap \chi = \varnothing$ and an equational theory E, two terms $t, u \in \mathcal{T}_\Sigma(\chi)$ are (Σ, E)-unifiable if there is at least a mapping σ such that $t\sigma =_E u\sigma$.

Normally, unification modulo theories is undecidable [68]: thus Tamarin recommends users to only define *subterm convergent theories* to ensure termination, even though the software never actually checks this constraint.

Definition 2.3.7 (Convergent Theory)**.** Before defining what a convergent theory is, we have to define both *terminating* and *confluent* theories:

- A terminating theory is an equational theory which ensures that each term has a *normal form* that can be reached through an arbitrary (but finite) number of substitutions.
- A confluent theory is an equational theory that ensures that if a term t can be rewritten as both terms t_1 and t_2, then there must be a fourth term t' that can be reached through an arbitrary number of substitutions from both t_1 and t_2.

A convergent theory is an equational theory that is both terminating and convergent.

Definition 2.3.8 (Subterm Convergent Theory)**.** A subterm convergent theory is an equational theory that is convergent and, for each equation $e : L \to R, e \in E$, R is either ground and in normal form or a proper subterm of L.

Tamarin uses equational theories to model algebraically the cryptographic primitives, such as symmetrical/asymmetrical encryption/decryption, hashing, signing, pair construction/deconstruction, elevation to power, etc... 10 built-in theories are provided to allow the user to easily formalize cryptographic protocols:

1. `hashing`
2. `asymmetric-encryption`
3. `signing`
4. `revealing-signing`
5. `symmetric-encryption`
6. `diffie-hellman`
7. `bilinear-pairing`
8. `xor`
9. `multiset`
10. `reliable-channel`

2.3.3 Default facts and rules

Tamarin provides 4 built-in special facts that allow to model the generation and transmission of messages through the network:

- `Fr(~msg)` allows to create new messages `msg`. Note that each value is freshly generated (let us recall that Tamarin allows an infinite number of nonces), so the same ground value can not appear in two different instances of `Fr(~msg)`.
- `K(msg)` indicates that the attacker deduced `msg` from the network.
- `Out(msg)` allows machines to send messages to the network.
- `In(msg)` allows machines to query the network for incoming messages.

These facts are defined through default multiset rewriting rules according to Dolev Yao's assumptions:

- `[] --[]-> [ Fr(~msg) ]` allows for the generation of new fresh values.
- `[ Fr(~msg) ] --[]-> [ K(~msg) ]` allows for the generation of new fresh values by the attacker.
- `[ Out(msg) ] --[]-> [ K(msg) ]` allows the attacker to eavesdrop all messages travelling through the network.
- `[ K(msg) ] --[ K(msg) ]-> [ In(msg) ]` allows user to retrieve messages from the attacker-controlled network.
- `[] --[]-> [ K($x) ]` allows the attacker to discover all public names.
- `[ K(x1,...,xn) ] --[]-> [ K(f(x1,...,xn)) ]` allows the attacker to apply n-ary functions to arguments he already knows.

2.3.4 Trace properties

As previously anticipated, in Tamarin security properties are expressed as guarded fragments of first order logic. The syntactical construct adopted by Tamarin to indicate a property to prove is `lemma`. Lemmas can be either of type `exists-trace`, which are demonstrated by finding a single valid protocol trace verifing the formula, or `all-traces`, which are proved by negating the property and trying to find a counterexample. To specify a lemma, we can combinate the following atoms:

- false $\bot$;
- logical operators $\neg \ \wedge \ \vee \ \implies$;
- quantifiers and variables $\forall \ \exists \ a \ b \ c$;
- term equality $t_1 \approx t_2$;
- time point ordering and equality $i < j$ and $i = j$;
- action facts at time points $F@i$ (for an action fact F at timepoint i).

Keeping in consideration that also properties can define sets of traces similarly to protocol rules, we can define correctness as follows:

Definition 2.3.9 (Correctness)**.** Given a set of protocol rules P and a property to prove ϕ, P is correct in regards to ϕ if and only if the set of traces generated by P is a subset of the one generated by ϕ:

$$P \vDash \phi \iff traces(\phi) \subseteq traces(P)$$

On the contrary, if $P \nvDash \phi$, all traces belonging to $traces(P)/traces(\phi)$ represent valid attacks.

Note that property proving is done by Tamarin through *constraint systems* resolving. In turn, these systems are simplified by executing a backwards search within the relative protocol's *dependencies graph* (a graph that, for each fact used by a Tamarin *theory*, describes the list of possible rules usable to obtain it). The theoretical foundations behind property proving are beyond the scope of this thesis, but are explained in detail in [67] [59]. As end users, we can be satisfied by knowing that Tamarin's verification algorithm, although does not guarantee termination, is both *sound* and *complete* [67] [59] [50].

2.3.5 Observational equivalence properties

When operated in *Observational Equivalence Mode* (refer to the official documentation [71] for further reference), Tamarin allows for the use of the `diff/2` operator [41], that permits to demonstrate *diff equivalence* properties. To understand what a diff equivalence property is, we have to firstly introduce the concept of *trace equivalence*

Definition 2.3.10 (Trace equivalence)**.** Two different protocols P_1, P_2 are trace equivalent if and only if for each trace of P_1 exists a trace of P_2 so that the messages exchanged during the two executions are indistinguishable.

This is the weakest (and thus most suitable for security verification) form of observational equivalence. Since many problems in this category are undecidable in nature [39], Tamarin only allows diff equivalences in aid of termination.

Definition 2.3.11 (Diff equivalence)**.** Two protocols P_1, P_2 are diff-equivalent if and only if they have the same structure and differ only by the messages exchanged.

Thus, the two protocol have the same structure during execution.

The previously introduced `diff(left, right)` operator allows to define a protocol rule that can be instantiated either with the `left` or `right` value. For each one of them, Tamarin constructs the relative dependencies graph and checks if they are equivalent, which is a sufficient criterion for observational diff-equivalence. Please note that a formal definition of *dependencies graph equivalence*, along with the subsequent demonstration of sufficiency are available in the introductory paper for the observational equivalences in Tamarin written by D. Basin, J. Dreier and R. Sasse [41].

2.3.6 Aiding termination

Since protocol verification is an inherently undecidable problem, Tamarin provides some additional functionalities engineered to aid termination.

Source lemmas

As explained before, Tamarin elaborates the dependencies graph related to a theory before beginning constraint solving. Since the prover uses an untyped system, in certain cases it is not able to deduce the source of one or more facts, causing partial deconstructions. As explained by V. Cortier [73], for example, this situation occurs whenever the same message has to travel across the network multiple times. To mitigate this issue, Tamarin allows to define a lemma with the additional tag `[sources]`, which is automatically proved during the precomputation phase and allows to declare the origin of one or more messages.

An example of this type of lemma, along with its explanation, will be provided in section 4.3.7

Note that V. Cortier, S. Dealune and J. Dreier proposed an algorithm for automatic source lemma generation [73] that has already been integrated in Tamarin (run the program with the additional `--auto-sources` tag to execute the extension), but sometimes it leads to non-termination during the precomputation phase.

Oracles

During constraint-solving, Tamarin uses its built-in *smart* heuristic (refer to the relevant section of the manual [71] for further reference) to sort the list of security goals to check. Sometimes, though, the algorithm prioritizes the wrong goals, leading to loops in the search and thus to non-termination. In an attempt to prevent this behavior, Tamarin provides also two variants of the *consecutive* heuristic, which guarantee to avoid delaying any goal infinitely to exclude starvation. This causes bigger proofs and still fails to solve the problem sometimes.

By knowing how to manually guide the search (for example after doing some practice with the built-in interactive mode), we can code (in any programming language of choice) an external *oracle*, which

will be automatically run by the prover to determine the right priority objective to solve within a list of current goals. This user-defined software receives the name of the lemma and the indexed goal list (sorted by the smart heuristic) as input, and returns the re-ranked list (or, alternatively, only its first element) as output. Note that oracles are generally stateless: for each step of the proof, the program is executed from scratch, with only the name of the considered lemma and the list of current security goals as input.

Examples of oracles will be introduced in sections 4.4 and 5.4.7.

Restrictions

Similarly to lemmas, *restrictions* are specified through guarded fragments of first order logic. The difference between the two is that while lemmas express properties to prove, a restriction limits the possible traces of a protocol to the executions that verify the specified formula.

Re-use lemmas

Finally, the last functionality we introduce are *re-use lemmas*: defined with the `[reuse]` keyword, these formulas, once proved, can be used by Tamarin in the demonstration of the subsequent specified lemmas.

3

Session: an application for private messaging

Due to the recent widespread of social networks and messaging apps, our default way of communication has progressively transitioned from face-to-face talking towards digital chatting through the network. Since on the internet it is quite easy to keep track of any information, this evolution brought up several concerns about the privacy of virtual conversations [38] [58] [43]. Nearly all of the viral platforms publicly available today feature at least some kind of security mispractice, such as a centralised architecture, closed source code, collection of personal information or weak encryption schemes/protocol, that puts the user's privacy in jeopardy.

As shown in table 3.1, almost all messaging apps' architectures allow governments (or, similarly, attackers) to retrieve information about their users and the relative conversations from the network. Ideally, a security-centric chat platform should be engineered in a way that makes conversations data (and metadata) inaccessible to any party except from the intended participants, even considering malevolent servers and ISPs.

	Information Accessed							
App	Subscriber Data	Message Sender-Receiver Data	Device Backup	IP Address	Encryption Keys	Date-Time Information	Registration Time Data	User's contacts
iMessage	✓	✓	✓	✗	✓	✓	✓	✓
Line	✓	✓	✗	✗	✗	✓	✓	✓
Signal	✗	✗	✗	✗	✗	✓	✓	✗
Telegram	✗	✗	✗	✗	✗	✗	✓	✗
Threema	✓	✗	✗	✗	✓	✓	✓	✗
Viber	✓	✗	✗	✓	✓	✓	✓	✗
WeChat	✓	✗	✗	✓	✗	✗	✓	✗
WhatsApp	✓	✓	✗	✗	✗	✓	✓	✓
Wickr	✓	✗	✗	✗	✗	✓	✓	✗

Table 3.1: Summary of the recently undisclosed FBI document *Lawful Access*, that outlines what data the government is able to retrieve from each chat platform. For more information, please refer to [29]

In 2013 [6], the idea of a truly secure chat app was proposed along with the Tox Protocol [27]. In the (advanced [26]) FAQ section of their website [25] is currently possible to read that their main goal was to create a distributed, E2E encrypted, onion routed P2P network. Unfortunately, as it is easy to

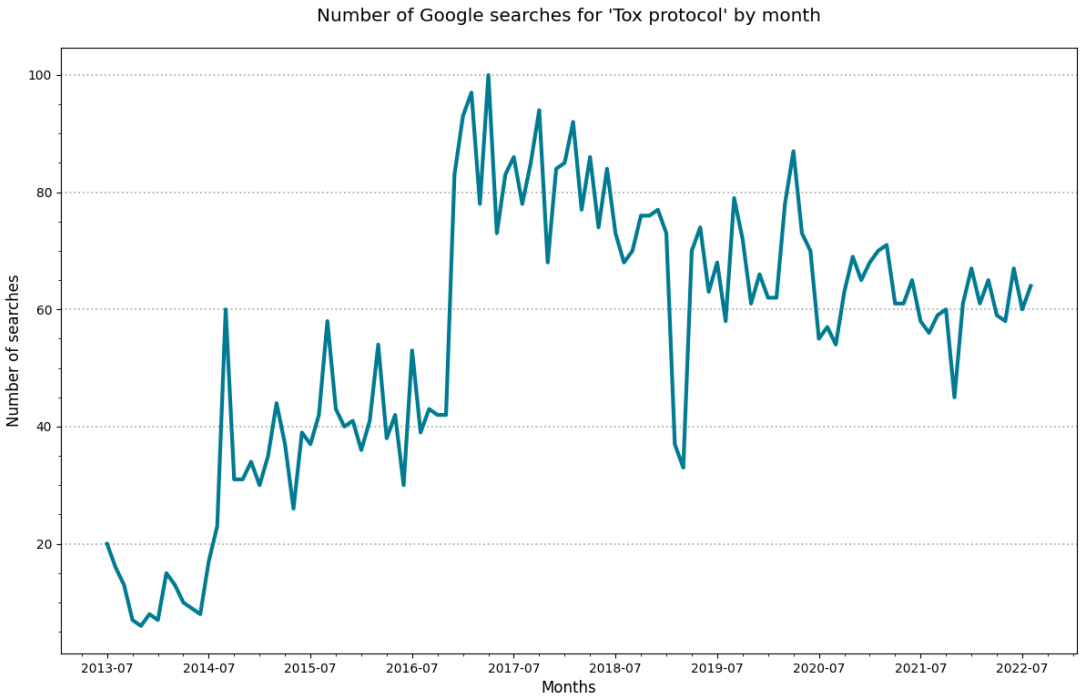

Figure 3.1: Trend of monthly searches worldwide for 'Tox protocol' according to Google Trends [11].

see from graph 3.1, after an initial spread Tox did not gain much traction, probably due to the lack of usability of its clients and the protocol itself. At the moment, after 6 years, its major Android client implementation Antox [2] has not been updated in almost half a decade and has never even reached a million downloads.

Since there was no clear alternative to Tox available, Oxen Privacy Tech Foundation's [14] development team decided to take inspiration from both worlds [1] to create an easy-to-use but extremely safe messaging application: Session. Their website [24] describes the platform as:

> ⟨⟨Session is an end-to-end encrypted messenger that minimises sensitive metadata, designed and built for people who want absolute privacy and freedom from any form of surveillance.⟩⟩

Since February 2020, Session's official Android client [19] has passed the 1 million downloads mark, with a mean rating on Google Play Store of 4.1 stars out of 5. Since the usability criticalities of Tox have been clearly overcomed (as demonstrated by its viral spread), we'll now focus on analyzing its security features. For further details, consult the official lightpaper [70], whitepaper [54], blog [23] and website [24].

3.1 No personal information involved

Normally, to sign up on a messaging service we provide personal information (name, email, telephone number, etc...), that are associated to our digital account in order to allow other users to contact us

[1] Note that Tox is actually introduced as an ispiration model in their official lightpaper [70].

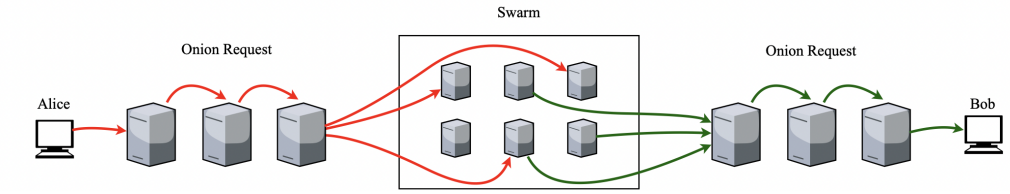

Figure 3.2: Simplified view of the E2E sending procedure. Image taken directly from Session's whitepaper [54].

on the platform. This is a problem since it de-anonymises accounts in order to increase usability. For malevolent agents it is easier to keep track of conversations if not properly encrypted, while for users it is harder to create new accounts when compromised. To mitigate both issues, Session does not require any information upon registration: during initialization the client generates a fresh pair of 128 bit encryption keys, whose public half (also called *Session ID*) will be used from that moment on to identify the relative account within the network. As a consequence, Session IDs are secure, recyclable and anonymous at the same time. Users have to exchange public keys in order to communicate, thus also eliminating the possibility of man-in-the-middle attacks that exploit contact registries.

Note that private keys can be restored through a mnemonic recovery phrase generated during sign-up, so the same account can be shared through multiple devices.

3.2 Decentralized Architecture

Having a centralized architecture allows for simpler routing protocols and managing politics, at the cost of a single point of failure and metadata leakage. To avoid this, Session features a decentralized network of *Service Nodes* that are used for message storing and retrieval. A small subset of nodes belonging to the network are associated to a user during its initialization: this set of servers is known as the user's *Swarm*. In order to send a message to a user, clients first have to retrieve a list of nodes belonging to its Swarm and forward the encrypted envelope to them through the network. Similarly, the receiver has to query the same nodes to recover its messages. Swarms are managed completely by the nodes themselves, but clients automatically report malfunctioning nodes in order to guarantee availability by excluding servers that do not perform well from the service.

Note that, similarly to clients, also nodes are identified within the network by the public half of their cryptographic keypair.

3.2.1 Onion Routing

In order to prevent metadata leakage, Session exploits its decentralized structure to add another layer of security through an implementation of onion routing. Messages from clients to nodes are dispatched after multiple hops within the network; in particular, each onion path is composed by three hops (including the destination node intended). In order to achieve both secrecy and anonymity, the client progressively encrypts the data in a layered structure and sends the whole ciphertext to a *guard node* (the first node of the path). This node will be able to decrypt only the external layer and will have to forward the internal ciphertext to the specified node, that will do the same as shown in picture 3.3. Only the receiving node is able to actually decrypt and read the data sent by the client. By adopting this method, assuming

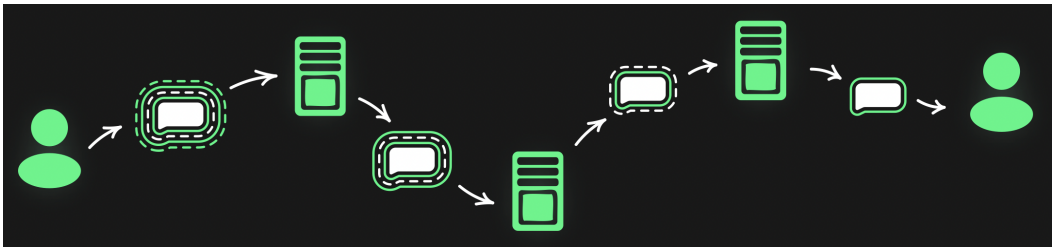

Figure 3.3: Simplified view of the routing protocol. Image taken directly from Session's lighpaper [70].

there are at least two honest nodes in the path (see relevant section 5.3), an attacker is not able to recover the IPs of both endpoints of the communication.

3.2.2 The Loki Blockchain

By looking at Session's onion routing somebody may think that it is almost identical to Tor's own protocol and thus conclude that it suffers from the same vulnerabilities: namely, Sybil attacks [46]. On the contrary, under the threat assumption of an attacker with limited economical resources, Session is protected from adversary monopolization of the network by its parent company's blockchain: Loki Blockchain [10]. In order to register a Service Node an agent has to conduct a staking transaction that requires him to buy and assign 15000 $OXEN (about €3428.39 at the moment according to [1]) to the node itself [15]. As explained in Oxen's public docs [13], this prevents the attacker from owning a large part of the network:

> ⟨⟨As an attacker accumulates $OXEN, the circulating supply decreases, in turn applying demand-side pressure and driving the price of $OXEN up. This effect spirals, making it increasingly costly for additional $OXEN to be purchased and thus making an attack prohibitively expensive.⟩⟩

Another positive side of the association of Loki Blockchain to Session's infrastructure is that nodes receive monetary rewards according to their service performances, thus incentivizing servers to behave honestly and provide an high standard of functioning.

3.3 Additional services

In addition to P2P chats, Session provides group chats and open groups. The main difference between the two is that whereas normal group chats are treated quite similarly to two-parties conversations with comparable security goals, open groups feature only transport-level encryption/anonymity and are managed externally to the Service Node network: these groups (meant for non-critical community exchange) are hosted on separated nodes that do not require the 15000 $OXEN transaction and are completely controlled by their host.

Anyway, the study of both kinds of group chats is out of the scope of this thesis.

3.4 Problems with the specification

By reading both Session's whitepaper [54] and technical deep dive [21], it is easy to find multiple inconsistencies regarding the E2E encryption protocol. Whereas in the first we can read about an adaptation of Signal's protocol [53] featuring the Double Ratchet Algorithm [49], in the second we learn about long-term keys asymmetric cryptography. In an article published in their blog [22] shortly after their technical dive they explained that they were going to make the transition to the new Session Protocol in order to simplify the experience for the end user, at the cost of loosing perfect forward secrecy and deniable encryption, previously provided by Signal's protocol (as demonstrated by N. Kobeissi [62]):

> ⟨⟨We believe the new possibilities and significantly streamlined encryption pipeline introduced by the Session Protocol make these tradeoffs well worth it for our users.⟩⟩

Since it was not clear at which point the development of the new protocol implementation was, along with the fact that there were not any clear explanation about how the clients are able to manage network discovery in a decentralized context (for example to find a recipients's Swarm), we decided to rebuild the protocol specification from the app's desktop source code, available at their public repository [12]. The following chapters will document this process and the subsequent symbolic verification in Tamarin of the protocol itself.

4

E2E Protocol

4.1 Reverse-engineering the protocol

Due to the issues introduced in section 3.4 , the first step necessary towards the verification of the security properties provided by Session is to unambiguously define the actual E2E protocol itself. To do so, we reverse engineered the code available at the desktop's app public repository [20]. The client was mainly programmed in JavaScript, with the front-end managed through React and the back-end powered by TypeScript (therefore having the additional safety checks provided by static type-checking [47]).

4.1.1 Sending procedure

Finding the correct entry point

Due to the sheer amount of files and folders present in the repository, finding the main function invoked in the sending procedure of a message could not be done through manual searching. Instead, since Session is a worldwide available app, it is possible to open one of the subfolders available in the `/_locales/` directory and search for the relevant translation of each button in a language of choice. Between the items listed, there is a line containing `sendMessage`, which defines the translation for the placeholder message initialised within the sending textbox. Using this keyword for a global search in the repository, we found the file `/ts/components/conversation/composition/CompositionBox.tsx`, which manages the UI for message typing. Within this file the next line is coded:

```
<SendMessageButton onClick={this.onSendMessage} />
```

Following this hint, we found the async function `onSendMessage()`, which gathers the plaintext and sending options from the UI and calls the actual sending procedure:

```
const extractedQuotedMessageProps = _.pick(
    'id',
    quotedMessageProps,
    'author',
    'text',
    'attachments');
```

```
const { attachments, previews } = await this.getFiles(linkPreview);
```

```
this.props.sendMessage({
    body: messagePlaintext,
    attachments: attachments || [],
    quote: extractedQuotedMessageProps,
    preview: previews,
    groupInvitation: undefined,
});
```

`sendMessage(msg)` then proceeds by adding the outgoing timestamp and the timer used for ephemeral messages:

```
this.clearTypingTimers();
const expireTimer = this.get('expireTimer');
const networkTimestamp = getNowWithNetworkOffset();
```

and calls another procedure that then manages the outgoing message queue. What follows next is a long waterfall of procedures invoked to manage different aspects of the dispatching of the message, which are out of the scope of this thesis.

Encryption procedure

Encryption is introduced later on during the long chain of functions by the `send()` procedure defined in `/ts/session/sending`. In the body of this function, an object of type `MessageEncrypter` is called for its method `encrypt()`, which manages padding and calls in turn `encryptUsingSessionProtocol()`. The following procedure is the one actually responsible for the encryption of the data:

```
export async function encryptUsingSessionProtocol(
    recipientHexEncodedX25519PublicKey: PubKey,
    plaintext: Uint8Array
): Promise<Uint8Array> {

    // 1. ) Retrieve Alice's own signing keypair
    const userED25519KeyPairHex = await UserUtils.getUserED25519KeyPair();

    // 2. ) Load the encryption library
    const sodium = await getSodium();

    // 3. ) Prepare the keys for encryption
    const userED25519PubKeyBytes = fromHexToArray(userED25519KeyPairHex.pubKey);
    const recipientX25519PublicKey = recipientHexEncodedX25519PublicKey.withoutPrefixToArray();
    const userED25519SecretKeyBytes = fromHexToArray(userED25519KeyPairHex.privKey);

    // 4. ) Create the content for the signature
    const verificationData = concatUInt8Array(
      plaintext,
      userED25519PubKeyBytes,
      recipientX25519PublicKey
    );

    // 5. ) Sign the verification data
    const signature = sodium.crypto_sign_detached(verificationData, userED25519SecretKeyBytes);

    // 6. ) Unite the payload for the message
    const plaintextWithMetadata = concatUInt8Array(plaintext, userED25519PubKeyBytes, signature);

```

```
30      // 7. ) Encrypt the payload
31      const ciphertext = sodium.crypto_box_seal(plaintextWithMetadata, recipientX25519PublicKey);
32
33      // 8. ) Return the encrypted payload to the sending procedure
34      return ciphertext;
35  }
```

Note that:

- at this point, `plaintext` also contains the sending options and the sending timestamp;
- further analysis of `crypto_box_seal()` is available in appendix A;
- the previous function definition is not complete: all parts related to exception handling have been removed for the sake of brevity.

Another aspect to address in this phase is the fact that the sender is able to use the receiver's public $X25519$ key, although apparently they only exchanged $ED25519$ keys. As F. Valsorda explained in his blog post [74], it is easy to map a public $X25519$ key to two $ED25519$ counterparts (with sign bit either 0 or 1): on the opposite, to each $pub\_ED25519$ key corresponds only a pub_X25519 equivalent. This translation is provided by *libsodium*'s API `crypto_sign_ed25519_pk_to_curve25519()` (documentation available at [5]).

Route discovery

After the completion of the encryption procedure, the previously introduced function `send()` calls `TEST_sendMessageToSnode()`, which, in turn, discovers the nodes that will store the message and forwards the `ciphertext`. Swarm retrieval is done through `getSwarmFor()`, defined in `/ts/session/apis/snode_api/snodePool.ts`:

```
1   export async function getSwarmFor(pubkey: string): Promise<Array<Data.Snode>> {
2       const nodes = await getSwarmFromCacheOrDb(pubkey);
3
4       // 1. ) See how many nodes belonging to the snode pool are still reachable
5       const goodNodes = randomSnodePool.filter(
6         (n: Data.Snode) => nodes.indexOf(n.pubkey_ed25519) !== -1
7       );
8
9       // 2. ) If enough nodes are present, there is no need to execute Network Discovery ...
10      if (goodNodes.length >= minSwarmSnodeCount) {
11        return goodNodes;
12      }
13
14      // 3. ) ... else, request a new node list from the network
15      const freshNodes = _.shuffle(await requestSnodesForPubkey(pubkey));
16      const edkeys = freshNodes.map((n: Data.Snode) => n.pubkey_ed25519);
17      await internalUpdateSwarmFor(pubkey, edkeys);
18      return freshNodes;
19  }
```

In particular, `requestSnodesForPubKey()` begins a chain of functions that invokes a `snodeRpc()` to a random node with the `'get_snodes_for_pubkey'` parameter [1]. To retrieve the random node, the auxiliary `getRandomSnode()`

[1] Note that `snodeRpc()` is treated here as an abstraction and will be further discussed in chapter 5.1.1.

function is called. This procedure returns a Service Node either from the local database or from one of the official *Seed Nodes*. In the latter case, the actual query is done through a function defined in `/ts/session/apis/seed_node_api/SeedNode.ts`:

```
async function getSnodesFromSeedUrl(urlObj: URL): Promise<Array<any>> {

    // 1. ) Define the query parameters
    const params = {
        active_only: true,
        fields: {
            public_ip: true,
            storage_port: true,
            pubkey_x25519: true,
            pubkey_ed25519: true,
        },
    };

    // 2. ) Define the endpoint of the fetch() request
    const endpoint = 'json_rpc';
    const url = `${urlObj.href}${endpoint}`;

    // 3. ) Define the body of the request
    const body = {
        jsonrpc: '2.0',
        id: '0',
        method: 'get_n_service_nodes',
        params,
    };

    // 4. ) Return a custom sslAgent to verify the self-signed certificate of the Seed Node
    const sslAgent = getSslAgentForSeedNode(
        urlObj.hostname,
        urlObj.protocol !== Constants.PROTOCOLS.HTTP
    );

    // 5. ) Complete the definition of the HTTP headers
    const fetchOptions = {
        method: 'POST',
        timeout: 5000,
        body: JSON.stringify(body),
        headers: {
            'User-Agent': 'WhatsApp',
            'Accept-Language': 'en-us',
        },
        agent: sslAgent,
    };

    // 6. ) Execute the fetch request
    const response = await insecureNodeFetch(url, fetchOptions);

    // 7. ) Return the parsed results to the calling procedure
    const json = await response.json();
    const result = json.result;
    const validNodes = result.service_node_states.filter(
        (snode: any) => snode.public_ip !== '0.0.0.0'
        );
```

```
53      return validNodes;
54  }
```

Note that the code listed above is not the complete definition of the function: all parts related to exception handling have been removed for the sake of brevity.

Actual sending

Having all the relevant information about a Swarm at our disposal, the previosly introduced procedure `TEST_sendMessageToSnode()` can now manage the actual dispatching of the message. For each node belonging to the intended Swarm, through the auxiliary function `storeOnNode()`, it invokes a `snodeRpc()` procedure as follows :

```
1  const result = await snodeRpc({
2      method: 'store',                    // RPC method
3      params,                             // Sending parameters
4      targetNode,                         // Bob Swarm Node's public ED25519 key
5      associatedWith: params.pubKey,      // Bob's public ED25519 key
6  });
```

Note that, at this point, the `ciphertext` is an attribute of the variable `params`, along with the sending timestamp and a ttl counter.

4.1.2 Receiving messages

Entry point

Since there is no glaring UI element to look at for receiving messages (which is a 'passive' activity for the user), we started searching through the `/ts/session/apis/snode_apis/swarmPolling.ts` file. There the `start()` procedure handles the background polls for messages. After a waterfall of functions, `TEST_pollOnceForKey()` is called, which uses the previously introduced `getSwarmFor()` to find the client's own Swarm information.

Message retrieval

For each node belonging to the Swarm, the client fetches from the local database the hash of the last message retrieved, then invokes `retrieveNextMessage()` to send a `snodeRpc()` call with the node's public $ED25519$ key, his public $ED25519$ key, a signed nonce and `'retrieve'` as parameters:

```
1  export async function retrieveNextMessages(
2      targetNode: Snode,
3      lastHash: string,
4      associatedWith: string
5  ): Promise<Array<any>> {
6
7      // 1. ) Define the options of the query
8      const params = {
9          pubKey: associatedWith,         // Bob's public ED25519 key
10         lastHash: lastHash || '',       // Bob's last hash associated with this Swarm Node
11     };
12
13     // 2. ) Create a signed timestamp for authentication
14     const signatureParams = (await getRetrieveSignatureParams(params)) || {};
```

```
15
16      // 3. ) Send the query to the right Service Node
17      const result = await snodeRpc({
18          method: 'retrieve',
19          params: { ...signatureParams, ...params },
20          targetNode,                    // Bob Swarm Node's public ED25519 key
21          associatedWith,                // Bob's public ED25519 key
22          timeout: 4000,
23      });
24
25      // 4. ) Return the message retrieved, if any
26      const json = JSON.parse(result.body);
27      return json.messages || [];
28  }
```

Note that:

- `targetNode` refers to the Swarm Node's public $ED25519$ key, `lastHash` the last hash relevant to that node and `associatedWith` is the public $ED25519$ key of the receiver;
- the code listed is only a fraction of the actual function implemented: all parts related to exception handling have been removed for the sake of brevity.

The nonce sent to the Service Node is a simple signed Unix timestamp as we can see from the following function (from which we removed all error handling code):

```
1   async function getRetrieveSignatureParams(
2       params: RetrieveRequestParams
3   ): Promise<{ timestamp: number; signature: string; pubkey_ed25519: string } | null> {
4
5       // 1. ) Retrieve our signing keypair
6       const ourPubkey = UserUtils.getOurPubKeyFromCache();
7       const ourEd25519Key = await UserUtils.getUserED25519KeyPair();
8       const edKeyPrivBytes = fromHexToArray(ourEd25519Key?.privKey);
9
10      // 2. ) Retrieve the an update timestamp
11      const signatureTimestamp = getNowWithNetworkOffset();
12
13      // 3. ) Concatenate the request with the timestamp and encode it for signing
14      const verificationData = StringUtils.encode(`retrieve${signatureTimestamp}`, 'utf8');
15      const message = new Uint8Array(verificationData);
16
17      // 4. ) Sign the verification data and encode it for transmission
18      const sodium = await getSodiumRenderer();
19      const signature = sodium.crypto_sign_detached(message, edKeyPrivBytes);
20      const signatureBase64 = fromUInt8ArrayToBase64(signature);
21
22      // 5. ) Return the signed data
23      return {
24          timestamp: signatureTimestamp,
25          signature: signatureBase64,
26          pubkey_ed25519: ourEd25519Key.pubKey
27      };
28    }
```

Message Handling

After retrieving a new (encrypted) message, the function `decryptWithSessionProtocol()` available at `/ts/receiver/contentMessage.ts` handles the decryption and verification of authenticity of the `ciphertext` received:

```
export async function decryptWithSessionProtocol(
    envelope: EnvelopePlus,
    ciphertextObj: ArrayBuffer,
    x25519KeyPair: ECKeyPair,
    isClosedGroup?: boolean
): Promise<ArrayBuffer> {

    // 1. ) Retrieve Bob's encryption keypair
    const recipientX25519PrivateKey = x25519KeyPair.privateKeyData;
    const hex = toHex(new Uint8Array(x25519KeyPair.publicKeyData));
    const recipientX25519PublicKey = PubKey.remove05PrefixIfNeeded(hex);

    // 2. ) Initialize objects and variables
    const sodium = await getSodium();
    const signatureSize = sodium.crypto_sign_BYTES;
    const ed25519PublicKeySize = sodium.crypto_sign_PUBLICKEYBYTES;

    // 3. ) Decrypt the message
    const plaintextWithMetadata = sodium.crypto_box_seal_open(
        new Uint8Array(ciphertextObj),
        fromHexToArray(recipientX25519PublicKey),
        new Uint8Array(recipientX25519PrivateKey)
    );
    if (plaintextWithMetadata.byteLength <= signatureSize + ed25519PublicKeySize) {
        throw new Error('Decryption failed.');
    }

    // 4. ) Get the message parts
    const signatureStart = plaintextWithMetadata.byteLength - signatureSize;
    const signature = plaintextWithMetadata.subarray(signatureStart);
    const pubkeyStart = plaintextWithMetadata.byteLength - (signatureSize + ed25519PublicKeySize);
    const pubkeyEnd = plaintextWithMetadata.byteLength - signatureSize;
    const senderED25519PublicKey = plaintextWithMetadata.subarray(pubkeyStart, pubkeyEnd);
    const plainTextEnd = plaintextWithMetadata.byteLength - (signatureSize + ed25519PublicKeySize);
    const plaintext = plaintextWithMetadata.subarray(0, plainTextEnd);

    // 4. ) Verify the signature
    const isValid = sodium.crypto_sign_verify_detached(
        signature,
        concatUInt8Array(plaintext, senderED25519PublicKey, fromHexToArray(recipientX25519PublicKey)),
        senderED25519PublicKey
    );
    if (!isValid) {
      throw new Error('Invalid message signature.');
    }

    // 5. ) Get the sender's X25519 public key
    const senderX25519PublicKey = sodium.crypto_sign_ed25519_pk_to_curve25519(senderED25519PublicKey);
    if (!senderX25519PublicKey) {
      throw new Error('Decryption failed.');
    }
```

```
52
53      // 6. ) set the sender identity on the envelope itself.
54      if (isClosedGroup) {
55        envelope.senderIdentity = `05${toHex(senderX25519PublicKey)}`;
56      } else {
57        envelope.source = `05${toHex(senderX25519PublicKey)}`;
58      }
59
60      // 7. ) Return the message
61      return plaintext;
62    }
```

Note that the previous code listing is not complete: logging commands were left out for the sake of brevity.

Finally, once successfully received, decrypted and veryfied a message, the client updates the internal database for the last hash received from that particular Swarm Node.

4.2 Protocol summary

As introduced in the relevant section 3.1, Session's End-2-End protocol aims to guarantee all of its privacy features to its users without the need of out-of-band checks (such as the QR comparison provided by apps like WhatsApp, or Telegram). In order to achieve this, they have decided to associate to each user a quadruple of keys: a private $X25519$ key and its public counterpart, as well as a private $ED25519$ key and its corrisponding public key. Each node (both users and Service Nodes) is identified and can be reached in the network only by its public $ED25519$ key: this forces users to exchange public keys instead of identifiers (such as phone numbers or email addresses) that have to be subsequently associated by a potentially malicious server. As a consequence, when a user wants to send a message to another account, it already owns all the necessary information required to encrypt the communication and only needs to discover the path to deliver the message through the decentralized network. To do so, the sender initiates the following 3-phase protocol.

Note that from here on, we will use the standard Alice-Bob notation [63] to refer to the sender and the receiver users.

4.2.1 Network Discovery

Alice retrieves a list of active nodes in the network by querying a Seed Node. Each client owns a list of three Seed Nodes' URLs and self-signed TLS certificates hardcoded in memory: this allows Alice to establish a secure connection [37] with a trusted server to retrieve a list of (at least) 12 active Service Nodes in the network. Each server is represented by the following quadruple

$$\text{ServiceNode}_i = \langle \text{IP}_i, \text{port}_i, pub\_X25519_i, pub\_ED25519_i \rangle \tag{4.1}$$

IP_i and port_i are used for the actual http connection to the i^{th} server, while $X25519_i$ allows for symmetrical cryptography powered by an *Elliptic Curve Diffie Hellman* exchanges. Finally, as previously introduced, the $pub\_ED25519_i$ key allows the nodes to identify the i^{th} node in the network.

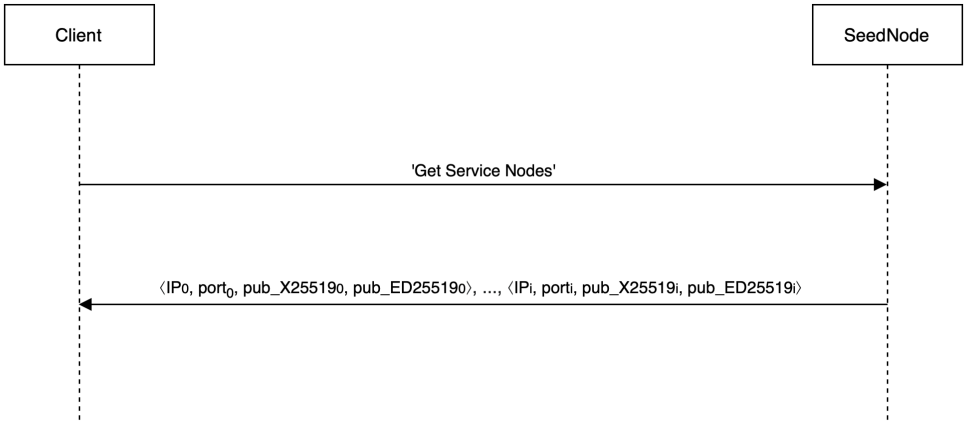

Figure 4.1: Summary of the Network Discovery sub-protocol

Note that this phase of the protocol is actually seldomly executed: each client has a local database of active Service Nodes that uses for almost each run of the E2E protocol. This step is needed during the first initialization of the client or if the numerosity of the local pool of nodes falls under a certain amount, since Service Nodes can be excluded by local databases if they fail to respond or deliver messages (in the desktop version the current minimum is set to 12).

4.2.2 Swarm Discovery

Since Session features a decentralized architecture, each client is associated to a group of servers called Swarm. Swarms vary dinamically depending on the status of the network and are completely managed by the network itself: from the client's perspective all we need to know is that for each E2E communication we need to determine the correct collection of nodes associated to our recipient.

When Alice has enough Service Nodes available, she queries a random node from her pool for Bob's Swarm. The node then responds with a list of nodes in the same form of 4.1. Considering that metadata anonymity is one of Session's major characteristics, further measures have to be provided during the execution of this phase to assure that Alice's identity ca not be traced back from her query for Bob's Swarm, since this would defy the entire purpose of the application. To guarantee the sender's anonymity, the same onion routing protocol 5.2 adopted later on to send the actual message is used by Alice during this exchange. In this chapter, we'll abstract from the underlying routing infrastructure by only considering the outermost layer of encryption: when a client is encrypting a piece of data d for a receiver Service Node$_\delta$, the message is sent as

$$\text{message\_sent} = \text{ECDH\_pubKey}_\delta \;||\; E_{K_\delta}(d) \;||\; pub\_ED25519_\delta \tag{4.2}$$

in which

- ECDH_pubKey$_\delta$ is the ephemeral public key that the receiving Service Node will have to use with its own private $X25519$ key to obtain K_δ through an Elliptic Curve Diffie Hellman exchange;
- K_δ is the symmetric E2E encryption key;

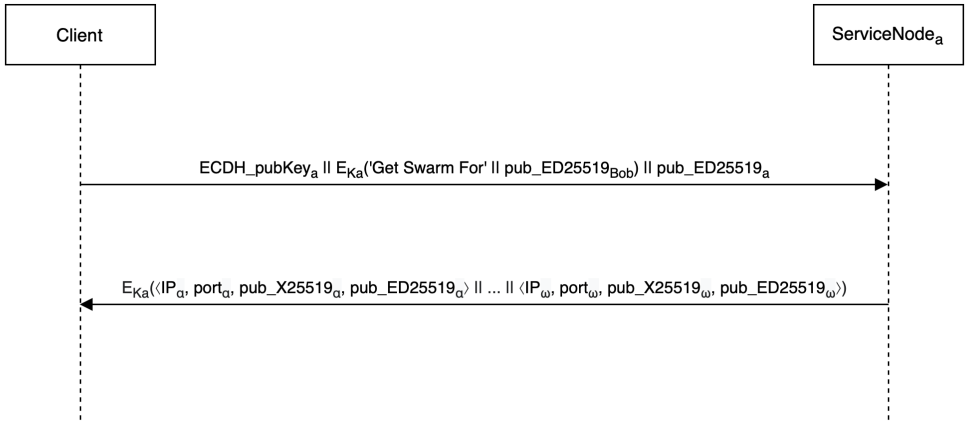

Figure 4.2: Summary of the Swarm Discovery sub-protocol

- $pub\_ED25519_\delta$ is included for routing purposes.

Note that also this phase is not always executed: Alice has a local database of Swarm nodes for each of its contacts, that is updated during each iteration of the protocol. This step is executed only if the number of Service Nodes belonging to a Swarm falls below a predetermined amount (in the desktop version the current minimum is set to 3).

4.2.3 Message Sending

Arrived at this point, Alice knows the public $ED25519$ and $X25519$ keys of some Service Nodes belonging to Bob's Swarm. To each one of them sends a sealed box, built as following.

The *plaintext* is concatenated with its sending *timestamp* (necessary for anti-replay protection, as it will be shown in section 4.4.3), along with other non-mandatory options (such as a timer to implement ephemeral messages). To provide authenticity (as will be demonstrated in section 4.4.2), the data is then signed along with the sender and the recipient's public information using Alice's private $ED25519$ key. Finally, everything is put together in a box, along with Alice's public $ED25519$ key. This allows Bob to determine the identity of the sender.

$$\begin{aligned} \text{message} &= \text{plaintext} \,||\, \text{timestamp} \,||\, \text{options} \\ \text{signature} &= Sig_{priv\_ED25519_{Alice}}(\text{message} \,||\, pub\_ED25519_{Alice} \,||\, pub\_X25519_{Bob}) \\ \text{box} &= \text{message} \,||\, pub\_ED25519_{Alice} \,||\, \text{signature} \end{aligned}$$

To achieve E2E cryptography the box is encrypted through libsodium's `crypto_box_seal()` API [17] invoked with Bob's public $X25519$ key. The encryption scheme adopted by this function is explained in further detail in appendix A. Meanwhile, the result of the encryption can be summarized as:

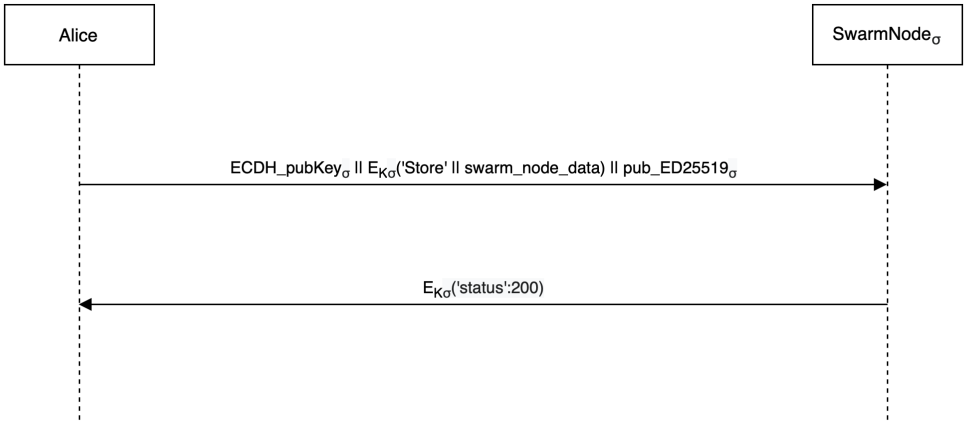

Figure 4.3: Summary of the Message Sending sub-protocol

$$\begin{aligned} \text{nonce} &= hash(\text{ephemeral\_key} \mid\mid pub\_X25519_{Bob}) \\ \text{encKey} &= \text{symKey} \mid\mid \text{nonce} \\ \text{sealed\_box} &= \text{ephemeral\_key} \mid\mid E_{\text{encKey}}(\text{box}) \end{aligned}$$

in which

- ephemeral_key is a public ephemeral $X25519$ key generated on the spot;
- symKey is a symmetric encryption key obtained through an $ECDH$ exchange between ephemeral_key's private counterpart and Bob's public $X25519$ key.

For dispatching purposes, the sealed_box is then concatenated with Bob's public $ED25519$ key, the same timestamp seen earlier and a ttl counter.

$$\text{swarm\_node\_data} = \text{sealed\_box} \mid\mid pub\_ED25519_{Bob} \mid\mid \text{timestamp} \mid\mid \text{ttl}$$

Finally, swarm_node_data is sent using Session's onion routing protocol (see the encryption scheme explained at 4.2) to each Service Node$_\sigma$ belonging to Bob's Swarm according to Alice's local database. If the exchange is executed correctly with Service Node$_\sigma$, Alice will receive a confirmation message E_{K_σ}('status' : 200). If not enough confirmations are received, Alice will execute another run of the protocol, excluding currently failing nodes.

4.2.4 Message Retrieval

Until now, we have seen the protocol's specification for sending a message to Bob's Swarm. On the other hand, Bob has to do a very similar run of this 3-phase protocol to retrieve his messages: firstly he may have to repeat steps 4.2.1 and 4.2.2 in order to recover its own Swarm. Then, he has to send to each Service Node$_\tau$ of the Swarm available in his local database the following message:

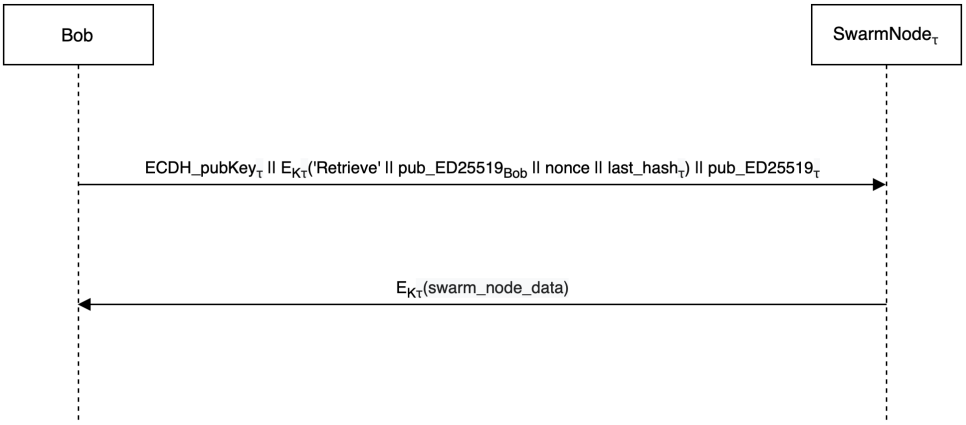

Figure 4.4: Summary of the Message Retrieval sub-protocol

$$\begin{aligned} \text{nonce} &= Sig_{priv\_ED25519_{Bob}}(\text{timestamp}) \\ \text{request} &= E_{K_\tau}(\text{'retrieve'} \,||\, pub\_ED25519_{Bob} \,||\, \text{nonce} \,||\, \text{last\_hash}_\tau) \\ \text{data\_sent\_to\_}\tau &= \text{ECDH\_pubKey}_\tau \,||\, \text{request} \,||\, pub\_ED25519_\tau \end{aligned}$$

in which last_hash$_\tau$ is the hash of the last message Bob has correctly retrieved from Service Node$_\tau$ and $Sig_{priv\_ED25519_{Bob}}$(timestamp) is a nonce for the server to check the authenticity of the request. In particular, this latter information has been recently included in the May 18th update, as introduced in a blog post [4] published on their website. Before this change, any user (or even external agent) could possibly download all stored data for a given $ED25519$ key by just sending a valid request to the right nodes. Furthermore, since Swarm Discovery does not require the identity of the querying client (for anonymity purposes), it would have been trivial for an attacker to obtain all the incoming inbox of a user without anybody noticing. The messages retrieved would have been useless of course since all Peer-to-Peer chats are E2E encrypted, but Session's team decided to mitigate any possible risk by adding this further precaution.

Given that the message sent by Alice at 4.2.3 is currently stored on Service Node$_\tau$, Bob will receive it as E_{K_τ}(sealed_box), completing a correct run of the protocol.

4.3 Protocol formalization

As previously introduced in 3.4, we have undertaken the formalization of both the E2E and onion routing protocols in Tamarin. We'll now show the corresponding rewriting rules needed for modeling each phase of the E2E protocol.

4.3.1 Formalizing Network Discovery

Firstly, we have to initialize the Seed Nodes and clients.

```
1 rule InitialiseSeedNode :
```

```
    let
        seedNode_pubKey = pk(~seedNode_prKey)
        seedNode_cert = aenc(<~seedNode_identity, seedNode_pubKey>, ~seedNode_prKey)
    in
    [
        Fr(~seedNode_prKey),
        Fr(~seedNode_identity)
    ]
    -->
    [
        !SeedNode(~seedNode_prKey, seedNode_pubKey, ~seedNode_identity, seedNode_cert),
        Out(<~seedNode_identity, seedNode_cert, seedNode_pubKey>)
    ]
```

```
rule InitialiseClient :
    let
        client_X25519_pubKey = 'g'^~client_X25519_prKey
        client_ED25519_pubKey = pk(~client_ED25519_prKey)
        swarmNode_X25519_pubKey = 'g'^~swarmNode_X25519_prKey
        swarmNode_ED25519_pubKey = pk(~swarmNode_ED25519_prKey)
    in
    [
        Fr(~client_X25519_prKey),
        Fr(~client_ED25519_prKey),
        Fr(~swarmNode_X25519_prKey),
        Fr(~swarmNode_ED25519_prKey),
        !SeedNode(seedNode_prKey, seedNode_pubKey, seedNode_identity, seedNode_cert)
    ]
    --[ IsAClient(client_ED25519_pubKey) ]->
    [
        !Client(~client_X25519_prKey, client_X25519_pubKey,
            ~client_ED25519_prKey, client_ED25519_pubKey),
        !ClientSeedNode(client_ED25519_pubKey, seedNode_pubKey),
        !SwarmNode(client_ED25519_pubKey, ~swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
            ~swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey)
    ]
```

As will be soon explained in section 4.4, `IsAClient(client_ED_key)` action fact will be necessary to discriminate between users and nodes in the network (since both are identified by $pub\_ED25519$ keys) within the formalization of security properties.

Note that the certificate and public key of the Seed Node are never actually sent on the network, but this modeling strategy was necessary to allow the public key to be known to everyone (including the attacker) without publishing the corresponding private key. Furthermore, it is possibile to notice that the Swarm Node is initialized along with the client. While in the real application the Service Nodes are distributed between Swarms dinamically, this semplification is needed to ensure that the node provided for Bob's Swarm remains the same during both the executions of the protocol.

Even though Tamarin provides primitives for many recurrent situations in cryptography, it lacks a built-in equational theory for modeling secure channels. Luckily, a quick workaround to this problem is listed within the official manual [71]:

```
rule ChannelOut_S :
    [ SecureOut(sender, receiver, message) ]
    -->
```

```
    [ Sec(sender, receiver, message) ]

rule ChannelIn_S :
    [ Sec(sender, receiver, message) ]
    -->
    [ SecureIn(sender, receiver, message) ]
```

The actual exchange that makes up the first phase of the protocol can be modeled through three rules:

```
rule SendServiceNodeRequest :
    [
        !Client(client_X25519_prKey, client_X25519_pubKey, client_ED25519_prKey, client_ED25519_pubKey),
        !ClientSeedNode(client_ED25519_pubKey, seedNode_pubKey)
    ]
    -->
    [
        SentServiceNodeRequest(client_ED25519_pubKey, seedNode_pubKey),
        SecureOut(client_ED25519_pubKey, seedNode_pubKey, 'get_serviceNodes')
    ]
```

```
rule AnswerServiceNodeRequest :
    let
        serviceNode_X25519_pubKey = 'g'^~serviceNode_X25519_prKey
        serviceNode_ED25519_pubKey = pk(~serviceNode_ED25519_prKey)
        serviceNode_information = <serviceNode_ED25519_pubKey,
            serviceNode_X25519_pubKey, seedNode_cert>
    in
    [
        !SeedNode(seedNode_prKey, seedNode_pubKey, seedNode_identity, seedNode_cert),
        SecureIn(client_ED25519_pubKey, seedNode_pubKey, 'get_serviceNodes'),
        Fr(~serviceNode_X25519_prKey),
        Fr(~serviceNode_ED25519_prKey)

    ]
    -->
    [
        !ServiceNode(~serviceNode_X25519_prKey, serviceNode_X25519_pubKey,
            ~serviceNode_ED25519_prKey, serviceNode_ED25519_pubKey),
        SecureOut(seedNode_pubKey, client_ED25519_pubKey, serviceNode_information)
    ]
```

```
rule ReceiveServiceNodeResponse :
    [
        !Client(client_X25519_prKey, client_X25519_pubKey, client_ED25519_prKey, client_ED25519_pubKey),
        !ClientSeedNode(client_ED25519_pubKey, seedNode_pubKey),
        !SeedNode(seedNode_prKey, seedNode_pubKey, seedNode_identity, seedNode_cert),
        SentServiceNodeRequest(client_ED25519_pubKey, seedNode_pubKey),
        SecureIn(seedNode_pubKey, client_ED25519_pubKey, <serviceNode_ED25519_pubKey,
            serviceNode_X25519_pubKey, seedNode_cert>)
    ]
    -->
    [
        ClientReadyToQuery(client_ED25519_pubKey, serviceNode_ED25519_pubKey, serviceNode_X25519_pubKey)
    ]
```

Some implementation notes about this phase:

- The fact that the Seed Node's TLS certificate corresponds with the one hardcoded in the client's memory is guaranteed by pattern-matching;
- In the real world application, a list of multiple nodes would be sent by the Seed Node to the client. Since Tamarin implements only tuples of pre-determined length, we simplified this situation, while allowing the network topology to stay dynamic by generating a new, fresh node at each iteration of this phase.
- From this moment on, we can use the fact `ClientReadyToQuery(client_ED_key, node_ED_key, node_X_key)` to require that `client_ED_key` has already concluded the Network Discovery 4.2.1 phase at least once.

Finally, notice that we modeled the $X25119$ exchange as a standard Diffie Hellman, although in practice its elliptic curve variant is used. ECDH implementation for Tamarin is currently underway [40], but in the meantime, as D. Jackson explained in [52], it is reasonable to use the built-in DH theory.

4.3.2 Formalizing Swarm Discovery

As anticipated in section 4.2.2 4.2.3 and 4.2.4, both the Swarm Discovery, Message sending and Message Retrieval phases are executed adopting the *Onion Routing* protocol 5.2. Since in this chapter we do not investigate anonymity properties, we'll treat the underlying routing protocol as an abstraction and we'll only show the innermost layer of encryption (the one shared between the client and the designated Service Node).

The actual Swarm request is modeled as following:

```
rule SendSwarmRequest :
    let
        k = serviceNode_X25519_pubKey^~ECDH_prKey
        ECDH_pubKey = 'g'^~ECDH_prKey
    in
    [
        ClientReadyToQuery(client_ED25519_pubKey, serviceNode_ED25519_pubKey, serviceNode_X25519_pubKey),
        !Client(bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey, bob_ED25519_pubKey),
        Fr(~ECDH_prKey)
    ]
    --[ QueryForSwarm(bob_ED25519_pubKey) ]->
    [
        SentSwarmRequest(client_ED25519_pubKey, bob_ED25519_pubKey, serviceNode_ED25519_pubKey, k),
        Out(<senc(<'get_swarm', bob_ED25519_pubKey>, k), ECDH_pubKey, serviceNode_ED25519_pubKey>)
    ]
```

```
rule AnswerSwarmRequest :
    let
        k = ECDH_pubKey^serviceNode_X25519_prKey
    in
    [
        !ServiceNode(serviceNode_X25519_prKey, serviceNode_X25519_pubKey,
            serviceNode_ED25519_prKey, serviceNode_ED25519_pubKey),
        !SwarmNode(bob_ED25519_pubKey, swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
            swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey),
        In(<senc(<'get_swarm', bob_ED25519_pubKey>, k), ECDH_pubKey, serviceNode_ED25519_pubKey>)
    ]
```

```
    -->
    [
        Out(senc(<'requested_swarm', swarmNode_ED25519_pubKey, swarmNode_X25519_pubKey>, k))
    ]
```

```
rule ReceiveSwarmResponse :
    [
        SentSwarmRequest(client_ED25519_pubKey, bob_ED25519_pubKey, serviceNode_X25519_pubKey, k),
        In(senc(<'requested_swarm', swarmNode_ED25519_pubKey, swarmNode_X25519_pubKey>, k))
    ]
    -->
    [
        ClientReadyToSend(client_ED25519_pubKey, bob_ED25519_pubKey,
            swarmNode_ED25519_pubKey, swarmNode_X25519_pubKey)
    ]
```

The previous choice of assigning statically a Swarm Node to each client at initialization was crucial to guarantee that each run of this phase will return the same node. Otherwise, it would not be trivial to assure that Alice will send the message to the same node that Bob will query to retrieve it.

Notice also that:

- the fact `ClientReadyToSend(sender_ED_key, receiver_ED_key, node_ED_key, node_X_key)` can be used from now on as a premise to require that a determined client has already executed phases 1 and 2 of the protocol.
- the `QueryForSwarm(client_ED_key)` action fact means that a client has sent a Swarm request for `client_ED_key`. This fact will be useful while demonstrating the sender anonymity property 4.4.4.

4.3.3 Formalizing Message Sending

In this subsection we'll formalize the actual sending of a message. To convey the idea that Alice wants to send a message containing `plaintext` to Bob, we define the following rule:

```
rule CreateMessage :
    [
        !Client(alice_X25519_prKey, alice_X25519_pubKey, alice_ED25519_prKey, alice_ED25519_pubKey),
        !Client(bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey, bob_ED25519_pubKey),
        Fr(~plaintext)
    ]
    -->
    [
        MessageToSend(alice_ED25519_pubKey, bob_ED25519_pubKey, ~plaintext)
    ]
```

The actual encryption and sending steps of the protocol are summarized in one cumulative rule:

```
rule SendMessage :
    let
        message = <plaintext, ~timestamp, ~options>
        signature = sign(<message, alice_ED25519_pubKey, bob_X25519_pubKey>, alice_ED25519_prKey)
        box = <message, alice_ED25519_pubKey, signature>
        ephemeral_key = 'g'^~ephemeral_prKey
        symKey = bob_X25519_pubKey^~ephemeral_prKey
        nonce = h(<ephemeral_key, bob_X25519_pubKey>)
        encKey = <symKey, nonce>
```

```
        sealed_box = <ephemeral_key, senc(box, encKey)>
        swarm_node_data = <sealed_box, bob_ED25519_pubKey, ~timestamp, ~ttl>
        k = swarmNode_X25519_pubKey^~ECDH_prKey
        ECDH_pubKey = 'g'^~ECDH_prKey
        data_sent = <ECDH_pubKey, senc(<'store', swarm_node_data>, k), swarmNode_ED25519_pubKey>
    in
    [
        MessageToSend(alice_ED25519_pubKey, bob_ED25519_pubKey, plaintext),
        ClientReadyToSend(alice_ED25519_pubKey, bob_ED25519_pubKey,
            swarmNode_ED25519_pubKey, swarmNode_X25519_pubKey),
        !SwarmNode(bob_ED25519_pubKey, swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
            swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey),
        !Client(alice_X25519_prKey, alice_X25519_pubKey, alice_ED25519_prKey, alice_ED25519_pubKey),
        !Client(bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey, bob_ED25519_pubKey),
        Fr(~timestamp), Fr(~ttl), Fr(~options),
        Fr(~ECDH_prKey), Fr(~ephemeral_prKey)
    ]
    --[
        StoreSent(data_sent, swarmNode_ED25519_pubKey),
        SealedBoxSent(alice_ED25519_pubKey, bob_ED25519_pubKey, sealed_box),
        BoxSent(alice_ED25519_pubKey, bob_ED25519_pubKey, box),
        MessageSentWithTimestamp(alice_ED25519_pubKey, bob_ED25519_pubKey, plaintext, ~timestamp),
        SentBoxToSwarm(bob_ED25519_pubKey, swarmNode_ED25519_pubKey)
    ]->
    [
        MessageWaitingForConfirm(alice_ED25519_pubKey, bob_ED25519_pubKey, plaintext, k),
        Out(data_sent)
    ]
```

Note that the naming convention follows the equations defined in subsection 4.2.3. There are multiple action facts declared within this rule:

- `StoreSent(data, node_ED_key)` is needed later on to write a source lemma required to resolve partial deconstructions 4.3.6;
- `SealedBoxSent(sender_ED_key, receiver_ED_key)` is used to write a sanity check lemma that assures that the protocol can be executed;
- `BoxSent(sender_ED_key, receiver_ED_key, box)`, `SentBoxToSwarm(receiver_ED_Key, node_ED_key)` and `MessageSentWithTimestamp(sender_ED_key, receiver_ED_key, plaintext, timestamp)` are then used to specify security properties.

To model the actual data retrieval and confirmation by the Swarm Nodes, we use two supplementary rules:

```
rule AnswerMessageSent :
    let
        k = ECDH_pubKey^swarmNode_X25519_prKey
        hash = h(sealed_box)
        data_arrived = <ECDH_pubKey, senc(<'store', <sealed_box, bob_ED25519_pubKey,
            ~timestamp, ttl>>, k), swarmNode_ED25519_pubKey>
    in
    [
        !SwarmNode(bob_ED25519_pubKey, swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
            swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey),
```

```
        In(data_arrived)
    ]
    --[ StoreArrived(data_arrived, swarmNode_ED25519_pubKey) ]->
    [
        !StoredMessage(swarmNode_ED25519_pubKey, sealed_box, hash, bob_ED25519_pubKey),
        Out(senc('received', k))
    ]
```

Note that, unlike all the other variables, `~timestamp` is still declared as a fresh term: this modeling decision enforces the idea that the server is able to check the value contained in the variable to ensure that it belongs to a recent timeframe.

```
rule ReceiveSentMessageResponse :
    [
        MessageWaitingForConfirm(alice_ED25519_pubKey, bob_ED25519_pubKey, plaintext, k),
        In(senc('received', k))
    ]
    -->
    []
```

The additional `StoreArried(data, node_ED_key)` fact is also required to specify the previously anticipated source lemma.

Notice also that the persistent fact `!StoredMessage(node_ED_key, sealed_box, hash, receiver_ED_key)` can be used from this moment on to require that a determined Swarm Node has already received a message intended for `receiver_ED_key`.

4.3.4 Formalizing Message Retrieval

Finally, we define three more rules to model Bob's retrieval of his own messages from the Swarm. To require that phases 1 and 2 have already been executed by Bob we re-use the fact `ClientReadyToSend(sender_ED_key, receiver_ED_key, node_ED_key, node_X_key)`, this time with Bob's key for both the sender and the receiver (since he queried the Service Node for his own Swarm).

```
rule RetrieveMessage :
    let
        k = swarmNode_X25519_pubKey^~ECDH_prKey
        ECDH_pubKey = 'g'^~ECDH_prKey
        nonce = sign(~timestamp, bob_ED25519_prKey)
    in
    [
        !Client(bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey, bob_ED25519_pubKey),
        ClientReadyToSend(bob_ED25519_pubKey, bob_ED25519_pubKey,
            swarmNode_ED25519_pubKey, swarmNode_X25519_pubKey),
        !SwarmNode(bob_ED25519_pubKey, swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
            swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey),
        Fr(~ECDH_prKey), Fr(~timestamp)
    ]
    -->
    [
        SentRetrievalRequest(bob_ED25519_pubKey, swarmNode_ED25519_pubKey, k),
        Out(<ECDH_pubKey, senc(<'retrieve', nonce, bob_ED25519_pubKey>, k), swarmNode_ED25519_pubKey>)
    ]
```

```
rule AnswerRetrieveRequest :
    let
        k = ECDH_pubKey^swarmNode_X25519_prKey
        nonce = sign(~timestamp, bob_ED25519_prKey)
        bob_ED25519_pubKey = pk(bob_ED25519_prKey)
    in
    [
        !SwarmNode(bob_ED25519_pubKey, swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
            swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey),
        !StoredMessage(swarmNode_ED25519_pubKey, sealed_box, hash, bob_ED25519_pubKey),
        In(<ECDH_pubKey, senc(<'retrieve', nonce, bob_ED25519_pubKey>, k), swarmNode_ED25519_pubKey>)
    ]
    -->
    [
        Out(senc(<'new_messages', sealed_box>, k))
    ]
```

Note that:

- the private-public key corrispondence and coherence within this rule is granted by pattern-matching;
- as seen in section 4.2.4, in the real protocol Bob also sends to each Swarm Node the hash of the last message retrieved from it. This value is then updated in Bob's local database as soon as a new message is retrieved from the node itself. To formalize this evolution, we would have to create a fact during client initialization to keep track of the last hash retrieved for each client [2]: upon successful retrieval, we would consume and substite it with another semantically equal fact with an updated hash. Having the same fact in both the premises and consequences of a rule brings inevitably to loops in the proof and often leads to non-termination. Thus, since we have no certainity about how the last hash is used by the server and it does not provide any information about the identity or credentials of a user anyway, we decided to leave out this information from our formalization;
- as explained in 4.3.3, `~timestamp` is declared as a fresh term to express that the server is able to check the age of the message.

```
rule ReceiveRetrieveResponse :
    let
        message = <plaintext, ~timestamp, options>
        alice_ED25519_pubKey = pk(alice_ED25519_prKey)
        signature = sign(<message, alice_ED25519_pubKey, bob_X25519_pubKey>, alice_ED25519_prKey)
        box = <message, alice_ED25519_pubKey, signature>
        ephemeral_key = 'g'^~ephemeral_prKey
        symKey = bob_X25519_pubKey^~ephemeral_prKey
        nonce = h(<ephemeral_key, bob_X25519_pubKey>)
        encKey = <symKey, nonce>
        sealed_box = <ephemeral_key, senc(box, encKey)>
    in
    [
        SentRetrievalRequest(bob_ED25519_pubKey, swarmNode_ED25519_pubKey, k),
        !Client(bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey, bob_ED25519_pubKey),
```

[2] Remember that we modeled just one Swarm Node for each user for semplicity

```
        In(senc(<'new_messages', sealed_box>, k))
    ]
    --[
        SealedBoxReceived(alice_ED25519_pubKey, bob_ED25519_pubKey, sealed_box),
        BoxReceived(alice_ED25519_pubKey, bob_ED25519_pubKey, box),
        MessageReceivedWithTimestamp(alice_ED25519_pubKey, bob_ED25519_pubKey, plaintext, ~timestamp)
    ]->
    []
```

Notice that through the incremental definition of variables within the `let ... in ...` construct we are able to model the verification tests done upon retrieval of a message: as explained before, pattern matching ensures the coherence of the data by excluding traces in which the ground istances of the variables do not respect the aforementioned equations.

4.3.5 Formalizing attacker capabilities

Since we ca not be sure that every entity in the protocol is honest, we have to write some additional rules to model malevolent agents and compromised information:

Client public keys compromises

Ideally, key exchange should be done out-of-band through a secure channel (e.g. in person), but sometimes this could not be the case: an user should be able to share his own public key freely (for example by writing in an Session's Open Group, see 3.3) without compromising the privacy of his conversations.

```
rule PublicClient:
    [ !Client(client_X25519_prKey, client_X25519_pubKey, client_ED25519_prKey, client_ED25519_pubKey) ]
    --[ PublicKeys(client_ED25519_pubKey) ]->
    [ Out(<client_ED25519_pubKey, client_X25519_pubKey>) ]
```

Client controlled by an attacker

Due to the open-source nature of Session, it is trivial for an attacker to forge the relative valid pairs of keys for new a client. Similarly, even though private keys are never sent through the network in this protocol, an attacker may compromise an honest client (for example by physically accessing the device). Both situations are modeled through the following rule:

```
rule DishonestClient :
    [ !Client(client_X25519_prKey, client_X25519_pubKey, client_ED25519_prKey, client_ED25519_pubKey) ]
    --[ Compromised(client_ED25519_pubKey) ]->
    [ Out(<client_X25519_prKey, client_X25519_pubKey, client_ED25519_prKey, client_ED25519_pubKey>) ]
```

Nodes controlled by an attacker

Analogously, we want to formalize the possibility of an attacker gaining control of a node:

```
rule DishonestSwarmNode :
    [ !SwarmNode(bob_ED25519_pubKey, swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
        swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey) ]
    --[ Dishonest(swarmNode_ED25519_pubKey) ]->
    [ Out(<swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
        swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey>) ]
```

```
rule DishonestServiceNode :
    [ !ServiceNode(serviceNode_X25519_prKey, serviceNode_X25519_pubKey,
        serviceNode_ED25519_prKey, serviceNode_ED25519_pubKey) ]
    --[ Dishonest(serviceNode_ED25519_pubKey) ]->
    [ Out(<serviceNode_X25519_prKey, serviceNode_X25519_pubKey,
        serviceNode_ED25519_prKey, serviceNode_ED25519_pubKey>) ]
```

Note that the semantics of the `Dishonest(pub_ED_key)` and `Compromised(pub_ED_key)` remains the same for both clients, Service Nodes and Swarm Nodes: if there is a multiset A_i in the i^{th} position of a trace and a key `k` so that `Dishonest(k)` $\in A_i \vee$ `Compromised(k)` $\in A_i$, then for each istant $j > i$ the attacker knows all keys (public and private) of `k`.

4.3.6 Addressing non-termination

As explained by V. Cortier, S. Delaune and J. Dreier, [73], since Tamarin implements an untyped system, when modeling a protocol in which the same message is sent more than once through the network partial deconstructions may occour. In this case in particular, the `sealed_box` sent by Alice to the Swarm Node is later on forwarded to Bob, making it impossible for the automatic prover to determine the initial source of the `sealed_box`. To resolve this, we defined an additional lemma that removes the ambiguity:

```
lemma types [sources] :
    "All #t data swarm . StoreArrived(data, swarm) @ #t ==>
        ((Ex #x . StoreSent(data, swarm) @ #x & #x < #t) |
        (Ex #y . KU(data) @ #y & KU(swarm) @ #y & #y < #t))"
```

This source lemma, that is automatically proved during the precomputation phase of the Tamarin theory, resolves the 432 partial deconstructions introduced by the *Raw sources*.

4.3.7 Checking the protocol

In order to check that the protocol was modeled correctly, we can prove the following `SanityCheck`:

```
lemma SanityCheck :
    exists-trace
        "Ex #a #b alice bob box .
            SealedBoxSent(alice, bob, box) @ #a &
            SealedBoxReceived(alice, bob, box) @ #b &
            (#a < #b)"
```

Intuitively, it states Alice can send a `box` to Bob and Bob can retrieve it from his Swarm (after it was sent of course).

Due to the high number of rules defined in the previous subsection, the smart heuristic provided within Tamarin is not able to find quickly a solution to this lemma. In order to prove it, we have two alternatives:

1. to use the interactive mode and manually guide the demonstration;
2. to comment out the rules defined in section 4.3.5 and auto-prove the formula (through this way the smart heuristic is able to find a valid trace within minutes). This trick can be considered valid since `SanityCheck` is annotated as a `exists-trace` lemma: contrary to property lemmas, which require that a determined formula holds for all possible traces, this lemma only needs one valid trace (that

matches a complete run of the protocol) to be verified. If a trace can be found with a set R_0 of rewriting rules, trying to prove the same lemma with a set of rules R_1 such as $R_0 \subset R_1$ is trivially possible by not applying the rules belonging to R_1/R_0.

4.4 Security Properties

After formalizing through multiset-rewriting rules the E2E protocol, we can now proceed with the actual verification of the security properties the official article [21] claims are guaranteed. As previously explained, the numerosity of the set of rules makes it difficult for the smart heuristic to auto-prove our lemmas. Considering that we can not exclude rules from `all-traces` demonstrations, we have to manually guide the proof to termination. Since this is a time-consuming process that requires a lot of trial and error even with the help of a proof assistant, we decided to write an oracle to make the verification automatic.

Before introducing the actual code, it is helpful to remind the mechanism through which Tamarin demonstrates property lemmas: given a formula, it negates it and tries to find a valid counterexample. If no counterexample is found among all the sub-cases in which the proof gets split, the property is considered verified. Thus, to speed up the proving process, we have to prioritize goals that we are sure (by knowledge of the problem or empirical experience with the interactive mode) are impossible to solve in order to quickly "close" a branch without further splitting it and carry on to the next case. Since this method can be used to support the demonstration of all properties in this section (except Sender Anonymity, which will be undertaken on its own), we decided to write a general oracle, which loads from a local file the correct goal-priority list based on the name of the lemma under verification. Priority lists are simply ordered list of goals, ranked in ascending order according to the number of steps required to reach a contradiction (for example, the first elements will cause a contradiction immediately, while the last ones will need numerous steps).

```
#!/usr/bin/python3

import sys, json

# Location of the file with the priority lists
PRIORITIES = './E2EPriorities.json'

# Auxiliary functions that, given a the name of a lemma, returns a ordered list of goals
def loadPriorities (lemmaName) :

    # Open a file and parse its contents as a Python dictionary
    with open(PRIORITIES) as file:
        oraclesPriorities = json.loads(file.read())

    # Select the correct priority list based on the name of the lemma
    priorities = oraclesPriorities[lemmaName]

    # Return the list
    return priorities

# Name of the current lemma
lemma = sys.argv[1]
```

```
23
24  # List of goals to solve
25  goals = sys.stdin.readlines()
26
27  try :
28      # If the name of the lemma is defined within the file load the relevant priority list ...
29      priorities = loadPriorities(lemma)
30  except :
31      # ... else revert to the standard smart heuristic
32      print('0')
33      exit(0)
34
35  # Search for the index of the highest priority goal present in the goals list
36  for i in range(len(priorities)) :
37      for j in range(len(goals)) :
38          # If found, print its index to solve it
39          if priorities[i] in goals[j] :
40              print(str(j))
41              exit(0)
42
43  # If no priority goal is found within the list, use the standard smart heuristic
44  print('0')
45  exit(0)
```

4.4.1 Message Secrecy

Message secrecy is proved through the following lemma:

```
1  lemma MessageSecrecy :
2      "not(
3          Ex #a #b #k alice bob box .
4              (not Ex #c . Compromised(bob) @ #c) &
5              BoxSent(alice, bob, box) @ #a &
6              BoxReceived(alice, bob, box) @ #b &
7              K(box) @ #k
8          )"
```

This lemma can be read as follows: there is no valid trace in which a user sends to an uncompromised client a `box` and the attacker is able to retrieve the content of said `box`. Note that:

- the formula does not require Alice not to have been compromised: since each message is encrypted with an ephemeral key obtained through a fresh ECDH exchange, an attacker would not be able to retrieve the symmetric key even if it knew Alice's private keys;
- as seen in the code listing at 4.3.3, `box` not only contains the actual plaintext and options of the message, but also its timestamp, Alice's $pub\_ED25519$ key and signature.

The priority list needed to prove this lemma is the following:

```
1  "Secrecy" : [
2      "!KU( ~ephemeral_prKey )",
3      "!KU( ~client_X25519_prKey.1 )",
4      "!KU( 'g'^(~ephemeral_prKey*~client_X25519_prKey.1) )",
5      "!KU( ~timestamp )",
```

```
    "!KU( ~plaintext )"
]
```

4.4.2 Message Authenticity

To demonstrate that an attacker is not able to forge messages we defined the following lemma:

```
lemma Authenticity :
    "not(
        Ex #i alice bob box .
            (Ex #c . IsAClient(alice) @ #c) &
            not (Ex #d . Compromised(alice) @ #d) &
            BoxReceived(alice, bob, box) @ #i &
            not (Ex #j . BoxSent(alice, bob, box) @ #j & #j < #i)
        )"
```

Intuitively, this formula states that if a user has retrieved a `box` from its Swarm, and the specified sender for the `box` is a client which has not been compromised, then the message has to be authentic.

Notice that we have to explicitly specify the fact that Alice is a client: since every node belonging to the network has both an $ED25519$ and a $X25519$ keypair, a Swarm Node could possibly create a valid message for Bob, sign it with its own $priv\_ED25519$ key and store it in the correct queue. This message would then be received by Bob as authentic even if it never was sent via `BoxSent(sender, receiver, box)`, thus causing Tamarin to flag the trace as an attack trace. Since we have no control or certainty about how the private keys of Service Nodes are kept, we can only demonstrate authenticity of messages coming from other users.

The priority list needed to prove this lemma is the following:

```
"Authenticity" : [
    "!KU( ~client_ED25519_prKey )",
    "!KU( sign(<<plaintext, ~timestamp, options>,",
    "!Client( bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey,",
    "!KU( senc(<'retrieve',",
    "!KU( senc(<'store',",
    "!KU( senc(<'new_messages'",
    "!KU( senc(<"
]
```

4.4.3 Anti-Replay

As seen in the protocol summary at 4.2.3, the E2E protocol implements timestamps, which allow for anti-replay resistance:

```
lemma AntiReplay :
    "not (
        Ex #i alice bob message timestamp .
            (Ex #c . IsAClient(alice) @ #c) &
            not (Ex #c . Compromised(alice) @ #c) &
            MessageReceivedWithTimestamp(alice, bob, message, timestamp) @ #i &
            not (Ex #j . MessageSentWithTimestamp(alice, bob, message, timestamp) @ #j)
    )"
```

This lemma is quite similar to the previous one: under the same assumptions (the sender is an uncompromised client), we know that each message received with a certain timestamp has to have been sent with the same timestamp through the legit sending procedure: thus, an attacker is not able to perform a replay attack by forwarding an old message with a new timestamp.

The priority list needed to prove this lemma is the following:

```
"AntiReplay" : [
    "!KU( ~client_ED25519_prKey )",
    "!KU( ~client_ED25519_prKey.2 )",
    "!KU( sign(<<message, ~timestamp, options>,",
    "!KU( sign(~timestamp.1, ~client_ED25519_prKey.2)",
    "!Client( bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey,",
    "!KU( senc(<'retrieve',",
    "!KU( senc(<'store',",
    "!KU( senc(<'new_messages'",
    "!KU( senc(<"
]
```

4.4.4 Sender anonymity

While IP-anonymity is mainly a concern of the underlying onion routing protocol and will be verified in the relevant section of this thesis 5.5, $ED25519$ key anonymity must be undertaken at this point.

Trace property formalization

At first we tried to express this security goal as a trace property, but, as we will shortly see, the assumptions required in order to obtain a meaningful proof undermine the theorem's practical utility within this context:

```
lemma SenderAnonymity :
    "not (
        Ex alice #t .
            IsAClient(alice) @ #t &
            not(Ex bob box #t #c . BoxSent(alice, bob, box) @ #t & Compromised(bob) @ #c) &
            not(Ex #c . PublicKeys(alice) @ #c) &
            not(Ex #c . Compromised(alice) @ #c) &
            not(Ex #c . QueryForSwarm(alice) @ #c) &
            (Ex #k . K(alice) @ #k)
    )"
```

Let us explain why we need all these facts to demonstrate such an *apparently* simple property:

- the `IsAClient(client_ED_key)` action fact is needed since Tamarin requires lemmas to be guarded fragments of first order logic;
- we have to specify that each intended receiver of an Alice's message must not have been compromised: if it were, the attacker could trivially deduce Alice's public key from a decrypted message;
- Alice must not have been compromised and/or publicly have distributed her keys;
- no-one in the network must have required the Swarm for Alice: if not, an attacker who has compromised the queried service node would easily gain knowledge of Alice's identity. Note that

this premise also prevents Alice from retrieving her own messages, since even she is not able to obtain her own Swarm.

Note that also an additional restriction needs to be added to the theory:

```
restriction SendTheMessageToTheRightNode :
    "All #t0 bob swarm . SentBoxToSwarm(bob, swarm) @ #t0 ==> Ex #t1 . ClientsSwarm(bob, swarm) @ #t1"
```

Without this restriction, a client could possibly send to a random swarm node (thus without querying for Alice's swarm) a message for Alice. By compromising the node's private keys, the attacker would be trivially able to deduce the receiver's public $ED25519$ key, thus causing a virtually correct attack to the property. Since such an execution would not make sense, we decided to limit the trace to only legit runs of the protocol.

Similarly to the other properties, also the automatic proof of this lemma requires the support of an oracle with priority list. In this case the list is very similar to the one introduced for the message secrecy demonstration 4.4.1:

```
"SenderAnonymity" : [
    "!KU( ~ephemeral_prKey )",
    "!KU( ~client_X25519_prKey.1 )",
    "!KU( 'g'^(~ephemeral_prKey*~client_X25519_prKey.1"
]
```

As one can see, this lemma provides pretty limited results: in practice, we can only demonstrate that Alice's public key remains unknown to an attacker until somebody responds to a message sent by her. While this conclusion could be useful in the development of future protocols that feature ephemeral identities (for example in blockchain transactions where pseudonyms are used only once before being discarded), it is not particularly useful within a messaging app context. The problem with our lemma is that, while we want to express anonymity as a relationship property (we are more interested in proving that nobody can deduce that Alice is communicating with Bob instead of demonstrating that it is impossible to know the presence of Alice in the network on its own), proving trace properties requires to specify (persistent) knowledge as multiple atomic (and thus unrelated) informations (i.e.: the attacker knows a message, a key, or, in this case, an identity).

Observational Equivalence formalization

As introduced in subsection 2.3.5, Tamarin allows for the specification of observational equivalences properties through the use of the `diff()` operator. In our case, this feature is useful since it lets us prove sender anonymity as a relationship property: instead of verifying that the attacker is not able to deduce a particular $ED25519$ public key from a limited run of the protocol, we are able to show that if two different clients send a message to an uncompromised user, the attacker is not able to distinguish between the two protocol executions.

In order to do so, we have to modify the `SendMessage` rewriting rule to accomodate the two different runs of the protocol:

```
rule SendMessage :
    let
        message1 = <plaintext1, ~timestamp, ~options>
        signature1 = sign(<message1, sender1_ED25519_pubKey, bob_X25519_pubKey>, sender1_ED25519_prKey)
```

```
        box1 = <message1, sender1_ED25519_pubKey, signature1>

        message2 = <plaintext2, ~timestamp, ~options>
        signature2 = sign(<message2, sender2_ED25519_pubKey, bob_X25519_pubKey>, sender2_ED25519_prKey)
        box2 = <message2, sender2_ED25519_pubKey, signature2>

        ephemeral_key1 = 'g'^~ephemeral_prKey1
        symKey1 = bob_X25519_pubKey^~ephemeral_prKey1
        nonce1 = h(<ephemeral_key1, bob_X25519_pubKey>)
        k1 = swarmNode_X25519_pubKey^~ECDH_prKey1
        ECDH_pubKey1 = 'g'^~ECDH_prKey1

        ephemeral_key2 = 'g'^~ephemeral_prKey2
        symKey2 = bob_X25519_pubKey^~ephemeral_prKey2
        nonce2 = h(<ephemeral_key1, bob_X25519_pubKey>)
        k2 = swarmNode_X25519_pubKey^~ECDH_prKey2
        ECDH_pubKey2 = 'g'^~ECDH_prKey2

        sealed_box1 = <ephemeral_key1, senc(<box1, bob_X25519_pubKey, symKey1, nonce1>, symKey1)>
        swarm_node_data1 = <sealed_box1, bob_ED25519_pubKey, ~timestamp, ~ttl>
        data_sent1 = <ECDH_pubKey1, senc(<'store', swarm_node_data1>, k1), swarmNode_ED25519_pubKey>

        sealed_box2 = <ephemeral_key2, senc(<box2, bob_X25519_pubKey, symKey2, nonce2>, symKey2)>
        swarm_node_data2 = <sealed_box2, bob_ED25519_pubKey, ~timestamp, ~ttl>
        data_sent2 = <ECDH_pubKey2, senc(<'store', swarm_node_data2>, k2), swarmNode_ED25519_pubKey>
    in
    [
        !SwarmNode(bob_ED25519_pubKey, swarmNode_X25519_prKey, swarmNode_X25519_pubKey,
            swarmNode_ED25519_prKey, swarmNode_ED25519_pubKey),
        !Client(sender1_X25519_prKey, sender1_X25519_pubKey, sender1_ED25519_prKey, sender1_ED25519_pubKey),
        !Client(sender2_X25519_prKey, sender2_X25519_pubKey, sender2_ED25519_prKey, sender2_ED25519_pubKey),
        MessageToSend(sender1_ED25519_pubKey, bob_ED25519_pubKey, plaintext1),
        MessageToSend(sender2_ED25519_pubKey, bob_ED25519_pubKey, plaintext2),
        ClientReadyToSend(sender1_ED25519_pubKey, bob_ED25519_pubKey,
            swarmNode_ED25519_pubKey, swarmNode_X25519_pubKey),
        ClientReadyToSend(sender1_ED25519_pubKey, bob_ED25519_pubKey,
            swarmNode_ED25519_pubKey, swarmNode_X25519_pubKey),
        !Client(bob_X25519_prKey, bob_X25519_pubKey, bob_ED25519_prKey, bob_ED25519_pubKey),
        Fr(~timestamp), Fr(~ttl), Fr(~options),
        Fr(~ECDH_prKey1), Fr(~ephemeral_prKey1),
        Fr(~ECDH_prKey2), Fr(~ephemeral_prKey2)
    ]
    --[
        StoreSent(diff(data_sent1, data_sent2), swarmNode_ED25519_pubKey)
    ]->
    [ Out(diff(data_sent1, data_sent2)) ]
```

As we can see, this rule allows the contextual sending of two different messages, encrypted with unrelated sets of keys, from two distinct senders, to the same Swarm Node for Bob.

Observational equivalences are proved in Tamarin through rule-equivalence by backwards search. This means that the premises of each rule are analyzed from any possible source, and the property in question is proved if and only if each run of the *left-hand side* of the protocol (i.e. the left argument of the `diff()` operator) is correctly mirrored by a run of the *right-hand side* (i.e. the right argument). If any of the two executions reveal more information than the other, then the automatically generated

`Observational_equivalence` lemma is considered false.

Unfortunately, due to the high number of `Out()` facts necessary to formalize this theory, the numerosity of possible sources for `In()` facts causes an exponential explosion of subcases, thus leading to non-termination during the proof of some of the rules.

4.4.5 Proof results

In this subsection we will summarize the results and performance metrics collected during the automatic proving of the aforementioned security properties. To gain some insightful data about Tamarin efficiency, we adopted a benchmarking approach similar to the one used by P. Lafourcade and M. Puys [55]: we measured time duration and peak memory usage for 20 executions of each proof using the GNU `time` command [48].

The tests were conducted on a 8GB M1 Macbook Air using Tamarin version 1.6.1.

			Duration [ms]				Peak RAM usage [kB]			
Lemma	# of steps	Oracle	Mean	Max	Min	Deviation	Mean	Max	Min	Deviation
SanityCheck [3]	34	✗	512342	525490	455140	15296	2117842	2198240	2002384	60718
Secrecy	84	✓	36256	39090	34020	915	451560	470880	433040	8822
Authenticity	81	✓	36967	37820	36010	359	453213	474032	437120	11097
AntiReplay	122	✓	36052	37070	31130	1280	452863	482176	430880	10635
SenderAnonymity	52	✓	35919	36980	31600	1260	451792	470864	428880	10014
SenderAnonymity (obs. eq.)	-	-	-	-	-	-	-	-	-	-

Table 4.1: Tamarin's performances during the verification of the E2E properties

At first sight it may seem that there was an error during the measure of the `SanityCheck`'s metrics: in comparison with the other lemmas belonging to the theory, proving this proposition requires both the least amount of steps and the most amount of computing resorces by a large margin. This behaviour can be explained by noticing that it is also the only `exists-trace` lemma for the E2E protocol (thus only requiring one correct sequence of rules application to be proven) and that it is the only one not supported by an underlying oracle. Keeping in mind that trying to prove the other formulas using exclusively Tamarin's smart heuristic fails due to non-termination, these results show how helpful custom oracles can be while verifying security properties with an automated framework.

Please note that each single proof was executed from scratch, thus the metrics presented include the precomputation phase overhead.

[3] We want to remember that this lemma was proved excluding the additional attacker rules from the theory.

5

Onion Routing Protocol

5.1 Reverse-engineering the protocol

As anticipated in the previous chapter, a good part of the E2E Protocol uses onion routing as an abstraction to guarantee anonymity during phases 2, 3 and 4 of the exchange. In this chapter we'll further analyze the underlying API to prove that, within certain assumptions 5.3, it conforms its security specification.

5.1.1 Sending procedure

The API call

As highlighted in 4.1, the code for the E2E Protocol calls function `snodeRpc()` (defined in `/ts/session/apis/snode_api/sessionRpc.ts`) in order to use the underlying routing protocol. This procedure standardizes the call parameters and forwards them to `lokiFetch()`, which checks if onion routing is disabled for the app (probably a left-over of previous coding iterations: disabling onion routing can be done only by setting to `false` a global variable which is not reached by any code in the entire repository) and proceeds to call `lokiOnionFetch()`. This function handles the creation of the onion path and the sending of the data.

Building the onion path

The management of available onion paths is done through `getOnionPath()`, which is defined in `/ts/session/onions/onionPath.ts`. This function keeps track of previous onion paths that have failed and returns a random path from a local pool of alternatives. If not enough paths are available (in the desktop version the current minimum is set to 2), a new one is created through `buildNewOnionPathsOneAtATime()`. This function calls a JavaScript worker, `buildNewOnionPathsWorker()`, which will handle the construction of routes in the background. In particular, this is done through the following algorithm:

```
1  // 1. ) Get an up to date list of snodes from cache, from db, or from the a seed node.
2  let allNodes = await SnodePool.getSnodePoolFromDBOrFetchFromSeed();
3    if (allNodes.length <= SnodePool.minSnodePoolCount) {
4      throw new Error(`Cannot rebuild path as we do not have enough snodes: ${allNodes.length}`);
5  }
6
7  // 2. ) Make sure there are enough guard nodes to build the paths
```

```
await OnionPaths.getGuardNodeOrSelectNewOnes();

// 3. ) Be sure to fetch again as the previous list might have been refreshed by selectGuardNodes
allNodes = await SnodePool.getSnodePoolFromDBOrFetchFromSeed();

// 4. ) get all Service Nodes minus the selected guardNodes
if (allNodes.length <= SnodePool.minSnodePoolCount) {
    throw new Error('Too few nodes to build an onion path. Even after fetching from seed.');
}

// 5. ) Make sure to not reuse multiple times the same subnet /24
const allNodesGroupedBySubnet24 = _.groupBy(allNodes, e => {
  const lastDot = e.ip.lastIndexOf('.');
  return e.ip.substr(0, lastDot);
});
const oneNodeForEachSubnet24KeepingRatio = _.flatten(
  _.map(allNodesGroupedBySubnet24, group => {
    return _.fill(Array(group.length), _.sample(group) as Data.Snode);
  })        // note that "_" is an imported object from Lodash's library
);
if (oneNodeForEachSubnet24KeepingRatio.length <= SnodePool.minSnodePoolCount) {
  throw new Error(
    'Too few nodes "unique by ip" to build an onion path. Even after fetching from seed.'
  );
}

// 6. ) Separate Guard Nodes from the rest of the nodes
let otherNodes = _.differenceBy(
  oneNodeForEachSubnet24KeepingRatio,
  guardNodes,
  'pubkey_ed25519'
);
const guards = _.shuffle(guardNodes);

// 7. ) Create the maximum amount of paths possible with the current set of available nodes
const nodesNeededPerPaths = 2;
const maxPath = Math.floor(Math.min(guards.length, otherNodes.length / nodesNeededPerPaths));
onionPaths = [];
for (let i = 0; i < maxPath; i += 1) {
  const path = [guards[i]];
  for (let j = 0; j < nodesNeededPerPaths; j += 1) {
    const randomWinner = _.sample(otherNodes) as Data.Snode;
    otherNodes = otherNodes.filter(n => {
      return n.pubkey_ed25519 !== randomWinner?.pubkey_ed25519;
    });
    path.push(randomWinner);
  }
  onionPaths.push(path);
}
```

Notice that:

- an onion path is made up of 3 nodes, including the receiving one;
- only one path per guard node (the first node of the routing phase) is allowed;

- each guard node has to belong to a different /24 subnet.

Under the assumption that only some of the nodes can be malevolent because of the staking resources needed 3.2.2, these requirements enforce the idea that only part of the traffic will be exposed to attacker inspection/modification.

Encrypting the payload

Before sending the data, the payload has to be encrypted in an incremental way: this is managed by `sendOnionRequest()`, called after a series of functions initiated by `lokiOnionFetch()`. Firstly, the plaintext is encrypted for the last hop of the routing protocol: the destination Service Node. This is done through `encryptForPubKey()` via the following function:

```
async function encryptForPubkey(pubkeyX25519str, payloadBytes) {

    // 1. ) Generate a random X25519 keypair on the spot
    const ephemeral = await generateEphemeralKeyPair();

    // 2. ) Obtain a symmetric key via an ECDH exchange with the recipient node's public X25519 key
    const pubkeyX25519Buffer = fromHexToArray(pubkeyX25519str);
    const symmetricKey = await deriveSymmetricKey(
        pubkeyX25519Buffer,
        new Uint8Array(ephemeral.privKey)
    );

    // 3. ) Encrypt the payload with the symmetric key and return the encrypted plaintext, along with the keys
    const ciphertext = await EncryptAESGCM(symmetricKey, payloadBytes);
    return { ciphertext, symmetricKey, ephemeralKey: ephemeral.pubKey };
}
```

Note that the previous code snippet does an ECDH exchange with the receiver's public $X25519$ key and an ephemeral key generated on the spot. This allows to have always different symmetric keys for each connection. `encryptForPubkey()` is then executed using `ciphertext` as the subsequent layer's `payloadBytes`: this process is managed by `buildOnionGuardNodePayload()`. The actual implemented algorithm can be seen in `/ts/session/apis/snode_api/onions.ts` in the code of `buildOnionCtxs()`.

Note that, along with each layer's ciphertext, also the relevant routing and decryption information (the intended node's $ED25519$ public key and the ephemeral public key for the server-side ECDH exchange) is encrypted for the next layer.

Actual sending

After encrypting the payload for the onion routing, the data is sent to the first node of the path (the guard node). Since all the network infrastructure is based on https, `sendOnionRequest()` also composes the correct URL to reach it as

```
const url = `https://${guardNode.ip}:${guardNode.port}/onion_req/v2`;
```

Finally, the request is sent through a wrapper of the `fetch()` API. The application only saves the symmetric key shared with the receiving node: this allows us to suppose that the responses are delivered in the inverse direction of the path with only one level of symmetrical encryption.

5.2 Protocol Summary

The onion routing protocol implemented by Session is quite simple and mimics Tor's functioning [64] in order to guarantee sender anonymity. In the following subsection we'll analyze how these security goals are achieved, but before proceeding we have to highlight that this protocol is executable only after (at least) a successful run of phase 1 of the E2E protocol 4.2.1: the sender needs a local pool of multiple Service Nodes available to construct the paths.

As a reminder, we recall that when a client "knows" a Service Node, it means that its IP, port and public $ED25519$ and $X25519$ keys are available in the local database.

5.2.1 Encryption

Let us assume that the sender wants to sent to Service Node$_\gamma$ a piece of data we'll call simply plaintext. The innermost layer of encryption can be defined as

$$\text{layer\_for\_}\gamma = \text{ephemeral\_key}_\gamma \;||\; E_{K_\gamma}(\text{plaintext}) \;||\; pub\_ED25519_\gamma \tag{5.1}$$

- ephemeral_key$_\gamma$ is the public key obtained through the ECDH exchange described in the previous section. Service Node$_\gamma$ will use it, along with its own private $X25519$ key to derive E_{K_γ};
- E_{K_γ} is the symmetric encryption key calculated via `deriveSymmetricKey()`, also shown in previous section;
- $pub\_ED25519_\gamma$ is the identifier used to route the message to Service Node$_\gamma$.

Let us also assume that the onion path previously constructed includes two more nodes, Service Node$_\alpha$ and Service Node$_\beta$, in this order. The innermost layer is then encrypted again for the other nodes belonging to the onion path selected. Keeping the same name convention as 5.1, the data is overall encrypted as

$$\text{layer\_for\_}\beta = \text{ephemeral\_key}_\beta \;||\; E_{K_\gamma}(\text{layer\_for\_}\gamma) \;||\; pub\_ED25519_\beta$$
$$\text{layer\_for\_}\alpha = \text{ephemeral\_key}_\alpha \;||\; E_{K_\gamma}(\text{layer\_for\_}\beta) \;||\; pub\_ED25519_\alpha$$

Altogether, the sending procedure will look like:

$$\text{sender} \rightarrow \text{Service Node}_\alpha : \text{layer\_for\_}\alpha$$
$$\text{Service Node}_\alpha \rightarrow \text{Service Node}_\beta : \text{layer\_for\_}\beta$$
$$\text{Service Node}_\beta \rightarrow \text{Service Node}_\gamma : \text{layer\_for\_}\gamma$$

The response will arrive to the sender through the communication channel with Service Node$_\alpha$ and will be symmetrically encrypted with E_{K_γ}.

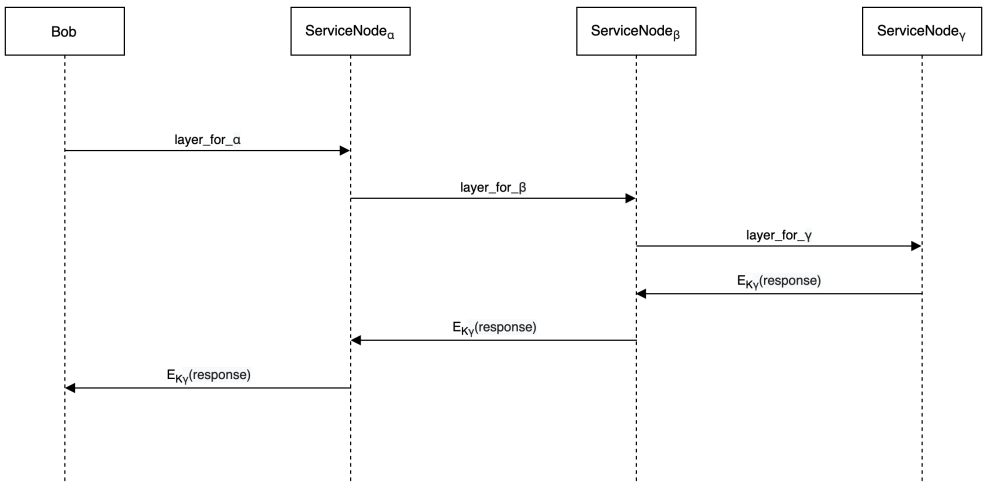

Figure 5.1: Summary of our assumption for the functioning of the Onion Routing protocol

Note that this encryption scheme is robust on paper, but for a true symbolic verification we should not trust any particular server since there is no certainity that it is running the code publicly available at [16].

5.3 Problems with the Dolev Yao model

As previously introduced in section 2.1, when verifying the security properties of abstract views of cryptographic protocols, it is common to operate within the Dolev-Yao's model [45]. This formal model expects clients (which are modeled as abstract machines) to send and receive messages (which are modeled as terms) to/from the network, that is completely controlled by the attacker. The malevolent agent is ubiquitous and can read, modify and synthetize any term beloging to a particular connection, but cannot break cryptography.

While this is a very consolidated paradigm, in our case we have to limit the assumptions made about the attacker's capabilities: as explained in Session's own whitepaper [54], the entire routing protocol would be useless against a malevolent agent that is in control of the entire network (a *Global Passive Adversary*, as called in their paper), since it could break anonymity through trivial traffic analysis. Similarly, an attacker that controls both ends of an onion routing tunnel (the guard and the receiving node) could quite easily deduce identity informations about the users of the communication (such as the public $ED25519$ key of the receiver and the IP address of the sender). Since Session claims to provide multiple precautions [15] to avoid these situations, we'll take in consideration an alternative thread model proposed in section 3.1 of [64]: given a constructed onion path with three different nodes, only one of them (without knowing priorly which) can be malevolent (and thus able to interfere with the traffic coming through it).

5.4 Protocol Formalization

5.4.1 Formalizing the sending primitives

Tamarin provides primitives to model standard input/output into and from an abstract machine. As explained in previous subsection 2.3.3, these built-in facts are complaint with the Dolev-Yao model, so are not useful in order to formalize a theory under different assumptions. To solve this problem, we created a couple of rules that allows us to limit the attacker's access to the network:

```
rule ModelOutput :
    [ ModelOut(sender, receiver, data, flow) ]
    -->
    [ Comm(sender, receiver, data, flow) ]
```

```
rule ModelInput :
    [ Comm(sender, receiver, data, flow) ]
    -->
    [ ModelIn(sender, receiver, data, flow) ]
```

The previous rules mimic the behaviour of the secure channel introduced in subsection 4.3.1. Note the additional `flow` term: to each layered message sent we associate an arbitrary label that identifies the connection between the sender and the receiving node. The `flow` term is not actually sent in the real world, but, as we will show shortly, it is needed in this situation to limit the number of encryption layers for a message.

We decided to use this formalization because it initially excludes the attacker from the communication: the interaction between the malevolent agent and the system can be completely modeled through additional rules and restrictions.

```
rule DishonestModelInput :
    [ Comm(sender, receiver, data, flow) ]
    --[ DishonestReceive(receiver) ]->
    [
        ModelIn(sender, receiver, data, flow),
        Out(<sender, data>)
    ]
```

```
rule DishonestModelOutput :
    [ K(sender), K(receiver), K(data), K(flow) ]
    --[ DishonestSend(sender) ]->
    [ ModelOut(sender, receiver, data, flow) ]
```

The last rules formalize the connection at an endpoint controlled by an attacker: if a malevolent agent has access to a Service Node, then it is able to inspect the incoming traffic and forge messages in outgoing connections. Note three things:

- the possibility of observing incoming data is portrayed through the `Out(data)` fact: since the built-in input/output facts for Tamarin are contextualized within the standard Dolev-Yao model, any message sent to the network is automatically read by the attacker;
- a malevolent agent is not able to deduce neither the `receiver` or the `flow` of an incoming connection. Whereas previous knowledge of the `receiver` is taken for granted since the attacker has to have

already compromised the node with all of its information, `flow` is not a "real" information sent through the network, thus nobody can actually deduce it in the real world;

- having two different copies (honest and dishonest) of the I/O facts causes non-determinism in the choice of rule to apply during the proof, consequently producing partial deconstructions. Unlike the previous chapter, where we managed to solve this problem through the use of a source lemma, in this case we must bear the additional complexity added to the proofs since we need both pairs of rules.

- the previous rules would actually allow an attacker to intercept and modify traffic in any node. To limit this capability to only compromised nodes, we need to write the following additional restiction

```
restriction OnlyCompromisedNodesCanBeDishonest :
    "(All #t node . DishonestSend(node) @ #t ==> Ex #d . Dishonest(node) @ #d & #d < #t) &
    (All #t node . DishonestReceive(node) @ #t ==>  Ex #d . Dishonest(node) @ #d & #d < #t)"
```

The `Dishonest()` fact indicates that an entity in the network was compromised/is malevolent. It will be introduced shortly in the next subsection.

5.4.2 Formalizing the initialization

Service Nodes are created and initialised in a very similar way to the rules introducted in previous section 4.3.1 for the E2E protocol:

```
rule InitialiseServiceNode :
    let
        serviceNode_X25519_pubKey = 'g'^~serviceNode_X25519_prKey
        serviceNode_ED25519_pubKey = pk(~serviceNode_ED25519_prKey)
    in
    [
        Fr(~serviceNode_X25519_prKey),
        Fr(~serviceNode_ED25519_prKey),
        Fr(~serviceNode_IP)
    ]
    -->
    [
        !ServiceNode(serviceNode_X25519_pubKey, ~serviceNode_X25519_prKey,
            serviceNode_ED25519_pubKey, ~serviceNode_ED25519_prKey, ~serviceNode_IP),
        Out(<serviceNode_X25519_pubKey, serviceNode_ED25519_pubKey, ~serviceNode_IP>)
    ]
```

Notice that in this formalization we included the nodes' IP: since we're analysing P2P connections on a lower level, logical addresses convey information about the identity of entities at stake.

We also added the `Out(node_X_Key, node_ED_Key, node_IP)` since we can not exclude the possibility that an attacker may deduce the presence of a new node from the network, for example executing a run of the second phase of the E2E protocol 4.2.1.

To allow an attacker to control a node, we wrote the following rule:

```
rule DishonestServiceNode :
    [
```

```
        !ServiceNode(serviceNode_X25519_pubKey, serviceNode_X25519_prKey,
            serviceNode_ED25519_pubKey, serviceNode_ED25519_prKey, serviceNode_IP)
    ]
    --[ Dishonest(serviceNode_IP) ]->
    [
        Out(<serviceNode_X25519_pubKey, serviceNode_X25519_prKey,
            serviceNode_ED25519_pubKey, serviceNode_ED25519_prKey, serviceNode_IP>)
    ]
```

The initialization of a client is portrayed through the next rule:

```
rule InitialiseClient :
    [ Fr(~sender_IP) ]
    -->
    [ !Client(~sender_IP) ]
```

Unlike the formalization of the E2E protocol, in this case we do not specify $ED25519$ and $X25519$ key pairs for the clients since we do not care about routing messages to users.

5.4.3 Formalizing Message Creation

Following the considerations done at 4.3.3, we wrote a couple of rules to model the creation of an onion path and a message to send, given a sender client and a receiver node:

```
rule CreateMessage :
    [
        !Client(sender_IP),
        !ServiceNode(receivingNode_X25519_pubKey, receivingNode_X25519_prKey,
            receivingNode_ED25519_pubKey, receivingNode_ED25519_prKey, receivingNode_IP),
        Fr(~message)
    ]
    -->
    [
        Data(sender_IP, receivingNode_ED25519_pubKey, ~message),
        NeedPath(sender_IP, receivingNode_ED25519_pubKey)
    ]
```

```
rule CreatePath :
    [
        NeedPath(sender_IP, receivingNode_ED25519_pubKey),
        !ServiceNode(firstNode_X25519_pubKey, firstNode_X25519_prKey,
            firstNode_ED25519_pubKey, firstNode_ED25519_prKey, firstNode_IP),
        !ServiceNode(secondNode_X25519_pubKey, secondNode_X25519_prKey,
            secondNode_ED25519_pubKey, secondNode_ED25519_prKey, secondNode_IP),
        !ServiceNode(receivingNode_X25519_pubKey, receivingNode_X25519_prKey,
            receivingNode_ED25519_pubKey, receivingNode_ED25519_prKey, receivingNode_IP)
    ]
    --[
        Neq(firstNode_ED25519_pubKey, secondNode_ED25519_pubKey),
        Neq(secondNode_ED25519_pubKey, receivingNode_X25519_pubKey),
        Neq(firstNode_ED25519_pubKey, receivingNode_ED25519_pubKey)
    ]->
    [
        OnionPath(sender_IP, firstNode_ED25519_pubKey,
            secondNode_ED25519_pubKey, receivingNode_ED25519_pubKey)
    ]
```

The action fact `Neq(node_a, node_b)` is useful to model the fact that the onion path has to be made of different nodes (as a matter of fact, as seen in 5.1.1, they ca not event belong to the same /24 subnet).

`Neq(node_a, node_b)` is a restriction taken from Tamarin's manual [71] and is written as follows:

```
restriction Inequality :
    "All x #i . Neq(x,x) @ #i ==> F"
```

5.4.4 Formalizing Message Sending

Given a message and an onion path for routing, the client is able to encrypt and send the message as seen in subsection 5.2.1. To dispatch the data, instead of the built-in `Out(data)` fact, it'll use our user-defined `ModelOut(sender, receiver, data)` in order to remain within our threat model.

```
rule SendMessage :
    let
        dh_pubKey_for_receivingNode = 'g'^~dh_prKey_for_receivingNode
        symKey_receivingNode = receivingNode_X25519_pubKey^~dh_prKey_for_receivingNode
        dh_pubKey_for_firstNode = 'g'^~dh_prKey_for_firstNode
        symKey_firstNode = firstNode_X25519_pubKey^~dh_prKey_for_firstNode
        dh_pubKey_for_secondNode = 'g'^~dh_prKey_for_secondNode
        symKey_secondNode = secondNode_X25519_pubKey^~dh_prKey_for_secondNode
        layer_for_receivingNode = <senc(<'store', message>, symKey_receivingNode),
            dh_pubKey_for_receivingNode, receivingNode_ED25519_pubKey>
        layer_for_secondNode = <senc(layer_for_receivingNode, symKey_secondNode),
            dh_pubKey_for_secondNode, secondNode_ED25519_pubKey>
        layer_for_firstNode = <senc(layer_for_secondNode, symKey_firstNode),
            dh_pubKey_for_firstNode, firstNode_ED25519_pubKey>
    in
    [
        Data(sender_IP, receivingNode_ED25519_pubKey, message),
        OnionPath(sender_IP, firstNode_ED25519_pubKey, secondNode_ED25519_pubKey, receivingNode_ED25519_pubKey),
        !ServiceNode(firstNode_X25519_pubKey, firstNode_X25519_prKey,
            firstNode_ED25519_pubKey, firstNode_ED25519_prKey, firstNode_IP),
        !ServiceNode(secondNode_X25519_pubKey, secondNode_X25519_prKey,
            secondNode_ED25519_pubKey, secondNode_ED25519_prKey, secondNode_IP),
        !ServiceNode(receivingNode_X25519_pubKey, receivingNode_X25519_prKey,
            receivingNode_ED25519_pubKey, receivingNode_ED25519_prKey, receivingNode_IP),
        Fr(~dh_prKey_for_receivingNode),
        Fr(~dh_prKey_for_firstNode),
        Fr(~dh_prKey_for_secondNode),
        Fr(~flow)
    ]
    --[
        MessageSent(sender_IP, receivingNode_IP, message),
        MessageSentThroughPath(sender_IP, firstNode_IP, secondNode_IP, receivingNode_IP),
        Sender(sender_IP)
    ]->
    [ ModelOut(sender_IP, firstNode_IP, layer_for_firstNode, ~flow) ]
```

Multiple action facts were defined within this rule:

- `MessageSent(sender, receiver, message)` is later on used for the definition multiple security properties and sanity checking the protocol;

- `MessageSentThroughPath(sender, guard, proxy, receiver)` will be shortly required to conclude the formalization of our threat model;
- `Sender(sender)` expresses that a certain IP address is associated to a client. This will be useful for specifying the Sender Anonymity lemma.

As expressed in 5.3, we assume that there is at most one malevolent node per onion path. To portray this limitation, we wrote the following restriction:

```
restriction NotMoreThanOneDishonestNodePerPath :
    "All sender guard proxy receiver #t .
        MessageSentThroughPath(sender, guard, proxy, receiver) @ #t
        ==>
        not (
            (Ex #x #y . (Dishonest(guard) @ #x & Dishonest(proxy) @ #y)) |
            (Ex #x #y . (Dishonest(guard) @ #x & Dishonest(receiver) @ #y)) |
            (Ex #x #y . (Dishonest(proxy) @ #x & Dishonest(receiver) @ #y))
        )"
```

5.4.5 Formalizing Message Forwarding

Assuming that each honest node decrypts the outermost layer of encryption and forwards the recovered ciphertext without other information to the next node in the path (as assumed in 5.2.1), we modeled onion forwarding through the following rule:

```
rule ForwardMessage :
    let
        symKey = dh_pubKey^serviceNode_X25519_prKey
        data_for_inner_layer = <encryptedData, dh_pubKey_nextNode, nextNode_ED25519_pubKey>
    in
    [
        !ServiceNode(serviceNode_X25519_pubKey, serviceNode_X25519_prKey,
            serviceNode_ED25519_pubKey, serviceNode_ED25519_prKey, serviceNode_IP),
        !ServiceNode(nextNode_X25519_pubKey, nextNode_X25519_prKey,
            nextNode_ED25519_pubKey, nextNode_ED25519_prKey, nextNode_IP),
        ModelIn(last_step_IP, serviceNode_IP, <senc(data_for_inner_layer, symKey),
            dh_pubKey, serviceNode_ED25519_pubKey>, flow)
    ]
    --[ MessageForwarded(serviceNode_IP, flow) ]->
    [ ModelOut(serviceNode_IP, nextNode_IP, data_for_inner_layer, flow) ]
```

The previously introduced `flow` term allows us now to bound the possible number of encryptions for each message:

```
restriction Only3Hops :
    "All flow node1 node2 node3 #t1 #t2 #t3 .
        (MessageForwarded(node1, flow) @ #t1 &
        MessageForwarded(node2, flow) @ #t2 &
        MessageForwarded(node3, flow) @ #t3) ==>
            (#t1 = #t2 | #t1 = #t3 | #t2 = #t3)"
```

Without this limitation, Tamarin could not exclude that a N-layered message cannot derive from a $N+1$-layered one, creating an the infinite application of the `ForwardMessage` rule and thus causing non-termination in the proof of `all-traces` properties.

5.4.6 Formalizing Message Receving

Finally, after the first hops of the onion path, the data sent through the rule defined at 5.4.4 should arrive at the intended receiving node. To model the arrival of a message, we added the last rule:

```
rule StoreMessage :
    let
        symKey = dh_pubKey^serviceNode_X25519_prKey
    in
    [
        !ServiceNode(serviceNode_X25519_pubKey, serviceNode_X25519_prKey,
            serviceNode_ED25519_pubKey, serviceNode_ED25519_prKey, serviceNode_IP),
        ModelIn(second_node_IP, serviceNode_IP, <senc(<'store', message>, symKey),
            dh_pubKey, serviceNode_ED25519_pubKey>)
    ]
    --[ MessageStored(serviceNode_IP, message) ]->
    []
```

The action fact `MessageStored(node, message)` will be useful in the following sections to define the security properties.

5.4.7 Addressing non termination

Even though the restriction `Only3Hops` presented in subsection 5.4.5 prevents Tamarin from entering loops of infinite message forwarding, the unsolved partial deconstructions cause a combinatory explosion of possible sources for each goal during the proof of the following lemmas. So, in order to guarantee termination and demonstrations of acceptable size, we decided to write an oracle to guide the automatic proofs. Unlike the script introduced in the previous chapter (see section 4.4), this oracle does not modify its behaviour depending on the lemma, since all of our demonstrations can be guided by prioritising the solution of `ModelIn()` facts and the application of the `Only3Hops` restriction (the syntax of the goal for this is `( (#vr.A, 0) ~~> (#vk.B, 0) )`, where `A` and `B` are natural numbers).

```
#!/usr/bin/python3

import sys

lemmas = ['SanityCheck', 'Secrecy', 'SenderAnonymity']

# Name of the current lemma
lemma = sys.argv[1]
# List of goals to solve
goals = sys.stdin.readlines()

if lemma in lemmas :
    # We want to prioritize goals that aim to solve the source for 'ModelIn' facts and goals containing '~~>'
    PriorityGoals = [index for (goal,index) in zip(goals, range(len(goals))) if ('ModelIn' in goal) or ('~~>' in goal)]
    if not PriorityGoals :
        # If no goals were found, return the standard goal found by the smart heuristic...
        # ... unless is a splitEqs goal ...
        if goals and 'splitEqs' in goals[0] :
            print('1')
        else :
            print('0')
```

```
        exit(0)
    else :
        # ... otherwise print the first occurrence of ModelIn
        print(str(PriorityGoals[0]))
        exit(0)
else :
    # If a new lemma is added to the theory, use the smart heuristic to try to solve it
    print('0')
    exit(0)
```

Note that we intentionally skipped the `splitEqs` goals: since these internally-generated facts tend to generate a huge amount of subcases during the solution of lemmas within this theory, skipping to the second goal according to the smart heuristic rank allows to find contradictions quickier and finish the proofs in fewer steps.

5.4.8 Checking the protocol

To check that the previously introduced rules are able to model at least a complete run of the protocol, we wrote a simple `exists-trace` lemma:

```
lemma SanityCheck :
    exists-trace
        "Ex #i #j sender node message .
            MessageSent(sender, node, message) @ #i &
            MessageStored(node, message) @ #j &
            #i < #j"
```

Similarly to the `SanityCheck` introduced in subsection 4.3.7, to prove this property Tamarin needs to find just one correct constraint system that satisfies the formula.

5.5 Security Properties

In this section we have decided to undertake the verification of three major security goals: message secrecy, relationship anonymity and, finally, sender IP anonymity.

5.5.1 Message Secrecy

Like the homonymous lemma presented in the previous chapter (see 4.4.1), we express message secrecy as a trace property. Its specification is:

```
lemma Secrecy:
    "not (
        Ex sender node message #i #j #k .
            MessageSent(sender, node, message) @ #i &
            MessageStored(node, message) @ #j &
            K(message) @ #k &
            not(Ex #d . Dishonest(node) @ #d)
        )"
```

Intuitively, we can read this proposition as ”an attacker can deduce the content of a message sent by a client only if it compromised the receiving node for that message”.

5.5.2 Relationship Anonymity

Relationship anonymity can be expressed as: "it is impossible for the attacker to deduce both ends of a communication". This property is easily formalized through the following rule:

```
lemma RelationshipAnonymity :
    "not (
        Ex sender node message #i #j #k .
            MessageSent(sender, node, message) @ #i &
            MessageStored(node, message) @ #j &
            K(sender) @ #k & K(node) @ #k
        )"
```

Unlike the other properties in this section, Tamarin's smart heuristic is able to prove the aforementioned theorem within seconds without the help of any oracle.

Note that this lemma assures us that even if the receiving node was malevolent (there is no restriction that imposes the last node to be honest), the attacker would not be able to recover the sender's IP address.

5.5.3 Sender Anonymity

Sender anonymity was listed as a security goal in the now-outdated Session's whitepaper [54]. They claim that only the first hop in an onion path is able to identify the sender's IP: considering our formalization, this is can be expressed by stating that if the attacker knows the sender's IP, it must have compromised the first node in a path.

```
lemma SenderAnonymity :
    "not (
        Ex #x #y sender .
            Sender(sender) @ #x &
            K(sender) @ #y &
            (not Ex #w #z guard proxy receiver .
                MessageSentThroughPath(sender, guard, proxy, receiver) @ #w &
                Dishonest(guard) @ #z)
        )"
```

Note that demonstrating this lemma does not actually prove anonymity in the grand scheme of the whole Session protocol: whereas IP-hiding is crucial for making traffic analysis harder, identity anonymity is proved from the more abstract E2E protocol analyzed in the previous chapter (see 4.4.4).

Anyway, this result is still relevant since, to our knowledge, it is the only current symbolic verification of an onion routed network.

5.5.4 Proof results

During the verification of the security properties presented within this chapter we benchmarked Tamarin with the same setup introduced in section 4.4.5. A brief summary of the results is presented in the following table.

As we could hypothesize, the unsolved partial deconstructions complicated the proof of `all-traces` lemmas: since Tamarin was unable to exclude multiple sources of data for each `ModelIn()` fact (due to the

			Duration [ms]				Peak RAM usage [kB]			
Lemma	# of steps	Oracle	Mean	Max	Min	Deviation	Mean	Max	Min	Deviation
SanityCheck	8	✓	389612	3001700	33860	883067	381534	395808	348672	13071
Secrecy	9515	✓	279938	295960	236920	17978	952947	978272	919872	19028
RelationshipAnonymity	734	✗	99208	945700	34100	206019	385370	398896	371248	7672
SenderAnonymity	115	✓	254917	2033270	33770	514803	384778	398848	357920	9775

Table 5.1: Tamarin's performances during the verification of the Onion Routing properties

presence of both the `ModelInput` and `DishonestModelInput` rules) during the precomputation phase, it had to include multiple subcases for each intermediate goal, thus generating very large demonstrations.

As we can see, while RAM usage remains relatively constant, time duration is characterized by a much higher standard deviation: this behaviour is mainly caused by hardware limitations and thermal throttling occoured only during a few executions (see the maxium time for both the sanity check and the sender anonymity properties in compared to the relative averages).

5.6 Future changes

According to their whitepaper [54], Session elaborated a future project for improving the anonymity behind onion routing: *MixMode*. This idea comes from the combination of two existing systems: *Bitmessage* and *Mix Networks*:

- Bitmessage [76] is a decentralized peer-to-peer protocol that mimics Bitcoin's transaction and block transfer system: each user has to both compute a proof-of-work before sending and forward messages on a best-effort basis. This flooding strategy, along with the fact that each message is sent without a destination address (receivers would recognise their messages by the fact that they would be able to decrypt them with their private keys) would make traffic analysis very difficult.

- Mix Networks are messaging systems made up of distributed nodes, called *Mixers*, that wait for a certain threshold of received ciphertexts to be surpassed before forwarding messages to the next hops. They dispatch their message queues in batches of messages, thus making traffic analysis harder since an attacker would have to trace back entire batches of data to follow the path of a single message. Mix Networks have already been studied from a security standpoint and have been proved useful to ensure anonymity properties [66].

Other precautions will be forcing idle clients to send dummy messages with no data to obfuscate real traffic and servers to answer to message retrieval queries with a fixed-sized response. Finally an information-theoretic *Private Information Retrieval* [31] scheme would be used to ensure anonymity while message retrieving.

According to the aforementioned whitepaper, MixMode will be able to protect users from global passive attackers and ensure that in a pool of N nodes queried for a private information in which the malevolent agent controls at most $N - 1$ servers, the privacy of the request will be guaranteed.

These proposals are surely interesting from a security-research perspective, but as today there is no official protocol to verify, so we will leave their verification for future work.

6

Conclusions

We firstly introduced the symbolic model, along with an automatic prover compliant to the Dolev Yao model's specification, Tamarin. Then we have given an overview of Session, an innovative messaging app that aims at improving digital communications by providing anonymity to peer-to-peer conversations. In the subsequent chapters, we reverse-engineered Session's protocol and analyzed its security properties by splitting it into two distinct sub-protocols and formalizing them using Tamarin. Our approach allowed us to demonstrate that, within some sensible restrictions, Session is able to guarantee message secrecy, authenticity and anti-replayability. The verification of these properties, along with the assurance of a correct implementation of the protocol provided by their code audit [28], should give confidence to their userbase that the platform is, at least, secure.

6.1 Unresolved issues

On the other hand, unfortunately, our anonymity verification did not provide all the expected results, thus the proof of this property as an observational equivalence is left as future work. Since Tamarin is not the only tool that allows for the verification of this kind of lemma [56], formalizing the E2E protocol within a prover based on a paradigm different from rule-rewriting may help in solving non-termination.

Similarly, during our in-depth study of the onion-routing protocol we were forced to restrict the possible execution traces by upper-bounding the number of hops for a certain message. Verifying the onion routing properties without this additional limitation may (or may not) provide a sounder proof for this theory. We were not able to do so because we needed multiple variants of the `ModelInput` and `ModelOutput` rules in order to formalize the attacker capabilities; thus, also in this case, modeling the protocol within a different specification language may lead to more conclusive results.

6.2 Future works

As explained in section 5.6, the Oxen Privacy Tech Foundation has published some ideas for future implementations of the onion routing protocol. These additional precautions may contibute to the platforms's overall anonymity, and proving their correctness may lead to new standards for more-secure onion routing protocols in general.

On top of that, Session currently provides two types of group chats: private groups and opengroups. This dissertation does not cover either of those, but could be an helpful reference for the analysis of the shared network-related phases (such as Network and Swarm discovery) and the reverse-engineering process in the case of a future study about the security properties guaranteed by their implementation of this feature.

A

crypto_box_seal for E2E encryption

Session low level encryption is provided by libsodium's library [8]. It allows to generate ephemeral $X25519$ keys, execute ECDH exchanges, sign messages via public key cryptography, encrypt data through multiple algorithms and much more. As we saw in sections 4.1.1 4.1.2, this library is responsible for the E2E encryption and decryption of the messages, thus we decided to take a closer look at the cryptographic scheme implemented. For further reference, refer to the official documentation [7] [17] and C-implementation repository [7]. Finally, note that Session's desktop app utilizes a Javascript wrapper (`libsodium-wrappers-sumo`, available at [9]) for the actual implementation, so some API calls will have slightly different signatures from the ones we will present in this appendix due to the additional functionality provided by higher-level languages.

A.1 Sealed Boxes

The main concept behind the encryption scheme is the *sealed box*. As explained in their relevant documentation [17],

> ⟨⟨Sealed boxes are designed to anonymously send messages to a recipient given their public key. Only the recipient can decrypt these messages using their private key. While the recipient can verify the integrity of the message, they cannot verify the identity of the sender.⟩⟩

Reading further in the documentation, we can see that a message m is encrypted as

$$\text{ephemeral\_pk} \;||\; \text{box}(m, \text{recipient\_pk}, \text{ephemeral\_sk}, \text{nonce} = \text{blake2b}(\text{ephemeral\_pk} \;||\; \text{recipient\_pk}))$$

and that the algorithms implemented are $X25519$ Diffie Hellman exchange and *XSalsa20-Poly1305* encryption. To unambiguously understand how the data is actually encrypted via the `box` function, we reverse-engineered the source code available at their public repository (note that originally, `crypto_box()` was implemented in the *NaCl* library [18], from which libsodium was initially forked).

A.1.1 Encryption procedure

To encrypt a payload into a sealed box, we can use the function `crypto_box_seal(c, m, mlen, pk)`, in which

- `c` is an unsigned char pointer that references the buffer to overwrite with the encrypted data; The buffer has to be of at least `crypto_box_seal_SEALBYTES + mlen` bytes.
- `m` is an unsigned char pointer referencing the data to encrypt.
- `mlen` is an unsigned long long that counts the significant bytes of `m`.
- `pk` is an unsigned char pointer referencing the recipient's public $X25519$ key.

```
1  int crypto_box_seal(unsigned char *c, const unsigned char *m,
2                      unsigned long long mlen, const unsigned char *pk)
3  {
4      unsigned char nonce[crypto_box_NONCEBYTES];
5      unsigned char epk[crypto_box_PUBLICKEYBYTES];
6      unsigned char esk[crypto_box_SECRETKEYBYTES];
7      int           ret;
8
9      // 1. ) Generate an ephemeral X25519 keypair and store it in buffers eps and esk
10     if (crypto_box_keypair(epk, esk) != 0) {
11         return -1;
12     }
13
14     // 2. ) Hash epk and pk and save the hash in buffer nonce
15     _crypto_box_seal_nonce(nonce, epk, pk);
16
17     // 3. ) Encrypt the plaintext. Note that an initial empty space is left at the head of the ciphertext buffer
18     ret = crypto_box_easy(c + crypto_box_PUBLICKEYBYTES, m, mlen,
19                           nonce, pk, esk);
20
21     // 4. ) Copy the ephemeral public key at the head of the ciphertext buffer
22     memcpy(c, epk, crypto_box_PUBLICKEYBYTES);
23
24     // 5. ) Delete the ephemeral keypair, along with the nonce
25     sodium_memzero(esk, sizeof esk);
26     sodium_memzero(epk, sizeof epk);
27     sodium_memzero(nonce, sizeof nonce);
28
29     return ret;
30 }
```

As previously anticipated, `crypto_box_keypair()` generates a fresh, random $X25519$ private key and then computes its public counterpart. `_crypto_box_seal_nonce(nonce, epk, pk)` concatenates `epk` and `pk` and hashes them through the *blake2* algorithm [51], which has been shown in literature to be more efficient than *SHA-256* in hashing single plaintexts [57].

The actual encryption is obtained through a series of function invocations that begins via `crypto_box_easy()` (which documentation is available at [3]). To highlight all the main steps without sacrificing brevity, we'll only show some key functions that are called, excluding all the error-handling code.

```
1  int crypto_box_detached(unsigned char *c, unsigned char *mac,
```

```
                    const unsigned char *m, unsigned long long mlen,
                    const unsigned char *n, const unsigned char *pk,
                    const unsigned char *sk)
{
    unsigned char k[crypto_box_BEFORENMBYTES];
    int           ret;

    // 1. ) Execute the ECDH exchange between pk and esk and store the result in k
    if (crypto_box_beforenm(k, pk, sk) != 0) {
        return -1;
    }

    // 2. ) Continue with the encryption
    ret = crypto_box_detached_afternm(c, mac, m, mlen, n, k);

    // 3. ) Overwrite the symmetric key k
    sodium_memzero(k, sizeof k);

    return ret;
}
```

In this code listing we can see that the key `k` is derived directly from the receiver's public $X25519$ key and the ephemeral secret generated previously. After the encryption, the symmetrical key obtained is also erased from memory, thus making it virtually impossible for a sender to open his own sealed boxes. Note also that we have now the additional parameter `mac`, which is a pointer to a buffer that will contain the *Message Authentication Code*. Since this buffer makes up the initial space of `c`, just after the memory left for the ephemeral public key, the complete message will have in the following form:

$$\text{sealed\_box} = \text{ephemeral\_pubKey} \;||\; MAC \;||\; \text{ciphertext}$$

Finally, after a few more steps, we come to the actual encryption-handling procedure:

```
int crypto_secretbox_detached(
    unsigned char *c,               // ciphertext buffer
    unsigned char *mac,             // MAC buffer
    const unsigned char *m,         // plaintext buffer
    unsigned long long mlen,        // length of m
    const unsigned char *n,         // nonce buffer
    const unsigned char *k          // symmetric key buffer
)
{
    crypto_onetimeauth_poly1305_state state;
    unsigned char                     block0[64U];
    unsigned char                     subkey[crypto_stream_salsa20_KEYBYTES];
    unsigned long long                i;
    unsigned long long                mlen0;

    // 1. ) Obtain the pseudo-random subsequence from the nonce and symmetric key
    crypto_core_hsalsa20(subkey, n, k, NULL);

    // Allow the m and c buffers to partially overlap, by calling memmove() if necessary.
    if (((uintptr_t) c > (uintptr_t) m &&
         (uintptr_t) c - (uintptr_t) m < mlen) ||
        ((uintptr_t) m > (uintptr_t) c &&
```

```
23          (uintptr_t) m - (uintptr_t) c < mlen)) {
24          memmove(c, m, mlen);
25          m = c;
26      }
27
28      memset(block0, 0U, crypto_secretbox_ZEROBYTES);
29      COMPILER_ASSERT(64U >= crypto_secretbox_ZEROBYTES);
30      mlen0 = mlen;
31
32      if (mlen0 > 64U - crypto_secretbox_ZEROBYTES) {
33          mlen0 = 64U - crypto_secretbox_ZEROBYTES;
34      }
35
36      // 2. ) Copy the first part of the message into block0
37      for (i = 0U; i < mlen0; i++) {
38          block0[i + crypto_secretbox_ZEROBYTES] = m[i];
39      }
40
41      // 3. ) Execute the XSalsa20 algorithm on block0 inplace using the pseudorandom sequence
42      crypto_stream_salsa20_xor(block0, block0, 64U, n + 16, subkey);
43      COMPILER_ASSERT(crypto_secretbox_ZEROBYTES >=
44                      crypto_onetimeauth_poly1305_KEYBYTES);
45
46      // 4. ) Initialize the creation of the MAC
47      crypto_onetimeauth_poly1305_init(&state, block0);
48
49      // 5. ) Copy the encrypted ciphertext into the intended buffer
50      for (i = 0U; i < mlen0; i++) {
51          c[i] = block0[crypto_secretbox_ZEROBYTES + i];
52      }
53
54      // 6. ) Reset block0 and repeat the procedure if there is more plaintext to encrypt
55      sodium_memzero(block0, sizeof block0);
56      if (mlen > mlen0) {
57          crypto_stream_salsa20_xor_ic(c + mlen0, m + mlen0, mlen - mlen0,
58                                       n + 16, 1U, subkey);
59      }
60
61      // 7. ) Erase the pseudo-random subsequence
62      sodium_memzero(subkey, sizeof subkey);
63
64      // 8. ) Compute the MAC and save it into the correct buffer
65      crypto_onetimeauth_poly1305_update(&state, c, mlen);
66      crypto_onetimeauth_poly1305_final(&state, mac);
67
68      // 9. ) Erase the MAC state
69      sodium_memzero(&state, sizeof state);
70
71      return 0;
72  }
```

Through the `crypto_core_hsalsa20()` function, the nonce and the symmetric key previously obtained through the ECDH exchange are combined into a pseudo-random bit sequence by applying XOR and rotation operations. This sequence will be subsequently used to encrypt the plaintext through an implementation of the *XSalsa20* stream cipher [34]. Along with the encryption procedure, in this

function we can also see that the *MAC* is produced and stored in the buffer pointed by `mac` through an execution of the *Poly1305* algorithm [33].

Having completed the reverse engineering of the `crypto_box_seal()` API, we can summarize the encryption scheme as:

$$\text{nonce} = h(\text{ephemeral\_pubKey} \mid\mid pub\_X25519_{Bob}) \tag{A.1}$$

$$\text{encKey} = \text{nonce} \oplus symKey \tag{A.2}$$

$$\text{ciphertext} = \text{plaintext} \oplus \text{encKey} \tag{A.3}$$

$$\text{sealed\_box} = \text{ephemeral\_pubKey} \mid\mid MAC \mid\mid \text{ciphertext} \tag{A.4}$$

As explained in Session's code audit [28], this scheme allows to avoid the use of the same encryption key every time, thus ensuring that even if an attacker manages to bruteforce the symmetric key of a sealed box, no correlation with other message sent can be found.

A.1.2 Encryption scheme simplification

Fortunately, Tamarin comes with all the needed default primitives to formalize the scheme presented in the previous chapter: having a built-in XOR equational theory, along with default pair construction and destruction operators, we are able to trivially model equations A.1, A.2, A.3 and A.4. The problem with this formalization is that during the pre-computation phase Tamarin identifies multiple possible term sources for the arguments of each application of the XOR operation, thus overly-complicating the proofs after. Since we are reasoning within the symbolic model, where the adopted term algebra guarantees that encrypted messages ca not be altered without the relevant key, we decided to simplify the encryption scheme as

$$\begin{aligned}
\text{nonce} &= h(\text{ephemeral_pubKey} \mid\mid pub_X25519_{Bob}) \\
\text{encKey} &= \text{nonce} \mid\mid symKey \\
\text{ciphertext} &= E_{\text{encKey}}(\text{plaintext}) \\
\text{sealed_box} &= \text{ephemeral_pubKey} \mid\mid \text{ciphertext}
\end{aligned}$$

Let us explain why these changes do not compromise the soundness of the proofs:

- substituting the XOR operator with a pair constructor in equation A.2 actually allows for less-restrictive assumptions: while both the $\oplus$ and $\mid\mid$ operators allow to construct a unified term by knowing two arguments A_1 and A_2 ($A_1 \oplus A_2$ and $A_1 \mid\mid A_2$), we (and thus the attacker) cannot deduce any information about a xor-ed term without knowing at least one of its argument. On the other end, by deducing a couple $A_1 \mid\mid A_2$ from a constraint system, we are able to coerce both A_1 and A_2. As a consequence, we can state that by replacing the XOR operation with the pair constructor we do not exclude any possible rightful deduction;

- since a computational analysis of the XSalsa20 cipher would be out of the scope of this thesis (and has already been done in [69] and [44]), we abstracted the encryption procedure by substituing the XOR operation of equation A.3 with a symmetric encryption;
- removing the Message Authentication Code from equation A.4 does not affect our formalization since, within the Dolev-Yao model, an attacker would not be able to modify part of an encrypted term without the relevant key anyway.

Bibliography

[1] 15000 oxen in eur according to cryptocurrency724. https://www.cryptocurrency724.com/convert-15000-loki-to-eur.html. Accessed: 2022-08-04.

[2] Antox. https://play.google.com/store/apps/details?id=chat.tox.antox&hl=it&gl=US. Accessed: 2022-08-04.

[3] Authenticated encryption. https://libsodium.gitbook.io/doc/public-key_cryptography/authenticated_encryption. Accessed: 2022-08-26.

[4] Behind the scenes: Session network and client update. https://getsession.org/blog/session-network-and-client-update. Accessed: 2022-08-04.

[5] Ed25519 to curve25519. https://libsodium.gitbook.io/doc/advanced/ed25519-curve25519. Accessed: 2022-04-26.

[6] Initial commit. https://github.com/irungentoo/toxcore/commit/f8ccb9adb99fc143e11927a461d06da1b3d5bcba. Accessed: 2022-08-04.

[7] Libsodium. https://github.com/jedisct1/libsodium. Accessed: 2022-08-26.

[8] Libsodium documentation. https://libsodium.gitbook.io/doc/. Accessed: 2022-08-26.

[9] libsodium-wrappers-sumo. https://www.npmjs.com/package/libsodium-wrappers-sumo. Accessed: 2022-08-26.

[10] Loki blockchain homepage. https://loki.network/2020/06/01/how-loki-blockchain-powers-session-lokinet-blink/. Accessed: 2022-08-04.

[11] Number of google searches for 'tox protocol' by month. https://trends.google.it/trends/explore?date=2013-06-24%202022-08-07&q=%2Fm%2F0_x5g0g. Accessed: 2022-08-04.

[12] Oxen. https://github.com/oxen-io. Accessed: 2022-04-06.

[13] Oxen docs. https://docs.oxen.io. Accessed: 2022-08-04.

[14] Oxen privacy tech foundation's website. https://optf.ngo. Accessed: 2022-08-04.

[15] Oxen service nodes. https://docs.oxen.io/about-the-oxen-blockchain/oxen-service-nodes. Accessed: 2022-08-04.

[16] oxen-storage-server. https://github.com/oxen-io/oxen-storage-server. Accessed: 2022-04-30.

[17] Sealed boxes. https://libsodium.gitbook.io/doc/public-key_cryptography/sealed_boxes. Accessed: 2022-08-26.

[18] Secretkey authenticated encryption: crypto_secretbox. https://nacl.cr.yp.to/secretbox.html. Accessed: 2022-08-26.

[19] Session - private messenger. https://play.google.com/store/apps/details?id=network.loki.messenger&gl=IT. Accessed: 2022-07-26.

[20] session-desktop. https://github.com/oxen-io/session-desktop. Accessed: 2022-08-17, last commit code: 67153bbb0782d825d51cbc6f395dc2a5ca22ac2b.

[21] Session protocol: Technical implementation details. https://getsession.org/blog/session-protocol-technical-information. Accessed: 2022-04-06.

[22] The session protocol: What's changing - and why. https://getsession.org/blog/session-protocol-explained. Accessed: 2022-04-06.

[23] Session's blog. https://getsession.org/blog. Accessed: 2022-07-26.

[24] Session's website. https://getsession.org. Accessed: 2022-07-26.

[25] Tox advanced faq. https://tox.chat/faq.html#techfaq. Accessed: 2022-08-04.

[26] Tox faq. https://tox.chat/faq.html. Accessed: 2022-08-04.

[27] The tox project. https://tox.chat/about.html. Accessed: 2022-08-04.

[28] Oxen session audit. Technical report, Quarkslab, Paris, France, Apr 2020.

[29] K. Andy. Fbi document says the feds can get your whatsapp data — in real time. *Rolling Stone*.

[30] A. Armando and L. Compagna. Satmc: A sat-based model checker for security protocols. volume 3229, pages 730–733, 09 2004.

[31] E. Kushilevitz B. Chor, O. Goldreich and M. Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50, 1995.

[32] C. Cremers B. Schmidt, S. Meier and D. Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 78–94, 2012.

[33] D.J. Bernstein. The poly1305-aes message-authentication code. In *Proceedings of the 12th International Conference on Fast Software Encryption*, FSE'05, pages 32–49. Springer-Verlag, 2005.

[34] D.J. Bernstein. *The Salsa20 family of stream ciphers*, pages 84–97. Lecture Notes in Computer Science. Springer, 2008.

[35] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 1:1–135, 10 2016.

[36] B. Bruno. Security protocol verification: Symbolic and computational models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, pages 3–29, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[37] J. Hoyland S. Scott C. Cremers, M. Horvat and T. van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery.

[38] J. Schrader N. Huaman Y. Acar A. L. Fehlhaber M. Wei B. Ur C. Stransky, D. Wermke and S. Fahl. On the limited impact of visualizing encryption: Perceptions of E2E messaging security. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 437–454. USENIX Association, Aug 2021.

[39] V. Cortier and S. Delaune. A method for proving observational equivalence. pages 266–276, 07 2009.

[40] C. Cremers and D. Jackson. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using diffie-hellman. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 78–7815, 2019.

[41] J. Dreier D. Basin and R. Sasse. Automated symbolic proofs of observational equivalence. pages 1144–1155, 10 2015.

[42] S. Mödersheim D. Basin and L. Vigano. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4:181–208, 01 2005.

[43] M. Debbabi and M. Rahman. The war of presence and instant messaging: right protocols and apis. pages 341 – 346, 02 2004.

[44] K. Deepthi and K. Singh. *Cryptanalysis of Salsa and ChaCha: Revisited*, pages 324–338. 05 2018.

[45] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[46] J. Douceur. The sybil attack. pages 251–260, 01 2002.

[47] M. Torgersen G. Bierman, M. Abadi. Understanding typescript. volume 8586, pages 257–281, 07 2014.

[48] GNU. time(1) - linux man page. `https://linux.die.net/man/1/time`.

[49] S. Coretti J. Alwen and Y. Yevgeniy. *The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol*, pages 129–158. 04 2019.

[50] S. Radomirovic J. Dreier, L. Hirschi and R. Sasse. Automated unbounded verification of stateful cryptographic protocols with exclusive or. pages 359–373, 07 2018.

[51] Z. Wilcox-O'Hearn J. P. Aumasson, S. Neves and C. Winnerlein. Blake2: simpler, smaller, fast as md5. pages 119–135, 06 2013.

[52] D. Jackson. Dennis jackson's public google group post. https://groups.google.com/g/tamarin-prover/c/zLVd-nD3j3g/m/jTX6_SCcAQAJ, May 2020. Accessed: 2022-05-05.

[53] B. Dowling-L. Garratt K. Cohn-Gordon, C. Cremers and D. Stebila. A formal security analysis of the signal messaging protocol. pages 451–466, 04 2017.

[54] M. Shishmarev K. Jefferys and S. Harman. Session: A model for end-to-end encrypted conversations with minimal metadata leakage. Technical report, Mar 2020.

[55] P. Lafourcade and M. Puys. Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In *8th International Symposium on Foundations and Practice of Security 8th International Symposium, FPS 2015*, pages 137–155, Clermont-Ferrand, France, October 2015. Springer.

[56] K. Bhargavan B. Blanchet C. Cremers K. Liao M. Barbosa, G. Barthe and B. Parno. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Paper 2019/1393, 2019.

[57] M. M. Karagöz M. M. Özcan, B. A. Ayaz and E. Yolaçan. Performance evaluation of sha-256 and blake2b in proof of work architecture. *Eskişehir Türk Dünyasi Uygulama ve Araştirma Merkezi Bilişim Dergisi*, 3(2):60 – 65, 2022.

[58] M. Mannan and P. Oorschot. Secure public instant messaging: A survey. 01 2004.

[59] S. Meier. *Advancing automated security protocol verification.* PhD thesis, ETH, 2013.

[60] M. Miculan and N. Vitacolonna. Automated symbolic verification of telegram's mtproto 2.0. *CoRR*, abs/2012.03141, 2020.

[61] P. Lincoln N. Durgin and J. Mitchell. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12:247–311, 02 2004.

[62] K. Bhargavan N. Kobeissi and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. pages 435–450, 04 2017.

[63] S. Mödersheim O. Almousa and L. Vigano. *Alice and Bob: Reconciling Formal Models and Implementation*, volume 9465, pages 66–85. 11 2015.

[64] N. Mathewson R. Dingledine and P. Syverson. Tor: The second-generation onion router. *Paul Syverson*, 13, Jun 2004.

[65] C. Cremers S. Meier, B. Schmidt and D. Basin. The tamarin prover for the symbolic analysis of security protocols. volume 8044, pages 696–701, 07 2013.

[66] K. Sampigethaya and R. Poovendran. A survey on mix networks and their secure applications. *Proceedings of the IEEE*, 94(12):2142–2181, 2006.

[67] B. Schmidt. *Formal analysis of key exchange protocols and physical protocols.* PhD thesis, ETH", 2012.

[68] M. Schmidt-Schauss. Unification in permutative equational theories is undecidable. *Journal of Symbolic Computation*, 8(4):415–421, 1989.

[69] M. Subhamoy, P. Goutam, and M. Willi. Salsa20 cryptanalysis: New moves and revisiting old styles. Cryptology ePrint Archive, Paper 2015/217, 2015.

[70] The Session Team. Session's lightpaper. Technical report, Oxen Privacy Tech Foundation, 2020.

[71] The Tamarin Team. *Tamarin-Prover Manual: Security Protocol Analysis in the Symbolic Model*, 2022.

[72] M. Turuani. The cl-atse protocol analyser. volume 4098, pages 277–286, 08 2006.

[73] S. Delaune V. Cortier and J. Dreier. *Automatic Generation of Sources Lemmas in Tamarin: Towards Automatic Proofs of Security Protocols*, pages 3–22. 09 2020.

[74] F. Valsorda. Using ed25519 signing keys for encryption. `https://words.filippo.io/using-ed25519-keys-for-encryption/`, May 2019. Accessed: 2022-04-26.

[75] N. Vitacolonna. Symbolic verification of security protocols with tamarin. `http://users.dimi.uniud.it/~angelo.montanari/SymbolicVerificationWithTamarin.pdf`, May 2021. Accessed: 2022-07-30.

[76] J. Warren. Bitmessage: A peer-to-peer message authentication and delivery system. `http://kevinriggen.com/files/bitmessage.pdf`, Nov 2012. Accessed: 2022-06-01.