Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cryptography in the Wild: Briar

Semester Project

Yuanming Song

March 5, 2023

Advisor: Prof. Dr. Kenny Paterson

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Briar is a secure peer-to-peer messaging application for higher-risk users. Briar users communicate directly by exchanging encrypted messages via Bluetooth, WiFi, Tor, etc. Briar can also synchronize messages among private group members or forum/blog subscribers via encrypted peer-to-peer communication.

The Briar team claims that Briar meets very high security standards, including confidentiality of message metadata and content, forward security, and resistance to denial-of-service attacks. To this end, they designed and implemented a set of protocols tailored to delay-tolerant peer-to-peer communication for Briar. However, there is little attention on whether these protocols are actually secure as claimed.

In this project, we examine Briar's use of cryptography and analyze the security of protocols in Briar. Our main contributions are as follows:

1. We provide a detailed documentation of Briar's protocol stack;
2. We give a series of arguments in favor of Briar's security by checking against possible attacks and performing formal verification. The overall positive results partly support Briar's security claims;
3. We discover three new vulnerabilities in Briar's protocol design and implementaion, and suggest possible fixes. The vulnerabilites are: 1) Bramble Handshake Protocol (BHP) is not forward secure, 2) the implementation of Bramble Synchronisation Protocol (BSP) is susceptible to denial-of-service attacks from malicious contacts, and 3) a malicious insider of a private group, forum, or blog can duplicate messages from any member as many times as desired. In addition, we identify the issue that the lack of out-of-band verification could lead to active man-in-the-middle attacks on introducing contacts in Briar. This issue was already known to Briar.

Contents

Contents **iii**

1 Introduction **1**

2 Preliminaries **5**
2.1 Strings and byte arrays 5
2.2 Encoding and decoding 5
2.3 Cryptographic primitives 6
2.4 Binary Data Format (BDF) 6

3 Overview of Briar **9**
3.1 Who's who in Briar 9
3.2 Protocol stack overview 10
3.3 Account creation and adding contact 11
3.4 Communication 11

4 Bramble QR Code Protocol (BQP) **13**
4.1 Constants 13
4.2 Preparation and connection establishment 13
4.3 Key agreement 14
4.4 Notes 16

5 Bramble Handshake Protocol (BHP) **17**
5.1 The protocol 17
5.2 Lack of forward security 18
5.3 Notes 19

6 Contact Exchange **21**
6.1 The protocol 21
6.2 Notes 22

7 Bramble Transport Protocol (BTP) 25
7.1 Key management protocol 25
7.1.1 Constants 25
7.1.2 Handshake mode 25
7.1.3 Rotation mode 26
7.2 Wire protocol 27
7.2.1 Stream and tag 28
7.2.2 Stream header 28
7.2.3 Frame 29
7.3 Notes 30

8 Bramble Synchronisation Protocol (BSP) 33
8.1 Concepts 33
8.2 Denial-of-service attack 34
8.3 Notes 35

9 Introduction Client 37
9.1 The protocol 37
9.2 Active man-in-the-middle attack 40
9.3 Notes 40

10 Briar Functionalities 43
10.1 Transport key agreement client 43
10.2 Messaging 44
10.3 Private groups, forums, and blogs 45
10.3.1 Message duplication attack 45
10.3.2 Notes 47

11 Discussion 49

12 Conclusion 51

A Tamarin code 53
A.1 Tamarin code for BQP 53
A.2 Tamarin code for BHP 54
A.3 Tamarin code for Briar introduction 57

B Code examples for message duplication 61

Bibliography 63

Chapter 1

Introduction

Briar[1] is a secure peer-to-peer messaging application designed to protect journalists, protestors and other higher risk groups of users against censorship and surveillance. In Briar, peers communicate directly by exchanging encrypted messages via a range of transports, including Bluetooth, WiFi, and Tor.[2] Briar also supports private groups, forums, and blogs, where it synchronizes messages among members or subscribers via encrypted peer-to-peer communication [14]. Notably, Briar does not adopt existing key agreement or communication protocols like TLS or the Signal protocol,[3] but instead develops a set of cryptographic protocols tailored for delay-tolerant peer-to-peer communication, including BQP (Chapter 4), BHP (Chapter 5), and BTP (Chapter 7).

As a messaging application designed against potentially powerful adversaries (e.g., nation-states), Briar claims to achieve a set of very strong security standards. In Briar, the traffic between peers is end-to-end encrypted and should be resistant against eavesdropping, tampering, and traffic analysis. Briar also claims to offer forward security for both bidirectional and unidirectional communications [10]. Finally, the decentralized nature of Briar should make it resistant to takedown orders, denial-of-service attacks, and Internet blackouts [14]. These claims, if true, would mean that Briar offers a higher level of security than almost all existing messaging applications.

The high risk of Briar's target user groups, the use of custom cryptographic protocols and the strong claims of security all motivate a thorough examination on the cryptography used in Briar. Furthermore, recent attacks [1, 3] on Bridgefy,[4] a mesh messaging application that shares many similarities to Briar, shows the importance of independent security assessments of Briar's

[1] https://briarproject.org/
[2] https://www.torproject.org/
[3] https://signal.org/docs/
[4] https://bridgefy.me/

cryptography. However, to our knowledge, the only public security analysis of Briar up to now is a security audit performed by Heiderich et al. from Cure53 [4], which was conducted around six years ago on a development version of Briar, covering only a subset of Briar's cryptographic protocols currently in use. Heiderich et al. also suggested in their report that performing another audit on Briar after its release was "highly advisable" [4].

We examined Briar's use of cryptography and Briar's protocols over a period of approximately five months as a semester project. The object of our security analysis is Briar for Android; Briar Mailbox[5] and Briar Desktop[6] are outside the scope of our analysis, as they are still under active development. However, the issues we discovered would all apply to Briar Desktop. Our documentation and findings in this report are valid for Briar 1.4.20,[7] released on 01-30-2023; for convenience, we refer to them in the present tense even if they may no longer be valid for newer versions of Briar.

The attack scenarios we consider in our analysis are 1) network attackers within the Dolev-Yao model [5], who have full control of the communication channel between Briar users, 2) loss of forward security, where attackers with compromised session keys or long-term secrets try to break security of past sessions, and 3) malicious insiders or contacts. Note that we did not consider compelled access or loss of post-compromise security, because Briar does not claim to be resistant to these attacks [11].

We manually reviewed Briar's specifications[8] and source code,[9] all of which are publicly available thanks to the Briar team. We also used the Tamarin prover,[10] a tool for automated protocol verification, to analyze selected protocols in Briar. In addition, we experimented on test devices to understand better Briar's inner workings and verify our attacks; no Briar users in real life were affected by our experiments.

In this report, we document in detail Briar's use of cryptography and Briar's protocol stack. Our report could serve as a reference for people interested in how Briar works to offer secure communication.

Our analysis shows that Briar generally follows good cryptography practices in its design and implementation; as a result, Briar is resistant to a number of attacks we have envisaged, which partly validates Briar's security claims. We present in this report a series of arguments in favor of Briar's security, including failed attacks and formal verification results.

[5] https://code.briarproject.org/briar/briar-mailbox/
[6] https://code.briarproject.org/briar/briar-desktop/
[7] https://code.briarproject.org/briar/briar/-/commits/release-1.4.20
[8] https://code.briarproject.org/briar/briar-spec/
[9] https://code.briarproject.org/briar/briar/
[10] http://tamarin-prover.github.io/

Nevertheless, we were able to discover four issues in Briar's protocol design and implementation that could undermine the security of Briar users:

1. Bramble Handshake Protocol (BHP) does not perform Diffie-Hellman key exchange between the ephemeral keys of the peers, and therefore fails to provide forward security when the handshake secret keys of both parties are compromised (Section 5.2);
2. The lack of out-of-band authentication gives a way for a malicious introducer to perform an active man-in-the-middle attack on the introduction of new contacts. The developers of Briar had already noticed this issue before our discovery but have not yet fixed it (Section 9.2);
3. The implementation of Bramble Synchronisation Protocol (BSP) is susceptible to a denial-of-service attack, where a victim's remote contact can cause the victim's application to crash repeatedly by sending large messages (Section 8.2);
4. In private groups, forums, and blogs, a malicious insider can duplicate messages (i.e., replay messages without modifying the timestamp) from any member as many times as the adversary desires.

We also give suggestions on how to fix these issues.

We contacted the developers of Briar on 14-02-2023, and notified them about our findings on 17-02-2023, where we suggested a 90-day disclosure period. The developers confirmed receipt and acknowledged the vulnerabilities on 17-02-2023. They fixed issues 3 and 4 in Briar 1.4.22 on 23-02-2023,[11] and plan to fix issue 1 within the 90-day deadline.

Chapter 2 lists some notations and primitives used in Briar protocols for reference. Chapter 3 provides a glossary and an overview of Briar's protocol stack. The subsequent chapters mainly document Bramble QR Code Protocol (BQP) (Chapter 4), Bramble Handshake Protocol (BHP) (Chapter 5), contact exchange (Chapter 6), Briar Transport Protocol (BTP) (Chapter 7), Bramble Synchronisation Protocol (BSP) (Chapter 8), introduction client (Chapter 9), and some Briar functionalities (Chapter 10). Finally, we discuss our findings in Chapter 11, and conclude our report in Chapter 12. We include the Tamarin code for formal verification in Appendix A, and example code snippets for message duplication in Appendix B.

[11] https://code.briarproject.org/briar/briar/-/commits/release-1.4.22

Chapter 2

Preliminaries

This chapter lists some notations and cryptographic primitives to appear in the following chapters.

2.1 Strings and byte arrays

- `"..."`: a string literal (e.g., `"briar"`);
- `[...]`: a byte array (e.g., `[0x62, 0x72, 0x69, 0x61, 0x72]`);
- `len(a)`: the number of bytes in the byte array `a`;
- `a||b`: the concatenation of byte arrays `a` and `b`;
- `a[x]`: the `x`-th byte (0-indexed) in the byte array `a`;
- `a[x:y]`: a byte array `[a[x], a[x+1], ..., a[y]]`; `x` (resp. `y`) can be omitted when it takes the value 0 (resp. `len(a)-1`).

2.2 Encoding and decoding

- `int16(x)`/`int32(x)`/`int64(x)`/`int`k`(x)`: the big-endian encoding of an unsigned 16/32/64/k-bit integer `x`;
- `utf-8(x)`: the UTF-8 encoding of a string `x`;
- `ISO-8859-1-decode(a)`: the ISO 8859-1[1] decoding of a byte array `a`;
- `BDF-encode(x)`/`BDF-decode(a)`: the encoding of a string `x` / decoding of a byte array `a` as specified in Briar's Binary Data Format (BDF) (see Section 2.4).

[1] https://en.wikipedia.org/wiki/ISO/IEC_8859-1

2.3 Cryptographic primitives

- BLAKE2b-256(x): the BLAKE2b-256[2] digest of a byte array x;
- BLAKE2b-256(k, x): the BLAKE2b-256 digest of a byte array x with a 32-byte key k;
- hash(label, $x_1, ..., x_n$): BLAKE2b-256(int32(len(utf-8(label)))‖ utf-8(label)‖int32(len(x_1))‖x_1‖...‖int32(len(x_n))‖x_n);
- mac(label, k, $x_1, ..., x_n$): BLAKE2b-256(k, int32(len(utf-8(label)))‖ utf-8(label)‖int32(len(x_1))‖x_1‖...‖int32(len(x_n))‖x_n);
- KDF(label, k, $x_1, ..., x_n$) : mac(label, k, $x_1, ..., x_n$);
- PRF(k, x) : BLAKE2b-256(k, x);
- ed25519-sign(sk, m): produces an Ed25519[3] signature of a message m with the signing key sk;
- ed25519-verify(sig, vk, m): checks if sig is a valid Ed25519 signature of message m using the verification key vk;
- sign(label, m, sk): ed25519-sign(sk, int32(len(utf-8(label)))‖ utf-8(label)‖int32(len(m))‖m);
- verify(sig, label, m, pk): ed25519-verify(sig, pk, int32(len(utf-8(label)))‖utf-8(label)‖int32(len(m))‖m);
- ENC(k, n, m): encrypts message a m with key k and nonce n using the XSalsa20Poly1305 cipher;[4]
- DEC(k, n, c): decrypts a ciphertext c with key k and nonce n using the XSalsa20Poly1305 cipher;
- keyGen() : generates a Curve25519[5] key pair (pk, sk);
- DH(sk, pk): performs X25519 key exchange, checks in constant time if the result is zero, aborting if so;
- verifyMac(m, label, k, $x_1, ..., x_n$): checks in constant time if m is equal to mac(label, k, $x_1, ..., x_n$).

2.4 Binary Data Format (BDF)

Binary Data Format (BDF) is a data format used in Briar for serialization. It supports the encoding and decoding of integers, floats, strings, byte arrays, as well as lists and dictionaries [6].

[2] https://www.blake2.net/
[3] https://ed25519.cr.yp.to/
[4] https://nacl.cr.yp.to/secretbox.html
[5] https://cr.yp.to/ecdh.html

In BDF, each encoded object starts with a byte that specifies its type, and, for some types, the length or the length-of-length of the encoded object. For example, INT_8 (0x21) indicates an integer of one byte, while INT_16 (0x22) indicates an integer of two bytes; the byte STRING_8 (0x41) indicates a BDF string whose length (after UTF-8 encoding) is represented as an one-byte integer. This byte is followed by an optional length field in case the length-of-length is specified, and then followed by the data of the specified length.

A BDF list starts with a byte LIST (0x60), which does not contain its length or length-of-length. It is followed by BDF encodings of elements in the list, and finally, an END (0x80) byte that represents the end of the list. Similarly, a BDF dictionary starts with a byte DICTIONARY (0x70), which is followed by pairs of keys and values, represented respectively as BDF strings and BDF-encoded objects. It also ends with the byte END (0x80). Both lists and dictionaries support nesting.

Table 2.1 shows the (canonical) BDF encoding of a BDF list that contains two elements, "eth" and 2023.

Table 2.1: The canonical BDF encoding of the BDF list of "eth" and 2023.

Raw	0x70	0x41	0x03	0x65	0x74	0x68	0x22	0x07	0xe7	0x80
Decoded	LIST	STRING_8	3	e	t	h	INT_16	2023		END

BSP clients (messaging, private groups, etc.) typically exchange messages between peers as encrypted BDF lists. In order to mitigate denial-of-sevice atatcks, Briar sets a nesting depth limit of 5 and a maximum buffer size of 64KiB for BDF-encoded messages received. The Briar specification stresses that "if data is to be hashed or signed, integers and lengths should be represented using the minimum number of bytes, and dictionary keys should be unique and sorted in lexicographic order" [6].

One can see that there is a certain level of flexibility in BDF. In fact, it is easy to find different BDF encodings that decode to the same object, provided that at most one of them does not contain invalid UTF-8 encodings and follows the rules described in the BDF specification (i.e., canonical). However, as we will see in Chapter 10, no such checks were in place for incoming messages as of Briar 1.4.20. Possible ways to produce different BDF encodings include leveraging integer overflow, increasing the length-of-length field (e.g., use INT_32 to represent a INT_16 value), and inserting unmappable or malformed bytes (e.g., 0x80) into the UTF-8 encoding of BDF strings (which is ignored by the decoder).

Chapter 3

Overview of Briar

3.1 Who's who in Briar

For readers' reference, we compiled a list of terms and their meanings in Briar from the Briar Wiki [13].

- **Binary Data Format (BDF):** A data format used in Briar for data serialization (Section 2.4);
- **Bramble:** The underlying framework of Briar for building decentralized applications;
- **Bramble Handshake Protocol (BHP):** A key agreement protocol in Briar that allows two peers having exchanged their handshake public keys to establish a shared secret key remotely (Chapter 5);
- **Bramble QR Code Protocol (BQP):** A key agreement protocol in Briar that allows two peers to establish an ephemeral shared key locally by scanning each other's QR code (Chapter 4);
- **Bramble Rendezvous Protocol (BRP):** A discovery protocol in Briar that allows two peers having exchanged their public keys to connect to each other in Tor;
- **Bramble Synchronisation Protocol (BSP):** An application layer data synchronization protocol in Briar that allows members of the same group to synchronize messages over delay-tolerant networks (Chapter 8);
- **BSP Client:** An application component in Briar that uses BSP to synchronize data between users;
- **Bramble Transport Protocol (BTP):** A transport layer security protocol in Briar (Chapter 7);

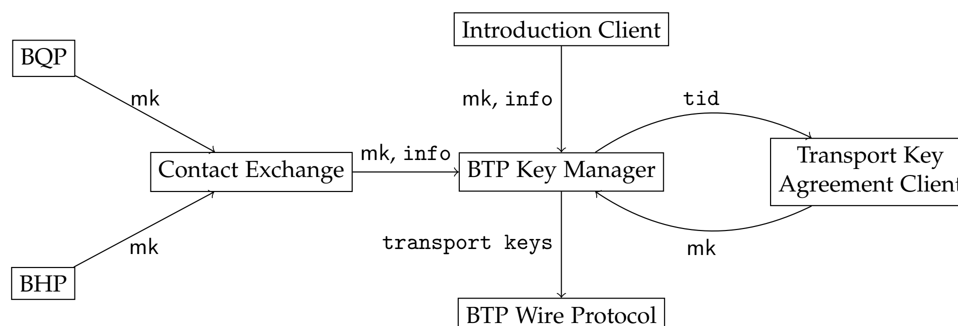

Figure 3.1: Relations between key agreement and contact exchange protocols in Briar. Here `mk` denotes the shared secret key, `info` denotes contact information (name, public key, etc.), and `tid` denotes the transport identifier.

- **Contact Exchange:** A protocol that allows peers having agreed on a shared secret key to exchange and verify contact details via BTP (Chapter 6).
- **Introduction Client:** A BSP client that allows a Briar user to introduce two of the user's contacts to each other (Chapter 9);
- **Transport Key Agreement Client:** A BSP client that establishes transport keys between two peers for transports that are added after their contact exchange (Section 10.1).

3.2 Protocol stack overview

Table 3.1: The Briar protocol stack.

				BSP Clients (Messaging,...)
BQP	BRP	BHP	Contact Exchange	BSP
-	-	BTP (handshake)	BTP (undocumented)[1]	BTP (rotation)
Bluetooth/WiFi	Tor		Bluetooth, WiFi, Tor, ...	

Table 3.1 provides a schematic overview of the Briar protocol stack; see also the wiki page about the Briar protocol stack [16] for more details. Figure 3.1 outlines the relations between key agreement and contact exchange protocols in Briar.

In the remaining sections of the chapter, we describe in a high level how Briar works. Readers may also refer to the Briar user manual [12].

[1]The way to manage BTP keys in the contact exchange protocol is undocumented in the Briar specifications and is likely an ad hoc solution; see Chapter 6.

3.3 Account creation and adding contact

When opening Briar for the first time, the user would be prompted to create an account by setting a nickname and a password. Briar creates the account and generates a signing key pair $(\mathsf{spk}, \mathsf{ssk})$ and a handshake key pair $(\mathsf{hpk}, \mathsf{hsk})$ for the account locally. There is no way to back up the account or tranfer the account to other devices; if the user forgets her password or loses her device, she would lose access to the account permanently and would have to register a new account.

After creating an account, the user may proceed to add contacts in the following ways:

1. Via QR code. Two peers nearby first scan each other's QR code, which contains a commitment to a ephemeral public key epk and transport descriptiors, and then derive a shared secret key mk in the key agreement phase of Bramble QR Code Protocol (BQP) over Bluetooth or WiFi. Note like unlike many other messaging applications, the QR code is re-generated every time and it is necessary for both parties to scan the QR code of their peer;

2. Via handshake links. Two peers exchange their handshake links out-of-band, which contain their handshake public keys. They then use Bramble Rendezvous Protocol (BRP) to establish a connection over Tor protected by a static key, and run Bramble Handshake Protocol (BHP) over the connection to derive a shared secret key mk;

3. Via introduction. A Briar user may introduce two of her contacts by first sending requests to them. If both contacts accept the introduction, the introducer may relay protocol messages between the contacts via established connections. In the introduction protocol, the contacts can derive a shared secret key mk unknown to the introducer and exchange their contact information.

For the first two options, the peers need to additionally perform contact exchange, where they exchange contact information records, signed with ssk and encrypted with a key derived from mk. This step is used to verify the peer's identity, exchange contact information, and negotiate the timestamp.

3.4 Communication

After adding a contact, the user may send messages to or receive messages from the contact. The messaging functionality is implemented as a BSP client, which synchronizes messages between the peers using Bramble Synchronisation Protocol (BSP).

Briar also supports private groups, forums, and blogs. They are also implemented as BSP clients, which synchronize messages between members or subscribers in the same BSP group. Only the creator can invite new members to a private group by sending a signed invitation token, while any forum subscriber may invite new subscribers to the forum. A blog is visible precisely to the user's contacts. While all members or subscribers of the private group/forum/blog can see all messages in the group, they may only perform syncrhonization between direct contacts in the group to whom they revealed the membership. Messages in the private group/forum/blog are BSP messages that generally include the group identifier, timestamp, and a BDF list of the message type, content, and the author's signature.

In a lower level, BSP runs over Bramble Transport Protocol (BTP), which secures the trasmitted data using session keys derived from the master key. BTP rotates session keys peroidically to provide forward security. BTP runs over Bluetooth, WiFi, or Tor connections between the peers.

Chapter 4

Bramble QR Code Protocol (BQP)

Bramble QR Code Protocol (BQP) is a protocol used in Briar to add contacts nearby. Two peers first scan each other's QR code, and then use the information from the QR code to establish communication and derive a shared secret key. In Briar, the peers use the key derived from BQP to perform contact exchange (Chapter 6) and set the key as the root key in Bramble Transport Protocol (BTP) rotation mode (Section 7.1.3). Readers may also refer to the BQP specification [8].

4.1 Constants

The current version of BQP is 4. The record types in BQP are `KEY` (`0x00`), `CONFIRM` (`0x01`), and `ABORT` (`0x02`). Labels in BQP are namespaced strings of the form `"org.briarproject.bramble.keyagreement/ABC"`, where `ABC` is the label name; we omit `"org.briarproject.bramble.keyagreement"` in the following sections for simplicity.

4.2 Preparation and connection establishment

Each peer first runs Algorithm 1 to generate a ephemeral key pair and derive a QR code containing a commitment to the ephemeral public key and transport descriptors. The commitment is the first 16 bytes of the labeled hash of the ephemeral public key. The peers then scan each other's QR code and parse the payload, rejecting payloads from different protocol versions.

After parsing payloads, the peers compare the lexicographical order of their commitments locally to determine their roles in the protocol, and try to use the transport descriptors to establish a connection within 60 seconds.

Algorithm 1: Generate key pair and QR code.

```
(epk, esk) = keyGen();
comm = hash(".../COMMIT", epk)[0:15];
descriptors = list((id1, transportDescriptor1), ..);
payload = 0x04||BDF-encode(list(comm) + descriptors);
createQrCode(ISO-8859-1-decode(payload));
```

4.3 Key agreement

Figure 4.1 shows the key agreement phase of BQP. Values sent and received here are BQP records of the form $\mathtt{0x04}\|\mathtt{type}\|\mathtt{int16}(\mathtt{len}(\mathtt{payload}))\|\mathtt{payload}$, where $\mathtt{type} \in \{\mathtt{KEY}, \mathtt{CONFIRM}, \mathtt{ABORT}\}$. Algorithm 2 checks BQP records, aborting when the type of incoming record is ABORT or unexpected. The party that aborts should send a record with type ABORT to its peer, who should also abort on receiving the record.

Algorithm 2: `readRecord(reader, expectedType)`

```
record = reader.readRecord(4);
if record == null then abort;
type = record.getRecordType();
if type == ABORT or type ≠ expectedType then abort;
return record.getPayload();
```

Alice first sends her ephemeral public key $\mathsf{epk}_\mathtt{A}$ to Bob, who checks that it matches the commitment $\mathtt{comm}_\mathtt{A}$ from the QR code payload of Alice. Bob then sends $\mathsf{epk}_\mathtt{B}$ to Alice, who checks the it matches $\mathtt{comm}_\mathtt{B}$. After receiving each other's public key, Alice and Bob respectively calculate the X25519 key exchange result of the local private key with the remote public key, $\mathtt{raw} = \mathtt{DH}(\mathsf{esk}_\mathtt{A}, \mathsf{epk}_\mathtt{B}) = \mathtt{DH}(\mathsf{esk}_\mathtt{B}, \mathsf{epk}_\mathtt{A})$. The raw secret is hashed together with the label, the protocol version, and the ephemeral public keys, to produce the shared secret $\mathtt{s}$.

To confirm that they have received the correct public key, Alice and Bob each derives a confirmation key ck from $\mathtt{s}$, and computes a message authentication code over the QR code payloads and ephemeral public keys, the difference being the exact order. Each party sends the confirmation to its peer for verification; if the verification fails, then the peer aborts the protocol. Finally, the master key mk is derived from the shared secret $\mathtt{s}$.

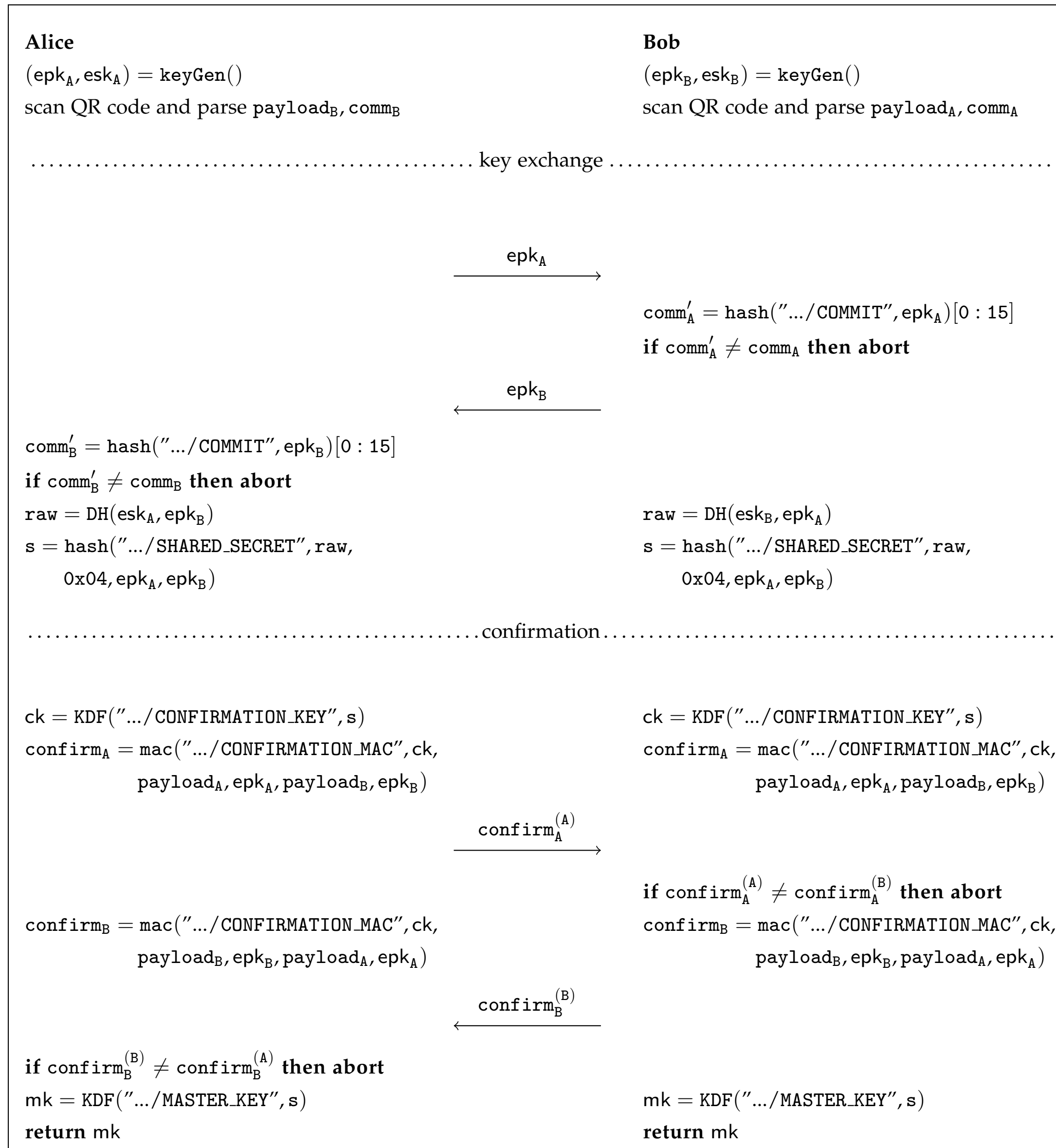

Figure 4.1: Key agreement phase of Bramble QR Code Protocol (BQP). Note that peers would check that records received are well-formed and abort if not.

4.4 Notes

- *Reversing commitment.* As the commitments have 128 bits, it should be infeasible to recover the committed ephemeral public key, but feasible to find two ephemeral public keys committed to the same value. The latter case, however, is not useful for the attacker, as the attacker cannot influence the ephemeral keys of the honest users. [8]
- *Role confusion.* The peers determine their roles by comparing the lexicographical order of their commitments, so a role confusion is not possible. However, it could be the case that two peers derive the same commitment value (one party cheating, or with negligible probability). As a result, each party thinks she is Bob and and waits for the remote peer, so the protocol times out after 60 seconds.
- *Reflection attack.* A reflection attack is not possible because the reflected party always thinks she is Bob and would ask for the committed public key first.
- *Small order subgroup attack.* A small order subgroup attack is unlikely because the private keys are clamped on generation, the raw X25519 key exchange aborts on zero result (Curve25519 also maps the infinity point to 0), and the ephemeral keys are also included in the hash for producing the shared key. The developers also noticed this when migrating to Curve25519.[1]
- *Labels and keys.* The labels for cryptographic operations in BQP are peoperly namespaced and not reused elsewhere. Briar generates a fresh ephemeral key pair each time and does not store the private key.
- *State machine.* The state machine of BQP is relatively simple and seems well-formed.
- *Formal verification.* A simplified version of BQP is verified by Tamarin, where we ignore the payloads and assume the commitments are exchanged over an authenticated channel (Appendix A.1). The simplified protocol satisfies key secrecy (that the attacker cannot recover the shared key `mk`) and injective agreement. The definition for injective agreement is quoted here in verbatim: "Whenever `a` completes a run of the protocol, apparently with `b` in role `B`, then `b` has previously been running the protocol, apparently with `a`, and `b` was acting in role `B` in his run, and the two principals agreed on the message `t`. Additionally, there is a unique matching partner instance for each completed run of an agent" [18]. The definitions and more information about these security properties can be found in Chapter "Property Specification" of the Tamarin prover manual [18].

[1] https://code.briarproject.org/briar/briar/-/issues/1163

Chapter 5

Bramble Handshake Protocol (BHP)

Bramble Handshake Protocol (BHP) is a protocol used in Briar to add remote contacts. Before running BHP, the peers should have exchanged their long-term public keys. This is done in Briar via exchanging the handshake link (Section 5.1) between peers. BHP runs on top of Bramble Transport Protocol (BTP) handshake mode (Section 7.1.2) and establishes an ephemeral shared secret key between two peers. Similar to BQP, the Briar peers use the derived key from BHP to perform contact exchange (Chapter 6) and set up transport keys (Section 7.1.3). Readers may also refer to the BHP specification [7].

5.1 The protocol

The current version of BHP is 0. The record types are `EPHEMERAL_PUBLIC_KEY` (0x00) and `PROOF_OF_OWNERSHIP` (0x01). Labels in BHP are namespaced strings of the form `"org.briarproject.bramble.handshake/ABC"`, where `ABC` is the label name. We omit `"org.briarproject.bramble.handshake"` in the following sections for simplicity.

The handshake link should be of the form `(briar://)?([a-z2-7]{53})`.[1] The latter part is the Base32 encoding of `0x00||pk`, where `0x00` is the format version of the handshake link. The handshake key pair and the handshake link stay unchanged for a Briar identity after creation, and there is currently no way to create a new handshake link except from creating a new account.

Assume that the peers have exchanged their handshake links and established a connection via Tor, which is done using Bramble Randezvous Protocol (BRP) in Briar. The connection is set up in BTP handshake mode, where the keys are statically derived from the long term handshake keys (see Section 7.1.2). Before starting the handshake, each peer compares the lexicographic order of the local public key and remote public key as byte

[1] An example: `briar://adplpxapai4rx37brjgiml3ixbufvof56baq7yz63gkxhhibm6qae`

arrays, taking the role of Alice if and only if the local public key is smaller. Let $(\mathsf{hpk_A}, \mathsf{hsk_A})$ and $(\mathsf{hpk_B}, \mathsf{hsk_B})$ be the static handshake key pairs of Alice and Bob, respectively. The protocol is shown in Figure 5.1.

First, Alice and Bob each generates an ephemeral key pair $(\mathsf{epk_A}, \mathsf{esk_A})$, and $(\mathsf{epk_B}, \mathsf{esk_B})$, and sends the ephemeral public key to its peer. Then, each party calculates the X25519 raw shared secret of the static keys and two pairs of static-ephemeral keys. The raw secrets are hashed together with the label, long term public keys, and ephemeral public keys to produce the master key mk.

Alice and Bob then prove to each other they derived the correct master key. Each party calculates a message authentication code over its label with the master key and sends it to its peer. The peer aborts if the verification fails. Finally, if the proof matches, each party returns the derived master key mk.

Similar to BQP, values sent and received are encrypted records of the form $\texttt{0x00}\|\texttt{type}\|\texttt{int16}(\texttt{len}(\texttt{payload}))\|\texttt{payload}$, where `0x00` is the protocol version in bytes, and $\texttt{type} \in \{\texttt{EPHEMERAL_PUBLIC_KEY}, \texttt{PROOF_OF_OWNERSHIP}\}$. Only records with the same version and expected types are accepted.

5.2 Lack of forward security

The BHP specification claims that "the shared key produced by BHP can be used to upgrade BTP to transport mode, which provides forward secrecy" [7]. However, for some unknown reason,[2] BHP is not forward secure as claimed.

If the long-term hanshake secret keys of Alice and Bob, $\mathsf{hpk_A}$ and $\mathsf{hpk_B}$, are both compromised, then the attacker could easily recompute from the protocol transcript the three X25519 key exchange results, $\texttt{raw}_0, \texttt{raw}_1, \texttt{raw}_2$, and recover the master key mk. The reason is that an ephemeral-to-ephemeral key exchange secret, $\texttt{DH}(\mathsf{esk_A}, \mathsf{epk_B})$, or equivalently, $\texttt{DH}(\mathsf{esk_B}, \mathsf{epk_A})$, is missing from the hash input. As a result, BHP is not forward secure, and BTP rotation mode initialized with the master key from BHP also fails to provide forward security. Note that while showing a design flaw in BHP, it should be difficult for attackers to exploit this vulnerability in practice, as attackers have to obtain the protocol transcript sent over Tor.

A simple fix would be to also incorporate $\texttt{DH}(\mathsf{esk_A}, \mathsf{epk_B}) = \texttt{DH}(\mathsf{esk_B}, \mathsf{epk_A})$ to the hash input for key derivation, and optionally remove $\texttt{DH}(\mathsf{hsk_A}, \mathsf{hpk_B}) = \texttt{DH}(\mathsf{hsk_B}, \mathsf{hpk_A})$ from the hash input. For users that have performed handshakes before the fix, Briar could either revoke old handshake key pairs, or

[2] A guess would be that BHP was added later than other protocols and is not yet thoroughly analyzed [11].

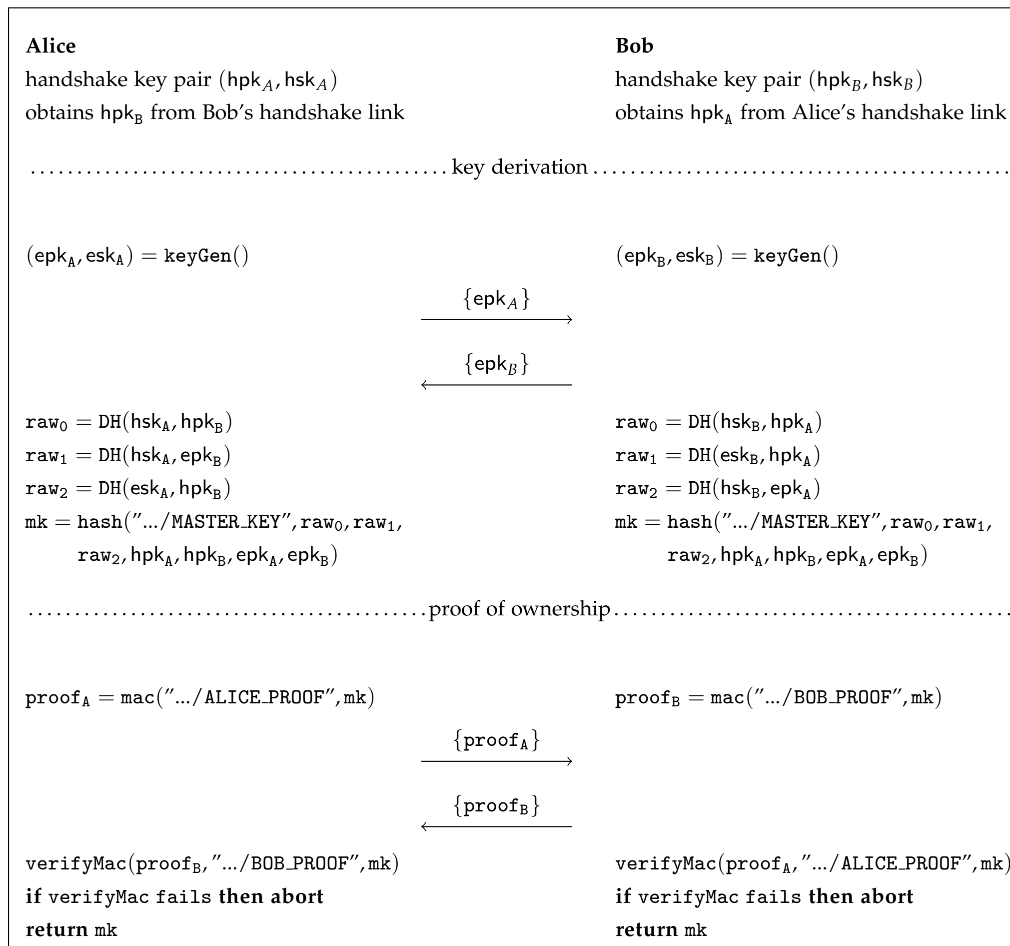

Figure 5.1: Bramble Handshake Protocol (BHP). Values on wire are encrypted in BTP handshake mode. Peers would check if records received are well-formed after decryption and abort if not.

derive fresh keys for all affected transports using the transport key agreement client (Section 10.1).

5.3 Notes

- *Self-handshake.* As Base32 encodes every five bits to one character, the handshake link can actually encode $53 \cdot 5 = 265$ bits of information, one bit more than the $33 \cdot 8 = 264$ bits being encoded. The extra bit is discarded during decoding and is not strictly enforced to be zero. Therefore, there are two equivalent ways of encoding the same handshake public keys that only differ by the last character (e.g., changing from `e` to `f`). This introduces the possibility of performing a self-handshake, as Briar only rejects handshake links that are exactly the

same as the user's own handshake link (after prepending `briar://` if needed). However, in this case, Briar cannot connect to the remote party via BRP, where each party performs a static key exchange on the handshake keys and derives an address according to their roles to establish a Tor hidden service. Since the attacker does not know the user's handshake private key and square-DH is assumed to be hard in Curve25519, the attacker cannot derive the hidden service address for the other role and therefore cannot connect to the user.

- *Duplicate handshake links.* Briar would check the parsed handshake public key from the link and see if it comes from an existing contact or pending contact (which is a remote peer to be added as a contact via BHP); if so, Briar asks the user if the two links come from the same person, and warns the user about a possible contact discovery attempt if the user answers no. Briar removes the new link in both cases.
- *Small order subgroup attack.* This is unlikely following the similar reasoning for BQP in Section 4.4.
- *Reflection.* It might be tempting to add a small order element to the public key, but then the derived keys in BTP handshake mode would still be different because the public keys are included in the hash input for deriving transport keys; see Section 7.1.2.
- *Labels and keys.* The labels for cryptographic operations in BHP are peoperly namespaced and not reused elsewhere. The long-term handshake private keys are stored in an encrypted database, the key of which is again encrypted with the user's password. Briar generates a fresh ephemeral key pair for a new handshake and does not store it.
- *State machine.* The state machine of BHP is simpler than BQP and seems well-formed.
- *Formal verification*. We verified a simplified version of BHP with Tamarin (Appendix A.2), which shows that BHP satisfies key secrecy and injective agreement, but does not provide forward security.

Chapter 6

Contact Exchange

Having derived a shared serect key mk via BQP (Chapter 4) or BHP (Chapter 5), Alice and Bob (keeping their roles in the previous protocol) start exchanging contact information via the same channel used for key agreement. This protocol is not covered in the Briar specifications.

6.1 The protocol

The current version of the protocol is 1. All records in contact exchange are of the type `CONTACT_INFO` (0x00). Labels are namespaced strings of the form `"org.briarproject.bramble.contact/ABC"`, where `ABC` is the label name. We omit `"org.briarproject.bramble.contact"` in the following sections for simplicity.

A hardwired mimimum reasonable time `MIN_REASONABLE_TIME_MS = 1,609,459,200,000` (which represents 1 Jan. 2021, 00:00:00 UTC) is in place to avoid too old timestamps from being negotiated. This also mitigates denial-of-service attacks from excessive key rotations.

The peers keep their roles of Alice and Bob in the previous protocol (BQP or BHP) used to derive the master key. Let $(\mathsf{spk_A}, \mathsf{ssk_A})$, $(\mathsf{spk_B}, \mathsf{ssk_B})$ be Ed25519 signing key pairs of the peers. The protocol is shown in Figure 6.1. Two peers first use the keys derived from the master key to set up a bidirectional secure communication channel via BTP. All subsequent communications in the protocol are via the encrypted channel. Then, each peer proves it owns the signing key pair by signing a nonce (a MAC over its label with mk) with its signing private key ssk. Each party packs the signing public key, the signature and some other contact information (transport properties, timestamp, ...) into a record and send the encrypted record to its peer. The peer checks that the entries in the record are correctly formatted and verifies the signature. The peers agree on a timestamp and abort

if the agreed timestamp is too old. Finally, each party adds contact information of its peer and uses mk as the root key to derive key sets in BTP rotation mode. Values sent and received are encrypted records of the form 0x01‖CONTACT_INFO‖int16(len(payload))‖payload.

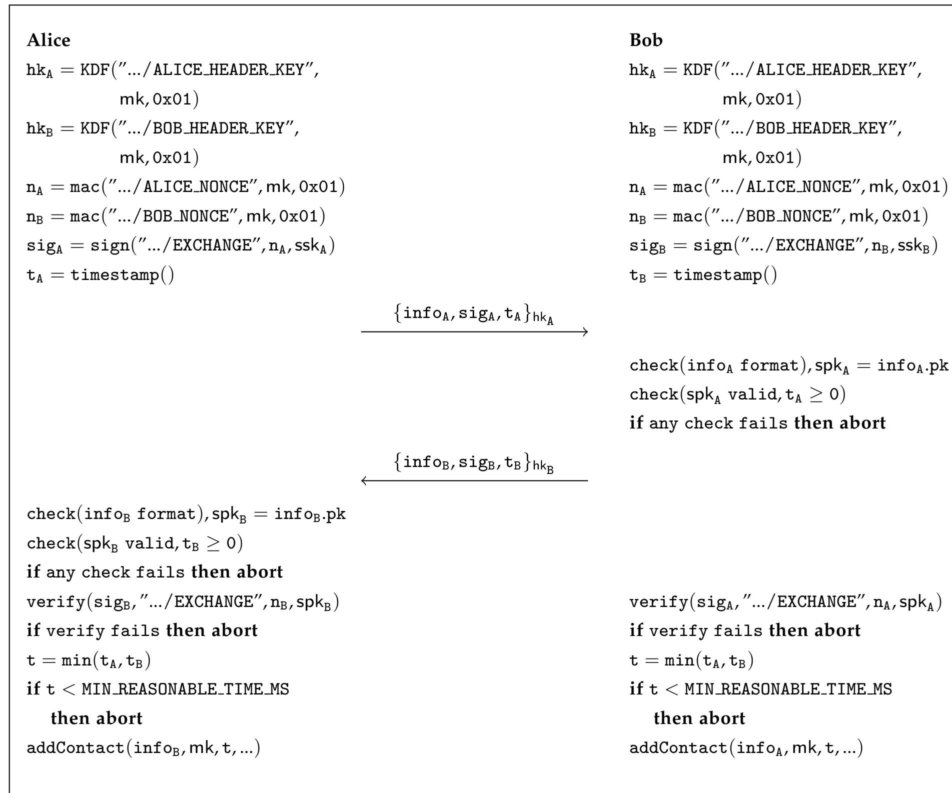

Figure 6.1: Briar contact exchange. Values on wire are records encrypted using BTP with the header keys specified in subscripts.

6.2 Notes

- *Custom BTP mode.* As mentioned before, the contact exchange protocol uses an ad hoc method of managing key sets for the underlying BTP, which is undocumented in the specification. However, it is not obvious that this would lead to any attacks.
- *Weak signature.* It is possible that one party deliberately uses a weak Ed25519 signing key pair such that she can create a valid signature without knowing the message (nonce here) to be signed, but it is not clear how this is going to be useful, since sending a well-formed record

already implies knowledge of the BTP keys and therefore knowledge of the master key and the nonce derived from it.

Chapter 7

Bramble Transport Protocol (BTP)

Bramble Transport Protocol (BTP) is a transport layer security protocol that allows two peers to build a secure channel for communication over some unidirectional or bidirectional transport. The underlying transport should be able to deliver in order a sequence of bytes (called a connection in Briar) on a best-effort basis. Connections may be discarded or reordered. Readers may also refer to the BTP specification [10].

7.1 Key management protocol

For each available transport, the peers create and maintain a set of keys using a time-based key management protocol described here. The transport properties used in key derivation and management are transport identifier `id` (e.g., `"org.briarproject.bramble.bluetooth"` for Bluetooth) and maximum latency `L` (e.g., hard-coded as 30,000 milliseconds, or 30 seconds, for Bluetooth, and usually 30 seconds for other transports).

7.1.1 Constants

The maximum clock difference `MAX_CLOCK_DIFFERENCE` is `86,400,000`, which is 24 hours in milliseconds. The clock difference could be a result of people not setting their time zones correctly. Labels in BTP are namespaced strings of the form `"org.briarproject.bramble.transport/ABC"`, where `ABC` is the label name. We omit `"org.briarproject.bramble.transport"` in the following sections for simplicity.

7.1.2 Handshake mode

Each party compares the lexicographical order of its handshake public key with the remote peer's handshake public key to decide on taking the role

of Alice or Bob. We denote the handshake key pairs of Alice and Bob as $(\mathsf{hpk_A}, \mathsf{hsk_A})$ and $(\mathsf{hpk_B}, \mathsf{hsk_B})$, respectively.

The key derivation process for Alice and Bob is shown in Algorithm 3; function `clock()` returns the elapsed time since the Unix epoch (1 Jan. 1970, 00:00:00 UTC) in milliseconds. To match the rotation mode, the key manager keeps incoming keys for the previous, current, and next time period and outgoing keys only for the current period. For the case where Alice and Bob have already added each other as a contact (still not fully implemented in Briar), we replace the label for deriving the pending contact root key pcrk with `".../CONTACT_ROOT_KEY"`. Algorithm 4 shows how to derive the header key and the tag key from the pending contact root key. As we also store the root key in handshake mode, the key sets can be easily updated at each interval by shifting key sets and computing new keys from the root key.

Algorithm 3: Key derivation in BTP handshake mode.

```
weAreAlice = hpk_local < hpk_remote;
raw = weAreAlice ?  DH(hsk_A, hpk_B) :  DH(hsk_B, hpk_A);
smk = hash(".../STATIC_MASTER_KEY", raw, hpk_A, hpk_B);
pcrk = mac(".../PENDING_CONTACT_ROOT_KEY", smk);
foreach transport (id, L) do
    P = ⌊clock() / (L + MAX_CLOCK_DIFFERENCE)⌋;
    if P < 1 then abort;
    inPrev = deriveHandshakeKeys(id, pcrk, !weAreAlice, P-1);
    inCurr = deriveHandshakeKeys(id, pcrk, !weAreAlice, P);
    inNext = deriveHandshakeKeys(id, pcrk, !weAreAlice, P+1);
    outCurr = deriveHandshakeKeys(id, pcrk, weAreAlice, P);
    addKeys(id, inPrev, inCurr, inNext, outCurr, pcrk, weAreAlice);
```

Algorithm 4: `deriveHandshakeKeys(id, pcrk, isAlice, P)`

```
TAG_LABEL = isAlice ?  ".../ALICE_HANDSHAKE_TAG_KEY":
 ".../BOB_HANDSHAKE_TAG_KEY";
HEADER_LABEL = isAlice ?  ".../ALICE_HANDSHAKE_HEADER_KEY":
 ".../BOB_HANDSHAKE_HEADER_KEY";
k_tag = KDF(TAG_LABEL, pcrk, utf-8(id), int64(P));
k_header = KDF(HEADER_LABEL, pcrk, utf-8(id), int64(P));
return (k_tag, k_header);
```

7.1.3 Rotation mode

Assume the peers have previously established a shared secret key rk and a timestamp `timestamp` (counted from the Unix epoch in milliseconds), and that they keep their respective roles of Alice and Bob in the previous pro-

tocol. The timestamp is negotiated by the peers to reduce the number of rounds for key rotation; otherwise they may have to rotate the keys starting from the Unix epoch, which has a higher cost and can be vulnerable to denial-of-service attacks.

Algorithm 5 shows the key derivation process. The key manager keeps incoming keys for the previous, current, and next period, because incoming messages should always fall into one of these three periods if our assumptions are correct. Algorithm 6 shows the key update process, in which we keep rotating the keys until we reach the current period. Algorithm 7 shows how we derive the header key and the tag key from the root key. Algorithm 8 rotates the key sets by computing the labeled hash with time period for each key respectively.

Algorithm 5: Key derivation in BTP rotation mode.

```
foreach transport (id, L) do
    P = ⌊timestamp / (L + MAX_CLOCK_DIFFERENCE)⌋;
    inPrev = deriveRotationKeys(id, rk, !weAreAlice);
    outPrev = deriveRotationKeys(id, rk, weAreAlice);
    inCurr = rotateKeys(inPrev, P);
    inNext = rotateKeys(inCurr, P+1);
    outCurr = rotateKeys(outPrev, P);
    keys = TransportKeys(id, inPrev, inCurr, inNext, outCurr, P);
    curP = ⌊clock() / (L + MAX_CLOCK_DIFFERENCE)⌋;
    updatedKeys = updateTransportKeys(keys, curP);
    addKeys(id, updatedKeys);
```

Algorithm 6: `updateTransportKeys(k=(id, inPrev, inCurr, inNext, outCurr, P), curP)`

```
if k.P >= curP then return k;
for p = k.P+1 to curP do
    inPrev, inCurr = inCurr, inNext;
    inNext = rotateKeys(inNext, p+1);
    outCurr = rotateKeys(outCurr, p);
return (id, inPrev, inCurr, inNext, outCurr, curP);
```

7.2 Wire protocol

The wire protocol delivers encrypted streams of data over some possibly insecure connections. Note that the protocol is not intended to deliver streams of data reliably; the data might be delayed, reordered, or deleted.

Algorithm 7: `deriveRotationKeys(id, rk, isAlice)`

```
TAG_LABEL = isAlice ? ".../ALICE_TAG_KEY": ".../BOB_TAG_KEY";
HEADER_LABEL = isAlice ? ".../ALICE_HEADER_KEY": ".../BOB_HEADER_KEY";
```
k_{tag} = `KDF(TAG_LABEL, rk, utf-8(id));`
k_{header} = `KDF(HEADER_LABEL, rk, utf-8(id));`
return (k_{tag}, k_{header});

Algorithm 8: `rotateKeys(`(k_{tag}, k_{header})`, P)`

k'_{tag} = `KDF(".../ROTATE",` k_{tag}`, int64(P));`
k'_{header} = `KDF(".../ROTATE",` k_{header}`, int64(P));`
return (k'_{tag}, k'_{header});

The current protocol version is 4.

7.2.1 Stream and tag

The underlying transport of BTP should be able to deliver a sequence of bytes in order, which is called a connection in Briar. BTP uses a connection to carry a sequence of encrypted and authenticated data, which is called a stream.

Each stream has a 32-bit stream number (encoded as a 64-bit integer), assigned by the key manager of the transport. Stream numbers are counted starting from zero in each time period. A stream consists of a tag, a stream header (Section 7.2.2), and one or more frames (Section 7.2.3).

A stream starts with a pseudo-random tag of `TAG_LEN = 16` bytes, which is of the form `PRF(`k_{tag}`, [0x00, 0x04]||int64(streamNumber))[:TAG_LEN-1]`. The function `PRF` is implemented with BLAKE2b (see Section 2.3). Each peer maintains a reordering window of size 32 for each time period (or to say each set of keys in use) to recognize a possibly out-of-order stream and avoid stream replays.

7.2.2 Stream header

The pseudo-random tag is followed by a stream header, which is of the form `nonce||ENC(`k_{header}`, nonce, [0x00, 0x04]||int64(streamNumber)||frameKey)`, where `nonce` is a secure random byte array of length 24, and `frameKey` is a secure random byte array of length 32. We see that the plaintext of a stream header has 2+8+32=42 bytes, and the stream header itself has 24+(42+16) = 82 bytes (the length of Poly1305 output is 16 bytes). Algorithm 9 shows the process of decrypting a stream header, which mostly follows the definition of the stream header.

Algorithm 9: `readStreamHeader()`

```
ciphertext = read(0, 82);
nonce = ciphertext[0:23];
plaintext = DEC(k_header, nonce, ciphertext[24:]);
if len(plaintext) != 32 then abort;
if plaintext[0:1] != [0x00, 0x04] then abort;
if plaintext[2:9] != int64(streamNumber) then abort;
frameKey = plaintext[10:];
```

7.2.3 Frame

Some frame constants are listed below.

```
NONCE_LEN = 24;
FRAME_HEADER_PLAINTEXT_LENGTH = 4;
FRAME_HEADER_LENGTH = 20;
MAX_FRAME_LENGTH = 1024;
MAX_PAYLOAD_LENGTH = 988;
```

The remainder of the stream consists of one or more frames, numbered from 0 to $2^{63} - 1$. Each frame has a fixed-length frame header and a variable-length frame body that may contain data and an optional padding. The optional padding is implemented but not used in Briar. The header and data are both encrypted and authenticated with `frameKey` in the stream header, albeit with different deterministic nonces. Nonces are not sent over the wire.

The frame header is of the form `ENC(frameKey, headerNonce, final||int15(len(data))||int16(len(padding)))`. Field `headerNonce` is a byte array of length 24, with first 8 bytes set to `int64(frameNumber)`, the first (most significant) bit then set to `1`, and the rest of the bytes set to `0x00`. Field `final` is a one-bit flag indicating if the current frame is the final frame. The frame header plaintext has 4 bytes and the frame header has 4+16=20 bytes. Note that we can overlap the frame number field with an additional flag in the nonce because the most significant bit is unused.

The frame data is of the form `ENC(frameKey, dataNonce, data||padding)`, where `dataNonce` is a byte array of length 24, with first 8 bytes set to `int64(frameNumber)`, and the rest of the bytes set as `0x00`. The maximum length of the encrypted frame is 1,024 bytes, so we have 1,024-20-16=988 bytes remaining for payload and optional padding in plaintext.

Algorithm 10 shows the process of reading a frame, where `read(x,y)` denotes reading `y` bytes starting from the `x`-th byte, $\vee$ denotes a bitwise-OR operation, and `[0x00]*(NONCE_LEN-8)` denotes an all-zero byte array of 16

bytes. On abort specified in the algorithm or error within the cryptographic primitives (decryption error in this case), the connection would be closed.

Algorithm 10: `readFrame()`

```
if finalFrame then return null;
if frameNumber < 0 then abort;
ciphertext = read(0, FRAME_HEADER_LENGTH);
nonce = int64(frameNumber)||[0x00]*(NONCE_LEN-8);
nonce[0] = nonce ∨ 0x80 ;
frameHeader = DEC(frameKey, nonce, ciphertext);
if len(frameHeader) != FRAME_HEADER_PLAINTEXT_LENGTH then abort;
finalFrame, payloadLen, paddingLen = decode(frameHeader);
if payloadLen + paddingLen > MAX_PAYLOAD_LENGTH then abort;
frameLen = FRAME_HEADER_LENGTH + payloadLen + paddingLen + 16;
ciphertext = read(FRAME_HEADER_LENGTH, frameLen);
nonce = int64(frameNumber)||[0x00]*(NONCE_LEN-8);
payload = DEC(frameKey, nonce, ciphertext);
if len(payload) != payloadLen + paddingLen then abort;
for i = 0 to paddingLen-1 do
    if payload[payloadLen + i] != 0x00 then abort;
frameNumber++;
return payload;
```

7.3 Notes

- *Underlying transport.* BTP operates on a sequence of bytes that can be delivered in order, so UDP should not be used for the underlying transport. Note that having an unreliable underlying transport such as UDP would not harm the security of BTP, as a malformed stream should cause BTP to abort and close the connection.
- *Payload length.* We can set the payload length in the frame header to be larger than the actual payload length, but we cannot hope to use this to read the program memory of the remote machine, because 1) Java is memory safe, 2) the function `read()` only returns the available bytes in the input stream, so `readFrame()` can only read bytes from the connection, and 3) with high probability the authenticated cipher would fail to decrypt the malformed payload and throw an error, causing the connection to be closed.
- *Deletion and truncation.* Each of the stream can be deleted or indefinitely delayed separately. Within a stream, the last frames can be truncated, but then the final frame flag would go missing. However, there seems to be no active checks on whether the final frame flag is

raised in the end (we can always think of the final frame as not yet arrived).

- *Duplication.* Since we keep a reordering window for each time period, the streams cannot be replayed. Frames within a stream cannot be duplicated either, because otherwise they would be decrypted with incorrect nonces.
- *Tag collision.* In an honest execution, the tags would only collide with negligible probability. An attacker may be able to create a tag collision intentionally (e.g., by brute-forcing the BTP handshake mode offline), but it would not be useful to the attacker, as the tag would be recognized as the context added more recently.
- *Key robustness.* As XSalsa20Poly1305 is used for encryption, it is possible for attackers to create two valid streams that differ in the tag and stream header but have the same frames. However, this does not benefit the attacker in any obvious way. We note that when Briar forwards a message received from a BSP group, the message would stay the same, but the underlying BTP stream sent would be different (and looks independent) from the BTP stream it received.

Chapter 8

Bramble Synchronisation Protocol (BSP)

Bramble Synchronisation Protocol (BSP) is an application layer data synchronization protocol in Briar that allows members of the same group to synchronize messages over delay-tolerant networks. Although, as the name suggests, there is very little cryptography involved in BSP, it may still be interesting to discuss how BSP provides security, since the underlying BTP can be unreliable. Readers may also refer to the BSP specification [9].

8.1 Concepts

Basically, BSP synchronizes *messages* among *clients* within the same *group*.

A BSP *client* is an application component in Briar that uses BSP to achieve its functionalities, such as the messaging client and the introduction client. Each type of BSP client has a client identifier, a major version number, and a minor version number. For example, the client identifier of the messaging client in Briar is `"org.briarproject.briar.messaging"`, and the current major and minor version numbers of the messaging client are respectively 0 and 3. BSP clients may only synchronize messages with other BSP clients of the same major version.

A *group* consists of one or more BSP clients of the same client identifier `client_id` and major version `major`. The group identifier is computed by each BSP client locally as `hash("org.briarproject.bramble/GROUP_ID", 0x01, int32(major), descriptor)`, where `0x01` is the group format version, and `descriptor` is the group descriptor decided by the clients. For example, the group descriptor of the messaging client is a BDF list of sorted author identfiers of the two peers, cast to a byte array.

A *message* in BSP consists of a group identifier `group_id`, a timestamp (a

64-bit integer cast to bytes that denotes milliseconds elapsed from the Unix epoch) `timestamp`, and the message body `body`. Each message has a message identifier `message_id`, which is not included in the message but can be uniquely computed as `hash("org.briarproject.bramble/MESSAGE_ID", 0x01, group_id, timestamp, root_hash)`, where `0x01` is the message format version, and `root_hash` is `hash("org.briarproject.bramble/MESSAGE_BLOCK", 0x01, body)`. It should be computationally infeasible to produce two different messages with the same message identifier if we assume that BLAKE2b is collision-resistant. Each client validates the received message according to the validation policy specified by the BSP client.

The wire protocol synchronizes messages using records of the form `0x00`‖`type`‖`int16(len(payload))`‖`payload`, where `0x00` is the current version of BSP, and `type` is one of {`ACK, MESSAGE, OFFER, REQUEST, VERSIONS, PRIORITY`}. As the name suggests, `ACK`, `OFFER`, and `REQUEST` records contain a list of message identifiers to acknowledge receipt, offer to the remote client, and request to the remote client, respectively. The `MESSAGE` record contains a BSP message. The `VERSIONS` record is implemented for future BSP protocols to be able to negotiate supported versions but is never sent by the current BSP protocol. The `PRIORITY` record type is not documented in the specification and is used to choose between redundant connections.

Messages in BSP can also have dependencies, represented as message identifiers. The exact way of representing and handling message dependencies is decided by the respective BSP client. For example, the messaging client does not implement any dependencies, while in the private group client, each post message depends on the previous message from the same author, and optionally, its parent message in the thread. A message with dependencies is first validated by the client, and then put on hold until all dependencies have arrived.

A BSP client only synchronizes messages in a group with a remote client if the remote client is controlled by a direct contact of the local identity, they are in the same group, and the local client shares the group to the remote client. The sharing policy is decided by each BSP client. Nevertheless, every member in the same group is entitled to view all messages within the group, and would eventually arrive at the same view if the synchronization network is connected.

8.2 Denial-of-service attack

The maximum record payload in BSP is 48KiB, which can be reached by `ACK`, `OFFER`, and `REQUEST` records. In contrast, a message has a 40-byte header and a body of at most 32KiB. The BSP record reader implementation in Briar

does not check if a message is too long before parsing.[1]

A remote contact can send a `MESSAGE` record of size between 32,809 bytes (32KiB+41B) and 49,152 bytes (48KiB) to the victim. The victim's application will throw an `IllegalArgumentException` when trying to parse this message,[2] which will halt the application. As long as the victim does not delete the malicious contact and they are both online, the malicious contact can repeatedly launch this attack, rendering the victim's application unusable.

To fix this issue, Briar could check whether the message is too long before parsing, or change/catch the `IllegalArgumentException` thrown from the aforementioned code.

8.3 Notes

- *Contact probing.* It might be tempting to (ab)use `OFFER` and `REQUEST` records or message dependencies to probe the message identifiers of the other groups. If we know the group identifier, the rough content and time interval of the message, we would have a good probability of reversing the message content by brute-forcing the message identifier, but this is not possible because the underlying database implementation separates the message space of different groups. For `OFFER` and `REQUEST` records, the client only looks for messages shared to the remote client; for dependencies, a dependency is only satisfied if the parent message has the specified identifier and is in the same group. It would also not work to change the group identifier in the message to the target group, because then the validation would fail.

[1] https://code.briarproject.org/briar/briar/-/blob/release-1.4.20/bramble-core/src/main/java/org/briarproject/bramble/sync/SyncRecordReaderImpl.java#L123

[2] https://code.briarproject.org/briar/briar/-/blob/release-1.4.20/bramble-core/src/main/java/org/briarproject/bramble/sync/MessageFactoryImpl.java#L57

Chapter 9

Introduction Client

In Briar, the introduction client is a BSP client that allows a Briar user to introduce two contacts to each other. Recall that a BSP client is an application component that uses Bramble Synchronisation Protocol (BSP) to synchronize data. A Briar user can act as an introducer by sending requests to two of its contacts (introducees). If both introducees accept the introduction, they can establish a shared secret key and exchange contact information by communicating with the introducer via the introduction client. Readers may also refer to the specification of the introduction client [15].

9.1 The protocol

All labels in the introduction client except `AUTHOR_ID` are of the form `"org.briarproject.briar.introduction/"+LABEL_NAME`. As an exception, we have `AUTHOR_ID = "org.briarproject.bramble/AUTHOR_ID"`.

Like in the contact exchange protocol (Section 6.1), a hardwired mimimum reasonable time `MIN_REASONABLE_TIME_MS = 1,609,459,200,000` (which represents 1 Jan. 2021, 00:00:00 UTC) is in place to avoid too old timestamps from being negotiated.

Each participant has a nickname `name`, a signing key pair $(\mathsf{spk}, \mathsf{ssk})$, and a unique identifier $\mathtt{id} = \mathtt{hash}(\mathtt{AUTHOR_ID}, \mathtt{int32}(1), \mathtt{utf\text{-}8}(\mathtt{name}), \mathsf{spk})$. A user knows the nickname, the signing public key and the identifier of its contacts.

Suppose that a user, as an introducer, would like to introduce two of the contacts to each other. The introducer compares the lexicographical order of the identifiers to decide who takes the role of each introducee, with the smaller one being Alice and the other being Bob. We use subscripts `I`, `A`, `B` to denote the values of or computed by the introducer, Alice, and Bob, respectively.

Messages are sent and received in this protocol via BSP, which runs over an encrypted and authenticated channel via Bramble Transport Protocol (BTP). The message body is a Bramble Data Format (BDF) list containing the message type, the session id $\texttt{hash}(\texttt{SESSION_ID}, \texttt{id}_\texttt{I}, \texttt{id}_\texttt{A}, \texttt{id}_\texttt{B})$, the identifier of the previous message, and other elements shown in the protocol description. We do not write down the complete record in the protocol description for simplicity.

The introducer fist sends a `REQUEST` message to each introducee, which contains the nickname and signing public key of its peer, and a text string (max. 31KiB) explaining the reason for introduction. Each introducee computes locally the identifier of the remote peer and decide on taking the role of Alice and Bob, and chooses to accept or decline the request.

Figure 9.1 shows the protocol of Briar introduction client when both introducees accepts the introduction. Each introducee sends an `ACCEPT` message to the introducer, which contains its ephemeral public key, timestamp, and transport properties. The introducer forwards the `ACCEPT` message to the other introducee.

In the case where one or more introducee declines, each declining introducee sends a `DECLINE` message to the introducer, which forwards the message and resets the state. The introducee receiving a `DECLINE` message should also reset the state if she has not already done so.

If both introducees accept and have received an `ACCEPT` message, then they would compute the shared secret key from the ephemeral key exchange. Each introducee then derives a MAC key `ck` from the shared key, uses the MAC key to authenticate the contact information `tomac`, uses the long-term signing key `ssk` to sign a nonce `n` derived from the mac key, and sends an `AUTH` message containing the message authentication code `h` and the signature `sig` to the introducer, who forwards it to the other introducee.

After sending and receiving the forwarded `AUTH` message, each introducee checks its validity and adds the remote introducee as a contact if it is valid. The introducee then sends an `ACTIVATE` message to the introducer that forwards it to the other introducee, and waits for the remote `ACTIVATE` message to be forwarded. On receiving the remote `ACTIVATE` message, the introducee checks that it is valid and sets the transport keys as active if necessary.

If the introducer aborts, possibly for a bad protocol state or wrong message format, then she would send an `ABORT` message to each of the introducee, who should also abort. An introducee that aborts would also send an `ABORT` message to the introducer, who forwards the `ABORT` message to the other introducee and aborts.

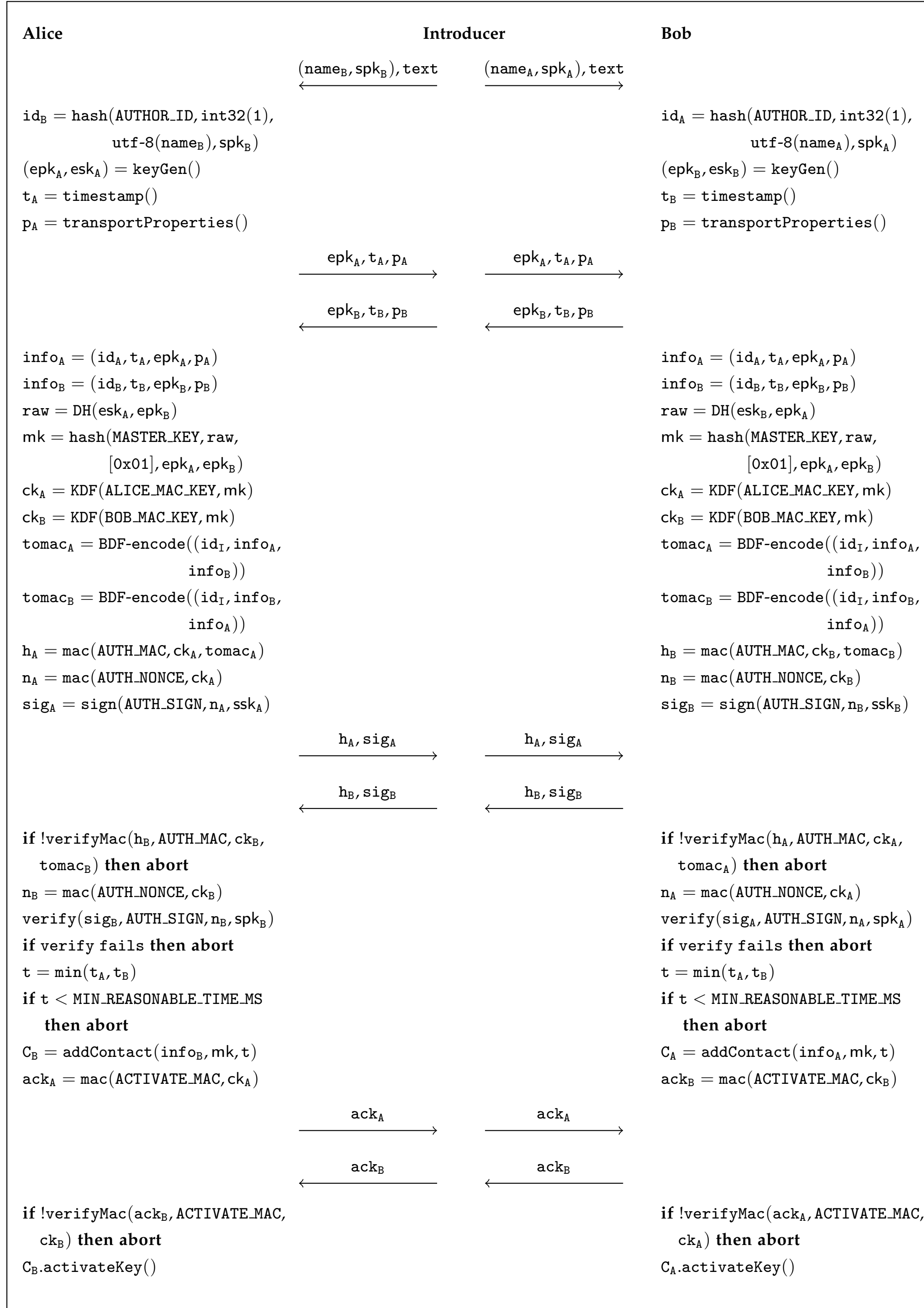

Figure 9.1: A protocol run of the Briar introduction client, where both introducees accept the introduction.

9.2 Active man-in-the-middle attack

As there is no out-of-band verification for the public keys (and thus identities) of the introducees, the introducer could launch a man-in-the-middle attack. However, because of the non-zero check on X25519 key exchange and the inclusion of key pairs into the hash input for key derivation, the introducer can only derive different master keys for the introducees, which is not preventable without introducing further ways of verification. This attack model is realistic as Briar is designed for high-risk individuals (e.g., protestors), so an infiltrator could first introduce two peers to each other with different master keys known to the infiltrator, and subsequently conduct a man-in-the-middle attack on the peers' communicatons.

In Briar, a trust indicator is shown beside each contact, which is `VERIFIED` (three green bars) for contacts added by BQP, and `UNVERIFIED` (two orange bars) for contacts added by all other methods. The developers also realized the lack of verification possibility for contacts added via introduction, but argued that the verification could be hard to implement, and QR-code based verification methods would still not resolve the fundamental issue here, as the malicious introducer could modify the introducee's negotiation of offline verification on Briar, such that each introducee meets with an attacker-controlled person offline instead of the perceived remote introducee.[1]

While out-of-band verifications could also be susceptible to some sophisticated attacks, it could still be a good idea to at least provide some form of out-of-band verifications and let the users decide, instead of relying on users to perform makeshift verifications themselves.

9.3 Notes

- *Similarity with BQP.* The introduction client is, in essence, BQP coupled with the contact exchange protocol. Therefore, the protocol in itself would not be easily exploitable if we assume the security of BQP and the contact exchange protocol.
- *Reflection attack.* A reflection attack is not possible because of the different labels used for the MAC keys of Alice and Bob, and the different order of the data to be authenticated.
- *Role confusion.* A difference between the introduction client and BQP is that how each introducee takes roles is independent from the ephemeral key pair, so it might be tempting for the introducer to modify the name to make both parties think they are Bob (or Alice), but ultimately the signature or MAC value would not pass verification.

[1] https://code.briarproject.org/briar/briar/-/issues/513

- *State machine.* The state machine is much more complicated than the previous protocols mentioned but seems well-formed.
- *Formal verification.* As expected, a simplified version of the introduction protocol in Tamarin (Appendix A.3) fails checks for key secrecy, injective agreement and forward security, but satisfies all these properties if we let the signing public keys be distributed by a trusted public-key infrastructure.

Chapter 10

Briar Functionalities

Most of the functionalities in Briar are implemented as BSP clients, which are at the topmost level of the protocol stack. Because of the asynchronous nature of Briar, these functionalities can be rather constrained and sometimes behave differently from other messaging applications. The Briar specification provides documentations for most of the BSP clients used in Briar.[1]

10.1 Transport key agreement client

The transport key agreement client is a BSP client that establishes keys for newly added transports [17]. As Briar uses separate key sets for different transports, when a new transport is added or supported by the peers after adding contact, it is necessary for the peers to agree on new keys for it.

Basically, each party first generates an ephemeral key pair and sends the ephemeral public key to its peer via BSP, which is again over BTP rotation mode on some existing connection. The parties use the lexicographic order of the ephemeral public keys to assign the roles of Alice and Bob. We denote the ephemeral key pairs of Alice and Bob as $(\mathsf{epk}_\mathtt{A}, \mathsf{esk}_\mathtt{A})$ and $(\mathsf{epk}_\mathtt{B}, \mathsf{esk}_\mathtt{B})$, respectively. Each party calculates the X25519 key exchange of the local private key with the remote public key, $\mathtt{raw} = \mathtt{DH}(\mathsf{esk}_\mathtt{A}, \mathsf{epk}_\mathtt{B}) = \mathtt{DH}(\mathsf{esk}_\mathtt{B}, \mathsf{epk}_\mathtt{A})$. The raw secret is hashed together with the label and the ephemeral public keys to produce the master key mk, which is then used to derive keys for the new transport.

Assuming that the underlying protocols are secure and the state machine of the transport key agreement client is correct, the key agreement process should be secure and provide forward security.

[1] https://code.briarproject.org/briar/briar-spec/-/tree/master/clients

10.2 Messaging

Two peers use a dedicated BSP group to synchronize private messages between them. To this end, each peer maintains a separate BSP client, called the messaging client, for each of its contacts. The client identifier is `"org.briarproject.briar.messaging"`, and the group descriptor is a BDF list (cast to a byte array) of sorted author identfiers of the two peers (see also Chapter 8). The messages are of the type `PRIVATE_MESSAGE` or `ATTACHMENT`. A `PRIVATE_MESSAGE` message contains the UTF-8 encoding of a text message (max. 30KiB after encoding), up to ten optional attachment headers, and an optional auto-delete timer. An attachment header consists of the message identifier of the respective attachment and the content type. An `ATTACHMENT` message contains of a byte array of the attachment and the content type. Note that Briar only supports attachments for private messages, not for private groups, forums or blogs, and the security implication of attachments is (recently) included in the Briar threat model [11] but not yet analyzed.

For each incoming message, the client checks that it is correctly formed, and that the timestamp is not too far in the future (max. 24 hours from now). There are no additional integrity checks involved, such as signatures. Messages are sorted according to their timestamps on display. The client tracks the lastest timestamp `t` among all conversations (messaging, group invitation, or sharing) with the remote peer, and sets the timestamp for the next outgoing message to be the maximum of the system clock and `t+1`.

Notes

- *Network attacks.* Assuming the security of BSP layered on BTP, the attacker can only drop messages or reorder the sequence of the message; in the latter case, the messages would eventually be displayed in the correct order. The attacker cannot drop parts of some particular message because no prefix of the encoded message is valid.
- *Message collision.* Two messages in the same group would have the same message identifier if they have the same timestamp and content (see Chapter 8). The direction of the message is irrelevant. In this case, BSP would think the two messages are the same and rejects the later message. This should be rather unlikely to happen in an honest execution. The developers noticed this and suggested incorporating a salt into the message or at the sync layer.[2]
- *Out-of-order timestamps.* A malicious peer may set arbitrary (valid) timestamps for her messages, but they would eventually be displayed in one fixed order. The sorting algorithm used in Briar is stable.

[2] https://code.briarproject.org/briar/briar/issues/1907

- *Attachment probing.* The messaging client explicitly checks whether an attachment header refers to an attachment in the same group, so we can not probe if an attachment is sent between two other parties.

10.3 Private groups, forums, and blogs

Private groups, forums, and blogs are quite similar except from how their members are controlled. As quoted from Briar FAQ, "A private group is a group chat where the admin decides who to invite. A forum is similar, but anyone can invite new members. A blog is a bit like a Telegram channel: you can write posts that all your contacts can see, and they can reshare individual posts or invite their contacts to subscribe to your feed."[3]

We take private groups as an example. The private group client is used to synchronize messages in the private group among devices. The client identifier is `"org.briarproject.briar.privategroup"`, and the group descriptor is a BDF list (cast to a byte array) of the creator information (format version, name, and sigining public key), group name, and a 32-byte salt. The message types include `JOIN` and `POST`. A `JOIN` message includes the member information, the invite token, and a member signature that includes the group identifier, timestamp, member information, and the group invite. The invite token is obtained using an additional private group sharing client between the creator and the invitee, which (in a simplified view) incorporates the invite timestamp, identifiers of the creator and the invitee, and the group identifier. A `POST` message includes the member information, an optional parent message identifier, the identifier of the previous message, text, and a signature additionally covering the group identifier and timestamp.

The forum client is similar to the private group client. It only has one message type, which is the same as the private group client except from not including the message type and the previous message identifier. The blog client only allows the blog author to create posts or comments, but the blog author may reblog posts or comments from other blogs, which are wrapped in the author's blog and should also be validated by subscribers.

10.3.1 Message duplication attack

The Bramble Synchronisation Protocol (BSP) synchronizes messages between clients within the same group, where each message has a unique identifier, which is a labeled hash digest that depends on the message format version, group identifier, timestamp, and the message body (see Chapter 8). BSP treats messages with the same identifier as duplicates, which prevents BSP clients from handling the same message multiple times. However, by

[3] https://code.briarproject.org/briar/briar/wikis/FAQ

tweaking the message encoding, it is possible for a malicious member or subscriber of a private group/forum/blog to duplicate messages from any author in the BSP group as many times as the adversary desires, without practical limit.

As mentioned in Section 2.4, it is easy to find different Binary Data Format (BDF) encodings that decode to the same data. Possible methods of producing different BDF encodings include leveraging integer overflows, increasing the length-of-length field, and inserting unmappable bytes into the UTF-8 encoding of strings (which is ignored by the decoder).

A malicious member could use these methods to alter the encoding of any message to produce a message with the same content and a different message body. By definition, the message identifier will also change for the modified message. As a result, the modified message would be perceived as a valid new message and shared within the private group/forum/blog along with the original message.

Take private groups as an example. The private group client produces a message by first computing the signature, and then encoding the message body as a BDF list. We can obtain a new message with a different message identifier and a valid signature by setting the message type as a very large number (e.g., $2^{32} + 1$), which is regarded as valid because of integer overflow. We can also insert 0x80 arbitrarily into the encoded text, or use `INT_16` instead of `INT_8` to encode the author format version, so on and so forth.

We include example code snippets for modifying private group posts (Listing B.1), forum posts (Listing B.2), and blog posts (Listing B.3). They do not cover all possible ways of duplicating messages.

A malicious insider can share the modified messages within the group. As long as the original message is valid, the modified messages would be valid. Members in the group will treat these two messages as different and display both messages in the application. For this attack to work in private groups, note that the private group client allows multiple messages with the same previous message id.

This attack can be used to create disruption to private groups/forums/blogs without exposing the identity of the culprit. Duplicate messages may also genuinely confuse users that only synchronize messages periodically, or are new to an existing private group/forum/blog.

Fixes with backward compatibility would be to compare the serialized message with its canonical serialization, look for possible overflows, and throw an exception for UTF-8 decoding errors. In the long run, it is probably a better idea to compute the signature only after serialization.

10.3.2 Notes

- *Circular dependency.* It might be tempting to create a circular structure in the thread, such that the application would run out of stack memory and crash when performing a depth-first search in order to render the threads. However, this is not possible because finding such a circle would break the underlying hash function used to compute message identifiers.
- *Timestamp overflow.* The private group client checks whether the timestamp is too far from the future, preventing a timestamp overflow possibly caused by a malicious member in the group, which would block the victim from communicating with other parties in the group.
- *Leaving a group.* A group member may announce to its peers or the creator that she is leaving the group, in which case the peers or the creator would stop sharing messages to the member. However, there is no real way of definitively leaving a group, as the leave message is not broadcast and can be reverted, and the invite token can be replayed.

Chapter 11

Discussion

We can attribute the overall security of Briar to the following factors:

- *Security and privacy by design.* Briar is a good example of security and privacy by design. Briar only implements a small subset of functions that would typically be available to most modern messaging applications. This significantly narrows Briar's attack vector. For example, Briar offers no security backup or account transfer as of today[1] in order to provide forward security, which is rather uncommon for messaging applications. In contrast, security backup is a common source of vulnerabilities in other messaging applications (e.g., Matrix [2]). It should be noted that the functionalities provided by Briar are sufficient for higher-risk users to perform basic activities.
- *Good cryptography practices.* The developers of Briar are evidently experienced with cryptography and peer-to-peer messaging. This can be seen from their wise choice of cryptographic primitives, good domain separations, and well-designed interaction and layering of a complex set of protocols. The redundancy in Briar's protocols weakens our attacks, and the versioning system in Briar smooths security updates.

On the other hand, the issues we identified also share some common pitfalls:

- *Insider attacks.* Three of the four vulnerabilities we identified are mounted by malicious insiders; that is, the attacker has to either be a contact of the victim, or in the same group as the victim. These are practical threats given the target group of Briar, but Briar seem to have paid less attention to them.
- *Input validation.* The peer-to-peer nature of Briar puts a heavier burden on Briar clients for input validation, from which we discovered two vulnerabilities.

[1] https://code.briarproject.org/briar/briar/-/issues/887

- *Encoding.* We identified an attack that originates from the encoding flexibility in BDF, and a potential issue that involves Base32 encoding. It is not the first time that Briar reported a bug related to its custom encoding format.[2] This may warn future developers against "rolling your own encoding" in addition to "rolling your own crypto."

- *Usable security.* From the usability perspective, Briar may need more work to nudge users to improve their security, especially for users that are not knowledgeable enough in security and privacy. For example, the current password policy in Briar is based on the number of unique characters in passwords, which accepts passwords with at least 6 unique characters, and flags passwords with at least 9 unique characters as "quite strong" (green bar). A look at the 10 most common passwords in the infamous RockYou leak shows that 9 of them would be accepted by Briar, and one (`"123456789"`) is even labeled as quite strong. While Briar has no central servers, offline password recovery is still a tangible threat given the high risk of compelled access for Briar's target user groups.

[2] https://code.briarproject.org/briar/briar/-/issues/1277

Chapter 12

Conclusion

In this project, we examined Briar's use of cryptography and analyzed the security of protocols in Briar. First, we provided a detailed documentation of Briar's protocol stack, which may be useful for future security analyses of Briar. Second, we gave a series of arguments in favor of Briar's security; the overall positive result partly supports Briar's security claims and may prompt more people to adopt Briar. Finally, we discovered several issues in Briar's protocol design and implementaion, and suggested possible fixes.

We conclude that Briar's use of cryptography is in general secure, and that Briar is suitable for higher-risk individuals from the cryptographic point of view. However, due to the author's lack of experience and time, we would still suggest another round of analysis on Briar's use of cryptography.

Some directions for future work include 1) formally proving the security of Briar's cryptographic protocols, especially BQP, BHP, and BTP, 2) exploring the possibility of cross-protocol interaction attacks, 3) investigating to what extent Briar protects the privacy of its users, including the feasibility of traffic analysis and tracing, and 4) analyzing the security of Briar Mailbox.

Appendix A

Tamarin code

A.1 Tamarin code for BQP

Listing A.1: BQP.spthy

```
theory BQP
begin

builtins: diffie-hellman, hashing

functions: commit/1, MAC/2

rule Init:
  let epkA = 'g'^~eskA
      epkB = 'g'^~eskB
  in
  [ Fr(~eskA), Fr(~eskB) ]
  -->
  [ Alice_Init($A, ~eskA, epkA, $B, commit(epkB)),
    Bob_Init($B, ~eskB, epkB, $A, commit(epkA)) ]

rule Alice_Sends_Pk:
  [ Alice_Init($A, ~eskA, epkA, $B, commB) ]
  -->
  [ Out(epkA),
    Alice_Awaits_Pk($A, ~eskA, epkA, $B, commB) ]

rule Bob_Derives_Key:
  let commA = commit(epkA)
      raw = epkA^~eskB
      s = h(<'SHARED_SECRET', raw, epkA, epkB>)
      ck = MAC('CONFIRMATION_KEY', s)
      confa = MAC(<'CONFIRMATION_MAC', epkA, epkB>, ck)
      mk = MAC('MASTER_KEY', s)
  in
  [ Bob_Init($B, ~eskB, epkB, $A, commA),
    In(epkA) ]
  --[ Neq(raw, 1), RunningR($B, $A, mk) ]->
  [ Out(epkB),
    Bob_Awaits_Conf($B, ~eskB, epkB, $A, epkA, s, ck, mk, confa) ]

rule Alice_Derives_Key:
  let commB = commit(epkB)
      raw = epkB^~eskA
      s = h(<'SHARED_SECRET', raw, epkA, epkB>)
      ck = MAC('CONFIRMATION_KEY', s)
      confa = MAC(<'CONFIRMATION_MAC', epkA, epkB>, ck)
      mk = MAC('MASTER_KEY', s)
  in
  [ Alice_Awaits_Pk($A, ~eskA, epkA, $B, commB),
```

```
46        In(epkB) ]
47      --[ Neq(raw, 1), RunningI($A, $B, mk) ]->
48      [ Out(confa),
49        Alice_Awaits_Conf($A, ~eskA, epkA, $B, epkB, s, ck, mk) ]
50
51    rule Bob_Receives_Conf:
52      let confb = MAC(<'CONFIRMATION_MAC', epkB, epkA>, ck)
53      in
54      [ Bob_Awaits_Conf($B, ~eskB, epkB, $A, epkA, s, ck, mk, confa),
55        In(confa) ]
56      --[ FinishedR($B),
57          SecretR($A, $B, mk),
58          CommitR($B, $A, mk)
59        ]->
60      [ Out(confb) ]
61
62    rule Alice_Receives_Conf:
63      let confb = MAC(<'CONFIRMATION_MAC', epkB, epkA>, ck)
64      in
65      [ Alice_Awaits_Conf($A, ~eskA, epkA, $B, epkB, s, ck, mk),
66        In(confb) ]
67      --[ FinishedI($A),
68          SecretI($A, $B, mk),
69          CommitI($A, $B, mk)
70        ]->
71      []
72
73    restriction Inequality:
74      "All x #i. Neq(x,x) @ #i ==> F"
75
76    lemma executableI:
77    exists-trace "Ex #i A. FinishedI(A) @ i"
78
79    lemma executableR:
80    exists-trace "Ex #i B. FinishedR(B) @ i"
81
82    lemma key_secrecyI:
83    "All #i A B k.
84     (SecretI(A, B, k) @ i) ==> not (Ex #j. K(k) @ j)"
85
86    lemma key_secrecyR:
87    "All #i A B k.
88     (SecretR(A, B, k) @ i) ==> not (Ex #j. K(k) @ j)"
89
90    lemma agreementR:
91    "All #i A B k.
92      (CommitR(B, A, k) @ i)
93      ==> (Ex #j. RunningI(A, B, k) @ j
94          & j < i
95          & not (Ex A2 B2 #i2. CommitR(B2, A2, k) @i2
96                & not (#i2 = #i)
97                )
98          )"
99
100   lemma agreementI:
101   "All #i A B k.
102     (CommitI(A, B, k) @ i)
103     ==> (Ex #j. RunningR(B, A, k) @ j
104         & j < i
105         & not (Ex A2 B2 #i2. CommitI(A2, B2, k) @i2
106               & not (#i2 = #i)
107               )
108         )"
109
110   end
```

A.2 Tamarin code for BHP

Listing A.2: BHP.spthy

```
1   theory BHP
2   begin
```

```
3
4   builtins: diffie-hellman, symmetric-encryption, hashing
5
6   functions: MAC/2
7
8   rule genkey:
9   let pkA = 'g'^~kA in
10     [ Fr(~kA) ]
11     -->
12     [ !Key($A, ~kA),
13       !Pk($A, pkA),
14       Out(pkA) ]
15
16  rule Alice_Sends_Pk:
17    let epkA = 'g'^~eskA
18        raw = hpkB^~hskA
19        hsk = h(<'STATIC_MASTER_KEY', raw, hpkA, hpkB>)
20    in
21    [ Fr(~eskA),
22      !Key($A, ~hskA),
23      !Pk($A, hpkA),
24      !Pk($B, hpkB)
25    ]
26    --[ Neq(raw, 1) ]->
27    [ Out(senc(epkA, hsk)),
28      Alice_Awaits_Pk($A, ~hskA, hpkA, ~eskA, epkA, $B, hpkB, hsk) ]
29
30  rule Bob_Responds_Pk:
31    let epkB = 'g'^~eskB
32        raw = hpkA^~hskB
33        hsk = h(<'STATIC_MASTER_KEY', raw, hpkA, hpkB>)
34        epkA = sdec(CepkA, hsk)
35        raw1 = hpkA^~eskB
36        raw2 = epkA^~hskB
37        mk = h(<'MASTER_KEY', raw, raw1, raw2, hpkA, hpkB, epkA, epkB>)
38    in
39    [ Fr(~eskB),
40      !Key($B, ~hskB),
41      !Pk($A, hpkA),
42      !Pk($B, hpkB),
43      In(CepkA)
44    ]
45    --[ Neq(raw, 1), Neq(raw1, 1), Neq(raw2, 1),
46        RunningR($B, $A, mk) ]->
47    [ Out(senc(epkB, hsk)),
48      Bob_Awaits_Proof($B, ~hskB, ~eskB, epkB, $A, hpkA, epkA, hsk, mk) ]
49
50  rule Alice_Derives_Key:
51    let epkB = sdec(CepkB, hsk)
52        raw = hpkB^~hskA
53        raw1 = epkB^~hskA
54        raw2 = hpkB^~eskA
55        mk = h(<'MASTER_KEY', raw, raw1, raw2, hpkA, hpkB, epkA, epkB>)
56        proofA = MAC('ALICE_PROOF', mk)
57    in
58    [ Alice_Awaits_Pk($A, ~hskA, hpkA, ~eskA, epkA, $B, hpkB, hsk),
59      In(CepkB)
60    ]
61    --[ Neq(raw, 1), Neq(raw1, 1), Neq(raw2, 1),
62        RunningI($A, $B, mk) ]->
63    [ Out(senc(proofA, hsk)),
64      Alice_Awaits_Proof($A, ~hskA, hpkA, ~eskA, epkA, $B, hpkB, epkB, hsk, mk) ]
65
66  rule Bob_Sends_Proof:
67    let proofA = sdec(CpA, hsk)
68        proofB = MAC('BOB_PROOF', mk)
69    in
70    [ Bob_Awaits_Proof($B, ~hskB, ~eskB, epkB, $A, hpkA, epkA, hsk, mk),
71      In(CpA)]
72    -->
73    [ Out(senc(proofB, hsk)),
74      Bob_Verifies_Proof($B, ~hskB, ~eskB, epkB, $A, hpkA, epkA, hsk, mk, proofA) ]
75
76  rule Bob_Finishes:
```

```
77    [ Bob_Verifies_Proof($B, ~hskB, ~eskB, epkB, $A, hpkA, epkA, hsk, mk, proofA) ]
78    --[ Eq(proofA, MAC('ALICE_PROOF', mk)),
79        FinishedR($B),
80        SecretR($A, $B, mk),
81        CommitR($B, $A, mk)
82      ]->
83    []
84
85  rule Alice_Finishes:
86    let proofB = sdec(CpB, hsk) in
87    [ Alice_Awaits_Proof($A, ~hskA, hpkA, ~eskA, epkA, $B, hpkB, epkB, hsk, mk),
88      In(CpB) ]
89    --[ Eq(proofB, MAC('BOB_PROOF', mk)),
90        FinishedI($A),
91        SecretI($A, $B, mk),
92        CommitI($A, $B, mk)
93      ]->
94    []
95
96  rule Compromise:
97  [ !Key(A, ~skA) ]
98  --[ Compromised(A) ]->
99  [ Out(~skA) ]
100
101 restriction equality:
102 "All x y #i. Eq(x,y) @i ==> x = y"
103
104 restriction Inequality:
105   "All x #i. Neq(x,x) @ #i ==> F"
106
107 lemma executableI:
108 exists-trace "Ex #i A. FinishedI(A) @ i & not (Ex #j B. Compromised(B)@j)"
109
110 lemma executableR:
111 exists-trace "Ex #i B. FinishedR(B) @ i & not (Ex #j B. Compromised(B)@j)"
112
113 lemma key_secrecyI:
114 "All #i A B k.
115  (SecretI(A, B, k) @ i &
116   not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
117   ==> not (Ex #j. K(k) @ j)"
118
119 lemma key_secrecyR:
120 "All #i A B k.
121  (SecretR(A, B, k) @ i &
122   not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
123   ==> not (Ex #j. K(k) @ j)"
124
125 lemma agreementR:
126 "All #i A B k.
127   (CommitR( B, A, k) @ i &
128    not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
129    ==> (Ex #j. RunningI(A, B, k) @ j & j < i)"
130
131 lemma agreementI:
132 "All #i A B k.
133   (CommitI( A, B, k) @ i &
134    not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
135    ==> (Ex #j. RunningR(B, A, k) @ j & j < i)"
136
137 lemma injectiveR:
138   "All A B k #i.
139     (CommitR(B, A, k) @i &
140     not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
141     ==> (Ex #j. RunningI(A, B, k) @j
142         & j < i
143         & not (Ex A2 B2 #i2. CommitR(B2, A2, k) @i2
144                            & not (#i2 = #i)))"
145
146 lemma injectiveI:
147   "All A B k #i.
148     (CommitI(A, B, k) @i &
149     not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
150     ==> (Ex #j. RunningR(B, A, k) @j
```

```
        & j < i
        & not (Ex A2 B2 #i2. CommitI(A2, B2, k) @i2
                           & not (#i2 = #i)))"

lemma secrecy_PFS_R:
 " All A B k #i.
    SecretR(A, B, k) @i &
      not ((Ex #l. (Compromised(A) @ l & l<i )) | (Ex #m. (Compromised(B) @ m & m
           < i)) )
    ==> not (Ex #j. K(k)@j )"

lemma secrecy_PFS_I:
 " All A B k #i.
    SecretI(A, B, k) @i &
      not ((Ex #l. (Compromised(A) @ l & l<i )) | (Ex #m. (Compromised(B) @ m & m
           < i)) )
    ==> not (Ex #j. K(k)@j )"

end
```

A.3 Tamarin code for Briar introduction

Listing A.3: introduction.spthy

```
// grossly simplified
theory BIC
begin

builtins: diffie-hellman, signing, hashing

functions:  MAC/2

rule genkey:
   [ Fr(~kA) ]
   -->
   [ !Key($A, ~kA),
     !Pk($A, pk(~kA)),
     Out(pk(~kA)) ]

rule Introducer_Sends_Request:
  [ !Pk($A, spkA),
    !Pk($B, spkB) ]
  -->
  [ Out(spkA), Out(spkB) ]


rule Alice_Accepts:
  let epkA = 'g'^~eskA in
  [ Fr(~eskA),
    In(spkB)
    //!Pk($B, spkB)
    ]
  -->
  [ Out(epkA),
    Alice_Awaits_Epk($A, ~eskA, epkA, $B, spkB) ]

rule Bob_Accepts:
  let epkB = 'g'^~eskB in
  [ Fr(~eskB),
    In(spkA)
    //!Pk($A, spkA)
  ]
  -->
  [ Out(epkB),
    Bob_Awaits_Epk($B, ~eskB, epkB, $A, spkA) ]

rule Alice_Authenticates:
  let raw = epkB^~eskA
      mk = h(<'MASTER_KEY', raw, epkA, epkB>)
      ckA = MAC('ALICE_MAC_KEY', mk)
      ckB = MAC('BOB_MAC_KEY', mk)
      hA = MAC(<'AUTH_MAC', epkA, epkB>, ckA)
```

```
49          nA = MAC('AUTH_NONCE', ckA)
50          sigA = sign(<'AUTH_SIGN', nA>, ~sskA)
51      in
52      [ Alice_Awaits_Epk($A, ~eskA, epkA, $B, spkB),
53        !Key($A, ~sskA),
54        In(epkB) ]
55      --[ Neq(raw, 1), RunningI(pk(~sskA), spkB, mk)
56          //RunningI($A, $B, mk)
57        ]->
58      [ Out(<hA, sigA>),
59        Alice_Awaits_Auth($A, ~sskA, ~eskA, epkA, $B, spkB, epkB, mk, ckA, ckB) ]
60
61    rule Bob_Authenticates:
62      let raw = epkA^~eskB
63          mk = h(<'MASTER_KEY', raw, epkA, epkB>)
64          ckA = MAC('ALICE_MAC_KEY', mk)
65          ckB = MAC('BOB_MAC_KEY', mk)
66          hB = MAC(<'AUTH_MAC', epkA, epkB>, ckB)
67          nB = MAC('AUTH_NONCE', ckB)
68          sigB = sign(<'AUTH_SIGN', nB>, ~sskB)
69      in
70      [ Bob_Awaits_Epk($B, ~eskB, epkB, $A, spkA),
71        !Key($B, ~sskB),
72        In(epkA) ]
73      --[ Neq(raw, 1), RunningR(pk(~sskB), spkA, mk)
74          //RunningR($B, $A, mk)
75        ]->
76      [ Out(<hB, sigB>),
77        Bob_Awaits_Auth($B, ~sskB, ~eskB, epkB, $A, spkA, epkA, mk, ckA, ckB) ]
78
79    rule Alice_Acks:
80      let nB = MAC('AUTH_NONCE', ckB)
81          ackA = MAC('ACTIVATE_MAC', ckA)
82      in
83      [ Alice_Awaits_Auth($A, ~sskA, ~eskA, epkA, $B, spkB, epkB, mk, ckA, ckB),
84        In(<hB, sigB>) ]
85      --[ Eq(MAC(<'AUTH_MAC', epkA, epkB>, ckB), hB),
86          Eq(verify(sigB, <'AUTH_SIGN', nB>, spkB), true) ]->
87      [ Out(ackA),
88        Alice_Awaits_Ack($A, ~sskA, ~eskA, epkA, $B, spkB, epkB, mk, ckA, ckB) ]
89
90    rule Bob_Acks:
91      let nA = MAC('AUTH_NONCE', ckA)
92          ackB = MAC('ACTIVATE_MAC', ckB)
93      in
94      [ Bob_Awaits_Auth($B, ~sskB, ~eskB, epkB, $A, spkA, epkA, mk, ckA, ckB),
95        In(<hA, sigA>) ]
96      --[ Eq(MAC(<'AUTH_MAC', epkA, epkB>, ckA), hA),
97          Eq(verify(sigA, <'AUTH_SIGN', nA>, spkA), true) ]->
98      [ Out(ackB),
99        Bob_Awaits_Ack($B, ~sskB, ~eskB, epkB, $A, spkA, epkA, mk, ckA, ckB) ]
100
101   rule Alice_Finishes:
102     [ Alice_Awaits_Ack($A, ~sskA, ~eskA, epkA, $B, spkB, epkB, mk, ckA, ckB),
103       In(ackB) ]
104     --[ Eq(MAC('ACTIVATE_MAC', ckB), ackB),
105         FinishedI(pk(~sskA)), SecretI(pk(~sskA), spkB, mk), CommitI(pk(~sskA), spkB,
            mk)
106         //FinishedI($A), SecretI($A, $B, mk), CommitI($A, $B, mk)
107       ]->
108     []
109
110   rule Bob_Finishes:
111     [ Bob_Awaits_Ack($B, ~sskB, ~eskB, epkB, $A, spkA, epkA, mk, ckA, ckB),
112       In(ackA) ]
113     --[ Eq(MAC('ACTIVATE_MAC', ckA), ackA),
114         FinishedR(pk(~sskB)), SecretR(pk(~sskB), spkA, mk), CommitR(pk(~sskB), spkA,
            mk)
115         //FinishedR($B), SecretR($A, $B, mk), CommitR($B, $A, mk)
116       ]->
117     []
118
119   rule Compromise:
120   [ !Key(A, ~skA) ]
```

```
121 --[ Compromised(pk(~skA)) //Compromised(A)
122   ]->
123 [ Out(~skA) ]
124
125 restriction equality:
126   "All x y #i. Eq(x,y) @i ==> x = y"
127
128 restriction Inequality:
129   "All x #i. Neq(x,x) @ #i ==> F"
130
131
132 lemma executableI:
133 exists-trace "Ex #i A. FinishedI(A) @ i & not (Ex #j B. Compromised(B)@j)"
134
135 lemma executableR:
136 exists-trace "Ex #i B. FinishedR(B) @ i & not (Ex #j B. Compromised(B)@j)"
137
138 lemma key_secrecyI:
139 "All #i A B k.
140  (SecretI(A, B, k) @ i &
141   not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
142   ==> not (Ex #j. K(k) @ j)"
143
144 lemma key_secrecyR:
145 "All #i A B k.
146  (SecretR(A, B, k) @ i &
147   not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
148   ==> not (Ex #j. K(k) @ j)"
149
150 lemma agreementR:
151 "All #i A B k.
152   (CommitR( B, A, k) @ i &
153    not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
154    ==> (Ex #j. RunningI(A, B, k) @ j & j < i)"
155
156 lemma agreementI:
157 "All #i A B k.
158   (CommitI( A, B, k) @ i &
159    not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
160    ==> (Ex #j. RunningR(B, A, k) @ j & j < i)"
161
162 lemma injectiveR:
163   "All A B k #i.
164     (CommitR(B, A, k) @i &
165     not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
166     ==> (Ex #j. RunningI(A, B, k) @j
167         & j < i
168         & not (Ex A2 B2 #i2. CommitR(B2, A2, k) @i2
169                            & not (#i2 = #i)))"
170
171 lemma injectiveI:
172   "All A B k #i.
173     (CommitI(A, B, k) @i &
174     not ((Ex #l. Compromised(A) @ l) | (Ex #m. Compromised(B) @ m)))
175     ==> (Ex #j. RunningR(B, A, k) @j
176         & j < i
177         & not (Ex A2 B2 #i2. CommitI(A2, B2, k) @i2
178                            & not (#i2 = #i)))"
179
180 lemma secrecy_PFS_R:
181  " All A B k #i.
182     SecretR(A, B, k) @i &
183       not ((Ex #l. (Compromised(A) @ l & l<i )) | (Ex #m. (Compromised(B) @ m & m
            < i)) )
184     ==> not (Ex #j. K(k)@j )"
185
186 lemma secrecy_PFS_I:
187  " All A B k #i.
188     SecretI(A, B, k) @i &
189       not ((Ex #l. (Compromised(A) @ l & l<i )) | (Ex #m. (Compromised(B) @ m & m
            < i)) )
190     ==> not (Ex #j. K(k)@j )"
191
192 end
```

Appendix B

Code examples for message duplication

Listing B.1: Duplicating private group posts by representing the message type (POST, or 1) with INT_16 instead of INT_8.

```
1  private Message dupGroupPost(Message m) throws FormatException {
2          BdfList body = clientHelper.toList(m.getBody());
3
4          BdfList memberList = body.getList(1);
5          byte[] parentId = body.getOptionalRaw(2);
6          byte[] previousMessageId = body.getRaw(3);
7          String text = body.getString(4);
8          byte[] signature = body.getRaw(5);
9
10         BdfList newBody = BdfList.of(
11                         2023, // placeholder for INT_16
12                         memberList,
13                         parentId,
14                         previousMessageId,
15                         text,
16                         signature
17         );
18
19         byte[] raw = clientHelper.toByteArray(newBody);
20         // LIST_START (0th), INT_16 (1st), value (2nd-3rd), ....
21         raw[2] = 0x00;
22         raw[3] = 0x01;
23         Message newMessage = clientHelper.createMessage(m.
               getGroupId(), m.getTimestamp(), raw);
24         return newMessage;
25 }
```

Listing B.2: Duplicating forum posts by appending 0x80 to the end of the UTF-8 encoded text.

```
1  private Message dupForumPost(Message m, BdfList body) throws
       FormatException {
2
```

```
3          byte[] parent = body.getOptionalRaw(0);
4          BdfList authorList = body.getList(1);
5          String text = body.getString(2);
6          byte[] sig = body.getRaw(3);
7
8          BdfList newBody = BdfList.of(parent, authorList, text + '
               a', sig);
9          byte[] raw = clientHelper.toByteArray(newBody);
10         // ... 'a', RAW_8, RAW_LEN, sig, END
11         int offset = raw.length - sig.length - 4;
12         raw[offset] = (byte) 0x80;
13         Message newMessage = clientHelper.createMessage(m.
               getGroupId(), m.getTimestamp(), raw);
14         return newMessage;
15 }
```

Listing B.3: Duplicating blog posts or comments by overflowing the message type field.

```
1  private Message dupBlogPostOrComment(Message m, BdfList body)
       throws FormatException {
2          BdfList newBody = new BdfList(body);
3          int type = body.getLong(0).intValue();
4          Long newType = 4294967296L + type;
5          newBody.remove(0);
6          newBody.add(0, newType);
7
8          byte[] raw = clientHelper.toByteArray(newBody);
9          Message newMessage = clientHelper.createMessage(m.
               getGroupId(), m.getTimestamp(), raw);
10         return newMessage;
11 }
```

Bibliography

[1] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking Bridgefy. In *CT-RSA*, volume 12704 of *Lecture Notes in Computer Science*, pages 375–398. Springer, 2021.

[2] Martin R Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable cryptographic vulnerabilities in Matrix. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1419–1436. IEEE Computer Society, 2022.

[3] Martin R. Albrecht, Raphael Eikenberg, and Kenneth G. Paterson. Breaking Bridgefy, again: Adopting libsignal is not enough. In *USENIX Security Symposium*, pages 269–286. USENIX Association, 2022.

[4] Cure53, M. Heiderich, A. Aranguren, A. Inführ, F. Fäßler, C. Kean, and N. Kobeissi. *Pentest-Report Briar Project App & Protocol 03.2017*, March 2017. https://cure53.de/pentest-report_briar.pdf.

[5] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.

[6] The Briar Team. *Binary Data Format, version 1*. https://code.briarproject.org/briar/briar-spec/blob/master/BDF.md (retrieved 05-03-2023).

[7] The Briar Team. *Bramble Handshake Protocol, version 0*. https://code.briarproject.org/briar/briar-spec/blob/master/protocols/BHP.md (retrieved 05-03-2023).

[8] The Briar Team. *Bramble QR Code Protocol, version 4*. https://code.briarproject.org/briar/briar-spec/-/blob/master/protocols/BQP.md (retrieved 05-03-2023).

[9] The Briar Team. *Bramble Synchronisation Protocol, version 0*. https://code.briarproject.org/briar/briar-spec/blob/master/protocols/BSP.md (retrieved 05-03-2023).

[10] The Briar Team. *Bramble Transport Protocol, version 4*. https://code.briarproject.org/briar/briar-spec/blob/master/protocols/BTP.md (retrieved 05-03-2023).

[11] The Briar Team. *Briar Threat Model*. https://code.briarproject.org/briar/briar/-/wikis/threat-model (retrieved 05-03-2023).

[12] The Briar Team. *Briar User Manual*. https://briarproject.org/manual/ (retrieved 05-03-2023).

[13] The Briar Team. *Briar Wiki*. https://code.briarproject.org/briar/briar/-/wikis/home (retrieved 05-03-2023).

[14] The Briar Team. *How it works*. https://briarproject.org/how-it-works/ (retrieved 05-03-2023).

[15] The Briar Team. *Introduction Client*. https://code.briarproject.org/briar/briar-spec/-/blob/master/clients/Introduction-Client.md (retrieved 05-03-2023).

[16] The Briar Team. *A Quick Overview of the Protocol Stack*. https://code.briarproject.org/briar/briar/-/wikis/A-Quick-Overview-of-the-Protocol-Stack (retrieved 05-03-2023).

[17] The Briar Team. *Transport Key Agreement Client*. https://code.briarproject.org/briar/briar-spec/blob/master/clients/Transport-Key-Agreement-Client.md (retrieved 05-03-2023).

[18] The Tamarin Team. *Tamarin-Prover Manual*. https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf (retrieved 05-03-2023).