Formalizing and Verifying the Security Protocols from the Noise Framework

Bachelor Thesis
A. Suter-Dörig
November 14, 2018

Advisors: Dr. L. Hirschi, Dr. R. Sasse
Professor: Prof. Dr. D. Basin
Department of Computer Science, ETH Zürich
Abstract

The potential use cases for cryptographic protocols vary in many ways. Adapting to these different situations requires either a versatile general purpose protocol or many different protocols tailored to the respective ways in which they are deployed.

While the former approach suffers from its own issues, the latter often leads to 'homegrown' protocols. Since cryptography is notoriously difficult to get right, the lack of scrutiny of such protocols usually receive often leads to vulnerabilities. The Noise protocol framework is an attempt to mitigate this problem.

The Noise framework provides an easy to use infrastructure to deploy Diffie-Hellman based key agreement and encrypted communication protocols. It offers a simple language to describe such protocols. In combination with five validity rules, this language supposedly guarantees certain security properties on the resulting protocols. Additionally, the framework specifies a number of such protocols and lists some of their supposed properties.

The contribution outlined in this work is a tool which generates symbolic models for protocols given as Noise patterns and verifies a host of properties on these models using the Tamarin prover.

This work also contains the results obtained by running the aforementioned tool on all protocols named in the Noise specification. While mostly agreeing with the properties asserted by the Noise specification, these results cover more properties and more protocols than the specification. This verification process has also brought to light some ambiguities in the definitions of the properties used in the Noise specification.
Acknowledgements

This thesis could not have come together in its current form without the help of many. While trying to keep this succinct, I would like to express my wholehearted gratitude to all involved.

Firstly, my thanks go to my advisors Lucca Hirschi and Ralf Sasse. Our weekly discussions have proven to be an invaluable help and were always insightful.

My appreciation also goes out to Christoph Sprenger for arranging this thesis for me, and Prof. David Basin for the opportunity of doing it in his group.

Thirdly, I thank my family for their support and Morina Suter also for the excuse for taking Saturdays off in the most hectic of times.

And finally, I am apparently contractually required to thank Pascal Störzbach for the use of his printer. While I am certain the above is the full extent of my legal obligations towards Pascal, I would also like to amend it with thanks for his always open ear and willingness to help.
Contents

1 Introduction 1
   1.1 Background .......................................... 1
   1.1.1 The Noise Framework .............................. 1
   1.1.2 Symbolic Verification .............................. 2
   1.2 Related Work ......................................... 3
   1.3 Contributions ........................................ 4

2 Noise 7
   2.1 Diffie-Hellman ....................................... 7
   2.2 Noise Patterns ...................................... 8
      2.2.1 Syntax .......................................... 8
      2.2.2 Pattern Validity ................................ 12
   2.3 Semantics ............................................ 13
      2.3.1 State ........................................... 13
      2.3.2 Functions ...................................... 14
      2.3.3 Transition ..................................... 16

3 Tamarin 23
   3.1 Attacker ............................................. 23
   3.2 Terms ................................................ 24
   3.3 Diffie-Hellman Support .............................. 26
   3.4 Multiset Rewriting .................................. 27
      3.4.1 Facts .......................................... 27
      3.4.2 Rewriting Rules ................................. 28
   3.5 Let Binding .......................................... 29
   3.6 Property Specification .............................. 30
      3.6.1 Action Facts ...................................... 30
      3.6.2 Specification Language ......................... 31
## Contents

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>3.6.3 First-Order Logic Formula</td>
</tr>
<tr>
<td>32</td>
<td>3.7 Oracle</td>
</tr>
<tr>
<td>35</td>
<td>4 Modelling</td>
</tr>
<tr>
<td>35</td>
<td>4.1 Generating Protocol Patterns</td>
</tr>
<tr>
<td>36</td>
<td>4.2 Generating Intermediate Language</td>
</tr>
<tr>
<td>38</td>
<td>4.3 Generating TAMARIN Models</td>
</tr>
<tr>
<td>39</td>
<td>4.3.1 Parsing Intermediate Language</td>
</tr>
<tr>
<td>39</td>
<td>4.3.2 Transform to Static Single Assignment Form (SSA)</td>
</tr>
<tr>
<td>39</td>
<td>4.3.3 Improve Readability</td>
</tr>
<tr>
<td>41</td>
<td>4.3.4 Optimize Decryption</td>
</tr>
<tr>
<td>42</td>
<td>4.3.5 Generate Blocks</td>
</tr>
<tr>
<td>43</td>
<td>4.3.6 Emit Equations</td>
</tr>
<tr>
<td>43</td>
<td>4.3.7 Emit Key- and Session Management Rules</td>
</tr>
<tr>
<td>44</td>
<td>4.3.8 Emit Protocol Rules</td>
</tr>
<tr>
<td>44</td>
<td>4.3.9 Emit Lemmas</td>
</tr>
<tr>
<td>44</td>
<td>4.4 Running TAMARIN</td>
</tr>
<tr>
<td>44</td>
<td>4.4.1 Oracle</td>
</tr>
<tr>
<td>45</td>
<td>4.5 Result Presentation</td>
</tr>
<tr>
<td>46</td>
<td>4.6 Model Design</td>
</tr>
<tr>
<td>46</td>
<td>4.6.1 Invalid DH keys</td>
</tr>
<tr>
<td>47</td>
<td>4.6.2 Dummy keys</td>
</tr>
<tr>
<td>49</td>
<td>4.6.3 Key Compromise</td>
</tr>
<tr>
<td>50</td>
<td>4.6.4 Differing Payload Sources</td>
</tr>
<tr>
<td>51</td>
<td>4.6.5 Transport Messages</td>
</tr>
<tr>
<td>53</td>
<td>5 Properties</td>
</tr>
<tr>
<td>53</td>
<td>5.1 Sanity Checks</td>
</tr>
<tr>
<td>54</td>
<td>5.1.1 Executability</td>
</tr>
<tr>
<td>54</td>
<td>5.1.2 ThreadStartedOnce</td>
</tr>
<tr>
<td>55</td>
<td>5.1.3 Thread Uniqueness</td>
</tr>
<tr>
<td>55</td>
<td>5.1.4 HashEqualityImpliesKeyEquality</td>
</tr>
<tr>
<td>55</td>
<td>5.1.5 NoKeyReuse</td>
</tr>
<tr>
<td>56</td>
<td>5.1.6 SNotSetTwiceAndSafe</td>
</tr>
<tr>
<td>56</td>
<td>5.1.7 RNotSetTwice</td>
</tr>
<tr>
<td>57</td>
<td>5.1.8 ENotSetTwiceAndSafe</td>
</tr>
<tr>
<td>57</td>
<td>5.1.9 RENotSetTwice</td>
</tr>
<tr>
<td>58</td>
<td>5.1.10 PrologueEquality</td>
</tr>
<tr>
<td>58</td>
<td>5.2 Key Agreement Properties</td>
</tr>
<tr>
<td>59</td>
<td>5.2.1 Key Secrecy</td>
</tr>
<tr>
<td>59</td>
<td>5.2.2 Key Uniqueness</td>
</tr>
<tr>
<td>60</td>
<td>5.2.3 Hash Uniqueness</td>
</tr>
<tr>
<td>60</td>
<td>5.3 Payload Properties</td>
</tr>
<tr>
<td>63</td>
<td>5.3.1 Destination</td>
</tr>
<tr>
<td>Contents</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>5.3.2 Source</td>
<td>70</td>
</tr>
<tr>
<td>5.3.3 Authentication</td>
<td>73</td>
</tr>
<tr>
<td>5.3.4 KCI Resistant Authentication</td>
<td>77</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>81</td>
</tr>
<tr>
<td>Bibliography</td>
<td>85</td>
</tr>
<tr>
<td>A Labels</td>
<td>93</td>
</tr>
<tr>
<td>B Intermediate Language Example</td>
<td>95</td>
</tr>
<tr>
<td>C Tamarin Model Example</td>
<td>101</td>
</tr>
<tr>
<td>D Results</td>
<td>129</td>
</tr>
<tr>
<td>D.1 One-Way Patterns</td>
<td>130</td>
</tr>
<tr>
<td>D.1.1 N</td>
<td>130</td>
</tr>
<tr>
<td>D.1.2 K</td>
<td>130</td>
</tr>
<tr>
<td>D.1.3 X</td>
<td>131</td>
</tr>
<tr>
<td>D.2 Fundamental Interactive Patterns</td>
<td>132</td>
</tr>
<tr>
<td>D.2.1 IX</td>
<td>132</td>
</tr>
<tr>
<td>D.2.2 IK</td>
<td>132</td>
</tr>
<tr>
<td>D.2.3 IN</td>
<td>133</td>
</tr>
<tr>
<td>D.2.4 XX</td>
<td>133</td>
</tr>
<tr>
<td>D.2.5 XK</td>
<td>134</td>
</tr>
<tr>
<td>D.2.6 XN</td>
<td>134</td>
</tr>
<tr>
<td>D.2.7 KX</td>
<td>135</td>
</tr>
<tr>
<td>D.2.8 KK</td>
<td>135</td>
</tr>
<tr>
<td>D.2.9 KN</td>
<td>136</td>
</tr>
<tr>
<td>D.2.10 NX</td>
<td>136</td>
</tr>
<tr>
<td>D.2.11 NK</td>
<td>137</td>
</tr>
<tr>
<td>D.2.12 NN</td>
<td>137</td>
</tr>
<tr>
<td>D.3 Deferred Patterns</td>
<td>138</td>
</tr>
<tr>
<td>D.3.1 IX1</td>
<td>138</td>
</tr>
<tr>
<td>D.3.2 IK1</td>
<td>138</td>
</tr>
<tr>
<td>D.3.3 XX1</td>
<td>139</td>
</tr>
<tr>
<td>D.3.4 XK1</td>
<td>139</td>
</tr>
<tr>
<td>D.3.5 KX1</td>
<td>140</td>
</tr>
<tr>
<td>D.3.6 KK1</td>
<td>140</td>
</tr>
<tr>
<td>D.3.7 NX1</td>
<td>141</td>
</tr>
<tr>
<td>D.3.8 NK1</td>
<td>141</td>
</tr>
<tr>
<td>D.3.9 IIX</td>
<td>142</td>
</tr>
<tr>
<td>D.3.10 I1K</td>
<td>143</td>
</tr>
<tr>
<td>D.3.11 I1N</td>
<td>144</td>
</tr>
<tr>
<td>D.3.12 IIX1</td>
<td>145</td>
</tr>
</tbody>
</table>
## Contents

D.3.13 I1K1 .............................................. 146  
D.3.14 X1X .............................................. 147  
D.3.15 X1K .............................................. 147  
D.3.16 X1N .............................................. 148  
D.3.17 X1X1 ............................................. 148  
D.3.18 X1K1 ............................................. 149  
D.3.19 K1X .............................................. 149  
D.3.20 K1K .............................................. 150  
D.3.21 K1N .............................................. 151  
D.3.22 K1X1 ............................................. 152  
D.3.23 K1K1 ............................................. 153  

D.4 Modified Patterns .................................... 155  
D.4.1 Npsk0 ............................................. 155  
D.4.2 Kpsk0 ............................................. 156  
D.4.3 Xpsk1 ............................................. 156  
D.4.4 XXfallback ....................................... 157  
D.4.5 XXfallback+psk0 ................................... 157  
D.4.6 NNpsk0 .......................................... 158  
D.4.7 NNpsk2 .......................................... 159  
D.4.8 NKpsk0 .......................................... 160  
D.4.9 NKpsk2 .......................................... 161  
D.4.10 NXpsk2 .......................................... 162  
D.4.11 XNpsk3 .......................................... 163  
D.4.12 XKpsk3 .......................................... 164  
D.4.13 XXpsk3 .......................................... 165  
D.4.14 KNpsk0 .......................................... 166  
D.4.15 KNpsk2 .......................................... 167  
D.4.16 KKpsk0 .......................................... 168  
D.4.17 KKpsk2 .......................................... 168  
D.4.18 XKpsk2 .......................................... 169  
D.4.19 INpsk1 .......................................... 170  
D.4.20 INpsk2 .......................................... 171  
D.4.21 IKpsk1 .......................................... 172  
D.4.22 IKpsk2 .......................................... 172  
D.4.23 IXpsk2 .......................................... 173
Chapter 1

Introduction

With the omnipresence of telecommunication and an increasing awareness of the advantages of protecting data, the trend towards encrypting more and more forms of communication continues. The various situations in which data is encrypted, however, impose widely differing requirements on cryptographic algorithms and protocols. It is therefore not uncommon for custom protocols to be developed to fulfill these specific requirements. Both designing secure cryptographic protocols and properly implementing them is, however, notoriously difficult. So, unsurprisingly, such one-off protocols often end up with serious security issues.

1.1 Background

1.1.1 The Noise Framework

The Noise protocol framework [60] is an aid in designing secure protocols. It provides resources to easily design a wide variety of protocols and is the basis for multiple libraries [60] which provide a simple way of deploying such protocols. Noise is used by numerous applications, most notably

WireGuard VPN [5] for establishing VPN connections as well as

WhatsApp [72] for encrypting communication between clients and WhatsApp’s servers. The end-to-end encryption of WhatsApp messages is performed with a different protocol.

Noise provides a simple language in which two-party key agreement protocols can be specified. In combination with a few additional validity rules, this language is designed to prevent the protocols from containing subtle security-relevant flaws.

The framework’s specification defines five different levels of confidentiality and two levels of authentication. Somewhat unusually, these properties are
defined to be properties of single messages and not of the protocol as a whole.

Noise does not strictly separate between a phase in which keys are negotiated and one in which they are then used. Even messages in the *handshake phase* can be used to transport application data. It is, therefore, useful to know what guarantees a protocol provides for such data. Especially since the properties can vary drastically from one message to the next as the keys are gradually negotiated.

For 38 protocols, Noise lists how secure each message is in terms of these authentication and confidentiality levels. While the definition of these security properties does contain a brief explanation of how Noise protocols might achieve them, no rigorous justification is given for any message actually satisfying these properties.

Although the use of a protocol which was carelessly put together is strongly discouraged, the warning [64]

> "Users are recommended to only use the handshake patterns listed below, or other patterns that have been vetted by experts to satisfy the above checks."

might fall short since these five checks are simple (see Section 2.2.2). It would, therefore, seem that the 'expertise' required to verify that they are not violated can easily be obtained.

Users who design their own protocol, however, do not have any real guidance as to how secure the resulting protocol is. The same goes for the protocols defined in the specification whose properties are not listed. Additionally, the security levels that Noise uses are insufficient for some use cases.

Chapter 2 provides a more detailed explanation of Noise's workings.

### 1.1.2 Symbolic Verification

One way of analyzing the security of cryptographic protocols is to model them as a sequence of operations performed on a set of abstract symbols and some equations which can be applied to these symbols. For instance, symmetric encryption can be modelled with two functions, \( \text{enc} \) and \( \text{dec} \), and the equation [71]

\[
\text{dec}(\text{enc}(\text{data, key}), \text{key}) = \text{data}
\]

This expresses many properties of symmetric encryption, such as:

- Decryption (\( \text{dec} \)) is the inverse of encryption (\( \text{enc} \)).
- Both encryption and decryption require the (same) key.
Related Work

- Encryption ‘hides’ the encrypted data. \( \text{enc}(\text{data}, \text{key}) \) is treated as an opaque token and therefore reveals nothing about data.

Abstracting cryptographic function with equations over some symbols has several advantages:

- The resulting model is independent of the cryptographic algorithm which is used. This is particularly useful when verifying Noise protocols since these are specified for arbitrary algorithms. In fact, Noise officially endorses 16 different combinations of algorithms. The Noise wiki also specifies how to use an additional 284 combinations [66] and is not intended to be exhaustive.

By working with an abstraction of the cryptography, the properties of a protocol can be verified in general without having to do so separately for all of these combinations.

- The set of equations used to specify an algorithm in the symbolic model tend to be significantly simpler than a precise specification of the properties of a cryptographic algorithm. This not only makes the model generation easier but also allows for more efficient automatic verification.

The most notable downside of working with such an abstraction of cryptography is that it can hide possible attacks on the protocol which arise from the subtleties of how the underlying algorithms work. It is, for instance, common practice to model hash functions as a function whose output reveals nothing about its input. This is, however, not generally a requirement for a hash function to be considered ‘secure.’ A protocol relying on hash functions hiding their input could, therefore, pass symbolic verification with flying colours and yet still be easily broken when performed with a hash function which is considered to be safe.

This thesis presents a tool for verifying Noise protocols in the symbolic model.

1.2 Related Work

While this thesis was in its early stages, Noise Explorer [9] was published. Noise Explorer generates models of arbitrary Noise protocols that can be used to verify them formally. The models generated by Noise Explorer are for the symbolic verification tool ProVerif [1].

However, ProVerif lacks built-in support for Diffie-Hellman operations. Additionally, there are some limitations to the equations which can be used to model such primitives. Therefore, Noise Explorer’s model for Diffie-Hellman lacks some features such as the possibility of inverting private keys.
In addition, Noise Explorer only supports a single pre-shared key (PSK) per protocol and does not model the final key derivation step. This leads to some messages being encrypted with different keys than the Noise specification prescribes.

As of this writing, results obtained by running ProVerif on the models for 57 of the 61 protocols named in the Noise specification have been published [10], with new ones being added on occasion.

The properties Noise Explorer verifies are very close to the security property defined by Noise itself. Noise Explorer verifies all of the framework’s properties and only adds two new ones, which are closely modelled on similar ones from Noise.

1.3 Contributions

The goal of this work is to provide a tool to automatically verify what security-related properties a Noise protocol satisfies [70]. The actual verification is performed by the Tamarin prover [69].

Tamarin is a powerful verification tool for security protocols in the symbolic model. It has an extensive built-in model of Diffie-Hellman exponentiation, making it well suited for the verification of Noise protocols. Tamarin is further described in Chapter 3.

The verification process for Noise protocols consists of the following steps.

1. Translating a protocol name into Noise’s pattern language.

2. Translating such a pattern into a more expressive and well defined intermediate language.

3. Translating protocols from this intermediate language into a formal model of the protocol. In order for the next step to work, this model is in Tamarin’s security protocol theory format. This step also adds formalized versions of Noise’s security properties as well as of some additional properties to the model. These are then verified in the following step. A full list of the verified properties can be found in Chapter 5.

4. Running Tamarin on the model to verify which of the properties the protocol satisfies. In the process also providing Tamarin with the guidance it needs to check which properties hold.

5. Formatting Tamarin’s output in a simplified way.

A detailed description of the tool’s working is laid out in Chapter 4.

The ambiguity of the language in which Noise specifies its properties has lead to significant differences between the interpretation used by Noise Explorer and the one used by the Tamarin based tool. These differences
1.3. Contributions

are discussed in Chapter 5. Their effects on the verification result are highlighted in Appendix D.
Chapter 2

Noise

The Noise protocol framework is a framework for Diffie-Hellman based authenticated key agreement protocols. It offers infrastructure for two parties to authenticate one another and to negotiate cryptographic keys, which can then be used for further communication.

Depending on the requirements of a specific situation, Noise protocols can be used to authenticate a single, both or neither party. Noise also supports the use of pre-shared keys, which can, for instance, improve post-quantum resistance [6]. Additionally, protocols can be tailored to authenticate data that is communicated before the protocol run. The framework also provides resources for using already negotiated keys in various forms of data exchange [64].

2.1 Diffie-Hellman

Noise protocols rely heavily on Diffie-Hellman (DH) for both authentication and key derivation.

Using Diffie-Hellman, two parties can generate a shared secret, i.e. some value that they both know. However, even an attacker who can read all the data they exchange is unable to compute this secret. To this purpose, each party generates a private key \(k_a\) and \(k_b\) respectively and computes the corresponding public key \(g^{k_a}\) and \(g^{k_b}\) respectively using some publicly known generator \(g\). By exchanging their public keys, they can then both compute

\[
(g^{k_a})^{k_b} = g^{k_axk_b} = g^{k_bxk_a} = (g^{k_b})^{k_a}
\]

Since all these calculations are performed in a group in which computing the (discrete) logarithm is assumed to be infeasible, the attacker cannot find out the private keys and, without them, cannot compute the shared secret.
Noise also uses the fact that computing the shared secret requires the knowledge of at least one of the private keys for authentication. The basic idea being that, since computing $g^{k_a k_b}$ requires the knowledge of either $k_a$ or $k_b$, the owner of $k_a$ knows that the only other party that can know the secret is the owner of $k_b$. Therefore, if $k_a$’s owner can reliably link the public key $g^{k_b}$ to some party, it can authenticate that party using the shared secret $g^{k_a k_b}$.

2.2 Noise Patterns

The Noise framework is built around a language designed to intuitively describe Diffie-Hellman based protocols. When conforming to the five validity rules detailed below, protocols should both be executable and be resistant to the exploitation of weaknesses in the underlying cryptographic algorithms. In particular, such protocols should not be susceptible to weaknesses arising from the reuse of cryptographic keys.

2.2.1 Syntax

The Noise pattern language is modelled around the Alice and Bob notation, with all protocols being between Alice and Bob. Independent of who is the initiator (i.e. the sender of the first message) and who is the responder (i.e. the recipient of the first message), the protocol is always denoted such that Alice is on the left and Bob is on the right.

In the protocol named IN:

1. IN:
2. $\rightarrow e, s$
3. $\leftarrow e, ee, se$

for instance, Alice begins by sending both an ephemeral public key ($e$) and her long-term static public key ($s$) to Bob, who responds with an ephemeral public key ($e$) of his own. The shared secrets obtained by performing a Diffie-Hellman exponentiation between both ephemeral keys ($ee$) as well as Alice’s static and Bob’s ephemeral key ($se$) are then used to derive encryption keys.

The Noise terminology allows for both Alice and Bob to perform either a protocol run in either initiator or responder role.

1. IN:
2. $\leftarrow e, s$
3. $\rightarrow e, ee, es$

is, therefore, an equivalent specification of the IN protocol, although the protocol is initiated by Bob in this version.
Noise patterns include all information to fully specify the protocol:

1. The protocol name (`Noise_XXfallback+psk0_25519_AESGCM_SHA256`) identifies the protocol, as well as the cryptographic algorithms used.
2. The `:` separating the name from the rest of the pattern.
3. The (optional) pre-messages (`-> e`), which represent the knowledge one party (here Bob) has about the other party’s (here Alice’s) DH keys before the protocol execution begins.
4. The `...` separating the pre-messages from the handshake messages.
5. The handshake messages (`<- psk, e, ee, s, es` and `-> s, se`) specify the cryptographic operations performed and the messages sent during a protocol run.

**Protocol Name**

The protocol name in turn also consists of five, underscore(\_)-separated components:

In the above example, `Noise_XXfallback+psk0_25519_AESGCM_SHA256` is composed of

1. The noise prefix is always 'Noise'.
2. The handshake pattern name section (`XXfallback+psk0`) identifies the protocol. It again is made up of two parts:
   a) The actual handshake pattern name (`XX`), an uppercase alphanumeric ASCII string identifying the base structure of the protocol. For example, `XX` denotes the following pattern:

   1. `XX:
   2. `-> e
   3. `<- e, ee, s, es
   4. `-> s, se

   b) The pattern modifiers (`fallback` and `psk0`), zero or more, plus(+) separated lowercase alphanumeric ASCII strings beginning in a
letter. They modify the base pattern in some way. The Noise specification only defines the fallback and the psk modifiers, which turn the first message into a pre-message and mix a pre-shared key into the negotiated key respectively. There exist, however various extensions to Noise which specify other modifiers [57].

3. The DH algorithm name section (25519) specifies the group in which DH operations are performed.

As of revision 34, the Noise specification suggests using X25519 (25519) or X448 (448).

4. The cipher algorithm name section (AESGCM) specifies the Authenticated-Encryption with Associated-Data (AEAD) [67] scheme used for encryption. Such encryption schemes not only encrypt but also authenticate the encrypted data as well as some additional data.

As of revision 34, the Noise specification suggests using AES256 in GCM mode (AESGCM) or CHACHA20 with POLY1305 MAC (CHACHAPoly).

5. The hash algorithm name section (SHA256) specifies the hash algorithm used for authenticating the protocol transcript as well as key derivation.

As of revision 34, the Noise specification suggests using SHA-256 (SHA256), SHA-512 (SHA512), BLAKE2s (BLAKE2s) or BLAKE2b (BLAKE2b).

For simplicity, when discussing a protocol in the abstract, the noise prefix and the algorithm names can be omitted. A simple, unauthenticated DH key agreement therefore becomes

1 NN:
2 -> e
3 <- e, ee

The Protocol Name is separated from the rest of the pattern by a colon(:).

Pre-Messages

It is fairly common for protocol participants to know the other party’s static public key before the protocol run begins. In some cases, this pre-knowledge may also include the remote ephemeral public key. Noise acknowledges this with the concept of pre-messages. Conceptually these are messages sent and received before the protocol run begins. Therefore,

1 -> e
2 ...
There are six valid pre-messages:

1. \( \rightarrow e \)
   Bob knows Alice’s ephemeral \((e)\) public key.

2. \( \leftarrow e \)
   Alice knows Bob’s ephemeral \((e)\) public key.

3. \( \rightarrow s \)
   Bob knows Alice’s static \((s)\) public key.

4. \( \leftarrow s \)
   Alice knows Bob’s static \((s)\) public key.

5. \( \rightarrow e, s \)
   Bob knows both Alice’s ephemeral \((e)\) and static \((s)\) public key.

6. \( \leftarrow e, s \)
   Alice knows both Bob’s ephemeral \((e)\) and static \((s)\) public key.

Additionally, the following rules apply to pre-messages:

- At most one pre-message per direction \((\rightarrow / \leftarrow)\) is allowed in a pattern.
- Empty pre-messages are left out.
- If both pre-messages are present, the first one must have the same direction \((\rightarrow / \leftarrow)\) as the first handshake message.
- If any pre-messages are present, they are delimited from the handshake messages by three dots \((...\)\). The dots are omitted if the protocol does not require any pre-messages.

**Handshake Messages**

Messages are denoted by an arrow indicating whether they are sent by Alice to Bob \((\rightarrow)\) or vice versa \((\leftarrow)\) and a comma\((,\)\)-separated list of zero or more tokens. The tokens can be any of the following:

- \(e\) The sender’s ephemeral public key is sent.
- \(s\) The sender’s static public key is sent.
- \(ee, es, se \text{ or } ss\) DH-exponentiation is performed with the corresponding keys. Alice’s key is indicated by the first letter, Bob’s by the second.
- \(psk\) The pre-shared key (PSK) shared by Alice and Bob is mixed into the negotiated encryption key.
2. Noise

2.2.2 Pattern Validity

In order to be valid, a syntactically correct pattern must also follow these five validity rules:

1. DH operations are only performed between keys both parties have knowledge of.

2. Neither party sends one of its public keys more than once per protocol run.

3. No DH operation between the same two keys is performed more than once per protocol run.

4. Participants may not send their public key or a payload after performing a DH operation between a remote public key and their static key without also having performed a DH operation between the same remote key and their own ephemeral key. Meaning that
   a) After an se, Alice may only send her public key or a payload if an ee has also been performed
   b) After an ss, Alice may only send her public key or a payload if an es has also been performed
   c) After an es, Bob may only send his public key or a payload if an ee has also been performed
   d) After an ss, Bob may only send his public key or a payload if an se has also been performed

5. No participant sends its public key or a payload after processing a psk token unless the participant has sent its ephemeral key. For the purposes of this rule, ephemeral keys in pre-messages count as being ‘sent.’

Note that

- At the end of every handshake message, applications may append some arbitrary data. If the application has no need for this, a zero-length blob is used instead. These blobs are called the handshake payloads. The purpose of the validity rules 4 & 5 is to ensure that these are never encrypted with the same key.

- The negotiated keys are presumably eventually used to encrypt some messages called transport payloads. In order to conform to rules 4 & 5, for some patterns there may be a participant who must not use these keys.

- Since the message tokens are processed message by message and left to right, any token in a message (or in a preceding message) is processed before the handshake payload is sent. Similarly, any token to
2.3 Semantics

the left of an s is processed before the s. Therefore,

1 IXpsk0:
2 -> psk, e, s
3 <- e, ee, se, s, es

is valid, while

1 INVALID:
2 <- s
3 ...
4 -> psk, s, e
5 <- es, e
6 -> se, ss

violates validity rules 5, 4c and 4a.

2.3 Semantics

Tokens are processed one after another, each modifying the participants’ states according to rules as outlined below.

2.3.1 State

The state of a participant in a Noise protocol run consists of eight variables.

Symmetric Key

The encryption/decryption key for static DH keys and handshake payloads. Initialized with

k ← empty

Nonce

Counter-based nonce for encryption/decryption of static DH keys and handshake payloads. The nonce is initialized with

n ← 0

and is usually reset to 0 when k is updated.

Hashes

The ck hash is used for deriving symmetric encryption keys. h is a hash of the protocol transcript and is used to ensure both participants have the same
2. Noise

view of previous messages. They are initialized with

1 \( h \leftarrow \text{HASH}(\text{protocol\_name}) \)
2 \( c k \leftarrow h \)

if the protocol name (presumably in ASCII encoding) is longer than the
output of \text{HASH}. Otherwise, the protocol name is padded with zero bytes to
the output-size of \text{HASH}

1 \( h \leftarrow \text{protocol\_name} \| 0x00000000... \)
2 \( c k \leftarrow h \)

For example

1 \( h \leftarrow \text{HASH}'\text{Noise\_XXfallback+psk0\_25519\_AESGCM\_SHA256}'\)
2 \( c k \leftarrow h \)

**Asymmetric Keys**

Since either party to the protocol can have both an ephemeral and a static
DH key, a participant must be able to keep track of up to four public and
two private keys.

1. e A keypair consisting of the participant’s ephemeral private key and
the corresponding public key.

2. s A keypair consisting of the participant’s static private key and the
corresponding public key.

3. re The other participant’s ephemeral public key.

4. rs The other participant’s static public key.

All DH keys are initialized to empty

1 \( e \leftarrow \text{empty} \)
2 \( s \leftarrow \text{empty} \)
3 \( re \leftarrow \text{empty} \)
4 \( rs \leftarrow \text{empty} \)

**2.3.2 Functions**

Since Noise supports various cryptographic algorithms, the actual cryptog-
raphy is performed in functions whose behaviour depends on the chosen
algorithms and error handling strategy.
2.3. Semantics

Encryption

ENCRIPT(key, nonce, additional_data, plaintext)
uses key and nonce to encrypt plaintext and generates an authentication
tag for plaintext and additional_data. It returns some concatenation of
the resulting ciphertext and authentication tag.

Decryption

DECRIPT(key, nonce, additional_data, ciphertext)
uses key and nonce to decrypt ciphertext as returned by ENCRYPT and
validates the authentication tag for ciphertext and additional_data. It
returns the plaintext as passed to ENCRYPT if the authentication tag matches.
Otherwise, it signals an error, causing the protocol run to be aborted.

Hashing

HASH(data)
returns the hash of data.

HMAC

HMAC-HASH(key, data)
returns an HMAC tag for key and data using the hash function, i.e. HMAC-
HASH calculates the tag using \[53\]

```plaintext
1  ipad ← 0x36363636...
2  opad ← 0x5C5C5C5C...
3  tag ← HASH((key XOR opad) || HASH(data XOR ipad))
```

Diffie-Hellman Key Generation

GENERATE_KEYPAIR()
returns a private DH key and the corresponding public key.

Diffie-Hellman Exponentiation

DH(private_key, public_key)
returns the result of performing a Diffie-Hellman operation with the two
2. Noise

keys. i.e. returns public_key^{private_key} in the corresponding DH group. At least as long as public_key is a valid DH public key. Otherwise, it may either return any value that does not depend on private_key or signal an error and cause the protocol run to abort.

2.3.3 Transition

Using the cryptographic algorithms specified in the protocol name, Noise participants perform the protocol by processing first all pre-messages and then each message in turn by applying the following rules.

Pre-Messages

Pre-messages are processed by setting the corresponding DH-keys to the correct values and mixing the public keys into \( h \). Alice, for example, processes a \( a < s \) pre-message by setting her \( rs \) to Bob’s static key and setting her \( h \) to

\[
 h \leftarrow \text{HASH}(h || \text{bobs\_static\_public\_key})
\]

In addition to this, protocols using PSKs also mix any ephemeral keys which are in the pre-messages into \( ck \) by updating it with

\[
 ck \leftarrow \text{HMAC-HASH(HMAC-HASH(ck, public\_key), 0x01)}
\]

in order to randomize encryption keys otherwise only derived from the PSK. Note that at this point, \( ck \) is a hash over the protocol name and public keys, both of which an attacker can presumably find out. Since keys derived from such \( ck \)'s are occasionally used for encryption, encryption in a Noise protocol does not inherently provide any security guarantees.

More generally, pre-messages are processed by updating \( h \) for each public key in turn. The public keys are hashed in the order in which they appear in the protocol pattern (left to right, top to bottom). The hashing is performed as follows for the public key \( public\_key \) in the pre-message with direction \( message\_direction \):
2.3. Semantics

1. $h \leftarrow \text{HASH}(h \ || \ \text{public_key})$
2. IF pattern_contains_psk AND is_ephemeral(public_key)
   3. \hspace{1em} temp $\leftarrow$ HMAC-HASH(ck, public_key)
   4. \hspace{1em} ck $\leftarrow$ HMAC-HASH(temp, 0x01)
   5. \hspace{1em} k $\leftarrow$ HMAC-HASH(temp, ck || 0x02)
   6. \hspace{1em} n $\leftarrow$ 0

7. IF (i_am_alice = (message_direction = '->'))
   8. \hspace{1em} // local key
   9. \hspace{2em} IF is_ephemeral(public_key)
   10. \hspace{3em} e $\leftarrow$ load_ephemeral_keypair()
   11. \hspace{2em} ELSE
   12. \hspace{3em} s $\leftarrow$ load_static_keypair()
   13. ELSE
   14. \hspace{1em} // remote key
   15. \hspace{2em} IF is_ephemeral(public_key)
   16. \hspace{3em} re $\leftarrow$ public_key
   17. \hspace{2em} ELSE
   18. \hspace{3em} rs $\leftarrow$ public_key

Messages

A pattern’s messages are processed one after the other. For each message, a message buffer is successively filled and then sent. Each token causes data to be added to the message buffer, modifies the participants’ state in some way or does both.

\( e \)

The sender appends its ephemeral public key to the message. The recipient then sets his or her \( re \) to this public key.

\( s \)

The sender appends its static public key to the message. If \( k \) is already non-empty, the public key is first encrypted using \( k \). The recipient updates \( rs \) accordingly.

\( \text{psk} \)

The pre-shared key is mixed into \( k \) and \( ck \).

\( \text{ee} / \text{es} / \text{se} / \text{ss} \)

A DH operation is performed between the two corresponding keys. Alice’s key is indicated by the first letter, Bob’s by the second. The resulting shared secret is then mixed into \( k \) and \( ck \).

Finally, the payload is added to the message buffer. If \( k \) is non-empty, it is first encrypted with \( k \).
All the while, everything added to the message buffer as well as the PSKs are also mixed into h. The pseudocode below shows the precise operations used for this process.

For messages they send (i.e. -> messages for Alice and <- for Bob), participants perform the following:

```
message_buffer ← ''
FOR token IN message.tokens
    IF token = 'e'
        e ← GENERATE_KEYPAIR()
        message_buffer ← message_buffer || e.public_key
        h ← HASH(h || e.public_key)
        IF pattern_contains_psk
            temp ← HMAC-HASH(ck, e.public_key)
            ck ← HMAC-HASH(temp, 0x01)
            k ← HMAC-HASH(temp, ck || 0x02)
            n ← 0
        ELSEIF token = 's'
            s ← load_static_keypair()
            IF k = empty
                enc_s ← s.public_key
            ELSE
                enc_s ← ENCRYPT(k, n, h, s.public_key)
                n ← n + 1
            message_buffer ← message_buffer || enc_s
            h ← HASH(h || enc_s)
        ELSEIF token = 'psk'
            psk ← load_corresponding_psk()
            temp ← HMAC-HASH(ck, psk)
            ck ← HMAC-HASH(temp, 0x01)
            temp_h ← HMAC-HASH(temp, ck || 0x02)
            k ← HMAC-HASH(temp, temp_h || 0x03)
            n ← 0
            h ← HASH(h || temp_h)
        ELSE
            IF token = 'ee'
                private_key ← e.private_key
                public_key ← re
            ELSEIF token = 'ss'
                private_key ← s.private_key
                public_key ← rs
            ELSEIF (token = 'es') = i_am_alice
                private_key ← e.private_key
                public_key ← rs
```
2.3. Semantics

39 ELSE
40   private_key ← s.private_key
41   public_key ← re
42   dh_result ← DH(private_key, public_key)
43   temp ← HMAC-HASH(ck, dh_result)
44   ck ← HMAC-HASH(temp, 0x01)
45   k ← HMAC-HASH(temp, ck || 0x02)
46   n ← 0
47 IF k = empty
48   ciphertext ← payload
49 ELSE
50   ciphertext ← ENCRYPT(k, n, h, payload)
51   n ← n + 1
52   message_buffer ← message_buffer || ciphertext
53   h ← HASH(h || ciphertext)
54   send(message_buffer)
For messages they receive (i.e. \(<\rightarrow\) messages for Alice and \(\rightarrow<\) for Bob), participants perform the following:

```plaintext
message_buffer ← receive()
FOR token IN message.tokens
    IF token = 'e'
        (re, message_buffer) ← split_first_component(message_buffer)
        h ← HASH(h || re)
        IF pattern_contains_psk
            temp ← HMAC-HASH(ck, re)
            ck ← HMAC-HASH(temp, 0x01)
            k ← HMAC-HASH(temp, ck || 0x02)
            n ← 0
        ELSEIF token = 's'
            (enc_rs, message_buffer) ← split_first_component(message_buffer)
            IF k = empty
                rs ← enc_rs
            ELSE
                rs ← DECRYPT(k, n, h, enc_rs)
                n ← n + 1
            h ← HASH(h || enc_rs)
        ELSEIF token = 'psk'
            psk ← load_corresponding_psk()
            temp ← HMAC-HASH(ck, psk)
            ck ← HMAC-HASH(temp, 0x01)
            temp_h ← HMAC-HASH(temp, ck || 0x02)
            k ← HMAC-HASH(temp, temp_h || 0x03)
            n ← 0
            h ← HASH(h || temp_h)
        ELSE
            IF token = 'ee'
                private_key ← e.private_key
                public_key ← re
            ELSEIF token = 'ss'
                private_key ← s.private_key
                public_key ← rs
            ELSEIF (token = 'es') = i_am_alice
                private_key ← e.private_key
                public_key ← rs
            ELSE
                private_key ← s.private_key
                public_key ← re
```
2.3. Semantics

```
42   dh_result ← DH(private_key, public_key)
43   temp ← HMAC-HASH(ck, dh_result)
44   ck ← HMAC-HASH(temp, 0x01)
45   k ← HMAC-HASH(temp, ck || 0x02)
46   n ← 0
47   IF k = empty
48       payload ← message_buffer
49   ELSE
50       payload ← DECRYPT(k, n, h, message_buffer)
51       n ← n + 1
52   h ← HASH(h || message_buffer)
```

**Key Derivation**

Once all messages are sent, Noise derives two keys for the application to use. The corresponding nonces are set to 0

```
1    k1 ← HMAC-HASH(HMAC-HASH(ck, ''), 0x01)
2    k2 ← HMAC-HASH(HMAC-HASH(ck, ''), k1 || 0x02)
3    n1 ← 0
4    n2 ← 0

All state other than k1, k2, n1, n2, and h is discarded.
```
The NOISE verification tool presented in this thesis uses TAMARIN for the actual verification of whether the properties hold for a specific protocol. TAMARIN is a powerful tool to automatically prove the security of cryptographic protocols [71]. It has built-in support for various cryptographic primitives and commonly used operations such as encryption, hashing, signatures, xor, and more. With respect to the verification of the DH based NOISE protocols, however, TAMARIN’s greatest strength is its built-in support for Diffie-Hellman, which is more extensive than that of other formal protocol verification tools such as PROVERIF [1] or Scyther [3].

While TAMARIN’s default proof strategy performs well in many cases, proofs can also use a different built-in strategy, a user-provided custom-built strategy, or be guided manually.

3.1 Attacker

TAMARIN uses the Dolev–Yao attacker model [4]. In this model, the attacker completely controls the network and is also a regular participant in it. The attacker can, therefore, among other things

- deliver messages to the intended recipient (essentially behaving like a friendly network)
- drop messages
- deliver messages to a participant in the network who was not the intended recipient
- construct arbitrary messages of its own and send them
- duplicate and replay messages
- delay the delivery of messages
• start a protocol run with any other participant in the network
• cause a participant to initiate a protocol run with a participant of the attacker’s choosing
• deconstruct any message into its components
• keep a record of all previous network traffic
• compare message components to one another
• perform computations and evaluate functions

The attacker cannot, however, break cryptography in any form. TAMARIN’s built-in hashing and symmetric encryption, for example, defines the binary functions \( \text{senc} \) and \( \text{sdec} \) as well as the unary function \( h \) with the equation

\[
\text{sdec}(\text{senc}(m, k), k) = m
\]

Using this, the attacker cannot decrypt

\[ \text{senc(data1, key)} \]

to extract data1 without somehow knowing key. Similarly, an attacker who has learned

\[ h(data2) \]

cannot use this to find out anything about data2. An attacker who knows both \( h(data2) \) and data2, however, can use the latter to compute \( h(data2) \) for itself, notice that the two \( h(data2) \) are equal and conclude that \( h(data2) \) is indeed the hash of data2.

### 3.2 Terms

While messages in concrete protocol implementations are raw data, i. e. arrays of bytes, TAMARIN models them with terms. Terms in TAMARIN can have one of the following form (TAMARIN also supports other forms, but these are not used in this thesis and are therefore not further discussed here)

• Constant

'identifier1'

As their name implies, constants always have the same value. All such values are part of the initial knowledge of the attacker. Use cases include the protocol’s name and the generator for the Diffie-Hellman group which is used.

• Fresh Value
3.2. Terms

Any participant in the protocol can generate an arbitrary amount of new fresh values during every protocol run. These are not initially known to the attacker (though if they are sent in an insufficiently secure way, the attacker may be able to learn them). These properties make fresh values a good model for values which would be generated by a cryptographically secure (pseudo) random number generator in a concrete protocol implementation. Therefore, use cases include private keys and nonces.

• Free Variable

These are essentially placeholders for more specific terms. Use cases include specifying equations which can then be applied to a wide variety of terms.

\[
\text{sdec(senc(m, k), k) = m}
\]

for instance expresses that

\[
\text{sdec(senc(\text{'data'}, \sim key1), \sim key1) = \text{'data'}}
\]

and that

\[
\text{sdec(senc(\sim nonce, \sim key2), \sim key2) = \sim nonce}
\]

• Function

Unless otherwise specified, these terms are completely opaque to the attacker. Meaning that the only thing an attacker can tell about two such terms is whether they are identical (i.e. the same function applied to the same arguments) or not. An attacker who knows the value of all subterms can also compute the result of applying the function to them. By using the right equations, functions can, therefore, be used to model cryptographic primitives.

For some built-in functions, TAMARIN also uses operators. Two, in particular, should be noted at this point:

1. Diffie-Hellman Exponentiation (and Multiplication)
   
   See Section 3.3 below.

2. Pairing
3. TAMARIN

\[ \langle 'a', \sim b, c \rangle = \langle 'a', \langle \sim b, c \rangle \rangle = \text{pair}(\langle 'a', \text{pair}(\sim b, c) \rangle) \]

TAMARIN's built-in functions used in this thesis are

- h, a unary function without corresponding equation used to model hash functions
- pair, or more specifically \( \langle a, b \rangle \), a binary function with the equations [71]
  \begin{align*}
  1 & \text{fst}(\text{pair}(x, y)) = x \\
  2 & \text{snd}(\text{pair}(x, y)) = y
  \end{align*}

- TAMARIN's Diffie-Hellman model. See below.

3.3 Diffie-Hellman Support

A particular strength of TAMARIN is its built-in support of Diffie-Hellman. TAMARIN models DH using the ^ operator, the * operator and the unary inv function. The operators model their arithmetic counterparts and the inv function models a ^{-1}, i.e. the computation of the multiplicative inverse. These operators and the function are put in relation to one another with these equations [71]:

\begin{align*}
  1 & (x^y)^z = x^{y*z} \\
  2 & x^1 = x \\
  3 & x*y = y*x \\
  4 & (x*y)*z = x*(y*z) \\
  5 & x*1 = x \\
  6 & x*\text{inv}(x) = 1
\end{align*}

TAMARIN having built-in support for DH allows for the use of operators instead of being restricted to a function-based syntax such as
\[
\text{exp} (\text{exp}(x, y), z) = \text{exp}(x, \text{mult}(y, z))
\]

More importantly, DH being built-in allows TAMARIN to implement the model efficiently. While the equations listed above are logically equivalent to the ones used by TAMARIN internally, using them in this form would lead to inefficient proofs. In addition, using the built-in Diffie-Hellman model helps ensure that the set of all (meaning both built-in and user-defined) equations satisfies the finite variant property. This, in turn, is a requirement for TAMARIN to properly process the equations.

In order for symbolic verification tools to perform efficient verifications, they must impose restrictions on what equations they support. Because of this, the DH models used in many such tools lack support for associativity (i.e.
3.4 Multiset Rewriting

In TAMARIN, protocols are specified as a set of rules which modify a global state. This global state is represented by a multiset of facts.

**Definition 3.1** A multiset \( m \) over a set \( S \) is a function \( m : S \rightarrow \mathbb{N} \cup \{\infty\} \)

Intuitively, a multiset is an unordered collection of elements of \( S \) in which elements can occur multiple times or even infinitely often.

3.4.1 Facts

Facts are a named list of symbols. Or, more formally,

**Definition 3.2** For a fact symbol \( F \) of arity \( n \) and terms \( t_1, t_2, \ldots, t_n \),

\[
F(t_1, t_2, \ldots, t_n)
\]

is a fact.

For example,

\[
\text{Init}_1(\sim \text{tid}, \sim \text{sk})
\]

might be used to indicate that an initiator has just begun running the modelled protocol in a thread with id \( \sim \text{tid} \) and is using the private key \( \sim \text{sk} \).

TAMARIN models also use some special facts, which are treated differently by TAMARIN.

- \( \text{Fr}(\sim x) \)
  
  Can be used to obtain new fresh values. Essentially, there is always an endless supply of different \( \text{Fr}(\sim x) \) facts available in the state multiset.

- \( \text{In}(\text{msg}) \)
  
  Is used to model receiving a message from the attacker-controlled network.

- \( \text{Out}(\text{msg}) \)
  
  Is used to model sending a message over the network. It causes \( \text{msg} \) to be added to the attacker’s knowledge.
### 3.4.2 Rewriting Rules

Protocols are then specified as a set of multiset rewriting rules.

```plaintext
1 rule Client_1:
2   [ Fr(~k), !Pk(pkS) ]
3 -->
4   [ Client_1(~k), Out(aenc(~k, pkS)) ]
```

This models a protocol step where the client
1. generates a random \( k \)
2. looks up the server’s public key
3. asymmetrically encrypts \( k \) with the server’s public key
4. sends the encrypted \( k \) to the server
5. stores \( k \) for further use later in the protocol

This rule can be understood as modifying the state multiset as follows.

1. Remove one of the \( Fr \) facts from the state. Thereby binding the fact’s new fresh value to \( \sim k \).
2. Verify if there is a \( !Pk \) fact in the state. If so, bind the corresponding value to \( pkS \). If no such fact exists, the rule cannot be applied to the state.

Facts whose names begin with a \( ! \) are treated differently from regular facts. If such a fact occurs in the state, its multiplicity is always infinity, i.e. for a fact \( !F \) and a state \( m \),

\[
!F \in m \implies m(!F) = \infty
\]

Therefore, such facts
- can occur multiple times in the left-hand side of the rewriting rule (i.e. the part between the : and the -->) and
- cannot be removed from the state

Consequently,

```plaintext
1 rule Client_1_alternate:
2   [ Fr(~k), !Pk(pkS), !Pk(pkS), !Pk(pkS), !Pk(pkS) ]
3 -->
4   [ Client_1(~k), Out(aenc(~k, pkS)), !Pk(pkS) ]
```

is logically equivalent to the \( Client_1 \) rewriting rule. This also demonstrates that the rule names \( Client_1 \) / \( Client_1 \_alternate \) purely serve to enhance readability and have no bearing on the rule’s meaning.
3. Add an $\text{out}(\text{aenc}(\sim k, \text{pkS}))$ fact to the state. TAMARIN will then use some built-in rules to add $\text{aenc}(\sim k, \text{pkS})$ to the attacker's knowledge and generate an $\text{in}(\text{aenc}(\sim k, \text{pkS}))$ fact if necessary.

4. Store a $\text{Client}_1(\sim k)$ fact into the global state.

### 3.5 Let Binding

Specifying protocol steps like this works well for simple protocols. For more complex ones, however, it can become somewhat unwieldy.

```plaintext
1 rule NaxosR:
  2   in
  3     [  
  4       \text{In}(X),
  5       \text{Fr}(\sim\text{eskR}),
  6       \text{Fr}(\sim\text{tid}),
  7       \text{!Ltk}(\sim R, \sim lR),
  8       \text{!Pk}(\sim I, \text{pkI})
  9    ]
 10  -->
 11  [  
 12    \text{Out}(\text{'}g\text{'(h(<'A', \sim\text{eskR}, \sim lR>)})),
  13    \text{!SessionKey}(\sim\text{tid}, \sim R, \sim I,
  14         h(<
  15           \text{'}B\text{'},
  16           \text{pkI}^{h(<'A', \sim\text{eskR}, \sim lR>)},
  17           \text{X}^{\sim lR},
  18           \text{X}^{h(<'A', \sim\text{eskR}, \sim lR>)},
  19           \sim I, \sim R
  20        >)
  21     )
  22    ]
```

In order to ameliorate this situation, TAMARIN supports *let binding*:

---

**Note:** The above code snippet is an example of how to specify protocol steps using let binding in a formal specification language. Let binding allows for the introduction of new variables in a local scope, which can be useful for managing state and variables within protocol steps.
3. TAMARIN

1 rule NaxosR:
2 let
3    exR = h('<A', ¬eskR, lkR >)
4    hkr = 'g'¬exR
5    kR = h('<B', pkI¬exR, X¬lkR, X¬exR, ¬I, ¬R >)
6 in
7 [ In(X),
8    Fr( ¬eskR ),
9    Fr( ¬tid ),
10   !Ltk(¬R, lkR),
11   !Pk(¬I, pkI)
12 ]
13 -->
14 [ Out( hkr ),
15   !SessionKey(¬tid, ¬R, ¬I, kR)
16 ]

These are then applied by replacing all further occurrences of the identifiers on the left with the expression on the right of the =.

3.6 Property Specification

In TAMARIN, security properties are specified using first-order logic over timestamped action facts.

3.6.1 Action Facts

Unlike regular facts, action facts are not part of the state multiset. Instead, they are created whenever the corresponding rewrite rule is applied.

They are specified by including a list of action facts in the --> of a multiset rewriting rule.

1 rule Example:
2 [ State1(x), Fr(¬k) ]
3 -->[
4    ActionFact1(¬k),
5    ActionFact2(x)
6 ] -->
7 [ Out( enc(x,¬k) ) ]

When the Example rule is applied to the state at timestamp i, the ActionFact1(¬k)@i and ActionFact2(x)@i action facts are created.
TAMARIN also has a special action fact $K(x)$ which expresses that the attacker knows the value of $x$.

### 3.6.2 Specification Language

The property specifications in TAMARIN are called *lemmas* and look something like this:

```plaintext
1 lemma Example1:
2     exists-trace
3     "Ex m s #i. ActionFact1(s)@i & ActionFact2(m)@i"
```

Lemmas consist of three parts:

1. **Example1**
   
   The lemma's name. It only serves as a way to identify the lemma and has no actual effect.

2. **exists-trace**
   
   An optional *exists-trace* or *all-traces*. *all-traces* properties must hold for all possible combinations of protocol executions. *exists-trace* properties, on the other hand, only require that there is one such combination which satisfies the formula.

   If neither *exists-trace* nor *all-traces* is specified, TAMARIN defaults to *all-traces*.

3. "Ex m s #i. ActionFact1(s)@i & ActionFact2(m)@i"

   The first-order logic formula that specifies the property. In this case, the Example1 lemma is satisfied, if and only if there is some way to apply a rule to the state which generates both an $\text{ActionFact1}$ and an $\text{ActionFact2}$ fact. While the facts' arguments can be independent of one another, both facts must be created at the same point in time $i$ and therefore by a single application of a rule.

### 3.6.3 First-Order Logic Formula

TAMARIN supports first-order logic formulas consisting of the following components:

- **Quantifiers**

  Both existential (Ex) and universal (All) quantifiers are supported. The quantification can be over terms and timestamps. Timestamps must be prefixed with $#$ unless they directly follow an $@$ such as in $\text{ActionFact1(s)@i}$
3. TAMARIN

- **Logical Operators**

  TAMARIN supports conjunction (\&), disjunction (!), negation (not), and implication (=>)

- **(In)equalities**

  Supported are
  - term equality term1 = term2
  - timestamp equality #i = #j
  - timestamp inequality #i < #j

- **Timestamped Action Facts**

  e. g. ActionFact2(m)@i

### 3.7 Oracle

In general, the security of a protocol is an undecidable problem [8]. By using appropriate proof strategies, however, TAMARIN often succeeds despite this. Nevertheless, TAMARIN’s built-in strategies do not work for all protocols and properties. It can, therefore, be necessary to provide a custom strategy. The TAMARIN way of doing this is by providing it with an oracle. An oracle is a program which receives as its input a list of outstanding proof goals, e. g.

```
0: !KU(!priv_e.1) @ #vk.14
1 1: splitEqs(6)
2 2: splitEqs(7)
3 3: !KU( enc([], 'I0', 'T1', payload4)) @ #vk.1
4 4: !KU( enc([], 'I0', 'T1', payload3)) @ #vk.2
5 5: !KU( enc([], 'I0', h2([], payload2)) @ #vk.5
6 6: !KU( enc([], 'I0', h2([], 'g'^([])))) @ #vk.10
7 7: !KU( enc([], 'I0', h2([], payload1.1)) @ #vk.11
8 8: splitEqs(2)
9 9: splitEqs(4)
10 10: splitEqs(10)
```

and outputs their indexes in the order of the priority in which they should be solved.
3.7. Oracle

1 0
2 1
3 2
4 8
5 9
6 10
7 3
8 4
9 5
10 6
11 7

Since TAMARIN only ever uses the highest priority goal to continue the proof, the output can also be truncated to a single line.
Chapter 4

Modelling

The main contribution of this thesis is a tool which automatically verifies the security of Noise protocols [70]. As previously mentioned, this is done in a five-step process consisting of:

1. (Optionally) converting Noise protocol names into their patterns.
2. Compiling these patterns into an intermediate language.
3. Compiling the intermediate language into a TAMARIN protocol specification. Thereby also adding the property descriptions.
4. Running TAMARIN on said protocol description and providing an oracle where appropriate.
5. Formatting TAMARIN’s results.

This chapter looks at each of these steps in turn.

4.1 Generating Protocol Patterns

The names of Noise protocols published in the specification follow a strict naming scheme. This allows for the automated generation of the corresponding patterns.

In general, the base protocol names consist of two uppercase letters, which specify how the participants’ static public keys are exchanged. The first letter refers to the initiator’s key, the second to the responder’s. The letters have the following meaning:

- I

  The corresponding static public key is sent in the very first message. I can only be used for the initiator.
4. Modelling

- **X**

  The corresponding static public key is sent in a handshake message. However, the ee and es or se DH operations are performed first if the necessary keys are available to both parties.

- **K**

  The corresponding static public key is already known to the other participant and does not have to be transmitted.

- **N**

  The corresponding static public key is not used in the protocol.

Any DH, for which all keys are available (to both parties) are performed in the order ee, se, es, ss, assuming Alice is the initiator. If Bob is the initiator, the order is ee, es, se, ss. The ss DH operation is only done if either es or se has not been performed yet. Additional messages are added until all public keys which are used in the protocol have been sent.

If one or both of the letters is followed by a 1, the corresponding participant’s authentication is said to be deferred. If either authentication is deferred, no ss DH is performed. If Alice’s authentication is deferred, the se DH operation is done one message later than it normally would be. If necessary, an additional message is added to the protocol. The same goes for Bob’s authentication and es DHs.

Unlike the interactive protocols above, the names of one-way protocols only consist of one letter. Since the responder does not have the opportunity to contribute any key material during the protocol run, the initiator must already know the responder’s static key before the protocol run starts. Therefore, only the initiator’s behaviour is specified. Additionally, this letter may only be one of N, X, or K. However, for one-way protocols, X behaves as I does for interactive protocols.

The full rules as to how the protocol patterns can be generated are available in [64].

4.2 Generating Intermediate Language

In order to improve the reusability of some components, the compilation of the Noise patterns into a Tamarin model is performed in two steps. The first one translates Noise patterns into a custom intermediate language. This intermediate language is much more explicit about the cryptographic operations which are performed. The Noise token ee, for instance, is translated
4.2. Generating Intermediate Language

1 \( ee = \text{DH}(\text{priv}_e, \text{re}) \)
2 \( \text{temp} = \text{HMAC-HASH}(\text{ck}, \text{ee}) \)
3 \( \text{ck} = \text{HMAC-HASH}(\text{temp}, 1) \)
4 \( k = \text{HMAC-HASH}(\text{temp}, \text{ck} || 2) \)
5 \( n = 0 \)

In this intermediate language, protocols are specified separately for both participants. A participant’s role specification consists of

1. the participant’s name. i.e. either Alice or Bob

2. a list of the participant’s initial knowledge. This list must contain all data the participant requires to perform the protocol but is neither received nor computed during the protocol run. Specifically, this may include
   - \( \text{priv}_e / e \) the participant’s ephemeral private or public DH key respectively
   - \( \text{priv}_s / s \) the participant’s static private or public DH key respectively
   - \( \text{re} / \text{rs} \) the other participant’s ephemeral or static public DH key respectively if said key is in a pre-message of the protocol
   - \( \text{psk}N \) the \( N \)-th PSK used in the protocol
   - \( \text{payload}N \) the payload which will be sent with the \( N \)-th message of the protocol
   - \( \text{prologue} \) the prologue to the protocol, i.e. some data for which the protocol should ensure that it is equal for both participants.

3. a list of statements used to specify the participant’s behaviour during a protocol run. These statements may be of one of the following forms
   - an assignment statement as in the example above
     Assignments set the value of the variable named by the identifier to the left of the ‘=’ to the result of the expression on the right. Variables which have a special meaning (see the list of possible preknowledge above) cannot have their values changed once it is set. Additionally, \( e, s, \text{priv}_e, \text{priv}_s, \text{psk}N \) and \( \text{prologue} \) may not be assigned to at all.
   - a send statement (send(message))
     Sends are used to represent sending a message to the other participant in the protocol. The message consists of the evaluated results
4. Modelling

of the expressions which are the parameters of the send. For the purposes of the intermediate language, messages are assumed to

– be delivered synchronously
– never be dropped
– always be delivered in order
– not be tampered with

• a receive statement \( \text{receive}(\text{message}) \)

The recipient of a message can access it with a receive. receives take one or more identifiers as parameters. The variables named by these identifiers are set to the values in the message. For any valid protocol, receives will always eventually be successful. In other words: Protocol specifications, in which sends and receives do not properly match up are invalid. For example, this can be because both roles start with a receive or because a send and the corresponding receive do not have the same number of parameters.

• a return statement \( \text{return} (k1, k2) \)

The purpose of returns is to indicate data which is of interest outside the context of the protocol run such as the negotiated keys or received payloads. The Noise verification tool presented here only uses returns to prevent the dead code elimination optimization from being overzealous.

A full example of what a protocol looks like in the intermediate language can be found in Appendix B.

4.3 Generating Tamarin Models

For the actual analysis, the protocol description has to be translated into a Tamarin model.

Conceptually, the intermediate language to Tamarin model compilation process can be split into nine steps.

1. Parsing Intermediate Language
2. Transform to Static Single Assignment Form (SSA)
3. Improve Readability
4. Optimize Decryption
5. Generate Blocks
6. Emit Equations
4.3. Generating TAMARIN Models

7. Emit Key- and Session Management Rules
8. Emit Protocol Rules
9. Emit Lemmas

Each step is explained below.

An example of what the generated TAMARIN models look like can be found in Appendix C.

4.3.1 Parsing Intermediate Language

In order to be compiled into a TAMARIN model, a specification in the intermediate language first has to be parsed. This is done by a parser generated by the ANTLR [55] parser generator.

4.3.2 Transform to Static Single Assignment Form (SSA)

Assignments in the intermediate language are eventually translated into let bindings in the TAMARIN model. Because of the way they are implemented, two let bindings with the same identifier on their left-hand side can lead to results inconsistent with the intent of the model’s author [68]. Therefore, the protocol description is transformed into a form where no variable is assigned to multiple times. This is called a static single assignment form or SSA for short.

Due to the particularities of the intermediate language, most notably its lack of control flow structures, this can be achieved by simply using a new variable for every assignment destination. The remainder of the protocol specification is then updated to use the new variable.

4.3.3 Improve Readability

The TAMARIN model is primarily intended to be read by TAMARIN. However, keeping it human readable has the benefit of making it easy to verify its correctness or, alternatively, allows the model to be used as a well-formalized protocol specification. More importantly, NOISE’s reliance on hash chains makes some of TAMARIN’s output difficult to read. In order to somewhat improve readability, a few additional steps are performed.

Constant Propagation

Since the generated intermediate language is very similar to the way the protocol is defined in the NOISE specification, the ‘nonce’ n is assigned to every few statements. This can make it easy to misjudge what value the nonce has at any given point in the protocol. Constant propagation improves
4. Modelling

upon this situation by replacing any usage of \( n \) with the appropriate integer literal.

**Dead Code Elimination**

The intermediate language generation is somewhat verbose. Particularly after the constant propagation step, this leads to many assignments, whose values are never read. Such assignments do not contribute to the protocol in any way and are therefore removed.

**Shorten Constants**

In Noise, all hash chains begin with a hash of the, often lengthy, protocol name (e.g. ‘Noise_IKpsk2_DH_CIPHER_HASH’). However, the full name of the protocol rarely provides useful insights, and the hash chains are often unwieldy enough without. Therefore, the protocol name is shortened. Usually to ‘T2’.

**Simplify Hash Chains**

Noise protocols contain many expressions of the form

\[
\text{h('<h<h('<h('<h('<h('<h('<h('<h('<h('<h('<h('('T2'), e1'), e2'), e3'), e4'), e5')
\]

which Tamarin then pretty prints as

1
2
3
4

\[
\text{h('<h('<h('<h('<h('<h('<h('<h('<h('<h('<h('('T2'), e1'), e2'), e3'), e4'), e5')
\]

By introducing a new hash function \( h2 \) which takes two parameters, this formatting can be slightly improved to

1
2
3

\[
\text{h2(h2(h2(h2(h2(h2(h2(h2(h2(h2('('T2'), e1), e2), e3), e4), e5))}
\]

It is possible to specify protocols in the intermediate language where this modification leads to various issues. Therefore, this is only done if it can be determined that this modification is safe for a specific protocol. This is, however, always the case for protocol specifications generated by the toolchain presented here.

**Simplify HMAC Chains**

Noise protocols’ HMAC chains tend to be even more unreadable than the hash chains. Therefore,
4.3. Generating Tamarin Models

hmac(hmac(ck, expr, 'byte'))

is replaced with
hmac(ck, expr, 'byte')

Thereby simplifying

1 hmac(hmac(hmac(hmac(hmac(hmac(hmac('T2'), e1), 'I1'), e2),
2 'I1'),
3 e3),
4 'I1')

1 hmac(hmac(hmac('T2'), e1, 'I1'), e2, 'I1')

As with the simplification of hash chains, this cannot be done for every protocol which can be specified in the intermediate language. HMACs, however, can always be simplified in this manner for protocol specifications generated by the toolchain presented here.

4.3.4 Optimize Decryption

The authenticated encryption used by Noise is modelled with

- three functions
  - enc(key, nonce, ad, plaintext)
  - dec(key, ciphertext)
  - verify(key, nonce, ad, ciphertext)
- a constant
  verified
- two equations
  - dec(k, enc(k, n, ad, d)) = <d, ad, n>
  - verify(k, n, ad, enc(k, n, ad, d)) = verified
- and a restriction

1 restriction Equality:
2 "All a b #i. Equal(a, b)@i ==> a = b"
This restriction ensures that any rule which creates an \( \text{Equal}(a, b) \) action fact can only be applied if \( a \) and \( b \) are equal. Specifically, a rule with an

\[
\text{Equal}(\text{verify}(k, n, ad, ciphertext), \text{verified})
\]

fact can only be applied if

\[
ciphertext = \text{enc}(k, n, ad, plaintext)
\]

for some plaintext.

If corresponding \( \text{Equal} \) facts are added to all rules which decrypt something, this adequately models authenticated encryption with additional data [7].

1 rule DecryptionExample1:
2 \[\text{In(ciphertext), CipherState}(k, n, ad)\]
3 \[\text{--}[\text{Equal}(\text{verify}(k, n, ad, ciphertext), \text{verified})]\rightarrow\]
4 \[\text{Plaintext}(\text{fst}(\text{dec}(k, ciphertext)))\]

However, in cases like this, it is possible to replace ciphertext with

\[
\text{enc}(k, n, ad, plaintext)
\]

and let TAMARIN’s pattern matching on the received messages perform both the decryption and verification.

1 rule DecryptionExample2:
2 \[\text{In(\text{enc}(k, n, ad, plaintext)), CipherState}(k, n, ad)\]
3 \[\text{--}\rightarrow\]
4 \[\text{Plaintext}(plaintext)\]

Since the second version leads to significantly faster verifications, it is used whenever this is possible.

4.3.5 Generate Blocks

TAMARIN models most protocols as several multiset rewrite rules for each participant. The intermediate language, however, only uses one list of statements per participant. Therefore, both of these lists have to be split into several lists of statements, each of which is then translated into one rewrite rule.

This is done by splitting a participant’s statement list just before any receive which is preceded by a send. For example,
4.3. Generating Tamarin Models

Alice:

1. \( a = 'A' \)
2. \( \text{send('B')} \)
3. \( \text{send('C')} \)
4. \( d = 'D' \)
5. \( \text{receive(e)} \)
6. \( \text{send(a || d || e)} \)

is translated into

rule Alice1:

1. \([\text{Alice}_0()]\)
2. \(\rightarrow\)
3. \([\text{Out('B'), Out('C'), Alice}_1('A', 'D')]\)

rule Alice2:

1. \([\text{Alice}_1(a, d), \text{In(e)}]\)
2. \(\rightarrow\)
3. \([\text{Out('<a, d, e>')}\]

4.3.6 Emit Equations

Once all prerequisites are completed, the code generation begins with the function declarations and equations. The exact code depends on which of the changes from Section 4.3.3 were applied. Generally, the result looks similar to

builtins: diffie-hellman, hashing
functions: hmac/3, h2/2, enc/4, dec/2, verified/0, verify/4
equations: \(\text{dec(k, enc(k, n, ad, d)) = <d, ad, n>}, \text{verify(k, n, ad, enc(k, n, ad, d)) = verified}\)

meaning that the model uses Tamarin’s built-in Diffie-Hellman and hashing support as well as the functions and equations discussed in Sections 4.3.3 and 4.3.4.

4.3.7 Emit Key- and Session Management Rules

Apart from the multiset rewrite rules that model the actual protocol run, the Tamarin model also needs additional rules. Specifically, rules are needed to

- generate static keys for protocol participants and announce them to the world.
- model the compromise of these public keys.
• pair up participants and start protocol runs.

If the protocol uses PSKs, it also requires rules to generate them and to model PSK compromise.

### 4.3.8 Emit Protocol Rules

The intermediate language is translated into multiset rewrite rules statement by statement. Assignments are turned into let bindings. `send` and `receive` become `Out` and `In` facts respectively.

The rewrite rule is also labelled with appropriate action facts. For instance, the `RE` and `RS` labels are used to indicate when one party has received the other party’s ephemeral and static keys respectively. Similarly, `CommitPayload` is used to indicate when a payload is sent while `ReceivePayload` labels the payload’s receipt. The full list of used action facts can be found in Appendix A.

### 4.3.9 Emit Lemmas

Finally, a TAMARIN model must also specify the properties which should be verified. These are discussed in Chapter 5.

### 4.4 Running Tamarin

Once a model is generated, it is verified by invoking TAMARIN on it. For simple NOISE handshakes, this usually works fairly well. As of TAMARIN version 1.4.0, however, TAMARIN’s built-in proof strategy fails to efficiently verify some lemmas for certain protocols. This can lead to excessive memory usage to the point where the proof has to be aborted.

#### 4.4.1 Oracle

In cases where TAMARIN fails to complete a proof on its own, an oracle is used to guide the proof. Heuristics are used to determine when this is necessary.

By selectively using the oracle in this way, all 61 handshake patterns of the NOISE specification can be verified. However, by adding enough transport messages to the model, NOISE protocols can be made sufficiently complex that the verification fails even with the oracle. See Section 4.6.5 for more on transport messages.
4.5 Result Presentation

TAMARIN has no understanding of the meaning of the verified properties. Therefore, all it can do is simply list the results. This quickly becomes difficult to read. Particularly with complex protocols, which can easily have over 70 lemmas.

1. Executability (exists-trace): verified (10 steps)
2. ThreadStartedOnce (all-traces): verified (53 steps)
3. AliceThreadUniqueness (all-traces): verified (10 steps)
4. BobThreadUniqueness (all-traces): verified (10 steps)
5. HashEqualityImpliesKeyEquality (all-traces): verified (2 steps)
6. NoKeyReuse (all-traces): verified (2499 steps)
7. SNotSetTwiceAndSafe (all-traces): verified (29 steps)
8. RSNotSetTwice (all-traces): verified (42 steps)
9. ENotSetTwiceAndSafe (all-traces): verified (19 steps)
10. RENotSetTwice (all-traces): verified (21 steps)
11. PrologueEquality (all-traces): verified (400 steps)
12. AliceKeySecrecy (all-traces): verified (35 steps)
13. BobKeySecrecy (all-traces): verified (33 steps)
14. AliceKeyUniqueness (all-traces): verified (76 steps)
15. BobKeyUniqueness (all-traces): falsified - found trace (14 steps)
16. AliceHashUniqueness (all-traces): verified (6 steps)
17. BobHashUniqueness (all-traces): falsified - found trace (15 steps)
18. Payload1DestinationProperty1 (all-traces): verified (19 steps)
19. Payload1DestinationProperty2 (all-traces): verified (32 steps)
20. Payload1DestinationProperty3 (all-traces): falsified - found trace (15 steps)
21. Payload1DestinationProperty4 (all-traces): falsified - found trace (15 steps)
22. Payload1DestinationProperty5 (all-traces): falsified - found trace (15 steps)

Additionally, the full verification of a NOISE protocol requires two TAMARIN runs. See Section 4.6.4 for more on this.

In order to make it easier to assess and compare protocols at a glance, the NOISE verification tool combines the outputs of both TAMARIN runs. Since many of the lemmas should always hold, they are ignored as long as their verification results match expectations. Additionally, whenever one property implies another, only the strongest property that holds is listed. For example, the AuthenticationProperty of a message being listed as 3 means that the message’s payload satisfies authentication properties 1 through 3 but not 4.
The meaning of these properties can be found in Chapter 5. This formatting results in outputs similar to the following.

```
1 AliceHashUniqueness: yes
2 AliceKeySecrecy: yes
3 AliceKeyUniqueness: yes
4 BobHashUniqueness: no
5 BobKeySecrecy: yes
6 BobKeyUniqueness: no

7 AuthenticationProperty:
8 X:
9 <- s
10 ...
11 -> e, es, s, ss: 3
12
13 DestinationProperty:
14 X:
15 <- s
16 ...
17 -> e, es, s, ss: 2
18
19 KCIAuthenticationProperty:
20 X:
21 <- s
22 ...
23 -> e, es, s, ss: 0
24
25 SourceProperty:
26 X:
27 <- s
28 ...
29 -> e, es, s, ss: 1
```

### 4.6 Model Design

This section discusses some design decisions which have a significant effect on the resulting TAMARIN model.

#### 4.6.1 Invalid DH keys

During the course of a Noise protocol run, unencrypted Diffie-Hellman public keys are sent and received over the network. An attacker can, therefore,
substitute these with particularly weak DH keys. For example, an attacker could set such a key to 0. The shared secret derived from public key 0 and any private key \( k \) is

\[ 0^k = 0 \]

and is therefore known to the attacker.

The **Noise** specification allows two ways for dealing with weak or invalid public keys. The preferred way is to detect them but nevertheless continue the protocol run using some special value as the result of the DH-exponentiation. This special value must not depend on the private key. Alternatively, the protocol run can be aborted if an invalid public key is detected.

The **Tamarin** model uses the second option since it is a better fit for **Tamarin**’s Diffie-Hellman model. This is implemented by adding the two let bindings

```plaintext
1 \text{ re } = \text{'g'}^{\text{priv_re}}
2 \text{ rs } = \text{'g'}^{\text{priv_rs}}
```

to the beginning of all protocol rules. Using **Tamarin**’s pattern matching, these then ensure that the partner’s public keys (\( \text{re} \) and \( \text{rs} \)) are indeed elements of the DH group, i.e. \( \text{'g'}^{\text{something}} \).

### 4.6.2 Dummy keys

For some use cases, it may be desirable for parties to the protocol to be able to optionally authenticate themselves. For instance, because the other party requests this in the course of the handshake. However, it is a design goal for **Noise** to require that any protocol negotiation is completed before a **Noise** handshake begins [65].

**Static Keys**

The suggested solution to this is to use a **Noise** protocol that performs the authentication in question. Unless the authentication is requested, the corresponding static key can be replaced with a dummy value.

The specification also contains what essentially amounts to a suggested API for implementations of the **Noise** framework. However, this API requires that the local static key be provided before the protocol run begins. It is, therefore, unclear how an application might implement such optional authentication.

This is one of the reasons the **Noise** verification tool does not support such optional authentication. Another reason is that **Noise** does not provide any
guidance as to how the authentication should be negotiated. This negotiation is, therefore, entirely application dependent and could not be modelled in any generally applicable way.

**PSKs**

Similarly, the suggested way of optionally requiring a PSK is to use a protocol that uses a PSK and replace the PSK with a known constant, for example 0x0000... , whenever the PSK is not necessary. Unlike with the static keys, the dummy value for PSKs should always be known to both parties in advance, since Noise protocols do not transmit PSKs.

Also unlike with the static keys, the suggested API for specifying a PSK is for Noise to request it from the application once the PSK is needed [56]. This usage of PSKs is therefore supported by the suggested Noise API.

Nevertheless, optional PSKs are not supported by the Noise verification tool discussed here. As with dummy static keys, this would require a generic model for a negotiation protocol.

The Noise specification explicitly allowing weak PSKs, however, is one reason the verification tool does not consider the knowledge of a PSK to be sufficient for identifying a specific protocol participant. The other being that for PSKs to work, they must be known to at least two parties.

It is also not safe to assume that using a dummy PSK provides identical guarantees to a version of the protocol with the PSK removed.

Using a publicly known PSK does negate most advantages of having a PSK in the first place. However, Noise protocols with a PSK mix the ephemeral keys into the encryption key, while non-PSK protocols do not. This additional randomization of the key can prevent some obscure attacks. The attack presented below is unlikely to be of any practical significance, requires a stronger attacker model in which ephemeral keys can also be compromised and only serves as an illustration of how all-zero PSKs can make a difference.

An attacker who has compromised Alice’s ephemeral DH keys of multiple sessions of the Noise protocol

1. \[NN:\]
2. \[\rightarrow e\]
3. \[\leftarrow e, ee\]

...can cause Alice to reuse the same key and use the resulting messages to attack the underlying cryptography. More explicitly, an attacker who knows

- Bob’s public \(g^{k_{B,1}}\) and Alice’s private \(k_{A,1}\) keys of a previous session as well as
• Alice’s private key \((k_{A,2})\) of a session in which Alice has not yet received message 2 (line 3)
can pass off
\[
(g^{k_{B,1}})^{k_{A,1} * k_{A,2}} = g^{k_{B,1} * k_{A,1} * k_{A,2}}
\]
as Bob’s public key. Thereby making Alice reuse
\[
(g^{k_{B,1} * k_{A,1} * k_{A,2}})^{k_{A,2}} = g^{k_{B,1} * k_{A,1} * k_{A,2} * k_{A,2}} = g^{k_{B,1} * k_{A,1}}
\]
as the shared secret from which the final encryption keys are generated. However, since the attacker must essentially already know the encryption key in order to perform it, this attack is not all that worrying. Nevertheless, adding a PSK, e.g. by using

```plaintext
1 \texttt{\texttt{NNpsk0}}:
2 \texttt{\texttt{\rightarrow psk, e}}
3 \texttt{\texttt{\leftarrow e, ee}}
```

instead of the \texttt{NN} protocol solves this issue. Even if the PSK is 0x0000... . Since the ephemeral keys are also hashed into the keys with \texttt{NNpsk0}, reusing the same shared secret does not lead to the same encryption keys.

### 4.6.3 Key Compromise

\texttt{Noise} uses three different kinds of secrets from which the encryption keys are derived:

1. Ephemeral DH Keys
2. Static DH Keys
3. PSKs

The assumptions the \texttt{Noise} specification makes as to whether a secret can be compromised depend strongly on its type. This section discusses how these assumptions are stated in the specification and how this is reflected in the \texttt{Tamarin} model.

#### Ephemeral DH Keys

The \texttt{Noise} specification barely discusses the compromise of ephemeral keys. And where this is mentioned, it is to emphasize that even the strongest \textit{destination} (i.e. secrecy) property assumes that ephemeral keys are not compromised.

Therefore, ephemeral keys are modelled in such a way that their private keys cannot be compromised by an attacker.
4. Modelling

Static DH Keys

Static keys are very different in this respect. The security properties defined in the specification are characterized by which static keys can be compromised when without anything ‘bad’ happening. This implies that they can, in fact, be compromised.

Consequently, static keys can also be compromised in the model.

PSKs

In contrast to this, PSK compromise is not mentioned at all. However, the fact that PSKs are used over potentially long periods of time makes them just as likely to be compromised as static keys. Therefore, both Noise Explorer and the Noise verification tool presented in this thesis model attackers which can compromise PSKs.

This leaves the question of how the security properties need to be adjusted. Both Noise Explorer and the tool presented here essentially replace mentions of compromised and safe static keys with compromised and uncompromised protocol participants, respectively.

However, the two tools model a ‘compromised party’ differently. Noise Explorer considers a party as compromised only once an attacker has compromised both its static key and PSK. For the purposes of the Tamarin based tool, on the other hand, a party is compromised as soon as either its static key or any of its PSKs are compromised.

4.6.4 Differing Payload Sources

In order to make the verification results applicable in many situations, the attacker model should be as strong as reasonably possible. This includes giving the attacker full control over the payloads which are sent.

However, trying to prove secrecy of a value the attacker chose in the first place is hopeless. In order to both use a strong attacker model and obtain useful results for secrecy, two different Tamarin models are generated for each protocol. One uses payloads which are obtained from In(payload) facts and are, therefore, under attacker control. The other model uses a new fresh value for every payload.

While most lemmas are included in both model version, for the aforementioned reasons, secrecy (destination) properties are only verified with the latter. Authentication properties, including the source properties, on the other hand, are only verified in the model with attacker controlled payloads. This is primarily because they can become very numerous for longer protocols. Since the model using fresh payloads already tend to be more difficult to
verify, the authentication properties are only verified against the stronger attacker model.

### 4.6.5 Transport Messages

All handshake patterns mentioned in the *Noise* specification end once all DH-operations are performed. However, receiving and successfully decrypting a message encrypted with a key derived from the final DH-operation confirms that the partner has ended up with the same key. This, in turn, can help rule out some types of attacker interference in the protocol run. Because of this, some handshake patterns provide better guarantees after the negotiated keys are used once or twice.

In order to accommodate this, the verification tool can be configured to add *transport messages* to the generated *Tamarin* model. These transport messages only consist of a payload encrypted with a key negotiated by the preceding *Noise* handshake. Modelling such transport messages allows for the verification of the security properties obtained after the negotiated keys are successfully used.
This chapter presents the lemmas used by the Noise verification tool to formalize various security properties. The labels used in the lemmas are outlined in Appendix A. Appendix D lists which properties are satisfied by each of the standard Noise handshake patterns in turn.

5.1 Sanity Checks

Most sanity checks should always hold for all protocols. They are therefore often trivial. Errors in the TAMARIN model or the understanding of the protocol might, however, be caught by them.

This section also includes properties like NoKeyReuse, which do verify interesting and non-trivial security properties on the protocols. These properties should nevertheless hold for all Noise protocols. Either because the specification states this, or, as with HashEqualityImpliesKeyEquality, because failing to do so would be strongly counterintuitive and make suggested use cases of Noise insecure.

A special case is the PrologueEquality lemma. Unlike all other sanity checks, there are valid Noise handshake patterns which do not satisfy it. See Section 5.1.10 for details.

All sanity checks property were successfully proven for all Noise protocols named in the specification.
5. Properties

5.1.1 Executability

lemma Executability:
exists-trace
"
(Ex tidA tidB k1 k2 n1 n2 hash #i #j.
   AliceKeys(tidA, k1, k2, n1, n2, hash)@i &
   BobKeys(tidB, k1, k2, n1, n2, hash)@j &
   not (Ex s #i. Reveal(s)@i) &
   not (Ex psk #i. RevealPSK(psk)@i) &
   All tid1 tid2 role s1 s2 #i #j.
   Running(tid1, role, s1)@i & Running(tid2, role, s2)@j
   ==> tid1 = tid2 & s1 = s2 & #i = #j
"

This lemma verifies that it is possible for Alice and Bob to agree on the negotiated keys, nonces and hashes (lines 4–6) without any honest party being compromised (lines 7 & 8).

For exists-trace lemmas which hold, TAMARIN can generate a graph that depicts a protocol execution which satisfies the lemma. In order to make this graph more readable, the lemma has been strengthened to only allow for one instantiation of each of the protocol roles (lines 9–12).

5.1.2 ThreadStartedOnce

lemma ThreadStartedOnce:
"
All tid role1 role2 s1 s2 #i #j.
   Running(tid, role1, s1)@i & Running(tid, role2, s2)@j
   ==> #i = #j & role1 = role2 & s1 = s2
"

The ThreadStartedOnce lemma verifies that thread ids are unique and that no thread can be started multiple times.
5.1.3 Thread Uniqueness

1 lemma AliceThreadUniqueness:
2 "
3 All tid k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1 hash_2 #i #j.
4 AliceKeys(tid, k1_1, k2_1, n1_1, n2_1, hash_1)@i &
5 AliceKeys(tid, k1_2, k2_2, n1_2, n2_2, hash_2)@j
6 ==> #i = #j
7 "

AliceThreadUniqueness verifies whether the protocol ensures that no thread performing the Alice role can run to completion multiple times. BobThreadUniqueness does the same for threads performing the Bob role.

5.1.4 HashEqualityImpliesKeyEquality

1 lemma HashEqualityImpliesKeyEquality:
2 "
3 All tida tidb k1a k1b k2a k2b n1a n1b n2a n2b hasha hashb #i #j.
4 AliceKeys(tida, k1a, k2a, n1a, n2a, hasha)@i &
5 BobKeys(tidb, k1b, k2b, n1b, n2b, hashb)@j &
6 hasha = hashb
7 ==> k1a = k1b & k2a = k2b
8 "

This lemma verifies that as long as both Alice and Bob agree on the transmitted data and involved PSKs (i.e. all the data that is mixed into the handshake hash), they end up with the same keys.

5.1.5 NoKeyReuse

1 lemma NoKeyReuse:
2 "
3 All tid1 tid2 k n d1 d2 #i #j.
4 KeyUse(tid1, k, n, d1)@i & KeyUse(tid2, k, n, d2)@j
5 ==> d1 = d2 & #i = #j
6 "

Using the same key and nonce combination to encrypt several plaintexts
5. Properties

...can lead to serious issues. This lemma verifies that the protocol ensures this cannot happen.

5.1.6 SNotSetTwiceAndSafe

lemma SNotSetTwiceAndSafe:
"All tid pk1 pk2 #i #j.
(S(tid, pk1)@i & S(tid, pk2)@j) ==> pk1 = pk2 & #i = #j & (Ex #k. HonestS(pk1)@k)"

SNotSetTwiceAndSafe verifies whether the protocol ensures that no participant has more than one local static key per protocol run. It also checks that any static key used by a participant as its own was actually created by an honest party.

5.1.7 RSNotSetTwice

lemma RSNotSetTwice:
"All tid pk1 pk2 #i #j.
(RS(tid, pk1)@i & RS(tid, pk2)@j) ==> pk1 = pk2 & #i = #j"

RSNotSetTwice verifies whether the protocol ensures that no participant uses more than one remote static key per protocol run. Essentially it verifies that the protocol follows the ‘no key is sent twice’ validity rule for static keys.
5.1.8 ENotSetTwiceAndSafe

lemma ENotSetTwiceAndSafe:
" All tid pk1 pk2 #i #j.
(E(tid, pk1)@i & E(tid, pk2)@j) =>
pk1 = pk2 & #i = #j &
(Ex #k. HonestE(pk1)@k) &
not (Ex #k. Reveal(pk1)@k) "

ENotSetTwiceAndSafe verifies whether the protocol ensures that no participant has more than one local ephemeral key per protocol run. It also checks that any ephemeral key used by a participant as its own was actually created by an honest party and has not been leaked. The compromise of ephemeral keys is not modelled since revision 34 of NOISE does not consider this a likely threat. Upcoming versions of NOISE may change in this respect [61].

5.1.9 RENotSetTwice

lemma RENotSetTwice:
" All tid pk1 pk2 #i #j.
(RE(tid, pk1)@i & RE(tid, pk2)@j) =>
pk1 = pk2 & #i = #j "

RENotSetTwice verifies whether the protocol ensures that no participant uses more than one remote ephemeral key per protocol run. Essentially it verifies that the protocol follows the 'no key is sent twice' validity rule for ephemeral keys.
5. Properties

5.1.10 PrologueEquality

lemma PrologueEquality:
" 
All tidA tidB k1 k2 n1 n2 hashA hashB #i #j.
AliceKeys(tidA, k1, k2, n1, n2, hashA)@i &
BobKeys(tidB, k1, k2, n1, n2, hashB)@j &
not (Ex prologue #k #l.
  PrologueAlice(tidA, prologue)@k &
  PrologueBob(tidB, prologue)@l)
==>
(Ex s #k #l. S(tidA, s)@k & Reveal(s)@l) | 
(Ex rs #k #l. RS(tidA, rs)@k & Reveal(rs)@l) | 
(Ex s #k #l. S(tidB, s)@k & Reveal(s)@l) | 
(Ex rs #k #l. RS(tidB, rs)@k & Reveal(rs)@l) | 
(Ex psk #k #l. PSK(tidA, psk)@k & PSK(tidB, psk)@l & 
  RevealPSK(psk)@m)
"

The PrologueEquality lemma verifies that if the key agreement completes successfully, meaning both parties actually agree on the keys (lines 4 & 5), then both parties also agree on the prologue (lines 6–8), as long as no secret has been compromised (lines 10–16). Unlike all other sanity check lemmas, this lemma is not expected to hold for all valid NOISE protocols. Instead, it holds if and only if the NOISE pattern contains at least one ee, es, se, ss or psk token, i.e. if the negotiated key is expected to be secret.

5.2 Key Agreement Properties

In order to accommodate a wide variety of use cases, NOISE protocols vary widely in what properties they guarantee on the keys they generate. Because of this, some participants can even be forbidden from using the negotiated keys for encryption. The key secrecy and key uniqueness lemmas verify some properties intended to allow an informed decision as to how these keys may be used. The hash uniqueness properties can be used to ensure that the handshake hash is used properly. Although all these properties are verified from both Alice’s and Bob’s perspective, only Alice’s lemmas are presented below, since Bob’s are equivalent.
5.2. Key Agreement Properties

5.2.1 Key Secrecy

lemma AliceKeySecrecy:
"All tid k1 k2 n1 n2 hash #i.
AliceKeys(tid, k1, k2, n1, n2, hash)@i
==> ((not Ex #j. K(k1)@j) & (not Ex #j. K(k2)@j)) |
(Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)
"

AliceKeySecrecy verifies that, unless a party participating in the protocol run has been compromised (lines 8–10) or is malicious from the start (line 7), any key Alice agrees to (line 4) is not known to an attacker (line 6).

5.2.2 Key Uniqueness

lemma AliceKeyUniqueness:
"All tid_1 tid_2 k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1
hash_2 #i #j.
AliceKeys(tid_1, k1_1, k2_1, n1_1, n2_1, hash_1)@i &
AliceKeys(tid_2, k1_2, k2_2, n1_2, n2_2, hash_2)@j &
(k1_1 = k1_2 | k2_1 = k2_2)
==> tid_1 = tid_2"

In order to prevent reuse of the same key, either party may only use the keys obtained from the Noise handshake if they cannot be tricked into ending up with the same key twice. AliceKeyUniqueness verifies that the key is safe to use for Alice.

Note that Noise’s validity rules which are intended to prevent key reuse are more restrictive than strictly necessary. For example,
IXpsk2+ss+noes:
"-> e, s
<- e, ee, se, s, ss, psk"

is a valid handshake pattern which satisfies key uniqueness for both Alice and Bob. Nevertheless, Noise explicitly forbids Alice from using the negotiated keys [64]:

59
"After an "ss" token, [Alice] must not send a handshake payload or transport payload unless there has also been an "es" token."

5.2.3 Hash Uniqueness

\[
\text{lemma AliceHashUniqueness:} \quad \text{All } \text{tid}_1 \text{tid}_2 k1_1 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 \text{hash}_1 \text{hash}_2 \#i \#j. \\
\text{AliceKeys}(\text{tid}_1, k1_1, k2_1, n1_1, n2_1, \text{hash}_1)i & \\
\text{AliceKeys}(\text{tid}_2, k1_2, k2_2, n1_2, n2_2, \text{hash}_2)j & \\
\text{hash}_1 = \text{hash}_2 \\
\text{=>} \\
\text{tid}_1 = \text{tid}_2
\]

Noise provides a GetHandshakeHash() function essentially intended to uniquely identify a handshake, allowing applications to bind communications to the handshake. AliceHashUniqueness verifies that no two hashes obtained by parties who participate in the protocol in the Alice role are the same. Note that a protocol which satisfies AliceHashUniqueness does not guarantee that the handshake hash is a suitable identifier for the protocol run in all cases. If, for instance, BobHashUniqueness does not hold, Bob can still end up with the same hash on multiple runs. In such cases, the handshake hash is, therefore, an inadequate identifier.

5.3 Payload Properties

Unlike the previous security properties, payload properties are verified separately for every payload sent or received. The source and destination properties are taken from the Noise specification while the authentication and KCI resistant authentication properties provide a more fine-grained view on the level of authentication the protocols guarantee. The language of the specification of the source and destination properties, however, is open to interpretation. This leads to the lemmas presented here occasionally being different from other formalizations of the same properties, such as by Noise Explorer [9].

The interpretations of Noise’s properties presented below are based on the following considerations, here ordered by their weight in the interpretation.

1. The results of the lemmas match the results published in the Noise specification. At least for the 15 so-called fundamental patterns (See Appendix D, Sections 1 & 2). Assuming that the published results are correct, they are a great indicator of how the properties were meant.
to be understood. The results for the *deferred* patterns (Appendix D, Section 3) are not as indicative since they were generated by *Noise Explorer*. *Noise Explorer* is a third party tool. While its implementations of the properties do provide insight into other possible interpretations, they do not do so for any original intent. *Noise Explorer*’s results are therefore not taken into consideration.

2. *Noise*’s security properties are grouped into the five *destination* and two *source* properties. This grouping of the properties, their number based naming, and the way every payload is only said to ‘have’ one (presumably the highest) property per group give a very strong impression they are meant to be strictly hierarchical. Meaning that, for example, a payload which satisfies *destination property 5* is ‘safer’ than one that only satisfies *destination property 1*.

The properties are worded in a way that also allows them to be interpreted as ‘payloads are not able to satisfy more than one destination and one source property.’ Essentially the properties can be read as ‘the payload provides such level of security but does *not* provide this higher level.’ Instead of using this approach, the lemmas are constructed such that every property is strictly stronger than all preceding ones. Meaning that, for instance, a payload satisfying *source property 2* necessarily also satisfies *source property 1*.

This hierarchical relation leads to lemmas which are often similar to one another. The changes to the previous lemma are therefore highlighted in the lemmas below. Any part of a lemma that is not in the next weaker property’s lemma is typeset in green. Parts which are in the weaker property but not the current one are in red and struck out. If, for instance, example property 1 is

```plaintext
1 lemma ExampleProperty1:
2 "
3 All a #i.
4 FactA(a)#i
5 ==>  
6 (Ex b #j. FactB(b)#j) |  
7 (Ex c #j. FactC(c)#k)  
8 "
```

and example property 2 is
5. Properties

lemma ExampleProperty2:
" All a #i.
  FactA(a)@i
  ==> (Ex c #j. FactC(c)@j & #j < #i)
"

the latter is typeset as

lemma ExampleProperty2:
" All a #i.
  FactA(a)@i
  ==> (Ex b #j. FactB(b)@j) ↓
    (Ex c #j. FactC(c)@j & #j < #i)
"

3. The property definitions in Noise all follow a pattern. First, there is a short description of the property, which is typeset in bold. This is then followed by an explanation of how this property can be achieved and how it is different from the other properties.

All property interpretations used for the lemmas explained in the following two sections strictly follow the first two points. They are all based on the bold parts of the property definitions, although sometimes only loosely. The reason for the actual property definition not being considered more important is that they are often sufficiently ambiguous to require some additional interpretation guide in order for the resulting lemmas to be consistent. The non-bold parts of the property definitions were mostly ignored and only used to differentiate between otherwise equally well justifiable interpretations.

Since all payload properties are verified for every payload, many nearly identical lemmas are generated. Although this section primarily discusses the properties of payload 1, the properties are analogous for the other payloads.

In order to verify all lemmas described in this chapter, two, slightly different, Tamarin models are used. In one, the payloads are attacker controlled while in the other, they are fresh values generated by the sender. Unlike the previously mentioned properties, each payload property is only verified against one of the generated models. This is primarily since verifying the destination (i.e. secrecy) properties against a payload that is chosen by, and therefore known to, the adversary is obviously not useful. Consequently, the
destination properties are validated against the model using fresh payloads. The other payload properties are verified using the model with attacker controlled payloads.

5.3.1 Destination

As of revision 34, Noise calls the secrecy properties of a payload from the sender’s viewpoint destination properties. Noise defines five of them, which are laid out below and differ in whether they consider active or passive attackers and what secrets are assumed to be compromised at which point in time.

Destination Property 1

```
lemma Payload1DestinationProperty1:
  "
  All tid payload1 #i.
  CommitPayload(tid, 'payload1', payload1)@i =>
    not (Ex #j. K(payload1)@j) |
    (Ex re #j. RE(tid, re)@j & not (Ex #k. HonestE(re)@k)) |
    (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
    (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |
    (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
    (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)
  "
```

Noise defines destination property 1 as [64]

“Encryption to an ephemeral recipient. This payload has forward secrecy, since encryption involves an ephemeral-ephemeral DH ("ee"). However, the sender has not authenticated the recipient, so this payload might be sent to any party, including an active attacker.”

Which, for the purposes of the Payload1DestinationProperty1 lemma, is understood to mean that an attacker, who has not compromised any secret key involved in the protocol run, cannot find out the contents of the payload without actively interfering in the specific protocol run the payload is sent in.

However, Tamarin is not designed for the modelling of passive attackers and even less so for attackers who are passive for only a single session [71]. While it is possible, this lemma instead strengthens the property to only
5. Properties

- disallow something other than an honest party’s ephemeral key from being used as the remote ephemeral key for the party making the secrecy claim (line 7)
- disallow something other than an honest party’s static key from being used as the remote static key for the party making the secrecy claim (line 8)
- disallow the attacker compromising any static key or PSK involved in the protocol run, for which the claim is being analyzed (lines 9–12)

Similar changes were necessary for all destination properties that consider an attacker who does not interfere in a given session. Apart from destination property 1, this affects the implementation of weak forward secrecy for destination property 2 and destination property 3.

This approach was taken rather than using an actual passive attacker since

- the property as modelled is strictly stronger than the one specified. The change, therefore, does not hide a possible attack on the protocol
- the change does not lead to different results for any of the patterns named in the specification
- this allows for all properties to be run on basically the same model, only requiring the change in how payloads are generated. Modelling a passive attacker in TAMARIN would require a fundamentally different model for the various properties. Even for the various closely related destination properties.

**Destination Property 2**

```plaintext
lemma Payload1DestinationProperty2:
"
All tid payload1 #i.
CommitPayload(tid, 'payload1', payload1)#i
==> (Ex rs #j. RS(tid, rs)#j) &
    (not (Ex #j. K(payload1)#j)) |
    (Ex re #j. RE(tid, re)#j & not (Ex #k. HonestE(re)#k)) |
    (Ex rs #j. RS(tid, rs)#j & not (Ex #k. HonestS(rs)#k)) |
    (Ex s #j #k. S(tid, s)#j & Reveal(s)#k & #k < #i)) |
    (Ex rs #j #k. RS(tid, rs)#j & Reveal(rs)#k) |
    (Ex psk #j #k.
      PSK(tid, psk)#j & RevealPSK(psk)#k & #k < #i))
"
```

NOISE defines destination property 2 as [64]
“Encryption to a known recipient, forward secrecy for sender compromise only, vulnerable to replay. This payload is encrypted based only on DHs involving the recipient’s static key pair. If the recipient’s static private key is compromised, even at a later date, this payload can be decrypted. This message can also be replayed, since there’s no ephemeral contribution from the recipient.”

The Payload1DestinationProperty2 lemma implements this as weak forward secrecy for sender static key compromise to a known recipient. This means that, in order to be able to decrypt the payload (line 7), an attacker must either

- actively interfere with the session the payload is sent in or
- compromise the payload’s sender before (lines 10, 12 & 13) the payload is sent or
- compromise the payload’s recipient at any point in time (line 11).

Although, same as in destination property 1, the attacker model used does allow for some interference as long as all keys used were originally created by honest parties (lines 8 & 9).

The ‘to a known recipient,’ which is part of all destination properties except for the first, is understood to mean ‘if the sender knows the supposed recipient’s static key’ (line 6).

NOISE EXPLORER [9], however, understands destination property 2 to mean ‘strong forward secrecy for sender static key compromise.’ This differs from the interpretation laid out above in two significant ways. Firstly, the attacker in NOISE EXPLORER’s version is fully active and can manipulate any message in any way. Secondly, the recipient does not have to be ‘known.’ This is clearly a stronger property, and therefore the ‘safer’ interpretation for this purpose, however, it seems unlikely to be the intended one. It does arguably match the actual property description (in bold) better than a weak attacker. However, this leads to NOISE EXPLORER’s destination properties not being hierarchical. For example, the second payload of

```
1  I1K1:
2  <- s
3  ...
4  -> e, s
5  <- e, ee, es
6  -> se
```

satisfies destination property 3. Both according to NOISE EXPLORER and TAMARIN (using the lemma outlined below). It does not, however, satisfy NOISE
5. Properties

EXPLORER’s interpretation of destination property 2. This is a stark contrast to the hierarchical way Noise presents these properties (e.g. naming them ‘destination property 1’ through ‘destination property 5’, only ever listing the largest property number achieved by a payload, ...).

Note that the non-bold part of the property definition strongly implies that payloads satisfying destination property 3 do not satisfy destination property 2. However, Noise EXPLORER does not follow this approach either. In fact, Noise EXPLORER even explicitly ignores higher numbered properties that are achieved if lower ones are not. In the example above, for example, Noise EXPLORER lists payload 2 as only satisfying destination property 1, even though it passes the verification for destination property 3 [13].

The interpretation of destination property 2 using an attacker who is passive during the protocol session in question, however, can also lead to unintuitive results. Due to the ‘to a known recipient’ part of several destination properties being interpreted as knowing their static key, payload 2 (line 3) of

1 NN:
2  -> e
3  <- e, ee

ends up only achieving destination property 1. Payload 2 (line 5) of

1 K1N:
2  <- s
3  ... 
4  -> e
5  <- e, ee
6  -> se

on the other hand also satisfies Payload2DestinationProperty2 as well as Payload2DestinationProperty3, even though the two payload’s encryption keys are generated in basically the same way. The only difference being the protocol name, which is hashed into the key.

Noise EXPLORER, which simply ignores the ‘to a known recipient’ wording of all destination properties, somewhat sidesteps this issue with its more powerful attacker for destination property 2. Such an attacker can impersonate any participant whose static key has not yet been used for the key generation. Since Noise EXPLORER ignores the results of higher destination properties unless its version of destination property 2 is achieved, these kinds of payloads end up getting labelled as destination property 1 by Noise EXPLORER [22].

The Noise specification does list the supposed destination property level achieved by the payloads of the protocol above. However, this is unfortunately not helpful in clarifying this ambiguity. This is because in all the
cases where NOISE EXPLORER’s results differ from those obtained by verifying a payload’s destination property 2 lemma with TAMARIN the results from the specification were merely copied from NOISE EXPLORER [58] [64].

Destination Property 3

```
lemma Payload1DestinationProperty3:
  "All tid payload1 #i.
  CommitPayload(tid, 'payload1', payload1)#i
  =>
  (Ex rs #j. RS(tid, rs)#j) &
  (not (Ex #j. K(payload1)#j) |
  (Ex re #j. RE(tid, re)#j & not (Ex #k. HonestE(re)#k)) |
  (Ex rs #j. RS(tid, rs)#j & not (Ex #k. HonestS(rs)#k)) |
  (Ex s #j #k. S(tid, s)#j & Reveal(s)#k & #k < #i) |
  (Ex rs #j #k. RS(tid, rs)#j & Reveal(rs)#k & #k < #i) |
  (Ex psk #j #k. PSK(tid, psk)#j & RevealPSK(psk)#k & #k < #i))
```

NOISE defines destination property 3 as [64]

“Encryption to a known recipient, weak forward secrecy. This payload is encrypted based on an ephemeral-ephemeral DH and also an ephemeral-static DH involving the recipient’s static key pair. However, the binding between the recipient’s alleged ephemeral public key and the recipient’s static public key has not been verified by the sender, so the recipient’s alleged ephemeral public key may have been forged by an active attacker. In this case, the attacker could later compromise the recipient’s static private key to decrypt the payload. Note that a future version of NOISE might include signatures, which could improve this security property, but brings other trade-offs.”

The lemma implements this by requiring that an attacker who only passively records a specific session (lines 8 & 9) compromise a static key (lines 10 & 11) or a PSK before (lines 12 & 13) the payload is sent in order to read the payload (line 7). As with the previous lemmas, an active attacker with some restrictions is used instead of a passive one. This property also requires that the sender know the recipient’s static public key (line 6) at the point in time where the payload is sent.
5. Properties

Destination Property 4

lemma Payload1DestinationProperty4:
"All tid payload1 #i.
CommitPayload(tid, 'payload1', payload1)@i
==>
(Ex rs #j. RS(tid, rs)@j) &
(not (Ex #j. K(payload1)@j) |
(Ex re #j. RE(tid, re)@j & not (Ex #k. HonestE(re)@k)) |
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k & #k < #i) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k & #k < #i) |
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k & #k < #i))"

Noise defines destination property 4 as [64]

“Encryption to a known recipient, weak forward secrecy if the sender's private key has been compromised. This payload is encrypted based on an ephemeral-ephemeral DH, and also based on an ephemeral-static DH involving the recipient’s static key pair. However, the binding between the recipient’s alleged ephemeral public and the recipient’s static public key has only been verified based on DHs involving both those public keys and the sender’s static private key. Thus, if the sender’s static private key was previously compromised, the recipient’s alleged ephemeral public key may have been forged by an active attacker. In this case, the attacker could later compromise the intended recipient’s static private key to decrypt the payload (this is a variant of a “KCI” attack enabling a “weak forward secrecy” attack). Note that a future version of Noise might include signatures, which could improve this security property, but brings other trade-offs.”

This could be interpreted as ‘Encryption to a known recipient with weak forward secrecy even if the sender’s private key has already been compromised.’ However, this reading leads to results which do not match the destination property levels published by Noise.

Therefore, the lemma implements it as ‘Encryption to a known recipient with strong forward secrecy, unless the sender’s private key has been compromised’ instead. Or simply ‘Encryption to a known recipient with strong forward secrecy.’ Meaning that
5.3. Payload Properties

- the sender knows the recipient’s static public key (line 6) and that
- even an active attacker must either compromise the sender’s static key (line 10), the recipient’s static key (line 11) or their PSK (lines 12 & 13) before the payload is sent in order to be able to decrypt the payload (line 7). At least if the sender is not intentionally communicating with the attacker (line 9).

This interpretation of destination property 4 is also used by Noise Explorer and produces results which match those in the Noise specification.

**Destination Property 5**

```plaintext
lemma Payload1DestinationProperty5:
  "
  All tid payload1 #i.
  CommitPayload(tid, ‘payload1’, payload1)@i
  =>
  (Ex rs #j. RS(tid, rs)@j) &
  (not (Ex #j. K(payload1)@j) |
  (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
  (Ex s #j #k. S(tid, s)@j & Reveal(s)@k & #k < #i)) |
  (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k & #k < #i))
  
  Noise defines destination property 5 as [64]

  "Encryption to a known recipient, strong forward secrecy. This payload is encrypted based on an ephemeral-ephemeral DH as well as an ephemeral-static DH with the recipient’s static key pair. Assuming the ephemeral private keys are secure, and the recipient is not being actively impersonated by an attacker that has stolen its static private key, this payload cannot be decrypted."
```

However, ‘Encryption to a known recipient, strong forward secrecy’ is essentially a description of destination property 4 as implemented in the Payload1DestinationProperty4 lemma. On the other hand, the ‘if the sender’s private key has been compromised’ clause of destination property 4 can be understood as an attempt to differentiate destination properties 4 and 5. In this case, destination property 5 should be read as ‘Encryption to a known recipient, strong forward secrecy even if the sender’s static key has previously been compromised.’ Meaning that

- the sender must know the recipient’s static public key (line 6) and that
5. Properties

- even an attacker who has compromised the sender’s static private key, as well as all PKSs, must still compromise the recipient’s static public key (line 10) before the payload is sent in order to decrypt it (line 7). At least if the sender is not intentionally communicating with the attacker (line 8).

Since both some readings of the non-bold part of the property description and the obtained results also support this, the Payload1Destination-Property5 lemma implements this interpretation of destination property 5. Noise Explorer also does so.

However, this interpretation of destination property 5 is based on the description of destination property 4 at least as much as it is on its own. Other readings of destination property 5 could also be based on the non-bold part of the description. The property could then be understood to mean ‘Payload confidentiality is guaranteed as long as the attacker is not actively interfering in this particular protocol run. Even if the recipient’s static private key has already been compromised.’ This roughly translates to

```
lemma Payload1AlternateDestinationProperty5:
" All tid payload1 #i.
  CommitPayload(tid, 'payload1', payload1)#i
  ==> (Ex rs #j. RS(tid, rs)#j) &
       (not (Ex #j. K(payload1)#j) |
        (Ex re #j. RE(tid, re)#j & not (Ex #k. HonestE(re)#k)))
"
```

Since the model does not allow for ephemeral key compromise, however, this lemma tends to be relatively weak. On the protocols it was tested on, its results matched up with those of the Payload1DestinationProperty3 lemma. The same analogously goes for the other payloads of the protocols.

5.3.2 Source

In revision 34, Noise renamed its authentication properties to source properties [62]. However, Noise only talks about the payload’s sender being ‘authenticated,’ without specifying what this form of authentication means. Both the following lemmas and Noise Explorer implement it as ‘the payload was sent by the presumed sender and was intended as payload for the correct message.’ Meaning that if, for example, Alice ‘authenticates’ payload 1 as being from Bob and no relevant secrets have been compromised, Bob must actually have sent the same payload and intended it to be the payload of the first message. This interpretation does not require Alice to have been the intended recipient. Since, depending on the recipient, identical mes-
sages can have different meanings, such sender authentication is insufficient for some use cases.

The two source properties only differ in whether the payload can also be properly authenticated even when the recipient’s static private key is compromised. The reason the recipient’s private key is relevant to the authentication of the sender is that authentication is performed by proving the sender’s knowledge of the encryption key. The key is mostly derived from a number of shared secrets obtained by performing Diffie-Hellman operations. In order to compute such shared secrets obtained by DH, an attacker only needs to know either party’s private key and the other party’s public key. Since public keys are assumed to be publicly known, an attacker who knows a participant’s private key can use this to impersonate anyone else to this party. This form of attack is called key-compromise impersonation, or KCI for short. Noise’s remedy for KCI is the use of ephemeral keys, which are presumed to be uncompromisable.

Source Property 1

```
1 lemma Payload1SourceProperty1:
2 "
3 All tid payload1 #i.
4   ReceivePayload(tid, 'payload1', payload1)@i
5   =>
6     (Ex rs #j. RS(tid, rs)@j) &
7       ((Ex tidR rs #j #k #l. CommitPayload(tidR, 'payload1', payload1)@j &
8         RS(tid, rs)@k & S(tidR, rs)@l) |
9         (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
10        (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |
11        (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
12        (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k))
13 "
```

Noise describes source property 1 as [64]

“Sender authentication vulnerable to key-compromise impersonation (KCI). The sender authentication is based on a static-static DH ("ss") involving both parties’ static key pairs. If the recipient’s long-term private key has been compromised, this authentication can be forged. Note that a future version of Noise might include signatures, which could improve this security property, but brings other trade-offs.”

Meaning that both participants’ private keys (lines 11 & 12) and all PSKs
5. Properties

(line 13) must be safe in order for ‘authentication’ to be ensured (lines 7–9). In order for this property to make sense, it also requires that the recipient

- know who the sender is (line 6) and
- not be intentionally communicating with the attacker (line 10).

NOISE EXPLORER considers the sender to be identified if either

- a PSK was used or
- the recipient knows the senders public key.

However, PSKs are not necessarily sufficient to identify a sender for several reasons. Firstly, there are use cases for NOISE protocols where a participant can run the protocol both as initiator and as responder. In such a situation, an attacker may manipulate an honest participant into running the protocol with him- or herself. For protocols which rely solely on PSKs for authentication, this leads to the fooled participant misidentifying its partner. Additionally, PSKs in NOISE are not strictly required to be kept secret between two parties. For details on this see Section 4.6.2.

Source Property 2

1 lemma Payload1SourceProperty2:
2 "
3 All tid payload1 #i.
4 ReceivePayload(tid, 'payload1', payload1)@i
5 ==> (Ex rs #j. RS(tid, rs)@j) &
6 ((Ex tidR rs #j #k #l. CommitPayload(tidR, 'payload1', payload1)@j & RS(tid, rs)@k & S(tidR, rs)@l) |
7 (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
8 (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |
9 (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
10 (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k))
11 "

NOISE describes source property 2 as [64]

“Sender authentication resistant to key-compromise impersonation (KCI). The sender authentication is based on an ephemeral-static DH (“es” or “se”) between the sender’s static key pair and the recipient’s ephemeral key pair. Assuming the corresponding private keys are secure, this authentication cannot be forged.”
Meaning that only the sender’s static private key must remain secret (line 12) in order for the payload’s authentication to be reliable (lines 7–9). As with source property 1, the recipient has to know who (line 6) the non-attacker (line 10) sender is.

### 5.3.3 Authentication

NOISE’s source properties cover authentication both in an environment where all secrets are kept secret and where the recipient has been compromised. However, the single level of authentication it considers is fairly weak and may be insufficient for some use cases [51]. For instance, while it can be safely assumed that a payload with source property 1 was actually sent by the presumed sender, there are no guarantees as to who the intended recipient was.

Such guarantees can arguably be inferred from a combination of both the source and destination property of the payload [59]. However, the precise implications of any given combination of source and destination properties are not readily apparent.

Therefore, four more authentication properties are also verified for the protocols in both the scenario where no participant is compromised and the scenario where the attacker attempts a KCI. The lemmas below are based on the commonly used hierarchy of authentication properties by G. Lowe [54]. The most notable difference being that, instead of requiring the protocol to run to completion, the receiving of the payload is used to trigger the authentication claim.

**Authentication Property 1 – Aliveness**

```
lemma Payload1AuthenticationProperty1:

  All tid payload1 #i.
  ReceivePayload(tid, 'payload1', payload1)#i =>
    (Ex rs tidR roleR #j #k #l. RS(tid, rs)#j & HonestS(rs)#k & Running(tidR, roleR, rs)#l) |
    (Ex rs #j #k. RS(tid, rs)#j & Reveal(rs)#k) |
    (Ex rs #j. RS(tid, rs)#j & not Ex #k. HonestS(rs)#k) |
    (Ex s #j #k. S(tid, s)#j & Reveal(s)#k) |
    (Ex psk #j #k. PSK(tid, psk)#j & RevealPSK(psk)#k)
```

This property is based on the *aliveness* property. It asserts that receiving a
5. Properties

certain payload (line 4) ensures that the apparent sender (line 7) of that payload was active (‘alive’) at some point in time (line 8), assuming no secrets have been compromised (lines 9–12).

The results of the authentication property 1 lemmas match those of the source property 1 lemmas for all protocols and payloads they were tested against (i.e. all protocols named in the Noise specification). This is likely because successfully decrypting (for the purposes of this lemma ‘receiving’) a payload requires that the transcript hashes, which are the additional data for the AEAD encryption/decryption, match in both sender and recipient. Therefore, the structure of Noise protocol seems to enforce sufficient implicit verification of the payloads being interpreted properly, that these two lemmas are equivalent in most cases.

Authentication Property 2 – Weak Agreement

```
lemma Payload1AuthenticationProperty2:
"

All tid payload1 #i.

  ReceivePayload(tid, 'payload1', payload1)#i

  ==> (Ex s rs tidR roleR #j #k #l #m #n.

    RS(tid, rs)#j & HonestS(rs)#k &

    Running(tidR, roleR, rs)#l &

    RS(tidR, s)#m & S(tid, s)#n) |

  (Ex rs #j #k. RS(tid, rs)#j & Reveal(rs)#k) |

  (Ex rs #j. RS(tid, rs)#j & not Ex #k. HonestS(rs)#k) |

  (Ex s #j #k. S(tid, s)#j & Reveal(s)#k) |

  (Ex psk #j #k. PSK(tid, psk)#j & RevealPSK(psk)#k)
"
```

This property is based on the weak agreement property. It asserts that receiving a certain payload (line 4) ensures that the apparent sender of that payload tried to run the protocol (line 8) with the recipient (line 9) at least once, assuming no secrets have been compromised (lines 10–13). What role the presumed sender was running in or how far the sender’s protocol run got is not guaranteed.

As can be seen in Appendix D, all messages which were tested and found to satisfy authentication property 2 also satisfied authentication property 3. This is probably because of the same implicit verification of the payloads discussed in the Authentication Property 1 section above. It ensures that payloads which achieve the version of weak agreement used here also satisfy authentication property 3.
5.3. Payload Properties

**Authenticity Property 3 – Non-Injective Agreement**

```plaintext
lemma Payload1AuthenticationProperty3:
"All tid payload1 #i.

ReceivePayload(tid, 'payload1', payload1)#i

=> (Ex s rs tidR role roleR #j #k #l #m #n #o #p.
    RS(tid, rs)#j & HonestS(rs)#k &
    Running(tidR, roleR, rs)#l &
    RS(tidR, s)#m & S(tid, s)#n &
    Running(tid, role, s)#o & not role = roleR &
    CommitPayload(tidR, 'payload1', payload1)#p) |
(Ex rs #j #k. RS(tid, rs)#j & Reveal(rs)#k) |
(Ex rs #j. RS(tid, rs)#j & not Ex #k. HonestS(rs)#k) |
(Ex s #j #k. S(tid, s)#j & Reveal(s)#k) |
(Ex psk #j #k. PSK(tid, psk)#j & RevealPSK(psk)#k)

""
```

This property is based on the *non-injective agreement* property. It claims that receiving a certain payload as the payload to message n (line 4) ensures that

- the sender and recipient were running the protocol together and in opposite roles (line 7–10)

- the apparent sender of that payload sent it at least once (line 11)

- the message was intended as the payload of message n (line 11)

- the recipient was the intended destination (line 9)

At least as long as both parties keep their private keys and PSKs secret (lines 12–15).

Note that the lemma contains several redundant claims. Due to the way noise protocols are structured, for instance, one thread receiving payload 1 and the other sending payload 1 already requires them to be running in opposite roles. This requirement is nevertheless also explicitly part of the lemma, in order to keep it as close as possible to the *non-injective agreement* property as defined in [54].
5. Properties

Authentication Property 4 – Injective Agreement

lemma Payload1AuthenticationProperty4:

"All tid payload1 #i.

ReceivePayload(tid, 'payload1', payload1)#i =>

(Ex s rs tidR role roleR #j #k #l #m #n #o #p. RS(tid, rs)#j & HonestS(rs)#k & Running(tidR, roleR, rs)#l & RS(tidR, s)#m & S(tid, s)#n & Running(tid, role, s)#o & not role = roleR & CommitPayload(tidR, 'payload1', payload1)#p & ((Ex tid2 #q #r #t. ReceivePayload(tid2, 'payload1', payload1) #q & S(tid2, s)#r & RS(tid2, rs)#t & not tid = tid2) => (Ex tidR2 #q #r #t. CommitPayload(tidR2, 'payload1', payload1) #q & S(tidR2, rs)#r & RS(tidR2, s)#t & not tidR = tidR2)) | (Ex rs #j #k. RS(tid, rs)#j & Reveal(rs)#k) | (Ex rs #j. RS(tid, rs)#j & not Ex #k. HonestS(rs)#k) | (Ex s #j #k. S(tid, s)#j & Reveal(s)#k) | (Ex psk #j #k. PSK(tid, psk)#j & RevealPSK(psk)#k)"

This property is based on the injective agreement or simply agreement property. It claims that receiving the payload to message n (line 4) ensures that the apparent sender of that payload attempted to send it as the payload of message n at least once (line 11) with the recipient being the intended destination (lines 7–10). In addition, it asserts that if the recipient receives the same payload multiple times, the sender also sent it several times (lines 12–22). At least as long as both parties keep their private keys and PSKs secret (lines 23–26).

Note that, in order to stay in line with the message based view of the authentication properties as used in these lemmas, the ‘injective’ (i.e. replay safety) aspect of this property was significantly altered to no longer be protocol run based either.
5.3.4 KCI Resistant Authentication

The lemmas listed under *Authentication Property 1* through *Authentication Property 4* do not take key compromise impersonation (see the second paragraph of Section 5.3.2) into account. The ones below do, meaning that they assume that the recipient’s static private key and all PSKs have been compromised. These properties check the same authentication hierarchy as the plain authentication properties under the harsher assumption that only the sender’s static key and ephemeral keys can be relied upon.

The results are mostly analogous. In particular, the results of the KCI resistant version of *authentication property 1* match those of *source property 2*, and the results of the KCI resistant version of *authentication property 2* match the KCI resistant version of *authentication property 3*. Additionally, however, in their respective KCI resistant versions, the results of authentication properties 2 and 3 also match those of *authentication property 4*. This is essentially because both replay resistant authentication and KCI resistance are achieved by performing a DH exponentiation between the recipient’s ephemeral and the sender’s static key.

```
lemma Payload1KCIAuthenticationProperty1:
  "
  All tid payload1 #i.

  ReceivePayload(tid, 'payload1', payload1)@i
  =>

  (Ex rs tidR roleR #j #k #l.
   RS(tid, rs)@j & HonestS(rs)@k &
   Running(tidR, roleR, rs)@l) |

  (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |

  (Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

  "
```
5. Properties

lemma Payload1KCIAuthenticationProperty2:
"All tid payload1 #i.
ReceivePayload(tid, 'payload1', payload1)@i
==> (Ex s rs tidR roleR #j #k #l #m #n.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)
"}

lemma Payload1KCIAuthenticationProperty3:
"All tid payload1 #i.
ReceivePayload(tid, 'payload1', payload1)@i
==> (Ex s rs tidR role roleR #j #k #l #m #n #o #p.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n &
    Running(tid, role, s)@o & not role = roleR &
    CommitPayload(tidR, 'payload1', payload1)@p) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)
"
5.3. Payload Properties

lemma Payload1KCIAuthenticationProperty4:
" 
All tid payload1 #i.
ReceivePayload(tid, 'payload1', payload1)@i
  ==>
(Ex s rs tidR role roleR #j #k #l #m #n #o #p.
  RS(tid, rs)@j & HonestS(rs)@k &
  Running(tidR, roleR, rs)@l &
  RS(tidR, s)@m & S(tid, s)@n &
  Running(tid, role, s)@o & not role = roleR &
  CommitPayload(tidR, 'payload1', payload1)@p &
  ((Ex tid2 #q #r #t.
    ReceivePayload(tid2, 'payload1', payload1)
    @q &
    S(tid2, s)@r &
    RS(tid2, rs)@t &
    not tid = tid2)
  ==>
  (Ex tidR2 #q #r #t.
    CommitPayload(tidR2, 'payload1', payload1)
    @q &
    S(tidR2, rs)@r &
    RS(tidR2, s)@t &
    not tidR = tidR2))) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | (Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)
"

79
Chapter 6

Conclusions

The primary goal of this thesis is the formalization and verification of protocols developed with the Noise framework. To this purposes, two major contributions are presented here. Firstly, a formalization of Noise’s security properties introduced in Section 5.3. And secondly, a verification tool for Noise protocols discussed in Chapter 4.

Using this tool, the security properties of all Noise protocols mentioned in the Noise specification were checked. Moreover, all valid one-message Noise handshakes which are not fallback or PSK protocols were verified. While the verification of the valid two-message handshake patterns was started, most results are still outstanding as of this writing. The obtained results (see Appendix D) match those published in the specification for all patterns for which the author of Noise has listed the presumed security property.

However, the verification results matching the security properties asserted in the Noise specification should not be understood as a blanket confirmation of the protocols’ properties. Since the property definitions leave much room for interpretation, the lemmas used to verify them were chosen at least partially because they lead to matching results. Therefore, the results from these lemmas cannot really be used to validate the satisfied properties listed in the specification. At most, these results demonstrate that there exist reasonable interpretations of the security properties for which the results from the specification hold.

Previous to the publication of Noise Explorer’s results, guidance as to the expected properties was only available for 15 Noise handshake patterns. Unfortunately, these 15 patterns were not necessarily a good representation of all Noise handshakes. For example, none of them contained a PSK. As the expected results are a crucial part of interpreting Noise’s security properties, the meaning of Noise’s security properties is unclear for many handshake
patterns. It is, therefore, unsurprising that Noise Explorer’s results for several protocols differ slightly from those obtained from the lemmas in Chapter 5. Such differences are further discussed in Appendix D.

These difficulties and inconsistencies clearly demonstrate the need for more formal definitions of Noise’s security properties.

In addition, the security properties defined by Noise only cover a small range of potentially interesting properties, as discussed in Chapter 5. Adding more properties – possibly similar to those in Sections 5.2, 5.3.3 and 5.3.4 – to the specification would help in the selection of the appropriate handshake for certain applications.

While Noise’s security properties are not quite sufficient for many use cases, the pre-defined handshake patterns mentioned in the specification cover most possible applications.

The verification results of the 28 valid one-message Noise handshakes show that it is not possible to achieve better properties than the X and K protocols do. Similarly, N is the simplest pattern which achieves any security.

The only (valid) handshake patterns which have a notable advantage over the one-way protocols suggested in the specification (X, K and N) are

```plaintext
1 NOENCRIPTION:
2 ->
```

and

```plaintext
1 FASTX:
2 <- s
3 ...
4 -> e, s, es, ss
```

as well as equivalent variations thereof such as

```plaintext
1 FASTXalternate:
2 -> s
3 ...
4 <- s, e, ss, se
```

NOENCRIPTION allows for using Noise for unencrypted communication, should that need arise. Similarly, FASTX achieves the same source and destination properties as X but does not encrypt the initiator’s static key. This not only saves a few CPU cycles but also reduces the message size by 16 bytes since no authentication tag is generated for the static key. However, these potential performance gains come at the cost of completely exposing the initiator’s static public key to any attacker. Due to privacy concerns, this may be unacceptable for some applications.
Preliminary results for all 830 (non-psk and non-fallback) two-message handshakes mirror those for one-message handshakes.

While the verification tool presented in this thesis works reasonably well for most Noise protocols, it could still be improved upon in many ways. The most notable are

• Resource Usage

The verification of some protocols requires significant resources such as time or memory. This includes some commonly used protocols. The verification of Ik with two transport messages, for instance, required several days on a virtualized 6-core 3.1GHz Kaby Lake processor with 10 GB RAM. Similarly, the verification of other protocols can use more than 20GB of memory. Better oracles would likely be able to improve upon this situation.

• Oracle Usage Heuristics

The heuristics used to decide whether to use an oracle or Tamarin’s built-in proof strategy are overly simplistic. While they work well in most cases, they fail for some slightly strange protocols.

For example
1 INTERACTIVE1WAY:
2 \rightarrow e, s
3 \leftarrow e, s, ee, se, ss

Note that, as with all handshakes where this issue arises, this pattern is not endorsed by the Noise specification. This is unlikely to change since by replacing the ss with an es, the protocol’s properties could be significantly improved at no additional cost.

According to Noise’s rules, Alice may not use the negotiated key to encrypt data. Because of this, the heuristics misclassify this handshake as non-interactive. This leads to Tamarin’s built-in strategy being used and the verification failing.

• Additional Lemmas

The lemmas which are currently verified do not cover all potentially interesting properties of a protocol. For instance, confidentiality from a recipient’s point of view is not explicitly verified. Similarly, the current key agreement properties are rather bare bones.

• Invalid DH Keys

The model used by the verification tool aborts a protocol run once an invalid key is received. According to the Noise specification, however, the preferred course of action is to produce an all-zero DH-result
whenever such a key is used and to continue the protocol run. Since Noise’s preferred way of dealing with invalid keys is more prone to attacks, the model is not attack-preserving for all implementations. At least optionally allowing for a model which tolerates invalid or weak DH keys would help generate results which are applicable in all situations where Noise is used.

- Dummy Keys

   As discussed in Section 4.6.2, there are significant difficulties in modelling advanced Noise features such as dummy keys. These are, however, not insurmountable. By, for instance, providing a way to specify application-level negotiation protocols, such features could be modelled.

Some of these points may be difficult to implement efficiently with the current version of Tamarin. However, even with these improvements to the Noise verification tool implemented, some questions regarding the security of Noise protocols would remain unanswered.

While there are several extensions to Noise [57], this thesis only concerned itself with the core Noise specification. Some extensions only add new pattern modifiers and do not change the way a given pattern is interpreted [2]. If the pattern is specified manually, protocols using such extensions can be verified by existing tools. Most extensions, on the other hand, would require explicit support in order to be modelled [63].

Furthermore, the Noise specification contains provisions for combining several Noise handshakes into a compound protocol. WhatsApp, for instance, uses the Noise Pipes compound protocol [72], which combines the XX, IK and XXfallback handshakes. Since compound protocols use the same DH keys for the different handshakes, the security of the compound protocol cannot be inferred from the security of the individual handshakes. The verification of such protocols would consequently have to fall under the purview of future work.
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Formalizing and Verifying the Security Protocols from the Noise Framework

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
Suter-Dorig

First name(s):
Andris

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Köniz, 5.11.2018

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.
This appendix lists the transition labels used in the Tamarin models and gives a brief description of their meaning.

**AliceKeys(tid, k1, k2, n1, n2, hash)**

The thread with id tid has completed a handshake in the Alice role. She has computed the handshake hash to be hash, the negotiated keys to be k1 and k2, and the corresponding nonces to be n1 and n2. Since the label may be emitted after the first few transport messages, the nonces are not necessarily 0.

**BobKeys(tid, k1, k2, n1, n2, hash)**

Analogous to AliceKeys(tid, k1, k2, n1, n2, hash) above.

**CommitPayload(tid, name, payload)**

The thread with id tid is sending payload as the payload identified by name (e.g. 'payload1', 'payload2', 'payload3', ...).

**E(tid, e)**

The thread with id tid uses the DH public e as its ephemeral public key.

**Equal(a, b)**

The rule labelled in this way should only be applied if a and b are equal. This is enforced by the Equality restriction:

1. restriction Equality:
2. "All a b #i. Equal(a, b)@i ==> a = b"

It is only used when strange ciphertext constructions prevent the use of pattern matching for verifying the authenticity of encrypted data.
A. Labels

**HonestE(e)**

The DH public key e was originally created by an honest agent as its ephemeral key. Since ephemeral key compromise is not a threat noise attempts to protect against and is therefore not modelled, the corresponding private key cannot be known to the attacker.

**HonestS(s)**

The DH public key s was originally created by an honest agent as its static key. The key may, however, have been compromised since then.

**KeyUse(tid, key, nonce, data)**

The key key and nonce nonce where used to encrypt data by the thread with id tid.

**PrologueAlice(tid, prologue)**

The thread with id tid is running in the Alice role, and its prologue is prologue.

**PrologueBob(tid, prologue)**

Analogous to PrologueAlice(tid, prologue) above.

**RE(tid, re)**

The thread with id tid believes it is communicating with a thread that uses re as its ephemeral public key.

**ReceivePayload(tid, name, payload)**

The thread with id tid has received the message payload as the payload identified by name (e.g. 'payload1', 'payload2', 'payload3', ...).

**Reveal(s)**

The private key corresponding to the static public key s has been compromised and is known to the attacker.

**RevealPSK(psk)**

The pre-shared key psk has been compromised and is known to the attacker.

**RS(tid, rs)**

The thread with id tid believes it is communicating with a party that uses rs as its static public key.

**Running(tid, role, s)**

The thread with id tid is performed by the party with the static public key s and has started running in role role (either 'Alice' or 'Bob').

**S(tid, s)**

The thread with id tid uses the DH public s as its static public key.
Following is the generated protocol description for the IKpsk2 handshake in the intermediate language used by the verification tool presented in this thesis. This protocol description also includes the first two transport messages.

```plaintext
/*
IKpsk2:
<- s
...
-> e, es, s, ss
<- e, ee, se, psk
// Transport Message 1: ->
// Transport Message 2: <-
*/

Alice(
    // handshake prologue
    prologue,
    // remote static public key
    rs,
    // payload of message 1
    payload1,
    // ephemeral public key
    e,
    // ephemeral private key
```
B. Intermediate Language Example

```plaintext
priv_e,
// static public key
s,

// static private key
priv_s,

// 1. psk
psk1,

// payload of message 3
payload3
): // Initialize()
// or h = "Noise...", if HASHLEN is large enough
h = HASH("Noise_IKpsk2_DH_CIPHER_HASH")
n = 0
ck = h
h = HASH(h || prologue)

// premessage <- s
h = HASH(h || rs)

// send message 1: -> e, es, s, ss
h = HASH(h || e)
temp = HMAC-HASH(ck, e)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
es = DH(priv_e, rs)
temp = HMAC-HASH(ck, es)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
enc_s = ENCRYPT(k, n, h, s)
h = HASH(h || enc_s)
n = 1
ss = DH(priv_s, rs)
temp = HMAC-HASH(ck, ss)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
ciphertext1 = ENCRYPT(k, n, h, payload1)
```
h = HASH(h || ciphertext1)
n = 1
send(e, enc_s, ciphertext1)

// receive message 2: <- e, ee, se, psk
receive(re, ciphertext2)
h = HASH(h || re)
temp = HMAC-HASH(ck, re)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
ee = DH(priv_e, re)
temp = HMAC-HASH(ck, ee)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
se = DH(priv_s, re)
temp = HMAC-HASH(ck, se)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
temp = HMAC-HASH(ck, psk1)
ck = HMAC-HASH(temp, 1)
temp_h = HMAC-HASH(temp, ck || 2)
k = HMAC-HASH(temp, temp_h || 3)
n = 0
h = HASH(h || temp_h)
payload2 = DECRYPT(k, n, h, ciphertext2)
h = HASH(h || ciphertext2)
n = 1

// Split()
temp = HMAC-HASH(ck, "")
k1 = HMAC-HASH(temp, 1)
k2 = HMAC-HASH(temp, k1 || 2)
n1 = 1
n2 = 1
hash = h

// Transport Message 1: ->
ciphertext3 = ENCRYPT(k1, 0, "", payload3)
send(ciphertext3)

// Transport Message 2: <-
B. Intermediate Language Example

receive(ciphertext4)
payload4 = DECRYPT(k2, 0, "", ciphertext4)

// k1, n1: 1. negotiated key & corresponding nonce
// k2, n2: 2. negotiated key & corresponding nonce
// hash: the handshake hash
// payload2: payload received in message 2
// payload4: payload received in message 4
return (k1, n1, k2, n2, hash, payload2, payload4)

Bob(
   // handshake prologue
   prologue,
   // static public key
   s,
   // static private key
   priv_s,
   // payload of message 2
   payload2,
   // ephemeral public key
   e,
   // ephemeral private key
   priv_e,
   // 1. psk
   psk1,
   // payload of message 4
   payload4
): // Initialize()
   // or h = "Noise...", if HASHLEN is large enough
   h = HASH("Noise_IKpsk2_DH_CIPHER_HASH")
   n = 0
   ck = h
   h = HASH(h || prologue)
   // premessage <- s
h = HASH(h || s)

// receive message 1: -> e, es, s, ss
receive(re, enc_rs, ciphertext1)

h = HASH(h || re)
temp = HMAC-HASH(ck, re)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
es = DH(priv_s, re)
temp = HMAC-HASH(ck, es)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
rs = DECRYPT(k, n, h, enc_rs)
h = HASH(h || enc_rs)
n = 1
ss = DH(priv_s, rs)
temp = HMAC-HASH(ck, ss)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
payload1 = DECRYPT(k, n, h, ciphertext1)
h = HASH(h || ciphertext1)
n = 1

// send message 2: <- e, ee, se, psk
h = HASH(h || e)
temp = HMAC-HASH(ck, e)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
ee = DH(priv_e, re)
temp = HMAC-HASH(ck, ee)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
se = DH(priv_e, rs)
temp = HMAC-HASH(ck, se)
ck = HMAC-HASH(temp, 1)
k = HMAC-HASH(temp, ck || 2)
n = 0
temp = HMAC-HASH(ck, psk1)
ck = HMAC-HASH(temp, 1)
B. Intermediate Language Example

begin
  temp_h = HMAC-HASH(temp, ck || 2)
  k = HMAC-HASH(temp, temp_h || 3)
  n = 0
  h = HASH(h || temp_h)
  ciphertext2 = ENCRYPT(k, n, h, payload2)
  h = HASH(h || ciphertext2)
  n = 1
  send(e, ciphertext2)

  // Split()
  temp = HMAC-HASH(ck, "")
  k1 = HMAC-HASH(temp, 1)
  k2 = HMAC-HASH(temp, k1 || 2)
  n1 = 1
  n2 = 1
  hash = h

  // Transport Message 1: ->
  receive(ciphertext3)
  payload3 = DECRYPT(k1, 0, "", ciphertext3)

  // Transport Message 2: <-
  ciphertext4 = ENCRYPT(k2, 0, "", payload4)
  send(ciphertext4)

  // k1, n1: 1. negotiated key & corresponding nonce
  // k2, n2: 2. negotiated key & corresponding nonce
  // hash: the handshake hash
  // payload1: payload received in message 1
  // payload3: payload received in message 3
  return (k1, n1, k2, n2, hash, payload1, payload3)
end
Appendix C

Tamarin Model Example

Following is one of the two Tamarin models for the IKpsk2 handshake generated by the verification tool presented in this thesis. This model also includes the first two transport messages.

The model below uses attacker controlled message payloads and is used to verify all authentication properties. The other model uses sender generated fresh values as payloads and is used to verify the secrecy (destination) properties.

```haskell
/*
IKpsk2:
...
-> e, es, s, ss
<- e, ee, se, psk
// Transport Message 1: -> 
// Transport Message 2: <-
*/
theory IKpsk2
begin
builtins: diffie-hellman, hashing
functions: hmac/3, h2/2, enc/4, dec/2, verified/0, verify/4
equations: dec(k, enc(k, n, ad, d)) = <d, ad, n>, verify(k, n, ad, enc(k, n, ad, d)) = verified
rule InitPrincipal:
[Fr(~k)]
--[HonestS('g'~k)]->
[!KeyPair('g'~k, ~k), Out('g'~k)]
```

C. Tamarin Model Example

```plaintext
rule InitPairing:
  ![KeyPair(pk1, ~sk1), !KeyPair(pk2, ~sk2), Fr(~psk1)]
  -->
  ![Pairing(pk1, pk2, ~psk1), !Pairing(pk2, pk1, ~psk1), !PSKValue(~psk1)]

rule PSKReveal:
  ![PSKValue(~psk)]
  --[RevealPSK(~psk)]->
  [Out(~psk)]

// ephemeral public key
// ephemeral private key
// static private key
// handshake prologue
// 1. psk
// static public key
rule InitThreadAlice:
  let
    s = pkA
  in
    [Fr(~tid), !Pairing(pkA, pkB, ~psk1), Fr(~priv_e), !KeyPair(pkA, ~priv_s), In(prologue)]
  --[S(~tid, s), E(~tid, 'g'~priv_e), PrologueAlice(~tid, prologue), HonestE('g'~priv_e), PSK(~tid, ~psk1)]->
    ![AliceParams(~tid, 'g'~priv_e, ~priv_e, ~priv_s, prologue, ~psk1, s), Alice_0(~tid)]

// ephemeral public key
// ephemeral private key
// static private key
// handshake prologue
// 1. psk
// static public key
rule InitThreadBob:
  let
    s = pkB
  in
    [Fr(~tid), !Pairing(pkA, pkB, ~psk1), Fr(~priv_e), !KeyPair(pkB, ~priv_s), In(prologue)]
  --[S(~tid, s), E(~tid, 'g'~priv_e), PrologueBob(~tid, prologue), HonestE('g'~priv_e), PSK(~tid, ~psk1)]->
```

58  ![BobParams(~tid, 'g'~priv_e, ~priv_e, ~priv_s, 
prologue, ~psk1, s),
59  Bob_0(~tid)]
60
61 rule LtkReveal:
62  ![KeyPair(pkA, ~skA)]
63  ←[Reveal(pkA)]→
64  ![Out(~skA)]
65
66 rule Alice1:
67    let
68    // Group Element Checks
69    e = 'g'~priv_e
70    s = 'g'~priv_s
71    re = 'g'~priv_re
72    rs = 'g'~priv_rs
73
74    // Translated NIL Assignments
75    // Initialize()
76    // or h = "Noise...", if HASHLEN is large enough
77    // 'T2' = 'Noise_IKpsk2_DH_CIPHER_HASH'
78    h_1 = h('T2')
79    h_2 = h2(h_1, prologue)
80    // premessage <- s
81    h_3 = h2(h_2, rs)
82    // send message 1: -> e, es, s, ss
83    h_4 = h2(h_3, e)
84    ck_2 = hmac(h_1, e, 'I1')
85    es_1 = (rs~priv_e)
86    ck_3 = hmac(ck_2, es_1, 'I1')
87    k_1 = hmac(ck_2, es_1, 'I2')
88    enc_s_1 = enc(k_1, 'I0', h_4, s)
89    h_5 = h2(h_4, enc_s_1)
90    ss_1 = (rs~priv_s)
91    ck_4 = hmac(ck_3, ss_1, 'I1')
92    k_2 = hmac(ck_3, ss_1, 'I2')
93    ciphertext1_1 = enc(k_2, 'I0', h_5, payload1)
94    h_6 = h2(h_5, ciphertext1_1)
95    // send(e, enc_s, ciphertext1)
96    in
97    [Alice_0(~tid), !AliceParams(~tid, e, ~priv_e, ~priv_s, 
prologue, ~psk1, s), In(rs), In(payload1)]
98  ←[Running(~tid, 'Alice', s), CommitPayload(~tid, ' 
payload1', payload1), KeyUse(~tid, k_1, 'I0', s),

103
KeyUse(~tid, k_2, 'I0', payload1), RS(~tid, rs)] ->
99 [Alice_1(~tid, ck_4, h_6), !AliceConstrs1(~tid, payload1, rs), Out(<e, enc_s_1, ciphertext1_1>)]
100
101 rule Alice2:
102   let
103     // Group Element Checks
104     e = 'g'~priv_e
105     s = 'g'~priv_s
106     re = 'g'~priv_re
107     rs = 'g'~priv_rs
108
109     // Translated NIL Assignments
110     // receive message 2: <- e, ee, se, psk
111     // receive (re, ciphertext2)
112     h_2 = h2(h_1, re)
113     ck_2 = hmac(ck_1, re, 'I1')
114     ee_1 = (re~priv_e)
115     ck_3 = hmac(ck_2, ee_1, 'I1')
116     se_1 = (re~priv_s)
117     ck_4 = hmac(ck_3, se_1, 'I1')
118     ck_5 = hmac(ck_4, ~psk1, 'I1')
119     temp_h_1 = hmac(ck_4, ~psk1, 'I2')
120     k_1 = hmac(ck_4, ~psk1, 'I3')
121     h_3 = h2(h_2, temp_h_1)
122     // Authenticate payload2 = dec(k_1, 'I0', h_3, ciphertext2_1)
123     ciphertext2_1 = enc(k_1, 'I0', h_3, payload2)
124     hash = h2(h_3, ciphertext2_1)
125     // Split()
126     // 'T1' = ''
127     k1 = hmac(ck_5, 'T1', 'I1')
128     // 'T1' = ''
129     k2 = hmac(ck_5, 'T1', 'I2')
130     n1 = 'I1'
131     n2 = 'I1'
132     // Transport Message 1: ->
133     // 'T1' = ''
134     ciphertext3_1 = enc(k1, 'I0', 'T1', payload3)
135     // send(ciphertext3)
136     in
137     [Alice_1(~tid, ck_1, h_1), !AliceParams(~tid, e, ~priv_e , ~priv_s, prologue, ~psk1, s), In(<re, ciphertext 2_1>), In(payload3)]
rule Alice3:
    let
    // Group Element Checks
    e = 'g'^~priv_e
    s = 'g'^~priv_s
    re = 'g'^~priv_re
    rs = 'g'^~priv_rs

    // Translated NIL Assignments
    // Transport Message 2: <->
    // receive (ciphertext4)
    // 'T1' = ''
    // Authenticate payload4 = dec(k2, 'I0', 'T1',
    ciphertext4_1 = enc(k2, 'I0', 'T1', payload4)
    // k1, n1: 1. negotiated key & corresponding nonce
    // k2, n2: 2. negotiated key & corresponding nonce
    // hash: the handshake hash
    // payload2: payload received in message 2
    // payload4: payload received in message 4
    // return (k1, n1, k2, n2, hash, payload2, payload4)

    in
    [Alice_2(~tid), !AliceConsts2(~tid, hash, k1, k2, n1, n
    2, payload2, re, payload3), Out(ciphertext3_1)]

rule Bob1:
    let
    // Group Element Checks
    e = 'g'^~priv_e
    s = 'g'^~priv_s
    re = 'g'^~priv_re
    rs = 'g'^~priv_rs
C. Tamarin Model Example

173  // Translated NIL Assignments
174  // Initialize()
175  // or h = "Noise_...", if HASHLEN is large enough
176  // 'T2' = 'Noise_IKpsk2_DH_CIPHER_HASH'
177  h_1 = h('T2')
178  h_2 = h2(h_1, prologue)
179  // premessage <- s
180  h_3 = h2(h_2, s)
181  // receive message 1: <- e, es, s, ss
182  h_4 = h2(h_3, re)
183  ck_2 = hmac(h_1, re, 'I1')
184  es_1 = (re^∼priv_s)
185  ck_3 = hmac(ck_2, es_1, 'I1')
186  k_1 = hmac(ck_2, es_1, 'I2')
187  // Authenticate rs = dec(k_1, 'I0', h_4, enc_rs_1)
188  enc_rs_1 = enc(k_1, 'I0', h_4, rs)
189  h_5 = h2(h_4, enc_rs_1)
190  ss_1 = (rs^∼priv_s)
191  ck_4 = hmac(ck_3, ss_1, 'I1')
192  k_2 = hmac(ck_3, ss_1, 'I2')
193  // Authenticate payload1 = dec(k_2, 'I0', h_5, ciphertext1_1)
194  ciphertext1_1 = enc(k_2, 'I0', h_5, payload1)
195  h_6 = h2(h_5, ciphertext1_1)
196  // send message 2: <- e, ee, se, psk
197  h_7 = h2(h_6, e)
198  ck_5 = hmac(ck_4, e, 'I1')
199  ee_1 = (e^∼priv_e)
200  ck_6 = hmac(ck_5, ee_1, 'I1')
201  se_1 = (e^∼priv_e)
202  ck_7 = hmac(ck_6, se_1, 'I1')
203  ck_8 = hmac(ck_7, ∼psk1, 'I1')
204  temp_h_1 = hmac(ck_7, ∼psk1, 'I2')
205  k_3 = hmac(ck_7, ∼psk1, 'I3')
206  h_8 = h2(h_7, temp_h_1)
207  ciphertext2_1 = enc(k_3, 'I0', h_8, payload2)
208  hash = h2(h_8, ciphertext2_1)
209  // send(e, ciphertext2)
210  // Split()
211  // 'T1' = '
212  k1 = hmac(ck_8, 'T1', 'I1')
k2 = hmac(ck_8, 'T1', 'I2')
n1 = 'I1'
n2 = 'I1'
in [Bob_0(~tid), !BobParams(~tid, e, ~priv_e, ~priv_s, prologue, ~psk1, s), In(re, enc_rs_1, ciphertext_1_1), In(payload2)]

-- [Running(~tid, 'Bob', s), CommitPayload(~tid, 'payload 2', payload2), ReceivePayload(~tid, 'payload1', payload1), KeyUse(~tid, k_3, 'I0', payload2), RS(~tid, rs), RE(~tid, re)]->

[Bob_1(~tid), !BobConsts1(~tid, hash, k1, k2, n1, n2, payload1, re, rs, payload2), Out(<e, ciphertext2_1>)]

rule Bob2:
let

// Group Element Checks
e = 'g'^~priv_e
s = 'g'^~priv_s
re = 'g'^~priv_re
rs = 'g'^~priv_rs

// Translated NIL Assignments

// Transport Message 1: ->
// receive (ciphertext3)
// 'T1' = ''
// Authenticate payload3 = dec(k1, 'I0', 'T1', ciphertext3_1)
ciphertext3_1 = enc(k1, 'I0', 'T1', payload3)
// Transport Message 2: <-
// 'T1' = ''
ciphertext4_1 = enc(k2, 'I0', 'T1', payload4)
// send(ciphertext4)

// k1, n1: 1. negotiated key & corresponding nonce
// k2, n2: 2. negotiated key & corresponding nonce

// hash: the handshake hash
// payload1: payload received in message 1
// payload3: payload received in message 3
// return (k1, n1, k2, n2, hash, payload1, payload3)
C. TAMARIN MODEL EXAMPLE

249  [Bob_1(~tid), !BobConsts1(~tid, hash, k1, k2, n1, n2, payload1, re, rs, payload2), In(ciphertext3_1), In(payload4)]

250  -- [CommitPayload(~tid, 'payload4', payload4), ReceivePayload(~tid, 'payload3', payload3), KeyUse(~tid, k2, 'I0', payload4), BobKeys(~tid, k1, k2, n1, n2, hash)]->

251  [Out(ciphertext4_1)]

252  // The purpose of this rule is to ensure that the lemmas and restrictions are accepted, even if the protocol doesn't produce these labels

253  rule Dummy:

254  [DummyFact()]

255  -- [Equal('a', 'b'), KeyUse('tid', 'key', 'nonce', 'plaintext'), Running('tid', 'role', 'pk'), CommitPayload('tid', 'title', 'payload'), ReceivePayload('tid', 'title', 'payload'), E('tid', 'pk'), RE('tid', 'pk'), S('tid', 'pk'), RS('tid', 'pk'), PSK('tid', 'psk')]

256  ]->

257  []

258  restriction Equality:

259  "All a b #i. Equal(a, b)@i ==> a = b"

260  /*sanity checks*/

261  lemma Executability:

262  exists-trace

263  "(Ex tidA tidB k1 k2 n1 n2 hash #i #j. AliceKeys(tidA, k1, k2, n1, n2, hash)@i & BobKeys(tidB, k1, k2, n1, n2, hash)@j) &

264  not (Ex s #i. Reveal(s)@i) &

265  not (Ex psk #i. RevealPSK(psk)@i) &

266  All tid1 tid2 role s1 s2 #i #j.

267  Running(tid1, role, s1)@i & Running(tid2, role, s2)@j

268  ==> tid1 = tid2 & s1 = s2 & #i = #j"

269  lemma ThreadStartedOnce:
All tid role1 role2 s1 s2 #i #j.
Running(tid, role1, s1)@i & Running(tid, role2, s2)@j
==>
#i = #j & role1 = role2 & s1 = s2

lemma AliceThreadUniqueness:

All tid k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1 hash_2 #i #j.
AliceKeys(tid, k1_1, k2_1, n1_1, n2_1, hash_1)@i &
AliceKeys(tid, k1_2, k2_2, n1_2, n2_2, hash_2)@j
==>
#i = #j

lemma BobThreadUniqueness:

All tid k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1 hash_2 #i #j.
BobKeys(tid, k1_1, k2_1, n1_1, n2_1, hash_1)@i & BobKeys(
tid, k1_2, k2_2, n1_2, n2_2, hash_2)@j
==>
#i = #j

lemma HashEqualityImpliesKeyEquality:

All tid a tidb k1a k1b k2a k2b n1a n1b n2a n2b hasha hashb #i #j.
AliceKeys(tida, k1a, k2a, n1a, n2a, hasha)@i & BobKeys(
tidb, k1b, k2b, n1b, n2b, hashb)@j &
hasha = hashb
==>
k1a = k1b & k2a = k2b

lemma NoKeyReuse:

All tid1 tid2 k n d1 d2 #i #j.
KeyUse(tid1, k, n, d1)@i & KeyUse(tid2, k, n, d2)@j
==>
d1 = d2 & #i = #j
C. Tamarin Model Example

lemma SNotSetTwiceAndSafe:
" All tid pk1 pk2 #i #j.
  (S(tid, pk1)@i & S(tid, pk2)@j) =>
  pk1 = pk2 & #i = #j &
  (Ex #k. HonestS(pk1)@k) "

lemma RSNotSetTwice:
" All tid pk1 pk2 #i #j.
  (RS(tid, pk1)@i & RS(tid, pk2)@j) =>
  pk1 = pk2 & #i = #j "

lemma ENotSetTwiceAndSafe:
" All tid pk1 pk2 #i #j.
  (E(tid, pk1)@i & E(tid, pk2)@j) =>
  pk1 = pk2 & #i = #j &
  (Ex #k. HonestE(pk1)@k) &
  not (Ex #k. Reveal(pk1)@k) "

lemma RENotSetTwice:
" All tid pk1 pk2 #i #j.
  (RE(tid, pk1)@i & RE(tid, pk2)@j) =>
  pk1 = pk2 & #i = #j "

lemma PrologueEquality:
" All tidA tidB k1 k2 n1 n2 hash #i #j.
  AliceKeys(tidA, k1, k2, n1, n2, hash)@i & BobKeys(tidB, k1, k2, n1, n2, hash)@j &
  not (Ex prologue #k #l. PrologueAlice(tidA, prologue)@k & PrologueBob(tidB, prologue)@l) =>

(Ex s #k #l. S(tidA, s)@k & Reveal(s)@l) | (Ex rs #k #l. RS(tidA, rs)@k & Reveal(rs)@l) |
(Ex s #k #l. S(tidB, s)@k & Reveal(s)@l) | (Ex rs #k #l. RS(tidB, rs)@k & Reveal(rs)@l) |
(Ex psk #k #l. PSK(tidA, psk)@k & PSK(tidA, psk)@l & RevealPSK(psk)@m)

" 

/***********************
* security properties *
***********************
lemma AliceKeySecrecy:
" 
All tid k1 k2 n1 n2 hash #i.
AliceKeys(tid, k1, k2, n1, n2, hash)@i 
=>
((not Ex #j. K(k1)@j) & (not Ex #j. K(k2)@j)) |
(Ex s #j. RS(tid, s)@j & not (Ex #k. HonestS(rs)@k)) |
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)

lemma BobKeySecrecy:
" 
All tid k1 k2 n1 n2 hash #i.
BobKeys(tid, k1, k2, n1, n2, hash)@i 
=>
((not Ex #j. K(k1)@j) & (not Ex #j. K(k2)@j)) |
(Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)

lemma AliceKeyUniqueness:
" 
All tid_1 tid_2 k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1 hash_2 #i #j.
AliceKeys(tid_1, k1_1, k2_1, n1_1, n2_1, hash_1)@i &
AliceKeys(tid_2, k1_2, k2_2, n1_2, n2_2, hash_2)@j &
(k1_1 = k1_2 | k2_1 = k2_2)
=>
tid_1 = tid_2
C. Tamarin Model Example

lemma BobKeyUniqueness:

All tid_1 tid_2 k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1 hash_2 #i #j.

  BobKeys(tid_1, k1_1, k2_1, n1_1, n2_1, hash_1)i &
  BobKeys(tid_2, k1_2, k2_2, n1_2, n2_2, hash_2)j &
  (k1_1 = k1_2 | k2_1 = k2_2) =>
  tid_1 = tid_2

lemma AliceHashUniqueness:

All tid_1 tid_2 k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1 hash_2 #i #j.

  AliceKeys(tid_1, k1_1, k2_1, n1_1, n2_1, hash_1)i &
  AliceKeys(tid_2, k1_2, k2_2, n1_2, n2_2, hash_2)j &
  hash_1 = hash_2 =>
  tid_1 = tid_2

lemma BobHashUniqueness:

All tid_1 tid_2 k1_1 k1_2 k2_1 k2_2 n1_1 n1_2 n2_1 n2_2 hash_1 hash_2 #i #j.

  BobKeys(tid_1, k1_1, k2_1, n1_1, n2_1, hash_1)i &
  BobKeys(tid_2, k1_2, k2_2, n1_2, n2_2, hash_2)j &
  hash_1 = hash_2 =>
  tid_1 = tid_2

lemma Payload4SourceProperty1:

All tid payload4 #i.

  ReceivePayload(tid, 'payload4', payload4)i =>
  ((Ex rs #j. RS(tid, rs)j) &
    (Ex tidR rs #j #k #l. CommitPayload(tidR, 'payload4', payload4)j & RS(tid, rs)k & S(tidR, rs)l) |
    (Ex rs #j. RS(tid, rs)j & not (Ex #k. HonestS(rs)k)) |
lemma Payload4SourceProperty2:

\[ \forall \text{tid} \, \text{payload4} \, \#i. \quad \text{ReceivePayload}(\text{tid}, \text{'payload4'}, \text{payload4})@i \implies \]
\[ (\exists \text{rs} \, \#j \, (\text{RS}(\text{tid}, \text{rs})@j \land \text{Reveal}(\text{rs})@k) \lor\]
\[ (\exists \text{psk} \, \#j \, (\text{PSK}(\text{tid}, \text{psk})@j \land \text{RevealPSK}(\text{psk})@k)) \]

lemma Payload4AuthenticationProperty1:

\[ \forall \text{tid} \, \text{payload4} \, \#i. \quad \text{ReceivePayload}(\text{tid}, \text{'payload4'}, \text{payload4})@i \implies \]
\[ (\exists \text{s}, \text{rs}, \text{tidR}, \text{roleR} \, \#j \, \#k \, \#l. \quad \text{RS}(\text{tid}, \text{rs})@j \land \text{HonestS}(\text{rs})@k \land\]
\[ \text{Running}(\text{tidR}, \text{roleR}, \text{rs})@l) \lor\]
\[ (\exists \text{rs} \, \#j \, \#k. \quad \text{RS}(\text{tid}, \text{rs})@j \land \text{Reveal}(\text{rs})@k) \lor\]
\[ (\exists \text{rs} \, \#j. \quad \text{RS}(\text{tid}, \text{rs})@j \land \text{not } (\exists \#k. \text{HonestS}(\text{rs})@k)) \lor\]
\[ (\exists \text{psk} \, \#j \, \#k. \quad \text{PSK}(\text{tid}, \text{psk})@j \land \text{RevealPSK}(\text{psk})@k) \]

lemma Payload4AuthenticationProperty2:

\[ \forall \text{tid} \, \text{payload4} \, \#i. \quad \text{ReceivePayload}(\text{tid}, \text{'payload4'}, \text{payload4})@i \implies \]
\[ (\exists \text{s}, \text{rs}, \text{tidR}, \text{roleR} \, \#j \, \#k \, \#l \, \#m \, \#n. \quad \text{RS}(\text{tid}, \text{rs})@j \land \text{HonestS}(\text{rs})@k \land\]
\[ \text{Running}(\text{tidR}, \text{roleR}, \text{rs})@l \land\]
\[ \text{RS}(\text{tidR}, \text{s})@m \land \text{S}(\text{tid}, \text{s})@n) \lor\]
\[ (\exists \text{rs} \, \#j \, \#k. \quad \text{RS}(\text{tid}, \text{rs})@j \land \text{Reveal}(\text{rs})@k) \lor\]
\[ (\exists \text{rs} \, \#j. \quad \text{RS}(\text{tid}, \text{rs})@j \land \text{not } (\exists \#k. \text{HonestS}(\text{rs})@k)) \lor\]
\[ (\exists \text{psk} \, \#j \, \#k. \quad \text{PSK}(\text{tid}, \text{psk})@j \land \text{RevealPSK}(\text{psk})@k) \]
lemma Payload4AuthenticationProperty3:

All tid payload4 #i.

ReceivePayload(tid, 'payload4', payload4)@i

=>

(Ex s rs tidR role roleR #j #k #l #m #n #o #p.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@on &
    Running(tid, role, s)@o & not role = roleR &
    CommitPayload(tidR, 'payload4', payload4)@p) |

(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) |
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)
lemma Payload4KCIAuthenticationProperty1:

All tid payload4 #i.

ReceivePayload(tid, 'payload4', payload4)@i ==>

(Ex rs tidR roleR #j #k #l. RS(tid, rs)@j & HonestS(rs)@k & Running(tidR, roleR, rs)@l) | 
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | 
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload4KCIAuthenticationProperty2:

All tid payload4 #i.

ReceivePayload(tid, 'payload4', payload4)@i ==>

(Ex s rs tidR roleR #j #k #l #m #n. RS(tid, rs)@j & HonestS(rs)@k & Running(tidR, roleR, rs)@l & RS(tidR, s)@m & S(tid, s)@n) | 
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | 
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload4KCIAuthenticationProperty3:

All tid payload4 #i.

ReceivePayload(tid, 'payload4', payload4)@i ==>

(Ex s rs tidR role roleR #j #k #l #m #n #o #p. RS(tid, rs)@j & HonestS(rs)@k & Running(tidR, roleR, rs)@l & RS(tidR, s)@m & S(tid, s)@n & Running(tid, role, s)@o & not role = roleR & CommitPayload(tidR, 'payload4', payload4)@p) | 
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | 
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)
lemma Payload4KCIAuthenticationProperty4:

"All tid payload4 #i.

ReceivePayload(tid, 'payload4', payload4)@i

==> (Ex s rs tidR role roleR #j #k #l #m #n #o #p. RS(tid, rs)@j & HonestS(rs)@k & Running(tidR, roleR, rs)@l & RSj(tidR, s)@m & S(tid, s)@n & Running(tid, role, s)@o & not role = roleR & CommitPayload(tidR, 'payload4', payload4)@p & ((Ex tid2 #q #r #t. ReceivePayload(tid2, 'payload4', payload4)@q & S(tid2, s)@r & RS(tid2, rs)@t & not tid = tid2) ==> (Ex tidR2 #q #r #t. CommitPayload(tidR2, 'payload4', payload4)@q & S(tidR2, rs)@r & RS(tidR2, s)@t & not tidR = tidR2)) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | (Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)"

lemma Payload2SourceProperty1:

"All tid payload2 #i.

ReceivePayload(tid, 'payload2', payload2)@i

==> (Ex rs #j. RS(tid, rs)@j) & ((Ex tidR rs #j #k #l. CommitPayload(tidR, 'payload2', payload2)@j & RS(tid, rs)@k & S(tidR, rs)@l) | (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) | (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k))"

lemma Payload2SourceProperty2:
lemma Payload2AuthenticationProperty1:
  All tid payload2 #i.
  ReceivePayload(tid, 'payload2', payload2)@i
  =>
  (Ex rs #j. RS(tid, rs)@j) &
  ((Ex tidR rs #j #k #l. CommitPayload(tidR, 'payload2', payload2)@j & RS(tid, rs)@k & S(tidR, rs)@l) |
   (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
   (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k))

lemma Payload2AuthenticationProperty2:
  All tid payload2 #i.
  ReceivePayload(tid, 'payload2', payload2)@i
  =>
  (Ex s rs tidR roleR #j #k #l. RS(tid, rs)@j & HonestS(rs)@k &
   Running(tidR, roleR, rs)@l) |
  (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
  (Ex s #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) |
  (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |
  (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)

lemma Payload2AuthenticationProperty3:
  All tid payload2 #i.
  ReceivePayload(tid, 'payload2', payload2)@i
C. Tamarin Model Example

\[
\text{Lemma Payload2AuthenticationProperty4:}
\]

\[
\text{All rid payload2 i.}
\]

\[
\text{ReceivePayload(rid, 'payload2', payload2)@i} \Rightarrow
\]

\[
\text{CommitPayload(ridR, 'payload2', payload2)@p} \land
\]

\[
\text{RS(tid, ridR, roleR, roleR, rid)@l} \land
\]

\[
\text{RS[tidR, s]@m \land S(tid, s)@n} \land
\]

\[
\text{Running(tid, role, s)@o \land not role = roleR} \land
\]

\[
\text{not role = roleR} \land
\]

\[
\text{CommitPayload(tidR, 'payload2', payload2)@p) |}
\]

\[
\text{(Ex rs #j #k. RS[tid, s]@l \land Reveal(s)@k) |}
\]

\[
\text{(Ex rid #j. RS[tid, rid, role, role, rid]@l \land}
\]

\[
\text{not Ex #k. HonestS(rs)@k) |}
\]

\[
\text{(Ex s #j #k. S[tid, s]@l \land Reveal(s)@k) |}
\]

\[
\text{(Ex psk #j #k. PSK[tid, psk]@l \land RevealPSK(psk)@k)
\}"

\[
\text{Lemma Payload2KCIAuthenticationProperty1:}
\]
All tid payload2 #i.

ReceivePayload(tid, 'payload2', payload2)@i

==> (Ex rs tidR roleR #j #k #l.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload2KCIAuthenticationProperty2:

All tid payload2 #i.

ReceivePayload(tid, 'payload2', payload2)@i

==> (Ex s rs tidR roleR #j #k #l #m #n.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload2KCIAuthenticationProperty3:

All tid payload2 #i.

ReceivePayload(tid, 'payload2', payload2)@i

==> (Ex s rs tidR role roleR #j #k #l #m #n #o #p.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n &
    Running(tid, role, s)@o & not role = roleR &
    CommitPayload(tidR, 'payload2', payload2)@p) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload2KCIAuthenticationProperty4:

All tid payload2 #i.

ReceivePayload(tid, 'payload2', payload2)@i

==>
lemma Payload3SourceProperty1:
    All tid payload3 #i.
    ReceivePayload(tid, 'payload3', payload3)@i
    ==> (Ex rs #j. RS(tid, rs)@j &
    ((Ex tidRs rs #j #k #l. CommitPayload(tidRs, 'payload3', payload3)@j &
        RS(tid, rs)@k & S(tidRs, rs)@l) |
     (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
     (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k)) |
    (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k))

lemma Payload3SourceProperty2:
    All tid payload3 #i.
    ReceivePayload(tid, 'payload3', payload3)@i
    ==> (Ex rs #j. RS(tid, rs)@j) &
    ((Ex tidRs rs #j #k #l. CommitPayload(tidRs, 'payload3', payload3)@j &
        RS(tid, rs)@k & S(tidRs, rs)@l) |
     (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) |
     (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k)) |
    (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k))
lemma Payload3AuthenticationProperty1:

\[\text{All } tid \text{ payload3 } \#i.\]
\[\text{ReceivePayload}(tid, 'payload3', payload3)@i \implies (Ex rs tidR roleR \#j \#k \#l. RS(tid, rs)@j & HonestS(rs)@k & Running(tidR, roleR, rs)@l) \lor (Ex rs \#j \#k. RS(tid, rs)@j & Reveal(rs)@k) \lor (Ex rs \#j. RS(tid, rs)@j & not Ex \#k. HonestS(rs)@k) \lor (Ex s \#j \#k. S(tid, s)@j & Reveal(s)@k) \lor (Ex psk \#j \#k. PSK(tid, psk)@j & RevealPSK(psk)@k)\]

lemma Payload3AuthenticationProperty2:

\[\text{All } tid \text{ payload3 } \#i.\]
\[\text{ReceivePayload}(tid, 'payload3', payload3)@i \implies (Ex s rs tidR roleR \#j \#k \#l \#m \#n. RS(tid, rs)@j & HonestS(rs)@k & Running(tidR, roleR, rs)@l & RS(tidR, s)@m & S(tid, s)@n) \lor (Ex rs \#j \#k. RS(tid, rs)@j & Reveal(rs)@k) \lor (Ex rs \#j. RS(tid, rs)@j & not Ex \#k. HonestS(rs)@k) \lor (Ex s \#j \#k. S(tid, s)@j & Reveal(s)@k) \lor (Ex psk \#j \#k. PSK(tid, psk)@j & RevealPSK(psk)@k)\]

lemma Payload3AuthenticationProperty3:

\[\text{All } tid \text{ payload3 } \#i.\]
\[\text{ReceivePayload}(tid, 'payload3', payload3)@i \implies (Ex s rs tidR role roleR \#j \#k \#l \#m \#n \#o \#p. RS(tid, rs)@j & HonestS(rs)@k & Running(tidR, roleR, rs)@l & RS(tidR, s)@m & S(tid, s)@n \& Running(tid, role, s)@o & not role = roleR & \]
CommitPayload(tidR, 'payload3', payload3)@p) | 
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | 
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) | 
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | 
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)

lemma Payload3AuthenticationProperty4:
" 
All tid payload3 #i.
ReceivePayload(tid, 'payload3', payload3)@i 
==> 
(Ex s rs tidR role roleR #j #k #l #m #n #o #p.
   RS(tid, rs)@j & HonestS(rs)@k & 
   Running(tidR, roleR, rs)@l & 
   RS(tidR, s)@m & S(tid, s)@n & 
   Running(tid, role, s)@o & not role = roleR & 
   CommitPayload(tidR, 'payload3', payload3)@p & 
   ((Ex tid2 #q #r #t. 
     ReceivePayload(tid2, 'payload3', payload3) 
     @q & 
     S(tid2, s)@r & 
     RS(tid2, rs)@t & 
     not tid = tid2) 
   ==> 
   (Ex tidR2 #q #r #t. 
    CommitPayload(tidR2, 'payload3', payload3) 
    @q & 
    S(tidR2, rs)@r & 
    RS(tidR2, s)@t & 
    not tidR = tidR2))) | 
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | 
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) | 
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | 
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)
"

lemma Payload3KCIAuthenticationProperty1:
" 
All tid payload3 #i.
ReceivePayload(tid, 'payload3', payload3)@i 
==> 
(Ex rs tidR roleR #j #k #l. 
 RS(tid, rs)@j & HonestS(rs)@k &
lemma Payload3KCIAuthenticationProperty2:

All tid payload3 #i.

ReceivePayload(tid, 'payload3', payload3)@i

⇒ (Ex s rs tidR roleR #j #k #l #m #n.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n) |
    (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
    (Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload3KCIAuthenticationProperty3:

All tid payload3 #i.

ReceivePayload(tid, 'payload3', payload3)@i

⇒ (Ex s rs tidR roleR #j #k #l #m #n #o #p.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n &
    Running(tid, role, s)@o & not role = roleR &
    CommitPayload(tidR, 'payload3', payload3)@p) |
    (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
    (Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload3KCIAuthenticationProperty4:

All tid payload3 #i.

ReceivePayload(tid, 'payload3', payload3)@i

⇒ (Ex s rs tidR roleR #j #k #l #m #n #o #p.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n &
    Running(tid, role, s)@o & not role = roleR &
    CommitPayload(tidR, 'payload3', payload3)@p &
lemma Payload1SourceProperty1: 

All tid payload1 #i.
ReceivePayload(tid, 'payload1', payload1)@i

==> (Ex rs #j. RS(tid, rs)@j) &
((Ex tidR rs #j #k #l. CommitPayload(tidR, 'payload1', payload1)@j & RS(tid, rs)@k & S(tidR, rs)@l1) | (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) | (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k)) | (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k))

lemma Payload1SourceProperty2:

All tid payload1 #i.
ReceivePayload(tid, 'payload1', payload1)@i

==> (Ex rs #j. RS(tid, rs)@j) &
((Ex tidR rs #j #k #l. CommitPayload(tidR, 'payload1', payload1)@j & RS(tid, rs)@k & S(tidR, rs)@l1) | (Ex rs #j. RS(tid, rs)@j & not (Ex #k. HonestS(rs)@k)) | (Ex s #j #k. S(tid, s)@j & Reveal(s)@k) | (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k)) | (Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k))
lemma Payload1AuthenticationProperty2:

All tid payload1 #i.

ReceivePayload(tid, 'payload1', payload1)@i

==> (Ex s rs tidR roleR #j #k #l #m #n.

RS(tid, rs)@j & HonestS(rs)@k &

Running(tidR, roleR, rs)@l &

RS(tidR, s)@m & S(tid, s)@n) |

(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |

(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) |

(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |

(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k) |

 lemma Payload1AuthenticationProperty3:

All tid payload1 #i.

ReceivePayload(tid, 'payload1', payload1)@i

==> (Ex s rs tidR role roleR #j #k #l #m #n #o #p.

RS(tid, rs)@j & HonestS(rs)@k &

Running(tidR, roleR, rs)@l &

RS(tidR, s)@m & S(tid, s)@n &

Running(tid, role, s)@o & not role = roleR &

CommitPayload(tidR, 'payload1', payload1)@p) |

(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |

(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) |

(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |

(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)
lemma Payload1AuthenticationProperty4:
" All tid payload1 #i.
ReceivePayload(tid, 'payload1', payload1)@i
==> (Ex s rs tidR role roleR #j #k #l #m #n #o #p.
RS(tid, rs)@j & HonestS(rs)@k &
Running(tidR, roleR, rs)@l &
RS(tidR, s)@m & S(tid, s)@n &
Running(tid, role, s)@o & not role = roleR &
CommitPayload(tidR, 'payload1', payload1)@p &
((Ex tid2 #q #r #t.
ReceivePayload(tid2, 'payload1', payload1)
  @q &
S(tid2, s)@r &
RS(tid2, rs)@t &
not tid = tid2)
==> (Ex tidR2 #q #r #t.
CommitPayload(tidR2, 'payload1', payload1)
  @q &
S(tidR2, rs)@r &
RS(tidR2, s)@t &
not tidR = tidR2))) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) |
(Ex s #j #k. S(tid, s)@j & Reveal(s)@k) |
(Ex psk #j #k. PSK(tid, psk)@j & RevealPSK(psk)@k)
"

lemma Payload1KCIAuthenticationProperty1:
" All tid payload1 #i.
ReceivePayload(tid, 'payload1', payload1)@i
==> (Ex rs tidR roleR #j #k #l.
RS(tid, rs)@j & HonestS(rs)@k &
Running(tidR, roleR, rs)@l) |
(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |
(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)
"

lemma Payload1KCIAuthenticationProperty2:
lemma Payload1KCIAuthenticationProperty3:

All tid payload1 #i.

ReceivePayload(tid, 'payload1', payload1)@i

==> (Ex s rs tidR roleR #j #k #l #m #n.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n) |

(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |

(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload1KCIAuthenticationProperty4:

All tid payload1 #i.

ReceivePayload(tid, 'payload1', payload1)@i

==> (Ex s rs tidR role roleR #j #k #l #m #n #o #p.
    RS(tid, rs)@j & HonestS(rs)@k &
    Running(tidR, roleR, rs)@l &
    RS(tidR, s)@m & S(tid, s)@n &
    Running(tid, role, s)@o & not role = roleR &
    CommitPayload(tidR, 'payload1', payload1)@p) |

(Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) |

(Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k)

lemma Payload1KCIAuthenticationProperty5:
C. Tamarin Model Example

1073       ==> 
1074       (Ex tidR2 #q #r #t. 
1075       CommitPayload(tidR2, 'payload1', payload1) 
1076           @q & 
1077    S(tidR2, rs)@r & 
1078     RS(tidR2, s)@t & 
1079     not tidR = tidR2))) | 
1080       (Ex rs #j #k. RS(tid, rs)@j & Reveal(rs)@k) | 
1081     (Ex rs #j. RS(tid, rs)@j & not Ex #k. HonestS(rs)@k) 
1082 " 
1083 end
Appendix D

Results

This appendix lists the verification results for patterns named in the Noise specification. The properties described in Chapter 5 are presented here in the following way.

- All integrity checks hold for all protocols. They are therefore omitted from the results.

- The three key agreement properties Key Secrecy, Key Uniqueness and Hash Uniqueness from both Alice’s and Bob’s perspective are listed in the Key Agreement Properties table.

- The payload properties are listed in the Payload Properties table together with the protocol’s pattern and the corresponding payload numbers. For each of the four categories ‘Destination Property’ (listed under Destination), ‘Source Property’ (Source), ‘Authentication Property’ (Authentication) and ‘KCI Resistant Authentication Property’ (KCI), only the highest achieved property is listed (or 0 if none of them hold for the corresponding payload). Since they all are strictly hierarchical, this also implies that all lower ones are achieved as well. In many protocols, the payload properties of transport messages are better than those of the last handshake message. Therefore, the first two transport messages are also listed in this table and indicated with $\rightarrow$ (transport) or $\leftarrow$ (transport) respectively. The properties listed for the second transport payload require that the first transport payload was successfully received and authenticated. For one-way protocols, only one transport message is listed since the responder cannot send one in return.

Differences between the results of Noise Explorer and those obtained using the Tamarin lemmas from Chapter 5 are highlighted and very briefly discussed.
D. Results

D.1 One-Way Patterns

Noise specifies three one-way protocols, which allow for zero-RTT encryption. However, since no DH exponentiation uses the responder’s (in the examples below Bob’s) ephemeral key, replaying the messages can lead Bob to use the same keys multiple times. In order to prevent key reuse, the responder may therefore not use the negotiated keys to encrypt anything.

D.1.1 N

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payload Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td># Message</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>e, es</td>
</tr>
<tr>
<td>(transport)</td>
</tr>
</tbody>
</table>

D.1.2 K

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payload Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td># Message</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>e, es, ss</td>
</tr>
<tr>
<td>(transport)</td>
</tr>
</tbody>
</table>
### D.1.3 X

#### Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

#### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>➔ e, es, s, ss</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>➔ (transport)</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
D. Results

D.2 Fundamental Interactive Patterns

The following twelve so-called fundamental interactive handshake patterns form the basis from which most Noise protocols are derived [64].

D.2.1 IX

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th># Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 → e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 ← e, ee, s, s, s</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3 → (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4 ← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

D.2.2 IK

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th># Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 → e, es, s, ss</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2 ← e, ee, se</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3 → (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4 ← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
D.2. Fundamental Interactive Patterns

D.2.3 IN

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 3 to also satisfy destination property 3, but not, however, destination property 2 [17].

D.2.4 XX

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination property 3, but not, however, destination property 2 [48].
D. Results

D.2.5 XK

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination property 3, but not, however, destination property 2 [43].

D.2.6 XN

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payloads 2, 3 and 5 to also satisfy destination property 3, but not, however, destination property 2 [46].
D.2.7 KX

Key Agreement Properties

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se, s, es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

D.2.8 KK

Key Agreement Properties

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es, ss</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
### D. Results

#### D.2.9 KN

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

#### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties' ‘to a known recipient’ clauses, it considers payload 3 to also satisfy destination property 3, but not, however, destination property 2 [26].

#### D.2.10 NX

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

#### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties' ‘to a known recipient’ clauses, it considers payloads 2 and 4 to also satisfy destination property 3, but not, however, destination property 2 [36].
D.2.11 NK

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payloads 2 and 4 to also satisfy destination property 3, but not, however, destination property 2 [29].

D.2.12 NN

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payloads 2, 3 and 4 to also satisfy destination property 3, but not, however, destination property 2 [33].
D. Results

D.3 Deferred Patterns

As of revision 34, the Noise specification defines 23 so-called deferred patterns. These do not perform es and se DH-operations in the earliest message possible but defer them to the following one in order to improve identity hiding properties.

D.3.1 IX1

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Key Secrecy</td>
</tr>
<tr>
<td>Key Uniqueness</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se, e</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

D.3.2 IK1

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Key Secrecy</td>
</tr>
<tr>
<td>Key Uniqueness</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se, es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
D.3.3 XX1

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ es, se</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination property 3, but not, however, destination property 2 [49].

D.3.4 XK1

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination property 3, but not, however, destination property 2 [44].
D. Results

### D.3.5 KX1

#### Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

#### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se, s</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

### D.3.6 KK1

#### Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

#### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se, es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
D.3. Deferred Patterns

D.3.7 NX1

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>e, ee, s</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>es</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>(transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>(transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Since NOISE EXPLORER does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payloads 2 and 4 to also satisfy destination property 3, but not, however, destination property 2 [37].

D.3.8 NK1

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>e, ee, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>(transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>(transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Since NOISE EXPLORER does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payloads 2 and 4 to also satisfy destination property 3, but not, however, destination property 2 [30].
D. Results

D.3.9 I1X

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of I1X as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Although Noise Explorer does note that payload 2 also satisfies destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model [15].
D.3. Deferred Patterns

D.3.10 I1K

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es, s</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of I1K as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es, s</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Although Noise Explorer does note that payload 2 also satisfies destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model [11].
D. RESULTS

D.3.11 I1N

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NOISE EXPLORER lists the source and destination properties of I1N as

<table>
<thead>
<tr>
<th>Source &amp; Destination Properties according to NOISE EXPLORER</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Although NOISE EXPLORER does note that payloads 2, 3 and 5 also satisfy destination property 3. Destination property 2 not being achieved in NOISE EXPLORER’s results is because it is implemented with an active attacker in that model. The additional destination property 3s stem from the way that the property is implemented in NOISE EXPLORER without requiring the payload’s recipient be identified [14].
D.3. Deferred Patterns

D.3.12 I1X1

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$e, s$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$e, ee, s$</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$se, es$</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>(transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>(transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of I1X1 as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$e, s$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$e, ee, s$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$se, es$</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>(transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>(transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Although Noise Explorer does note that payload 2 also satisfies destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model [16].
D. Results

D.3.13 I1K1

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Key Secrecy</td>
</tr>
<tr>
<td>Key Uniqueness</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of I1K1 as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>← s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, es</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Although Noise Explorer does note that payload 2 also satisfies destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model [12].
### D.3.14 X1X

**Key Agreement Properties**

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Payload Properties**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← se</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since **NOISE EXPLORER** does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination property 3, but not, however, destination property 2 [41].

### D.3.15 X1K

**Key Agreement Properties**

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Payload Properties**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← se</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since **NOISE EXPLORER** does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination
D. Results

property 3, but not, however, destination property 2 [39].

D.3.16 X1N

Key Agreement Properties

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>no</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ s</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← se</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payloads 2, 3 and 5 to also satisfy destination property 3, but not, however, destination property 2 [40].

D.3.17 X1X1

Key Agreement Properties

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ es, s</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← se</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination property 3, but not, however, destination property 2 [42].
D.3. Deferred Patterns

D.3.18 X1K1

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Key Secrecy</td>
</tr>
<tr>
<td>Key Uniqueness</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← se</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since Noise Explorer does not implement the destination properties’ ‘to a known recipient’ clauses, it considers payload 2 to also satisfy destination property 3, but not, however, destination property 2 [42].

D.3.19 K1X

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Key Secrecy</td>
</tr>
<tr>
<td>Key Uniqueness</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of K1X as
D. Results

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Although Noise Explorer does note that payload 2 also satisfies destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model [23].

D.3.20 K1K

Key Agreement Properties

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of K1K as
D.3. Deferred Patterns

Source & Destination Properties according to **Noise Explorer**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Although **Noise Explorer** does note that payload 2 also satisfies *destination property* 3. *Destination property* 2 not being achieved in **Noise Explorer**’s results is because it is implemented with an active attacker in that model [20].

**D.3.21 K1N**

**Key Agreement Properties**

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Payload Properties**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Noise Explorer** lists the source and destination properties of K1N as
D. Results

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td>...</td>
<td>1 → e</td>
<td>0 0</td>
</tr>
<tr>
<td>2 ← e, ee</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 → se</td>
<td>1 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 ← (transport)</td>
<td>5 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 → (transport)</td>
<td>1 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although Noise Explorer does note that payloads 2, 3 and 5 also satisfy destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model. The additional destination property 3s stem from the way that the property is implemented in Noise Explorer without requiring the payload’s recipient be identified [22].

D.3.22 K1X1

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td>...</td>
<td>1 → e</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>2 ← e, ee, s</td>
<td>3 0</td>
<td>0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 → se, es</td>
<td>3 2</td>
<td>4 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 ← (transport)</td>
<td>5 2</td>
<td>4 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 → (transport)</td>
<td>5 2</td>
<td>4 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of K1X1 as
Although Noise Explorer does note that payload 2 also satisfies destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model [24].

D.3.23 K1K1

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th># Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>left s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 → e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 ← e, ee, es</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3 → se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4 ← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5 → (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of K1K1 as
D. Results

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, es</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ se</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Although Noise Explorer does note that payload 2 also satisfies destination property 3. Destination property 2 not being achieved in Noise Explorer’s results is because it is implemented with an active attacker in that model [21].
D.4 Modified Patterns

Noise also defines two modifiers, which can be applied to the patterns listed above. The psk modifier allows for the use of a pre-shared key in the key derivation process, while the fallback modifier enables the reuse of an already known ephemeral key. This can save half a round trip and possibly obfuscate the protocol [64]. When combined, the modifiers allow for several hundred variations on the previously mentioned protocols. Following are the results of the ones named in the Noise specification.

D.4.1 Npsk0

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Key Secrecy</td>
</tr>
<tr>
<td>Key Uniqueness</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 → psk, e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1 → (transport)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Note that the missing authentication stems from the fact that PSKs are not treated as a secret that can be used to identify a partner.

Noise Explorer, which treats PSKs differently, lists the source and destination properties as [52]

<table>
<thead>
<tr>
<th>Source &amp; Destination Properties according to Noise Explorer</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
<tr>
<td>← s</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>1 → psk, e, es</td>
</tr>
<tr>
<td>1 → (transport)</td>
</tr>
</tbody>
</table>
## D. Results

### D.4.2 Kpsk0

#### Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

#### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ psk, e, es, ss</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>→ (transport)</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

### D.4.3 Xpsk1

#### Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

#### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es, s, ss, psk</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>→ (transport)</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
D.4. Modified Patterns

D.4.4 XXfallback

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>→ s, se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Note that, unlike most patterns listed here, XXfallback is listed in Bob-initiated form in order to underline its relation to the XX pattern.

D.4.5 XXfallback+psk0

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>← psk, e, ee, s, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>→ s, se</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Note that, unlike most patterns listed here, XXfallback+psk0 is listed in Bob-initiated form in order to underline its relation to the XX pattern.

NOISE EXPLORER has not published results for any fallback protocols [10]. Given that NOISE EXPLORER’s findings differ from those presented here for most psk protocols, it is likely that the results for XXfallback+psk0 would
D. Results

not match either. Especially since there are three differences in the results for XXpsk3 (see D.4.13), the most similar protocol Noise Explorer has published results for.

D.4.6 NNpsk0

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>psk, e</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>e, ee</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(transport)</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>(transport)</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that the missing authentication stems from the fact that PSKs are not treated as a secret that can be used to identify a partner.

Noise Explorer, which treats PSKs differently, lists the source and destination properties as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>psk, e</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>e, ee</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>(transport)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>(transport)</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

and also shows payloads 2 through 4 as satisfying destination property 5 [34].
D.4 Modified Patterns

D.4.7 NNpsk2

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, psk</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that the missing authentication stems from the fact that PSKs are not treated as a secret that can be used to identify a partner.

NOISE EXPLORER, which treats PSKs differently, lists the source and destination properties as

Source & Destination Properties according to NOISE EXPLORER

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, psk</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

and also shows payloads 3 and 4 as satisfying destination property 5 [35].
D. Results

D.4.8 NKpsk0

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ psk, e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of NKpsk0 as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ psk, e, es</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

and also shows payload 3 as satisfying destination property 5 [31].

Most differences are because of the way PSKs are treated differently. Noise Explorer considering payload 3 not to satisfy destination property 4, however, is because of how that property is implemented. For this payload, Noise Explorer implements destination property 4 such that it requires the payload be safe even from an attacker actively impersonating a compromised recipient.
D.4.9 NKpsk2

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, psk</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NOISE EXPLORER lists the source and destination properties of NKpsk2 as

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>← s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, psk</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

and also shows payload 3 as satisfying destination property 5 [32].

Most differences are because of the way PSKs are treated differently. Payload 1 not providing any confidentiality, for instance, is due to the way NOISE EXPLORER considers static DH keys as being unsafe in protocols that include a PSK. NOISE EXPLORER considering payload 3 not to satisfy destination property 4, however, is because of how that property is implemented. For this payload, NOISE EXPLORER implements destination property 4 such that it requires the payload be safe even from an attacker actively impersonating a compromised recipient.
D. Results

D.4.10 NXpsk2

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

## Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es, psk</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of NXpsk2 as

### Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es, psk</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

and also shows payload 3 as satisfying destination property 5 [38].

Most differences are because of the way PSKs are treated as sufficient to identify a party to the protocol. Noise Explorer considering payload 3 not to satisfy destination property 4, however, is because of how that property is implemented. For this payload, Noise Explorer implements destination property 4 such that it requires the payload be safe even from an attacker actively impersonating a compromised recipient.
D.4. Modified Patterns

D.4.11 XNpsk3

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>←, e, ee</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se, psk</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of XNpsk3 as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>←, e, ee</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se, psk</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

and also shows payload 2 as satisfying destination property 3 and payload 4 as satisfying destination property 5 [47].

Most differences are because of the way PSKs are treated differently and because of Noise Explorer’s implementation of the payload properties ignoring their ‘to a known recipient’ clause. Noise Explorer considering payload 4 not to satisfy destination property 4, however, is because of how that property is implemented. For this payload, Noise Explorer implements destination property 4 such that it requires the payload be safe even from an attacker actively impersonating a compromised recipient.
D. Results

D.4.12 XKpsk3

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>← s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se, psk</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of XKpsk3 as

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>← s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ e, es</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se, psk</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

and also shows payload 2 as satisfying source property 2 and destination property 3 [45].

The differences stem from the way Noise Explorer

- treats static DH keys as unsafe in PSK protocols, essentially only relying on them for security as long as the PSK has not been compromised either and
- does not model the ‘to a known recipient’ clauses of destination properties, therefore allowing for better destination property results.
D.4. Modified Patterns

D.4.13 XXpsk3

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se, psk</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Noise Explorer** lists the source and destination properties of XXpsk3 as

Source & Destination Properties according to **Noise Explorer**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, s, es</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ s, se, psk</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

and also shows payload 2 as satisfying source property 2 and destination property 3 [50].

The differences stem from the way **Noise Explorer**

- treats static DH keys as unsafe in PSK protocols, essentially only relying on them for security as long as the PSK has not been compromised either and

- does not model the ‘to a known recipient’ clauses of destination properties, therefore allowing for better destination property results.
D. Results

D.4.14 KNpsk0

### Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>→ s</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ psk, e</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Noise Explorer** lists the source and destination properties of KNpsk0 as

**Source & Destination Properties according to Noise Explorer**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>→ s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ psk, e</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

and also shows payload 4 as satisfying destination property 5 [27].

The differences stem from the way **Noise Explorer**

- treats PSKs as sufficient to identify a participant. This leads to the higher authentication results.
- does not model the ‘to a known recipient’ clauses of destination properties, therefore allowing for better destination property results.
- only requires that the PSK be safe for destination property 4 for messages sent by Alice, but not for messages sent by Bob. Thereby capping the destination property of ← messages at 3.
D.4.15 KNpsk2

### Key Agreement Properties

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td>1 → e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 ← e, ee, se, psk</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3 → (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4 ← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Noise Explorer** lists the source and destination properties of KNpsk2 as

| Source & Destination Properties according to Noise Explorer |
|---|---|---|---|
| # | Message | Destination | Source |
| 1 | → e | 0 | 0 |
| 2 | ← e, ee, se, psk | 3 | 1 |
| 3 | → (transport) | 5 | 2 |
| 4 | ← (transport) | 3 | 1 |

and also shows payload 4 as satisfying *destination property* 5 [28].

The differences stem from the way **Noise Explorer**

- treats PSKs as sufficient to identify a participant. This leads to the higher authentication results.

- does not model the ‘to a known recipient’ clauses of destination properties, therefore allowing for better destination property results.

- only requires that the PSK be safe for *destination property* 4 for messages sent by Alice, but not for messages sent by Bob. Thereby capping the destination property of ← messages at 3.
### D.4.16 KKPSK0

**Key Agreement Properties**

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Payload Properties**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td>→ psk, e, es, ss</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>← s</td>
<td>← e, ee, se</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

### D.4.17 KKPSK2

**Key Agreement Properties**

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Payload Properties**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ s</td>
<td>→ e, es, ss</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>← s</td>
<td>← e, ee, se, psk</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of KKPSK2 as [25].
The differences are due to \texttt{Noise Explorer} considering static DH keys ‘less safe’ in PSK protocols. The properties are formulated so that, in order to pass, a protocol must ensure that a static key being compromised does not provide an advantage to the attacker unless the PSK is also revealed.

\section*{D.4.18 KXpsk2}

\begin{center}
\begin{tabular}{ll}
\multicolumn{2}{c}{Key Agreement Properties} \\
\hline \\
& Alice & Bob \\
Key Secrecy & yes & yes \\
Key Uniqueness & yes & yes \\
Hash Uniqueness & yes & yes \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{lllll}
\multicolumn{5}{c}{Payload Properties} \\
\hline \\
# & Message & Destination & Source & Authentication & KCI \\
\hline \\
& $\rightarrow s$ & & & & \\
& ... & & & & \\
1 & $\rightarrow e$ & 0 & 0 & 0 & 0 \\
2 & $\leftarrow e$, ee, se, s, es, psk & 3 & 2 & 4 & 4 \\
3 & $\rightarrow$ (transport) & 5 & 2 & 4 & 4 \\
4 & $\leftarrow$ (transport) & 5 & 2 & 4 & 4 \\
\hline
\end{tabular}
\end{center}
D. Results

D.4.19 INpsk1

<table>
<thead>
<tr>
<th>Key Agreement Properties</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s, psk</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Noise Explorer lists the source and destination properties of INpsk1 as [18]

Source & Destination Properties according to Noise Explorer

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s, psk</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Additionally, payload 4 satisfies destination property 5 according to Noise Explorer.

Some properties not being achieved according to Noise Explorer is due to its model considering static DH keys ‘less safe’ in PSK protocols. The properties are formulated so that, in order to pass, a protocol must ensure that a static key being compromised does not provide an advantage to the attacker unless the PSK is also revealed. Some results being higher in Noise Explorer’s results is because it considers PSKs sufficient for identifying a protocol participant.
### D.4. Modified Patterns

#### D.4.20 INpsk2

**Key Agreement Properties**

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Payload Properties**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\rightarrow e, s$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$\leftarrow e, ee, se, psk$</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>$\rightarrow$ (transport)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>$\leftarrow$ (transport)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Noise Explorer** lists the source and destination properties of INpsk2 as [19].

**Source & Destination Properties according to Noise Explorer**

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\rightarrow e, s$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$\leftarrow e, ee, se, psk$</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$\rightarrow$ (transport)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>$\leftarrow$ (transport)</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Additionally, payload 4 satisfies *destination property 5* according to Noise Explorer.

Some properties not being achieved according to Noise Explorer is due to its model considering static DH keys ‘less safe’ in PSK protocols. The properties are formulated so that, in order to pass, a protocol must ensure that a static key being compromised does not provide an advantage to the attacker unless the PSK is also revealed. Some results being higher in Noise Explorer’s results is because it considers PSKs sufficient for identifying a protocol participant.
D. Results

D.4.21 IKpsk1

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, es, s, ss, psk</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer has not published results for IKpsk1. Given that their results for most psk patterns diverge from those presented here, Noise Explorer’s findings would likely differ from these.

D.4.22 IKpsk2

Key Agreement Properties

<table>
<thead>
<tr>
<th></th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, es, s, ss</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se, psk</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Noise Explorer has not published results for IKpsk2. Given that their results for most psk patterns diverge from those presented here, Noise Explorer’s findings would likely differ from these.
D.4. Modified Patterns

D.4.23 IXpsk2

Key Agreement Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Secrecy</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Key Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Hash Uniqueness</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Payload Properties

<table>
<thead>
<tr>
<th>#</th>
<th>Message</th>
<th>Destination</th>
<th>Source</th>
<th>Authentication</th>
<th>KCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>→ e, s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>← e, ee, se, s, es, psk</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>→ (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>← (transport)</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>