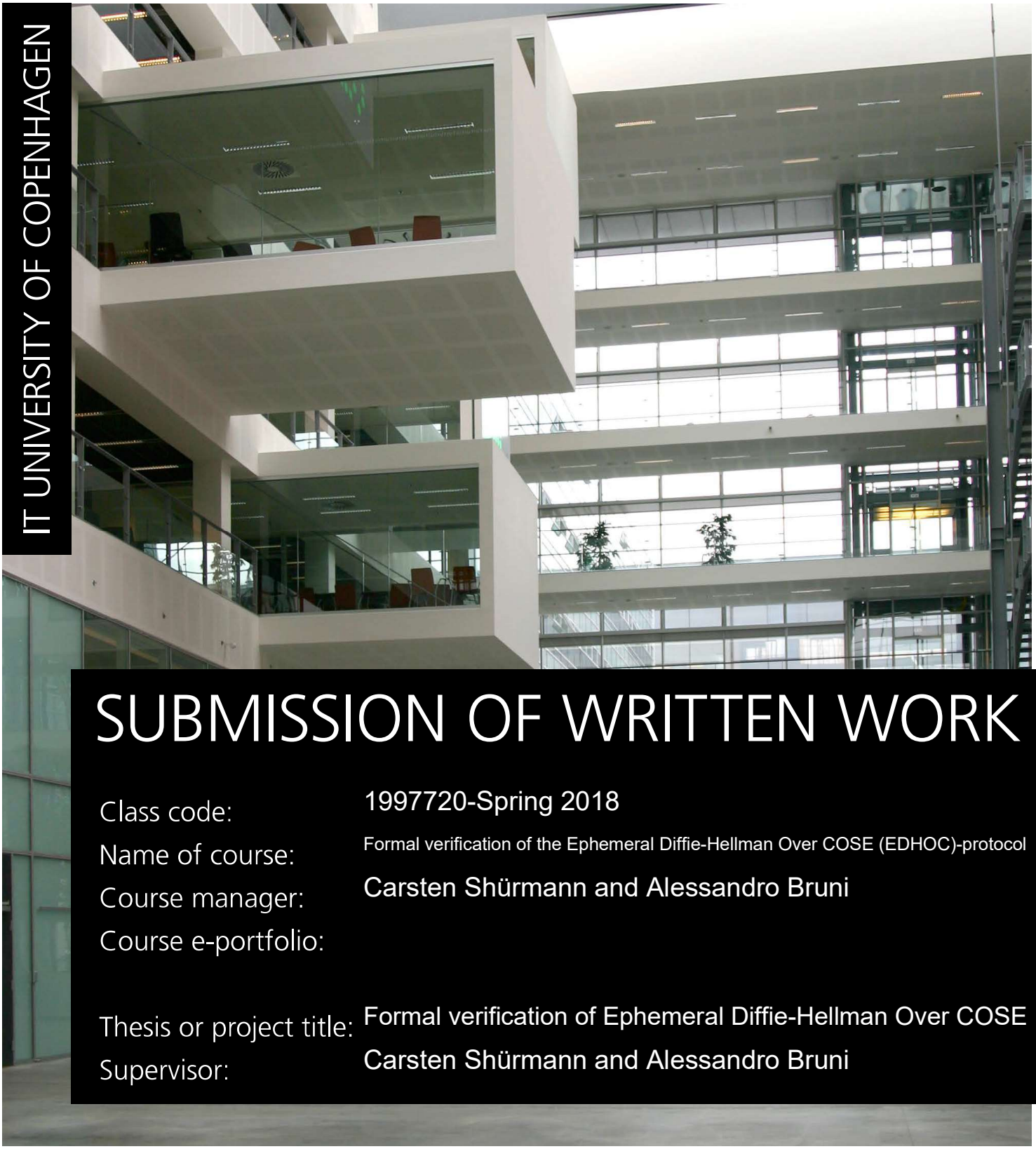
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:	1997720-Spring 2018
Name of course:	Formal verification of the Ephemeral Diffie-Hellman Over COSE (EDHOC)-protocol
Course manager:	Carsten Shürmann and Alessandro Bruni
Course e-portfolio:	
Thesis or project title:	Formal verification of Ephemeral Diffie-Hellman Over COSE
Supervisor:	Carsten Shürmann and Alessandro Bruni

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Theis Grønbech Petersen	23/12-1994	thep@itu.dk
2.	Thorvald Sahl Jørgensen	16/09-1995	thjo@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

IT UNIVERSITY OF COPENHAGEN

Bachelor Thesis

Formal Verification of Ephemeral Diffie-Hellman Over COSE

Authors:
Thorvald Sahl Jørgensen
Theis Grønbech Petersen

Supervisors:
Alessandro Bruni
Carsten Schürmann

IT University of Copenhagen
Copenhagen, Denmark

May 2018

Abstract

The Ephemeral Diffie-Hellman Over COSE (EDHOC)-protocol is an authentication protocol built for IoT. As a critical factor regarding security of devices in IoT is the limited computational power, it has been design to be very lightweight. EDHOC should ensure four security properties: Injective Agreement, Privacy, Confidentiality and Forward Secrecy. EDHOC has been proposed for standardization, making a formal verification of its given properties mandatory.

In this thesis we conduct a formal verification of EDHOC. This is done through a model created with the tool ProVerif. ProVerif is a modelling-tool that enables the automatic verification of encryption protocols. We managed to prove the properties of Injective Agreement and Privacy. This being said, we also found that EDHOC is unfortunately critically flawed with regards to both Confidentiality and Forward Secrecy. We therefore proposed and verified a solution that ensures all four properties.

Keywords: EDHOC, ProVerif, Automatic Verification, Internet of Things, IoT

Acknowledgements

We would like to express our gratitude for the supervision of this project by Carsten Schürmann and Alessandro Bruni. Their help through this thesis got us through many issues that we encountered during the development of the verification models.

The thesis has been made possible by the authors of the EDHOC protocol and especially by Göran Selander. We are grateful his interest in the project and for assisting our understanding of the protocol.

Finally, we wish to thank our family and friends for support and encouragement.

Contents

1 Introduction

Given the many cyberattacks on Internet of Things (IoT) devices, it is clear that security in IoT needs to be strengthened[1]. For this reason an authentication protocol has been proposed for standardization. This protocol, the Ephemeral Diffie-Hellman Over COSE (EDHOC)-protocol, is in need of a formal verification in order to ensure that it is able to supply secure communication in IoT.

In this bachelor project we have analyzed and formally verified the EDHOC-protocol. The verification consist of the development of a model in ProVerif to validate that the protocol works securely. The model is based on draft 08[2].

The critical factor regarding security of devices in IoT, is that these small devices have limited computational power. EDHOC is build to be very lightweight, making it ideal for use in IoT. The protocol exchanges three messages to establish session keys while transmitting application data.

EDHOC should secure the following four security properties: `Injective Agreement`, `Privacy`, `Confidentiality` and `Forward Secrecy`. The model

[1] Cloudflare Blog. 2018. Inside the infamous Mirai IoT Botnet: A Retrospective Analysis. [ONLINE] Available at: https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/. [Accessed May 2018].

[2] Ephemeral Diffie-Hellman Over COSE (EDHOC). draft-selander-ace-cose-ecdhe-08. [ONLINE] Available at: https://datatracker.ietf.org/doc/draft-selander-ace-cose-ecdhe/08. [Accessed 14 May 2018].

will be created using the tool ProVerif [1]. ProVerif is a modelling-language that enables the automatic verification of encryption protocols. This tool makes it possible to create a model of a protocol and run several queries to validate security properties. If ProVerif finds any vulnerability in the model, it will produce a list of the steps needed to exploit this. ProVerif is chosen as it seems to be the most mature modelling language for this type of task.

Creating the model, we have made some assumptions and simplifications. Due to time constraints as well as constraints of ProVerif, we have found it necessary to disregard certain elements. As our model does not cover *Algorithm Negotiation*, we assume that the algorithm negotiation results in algorithms widely considered secure. This also means that inner workings of those algorithms as well as Elliptic Curves lies beyond the scope of the project.

1.1 EDHOC compared to TLS 1.3

As EDHOC and TLS 1.3 have many aspects in common, the model of TLS 1.3 have been a source of inspiration[4]. This is due to the fact that TLS works quite similar to EDHOC in regard to encryption and the structure of messages. The model of EDHOC is much simpler and more lightweight than the already accepted TLS, as TLS 1.3 has to be backwards compatible with TLS 1.2. The backwards compatibility complicates the model significantly.

2

Ephemeral Diffie-Hellman Over COSE

The Ephemeral Diffie-Hellman Over COSE (EDHOC) protocol is designed by Göran Selander, John Mattsson and Francesca Palombini. It is intended as a lightweight protocol for use in IoT while maintaining the same security properties of heavier protocols. One key feature is to send as much encrypted and integrity protected data as possible, while simultaneously establishing sessions keys[2].

EDHOC is transferring application data in each message, this being encrypted and unencrypted data. The session keys between two hosts are established using a commonly shared secret obtained from the Diffie-Hellman key exchange, as well as some long term keys.

As the protocol is designed for IoT, it has to be as lightweight as possible while still secure. For the authors of EDHOC it is a discussion of security compared to performance. Looking at both variants it would be possible to use only long-term keys for encryption. This would make the protocol even more lightweight and would enable encryption from the first message. These two points are very important when it comes to creating a lightweight protocol. The sacrifice would be the property of `Forward Secrecy`. If the long term keys of a device were leaked, it would be possible to decrypt all previous sessions if they were recorded. The decision of the authors has been to ensure `Forward Secrecy` as opposed to improving the speed even further. The Diffie-Hellman key-exchange has been chosen in order to create unique

session-keys for each session.

The purpose of Diffie-Hellman is for two hosts to establish a shared secret, based on a commonly known integer (`g`). The two processes each generate a private integer: `x` and `y`. This is the private key of the given process. This key is used as an exponent to `g`. They respectively send g^x and g^y. Each process will use the integer received to make a new exponent with their private key. This will establish the integers g^{x^y} and g^{y^x}. These integers are equal and are only known to the hosts. The key exchange is modelled in Figure 2.1.

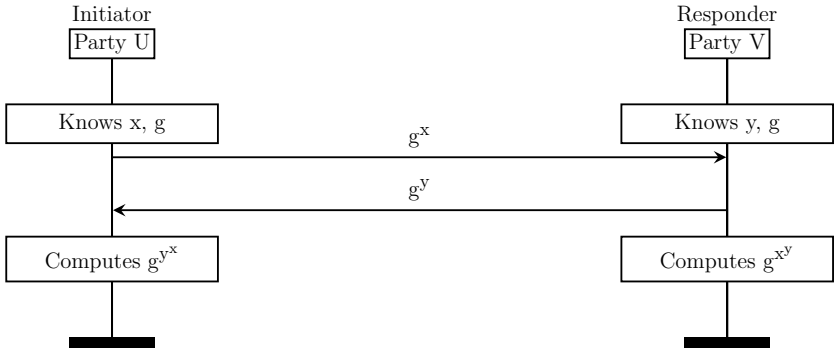


Figure 2.1: Basics of Diffie-Hellman

Using the Diffie-Hellman key along with the long term keys of each host ensures that if the long term keys are compromised, a potential attacker would not be able to read the information of previous sessions. This is due to the fact that the shared secret generated for the given session, will only be valid for that session and deleted afterwards.

2.1 Protocol

The core of EDHOC is Diffie-Hellman, but as this alone is not sufficient, the protocol has a verification-mechanism consisting of three elements. Firstly it will be verified that the session identifiers that were sent are identical to the ones received. Secondly the uniqueness of the nonce and session identifier received is verified. Thirdly the hash of previously sent data is validated by both hosts in order to ensure the two hosts agree on the content of the messages.

There are two different variants of the EDHOC protocol: A symmetric and an asymmetric variant. The symmetric variant uses a pre-shared key as salt when deriving the session keys from the key exchange. This will be described in detail in Section 2.1.1. The asymmetric variant uses public-private key pairs to create signatures in order to validate the authenticity of the messages. This will be described further in Section 2.1.2.

EDHOC uses three messages in order to establish shared ephemeral session keys. The first message is pure plain text. The following two consists of plain text data and an encryption. The data payload consists of information such as message types, nonces and the session identifiers in use. The encryption payloads contains application data. All the variables used in EDHOC have their abbreviations explained in Table 2.1.

The following is a description of the protocol between `Party U` and `Party V`. The symmetric variant of the protocol can be seen in Figure 2.2.

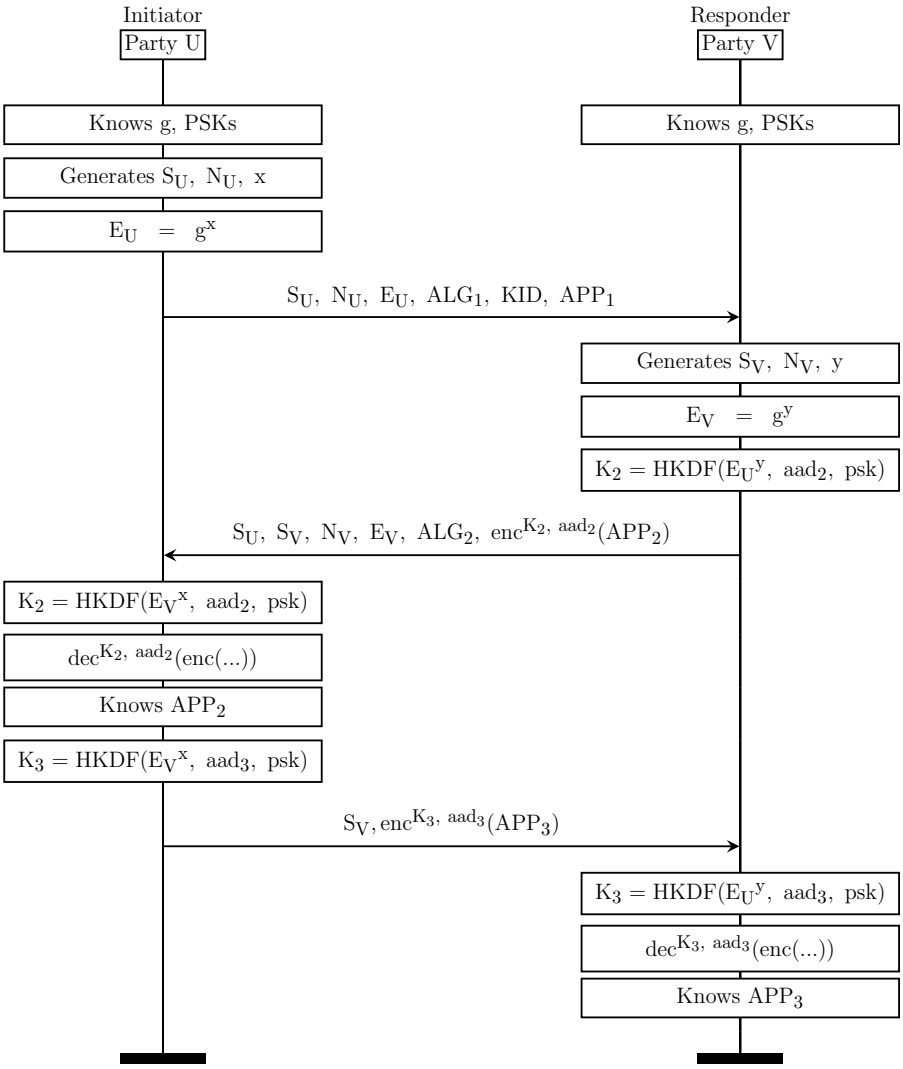


Figure 2.2: Model of the symmetric variant of EDHOC

`U` initiates the protocol by creating `message_1`. First `U` generates E_U for the Diffie-Hellman key exchange. `message_1` contains session data (session identifier and nonce), E_U, a set of supported algorithms and application data. `V` receives `message_1`, creates E_V and computes the shared secret. V selects an

algorithm from the set received from U to use in this session. From the shared secret and a hash (aad_2), V derives a key (K_2). The hash is based on the previously transmitted data as well as the unencrypted part of the new message.

`V` sends `message_2`, containing session data, E_V, the selected algorithms and application data encrypted with K_2 along with aad_2 to verify the integrity of the encryption. Upon reception `U` similarly computes the shared secret and derives K_2 using the shared secret and aad_2. K_2 is used to decrypt the encryption part of `message_2`.

Similarly `U` derives a key, K_3, from the shared secret and aad_3. `U` constructs `message_3` containing session data and application data encrypted with K_3 and aad_3. Upon reception `V` derives K_3, decrypts the application data and completes the protocol.

Variable	Description
S_X	Session identifier for Party X
N_X	Nonce generated by Party X
g	Common denominator for Diffie-Hellman
x, y	Private exponent for Diffie-Hellman
E_X	Diffie-Hellman public key for Party X
ALG_1	Set of supported algorithms
ALG_2	Algorithms selected for usage
APP_i	Application data for message_i
aad_i (i = 2 or 3)	Computed hash used in message_i
K_i (i = 2 or 3)	Derived key used in message_i
Symmetric	
KID	Pre-shared key identifier
PSKs	Set of pe-shared keys
psk	Pre-shared key used in the session
Asymmetric	
ID_X	Identifier for the public key of Party X
X	Private key for Party X
pkX	Public key for Party X
* X = U/V/M	

Table 2.1: Description of variables in EDHOC

2.1.1 Symmetric

The authentication of the symmetric variant is based on a set of pre-shared keys. The model is shown in Figure 2.2. The set of keys are utilized in the derivation of session keys. Here a given pre-shared key is used as salt. In `message_1` a key identifier, `KID` is sent along the other data. KID is used to identify a given key in the set of pre-shared keys. Only if in possession of the pre-shared key, one would be able to derive the session keys.

2.1.2 Asymmetric

The asymmetric variant is utilizing signatures as authentication. These are created using long-term public-private keys associated with each host. The model is shown in Figure 2.3.

`V` extends the encryption in `message_2` with a signature. This signature contains $\mathrm{ID_V}$ and APP_2. The signature is made with the private key of `V`. `U` will decrypt the encryption of `message_2` and verify the signature using the public key of `V`.

Similarly `U` extends the encryption in `message_3` with a signature containing $\mathrm{ID_U}$ and APP_3. The signature is made with the private key of `U`. Upon reception of `message_3`, party `V` will decrypt and verify the message and the protocol completes.

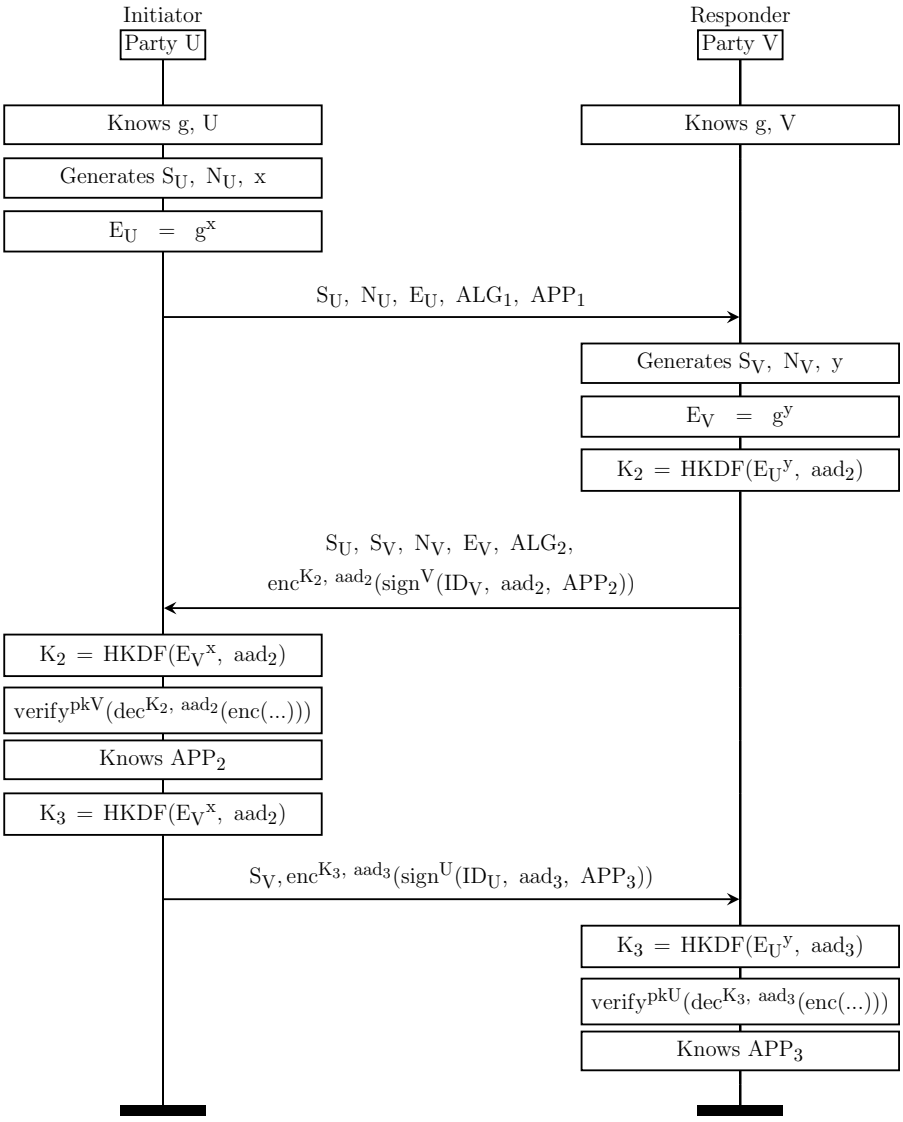


Figure 2.3: Model of the asymmetric variant of EDHOC

2.2 Properties

Given the documentation (and talks with the authors) EDHOC should adhere to the following properties [2, Section 8]. Note that the property of `Privacy` is only relevant for the asymmetric variant as the symmetric variant does not verify identities.

Injective Agreement: When a party completes a session with a given set of session identifiers, the opposite honest party must have initialized the session with the same session identifiers [5].

Privacy: The identity of a given host will first become evident when the identity of the counterpart has been verified.

Confidentiality: The encrypted data sent is unreadable to anyone but the honest participating parties in a given session [4].

Forward Secrecy: The encrypted data sent in past sessions remains unreadable to an attacker even if the long term keys are compromised.

3 ProVerif

ProVerif is a tool for automatic verification of cryptographic protocols [1]. It is a modelling-language, where only the behavior of the protocol and queries are modelled. Thus, ProVerif uses the model to attempt to establish a trace where the properties of the queries are violated.

The key feature of ProVerif is its ability to find attack traces based on a model. The developer does not have to program the attacker. The attacker in ProVerif can obtain all public information in the model and read the contents of all the messages sent on the public channel. With this information the attacker can use the public functions as well as processes defined in the model. Based on this set of options ProVerif will try to invalidate the queries. The attacker can create new messages with the observed information to trick the model into a state where its properties are invalidated.

3.1 Creating the model

In this section we will describe the functionality of ProVerif that has been used to create the model of EDHOC. In ProVerif every variable is considered bitstrings, however they can be defined with a custom type. Custom types enables the developer to set type limitations on functions.

Free/Const

`free` is used to declare a variable in global scope of the processes. An example is `free public: bitstring`. This will bind a new value of type `bitstring` to the name 'public'. By default, free values are publicly known - both to processes and to an attacker. A variable can however be declared private using the `[private]` keyword.

Similar to the `free` keyword, constants can be declared using the `const` keyword. Constants can be declared as `const c: bitstring`. This declares the constant *'c'* of type `bitstring`.

New

The `new` keyword creates a new local value with the scope of the process in which it is created. An example of this command is `new x : exponent`. This will result in a local value `x` of the type exponent.

Channels

A channel is used as a communication bridge between processes. A channel is a specific type declared by using the `channel` keyword with the `free` keyword. Data is sent through the channel using the `out` keyword. It takes a channel and a given bitstring to transfer messages over the channel.

An example of the usage of channels can be seen in Figure 3.1. Here a publicly known `channel` *'c'* is created and the message *'s'* is transferred through it. As the channel is publicly known, the processes running and an attacker can read all the data being transmitted through this channel.

```
free c: channel.

process
    new s : bitstring;
    out(c, s)
```

Figure 3.1: Example channel usage

Functions

Functions are used to create a new value from a given set of parameters. Functions are declared using the `fun` keyword and are reversed using the `reduc` keyword. Creating the model we have utilized these two functions. An example of a function is the encryption and decryption function shown in Figure 3.2.

```
fun enc(bitstring, key, bitstring): bitstring.
reduc forall x: bitstring, y: key, hash: bitstring;
            dec(enc(x, y, hash), y, hash) = x.
```

Figure 3.2: Functions in ProVerif

`enc` takes a given `bitstring`, a `key` and a `hash` and returns a cipher text in the form of a `bitstring`. When applying a given function to a set of values, the result will always be the same. The decryption is done using a `reduction function`: `dec`. Given the cipher text, the `key` and the `hash`, the original plain text is returned. Functions can also be declared with `[private]`, making the specific function unable to use for the attacker.

Equations

The `equation` keyword is used to define relations. Given the Diffie-Hellman key exchange it is essential that $g^{xy} = g^{yx}$. In order to create this relation,

the `equation` keyword is used. An example of this equation is shown in Figure 3.3.

```
const g: G.
fun exp(G, exponent): G.
equation forall x: exponent, y: exponent;
    exp(exp(g, x), y) = exp(exp(g, y), x).
```

Figure 3.3: Example of equation

Phases

`Phases` is a functionality that enables the developer to create multiple steps of the model. Some processes might run first, where-after another process begins. The `phase` keyword indicates that the two processes will never run simultaneously. One phase will have to be completed before the next can start. This feature can be used to model `Forward Secrecy`, as it requires that the protocol have come to an end before the keys are compromised. The use of phases are quite simple as seen in Figure 3.4.

```
( !process(secretKey)
| phase 1 ; out(c, secrectKey))
```

Figure 3.4: Example of phases

3.2 Creating the queries

Queries

Queries are the questions, we want answered based on the model. ProVerif works with two different types of queries.

`Attacker:` A query asking whether an attacker can obtain a given piece of information.

`Event:` A query asking about execution of different parts of code with given parameters.

Attacker

ProVerif uses the query `attacker` in an attempt to establish a trace for the model where an attacker can obtain a given piece of information. Figure 3.5 is an example of an `attacker` query.

```
query attacker(new secret).
```

Figure 3.5: Query to test the confidentiality of the secret

When the model is executed, ProVerif will try to establish a trace where an attacker can obtain this secret. If such a trace exists the query will return false and the trace containing all the steps needed for an adversary to obtain the information is displayed.

Event

`Events` are used to validate that a section of the code is being executed with a given set of parameters. An example of an event and corresponding validation can be seen in Figure 3.6. Events are in its essence flags that will be set, if the code reaches a given line. The queries check for these flags and their parameters.

```
event startProcess.
event endProcess.

query event(endProcess) ==> event(startProcess).
```

Figure 3.6: Validation of execution of events

This example will only be true if: For all cases where the event `endProcess` is executed, there has been a previous execution of the event `startProcess`. If, however, it is possible for ProVerif to establish a call stack where the `endProcess` event is the only event executed, the query will return false. This means that there is a path where an attacker is able to execute `endProcess` without executing the `startProcess`. These types of queries are often used for check of authentication.

3.3 Understanding the result

ProVerif has three different types of results.

RESULT [Query] is true: This means that there is no attack. ProVerif can prove that there exists no trace, where the given query can be violated.

RESULT [Query] is false: This means that there is an attack given the model. ProVerif will output the trace to do this attack. The developer will then have to figure out whether this is a real attack or a flaw of the model.

RESULT [Query] cannot be proved: This result means that ProVerif cannot prove that the query is true, but it can also not establish a trace where the query is false. ProVerif still provides an attack trace from which it might be possible manually to reconstruct either an attack or to verify the query. Also, small changes in the model might be able to make ProVerif give

a definitive answer, and thus to remove the uncertainty.

When ProVerif compiles the model, the output will be a numbered list of code statements as shown in Figure 3.7. These statements will be used as reference points in a potential trace. ProVerif works by deduction and the statements define starting points of this deduction.

```
{1}new skA: skey;
{2}let pkA: pkey = pk(skA) in
{3}out(c, pkA);

{4}new skB: skey;
[...]
```

Figure 3.7: Example of ProVerif statements

If the validation of the queries of the model fails, ProVerif provides a trace of how the query can be invalidated. This trace is build as a sequence of actions an attacker would need to do to break the property of the query. The trace will reference the statements from the model in order to clarify the source of this information and also reference the earlier steps in the attack trace. An example of the attack trace is displayed in Figure 3.8.

```
8. The message pk(skB[]) that the attacker may have by 1 may be received at
    input {8}.
The message sign(g,skB[]) that the attacker may have by 7 may be received at
     input {11}.
So the message enc(s[],exp(g,n0_881)) may be sent to the attacker at output
    {15}.
attacker(enc(s[],exp(g,n0_881))).

9. By 8, the attacker may know enc(s[],exp(g,n0_881)).
By 4, the attacker may know exp(g,n0_881).
Using the function dec the attacker may obtain s[].
attacker(s[]).
```

Figure 3.8: Example of ProVerif trace

3.4 Successful modelling

The success of the modelling is based on two main points. Firstly, we need to make sure that every piece of code is reachable. This can be done with a query asking whether an event has been executed. This event, `reach`, will be placed at the end of each process in order to ensure that the end of the process is actually reachable. Thus, if ProVerif establishes a trace to this point of execution, it can be concluded that the functions and logic can be executed. Based on this, we cannot however ensure that the logic is working exactly as it should according to the documentation.

Secondly, the queries defined in the model should be verified. The primary work of the developer consists of reading the output listed by ProVerif. If the code is executable and the query is returning true, one might conclude that the given property holds.

If however the query returns false, ProVerif will create an attack trace of how to violate the query. One will now need to read through the trace and conclude whether this is actually an attack or a flaw of the model. Another thing to keep in mind is that ProVerif is defined in such a way that it sometimes establishes a false attack trace - thus, it is unable to give a conclusive answer. This means that the models might require some fiddling in order to return a definite answer.

To validate a trace, one needs to read and understand the steps needed to produce this attack. The trace will list all the information an attacker would obtain and how this information could be used to break the property currently being verified. One would have to reason about the validity of this attack.

4

Modelling EDHOC using ProVerif

In this section we will cover the functionality in the model of EDHOC and explain how the protocol have been implemented in ProVerif. The theoretical explanation of the protocol is described in Section 2, this includes composition of messages and the derivation of the session keys.

As mentioned earlier, certain assumptions and simplifications is necessary in order to model EDHOC in ProVerif. Thus, we have disregarded some properties and components due to time constraints as well as constraints of ProVerif. As our model does not cover *Algorithm Negotiation*, we assume that the algorithm negotiation results in algorithms widely considered secure.

The two variants of the protocol is modelled slightly different, but consists of the same structure of the messages being sent. Thus, they differ in way of establishing the ephemeral session keys used in the encryption of each message, as well as the signatures used in the asymmetric variant. An example model is found in Appendix A and the rest is found on GitHub[1].

4.1 The Protocol

To model the process of encrypting messages with a given key, we model an encryption and a corresponding decryption function in ProVerif. The code

[1] `https://github.com/theisgroenbech/edhoc-proverif`

used for encryption and decryption is shown in Section 3. EDHOC is using *Integrity Protected Encryption* with the *Authenticated Encryption with Associated Data* (AEAD) algorithms when encrypting [2]. The function consists of three parameters: The data to be encrypted, a key used for the encryption and the hash as integrity validation.

An original message can be obtained through decryption, given an encrypted message, the key used and the hash. Normally one need not to provide the hash to decrypt. Thus, as we are never interested in decrypting a message that fails the verification, we have chosen to add the hash to the decryption. Due to the nature of ProVerif, the modelling of this separately would be unnecessarily complicated without achieving anything.

As mentioned EDHOC relies heavily on Diffie-Hellman. The core of the session key derivation is the shared secret of a Diffie-Hellman key establishment. The session keys (K_i where i = 2 or 3) are slightly different in the two variants of the protocol. The session keys are derived using the Hashed Message Authentication Code (HMAC)-based Key Derivation Function (HKDF)[3]. This is done using the function displayed in Figure 4.1. HKDF creates a session key based on the Diffie-Hellman shared secret and the hash.

```
fun HKDF(G, preSharedkey, bitstring): derivedKey.
```

Figure 4.1: Key derivation function for symmetric encryption

The symmetric variant also uses a pre-shared key as salt. The devices will be *born* with a set of these pre-shared keys. In order to agree on a given pre-shared key for a session, the initiator will send a `Key Identifier (KID)`. In order to identify a given key we have created a private function `identifyPreSharedKey`.

Signing and verification (Asymmetric only)

In the asymmetric variant, each process has a private-public key pair that is used to authenticate the data being sent. This is done using signatures. In order to model this, signing and verification functions need to be implemented. The signature function requires a private key and data. The signature is valid only if the verify function is called with the corresponding public key.

The implementation of these functions are similar the encryption and decryption functions. The two functions are displayed in Figure 4.2

```
fun sign(bitstring, skey): bitstring.
reduc forall x: bitstring, y: skey;
    verify(sign(x, y), pk(y)) = x.
```

Figure 4.2: Signing and verification

4.2 The Properties

As described in Section 2.2 the following properties should hold for EDHOC.

- `Injective Agreement`
- `Privacy`
- `Confidentiality`
- `Forward Secrecy`

This section will explain how these properties have been modelled using ProVerif, as well as the result of the modelling.

4.2.1 Injective Agreement

Injective Agreement: When a party completes a session with a given set of session identifiers, the opposite honest party must have initialized the session with the same session identifiers.

The property of `Injective Agreement` is modelled the following way: If the final message of party U is accepted, party V must have initialized the process with the same session IDs. Similarly if the final message of party V is accepted, party U must have initialized the process with same session IDs.

The model is based on events (described in Section 3). Each process has a start- and end-event with a set of parameters. For the start-event, the parameters will be the two hosts and the session ID of the given process. When the initiator process initiates the protocol, the event `startInitiator(...)` will be called. When the responder accepts the last message, the event `endResponder(...)` will be called. For the initiator it will be `startInitiator(U, V,` S_U`)`. For the end-event the set of parameters will be the two hosts and both session IDs. For the initiator the end-event will be `endInitiator(U, V,` S_U`,` S_V`)`.

```
(*Events*)
event startInitiator(host, host, bitstring).
event startResponder(host, host, bitstring).
event endInitiator(host, host, bitstring, bitstring).
event endResponder(host, host, bitstring, bitstring).

(*Injective Agreement*)
query U : host, V : host, S_U : bitstring, S_V : bitstring;
    event (endInitiator(U, V, S_U, S_V)) ==>
        event (startResponder(U, V, S_V)).
query U : host, V : host, S_U : bitstring, S_V : bitstring;
    event (endResponder(U, V, S_U, S_V)) ==>
        event (startInitiator(U, V, S_U)).
```

Figure 4.3: Injective Agreement

As a given host cannot know the session ID of the counterpart before the exchange of messages, it will only be possible to check one session ID per process. Figure 4.3 shows the code of the query: If `endResponder(..)` is called, the corresponding `startInitator(..)` must have been called by the counterpart. There can never exist a path where `endResponder` can be called without `startInitiator` has been called. The query is similar for the initiator.

Based on the model, the property of `Injective Agreement` holds for both the asymmetric and symmetric variant.

4.2.2 Privacy (Asymmetric only)

Privacy: The identity of a given host will first become evident when the identity of the counterpart has been verified.

Given the nature of this definition it is only possible to ensure privacy of

a single host in a given session. In EDHOC the privacy of the `initiator (U)` should remain private until it is certain of the identity of the `responder (V)`. Identity of a given host is defined as the ID_U and ID_V, which are IDs sent in the asymmetric variant. `Privacy` is modelled as a query, similarly to `Confidentiality`, checking if the attacker can obtain ID_U or ID_V (see Figure 4.4). ID_U and ID_V are created from the function `identifyPK`, which takes a private and public key and returns a value identifying a given process.

```
(*Privacy*)
query attacker(identifyPK(new skU, pk(new skU))));
      attacker(identifyPK(new skV, pk(new skV))).
```

Figure 4.4: Privacy

Based on the model the property of `Privacy` holds for the initiator for the asymmetric variant.

4.2.3 Confidentiality

Confidentiality: The encrypted data sent is unreadable to anyone but the honest participating parties.

`Confidentiality` is modelled by a check asking if an attacker can come into possession of data inside an encryption (see Figure 4.5). Based on the

```
(*Confidentiality*)
query attacker(new APP_2); attacker(new APP_3).
```

Figure 4.5: Confidentiality

model, the property of `Confidentiality` holds for the symmetric variant. In the asymmetric variant, it holds for APP_3 but for APP_2 it is critically flawed. This flaw will be described in Section 4.3.

4.2.4 Forward secrecy

Forward Secrecy: The encrypted data sent in past sessions remains unreadable to an attacker even if the long term keys are compromised.

`Forward Secrecy` is modelled with the usage of phases (see Figure 4.6). Both processes of the model start and complete the protocol, while an attacker observes all the messages sent. After the protocol has completed, the next phase of the protocol is started. In this phase the long term keys of the session are leaked. These consist of the private keys in the asymmetric variant and the pre-shared keys in the symmetric variant. The query to check `Forward Secrecy` is asking whether the attacker can obtain the same information from `Confidentiality` after the long term keys used have been leaked.

```
( !initiator(U, V, skU, pkU, pkV, pkIdV)
| !responder(V, U, skV, pkV, pkU, pkIdU)
| phase 1 ; out(c, skV) ; out(c, skU))
```

Figure 4.6: Perfect forward secrecy in asymmetric

Based on the model the property of `Forward Secrecy` is flawed for APP_2 in both the symmetric and the asymmetric variant. However, the flaw of the asymmetric variant is inherited from the property of `Confidentiality`. In both variants `Forward Secrecy` holds for APP_3.

4.3 Results

To summarize our findings, the properties of `Injective Agreement` and `Privacy (Asymmetric only)` hold. `Confidentiality` holds for the symmetric variant, however is critically flawed for the asymmetric variant. `Forward Secrecy` is flawed for both variants. Note that the flaw of the asymmetric

variant is inherited from the problems with `Confidentiality`. In Section 5 we will explain how the two variants can be exploited.

The flaw of `Confidentiality` in the asymmetric variant is severe. The documentation states that `message_2` contains encrypted application data, APP_2. The data of the encryption, however, cannot be considered confidential, as it is encrypted with a key that might be have been created in cooperation with an attacker. Consequently, the protocol can be exploited without leakage of long term keys or anything similar. This means that an attacker would be able to come into possession of - as well as to decrypt - `message_2` and thereby access APP_2. This is due to the fact that the party sending APP_2 has no way of verifying the identity of the counterpart at the time of sending the message. Thus, an attacker can initialize a session with an honest party and create the session key used for the encryption.

Looking at the flaw of `Forward Secrecy` in the symmetric variant, the problem is likewise related to APP_2. This application data cannot be considered secure, given the fact that an attacker can gain access to the application data of `message_2` if the long term keys are compromised. The attacker has to do an active attack before the keys are leaked in order to be able to access APP_2 - this will be described in Section 5. Only if the protocol has ended correctly, is it safe to assume that the data contained in APP_2 is not compromised. This, however, is problematic, as the party sending APP_2, has no way of verifying whom the data is sent to at the given time. Only when the protocol has come to an end, one can be sure of the identity of the counterpart and thereby the secrecy of the data.

The complete attack traces and results of all the queries in the models can be found in Appendix B.

Runtime

The runtime of ProVerif scales fast as the level of complexity in the model rises. Looking at Figure 4.1, it is clear that the time it takes ProVerif to generate the result of the EDHOC-models is insignificant. The execution of the TLS 1.3-model is significantly heavier with a runtime of 616 seconds[2]. The runtime is an indicator of the complexity of the model and thereby expandability of the model. If a model is very heavy, one might choose to model some aspects differently. As our model only takes ∼0.7 seconds to compute all results, it is quite expandable.

	Authentication	**Privacy**	**Confidentiality**	**Forward Secrecy**
Symmetric	0,148s	N/A	0,015s	0,191s
Asymmetric	0,107s	0,046s	0,085s	0,182s

Table 4.1: Runtime of ProVerif

[2]TLS 1.3 Repository: `https://github.com/inria-prosecco/reftls`

5 Vulnerabilities

As described in Section 4.2 there are flaws in two properties of EDHOC. For the symmetric variant `Forward Secrecy` is flawed. For the asymmetric variant `Confidentiality` is flawed, which means that `Forward Secrecy` is inherently flawed.

Exploitation of Asymmetric EDHOC

`Confidentiality` can only be ensured if the protocol has been completed. This means that at the time of transmitting the encrypted application data in `message_2`, `Confidentiality` can never be ensured. The attack is modelled in Figure 5.1

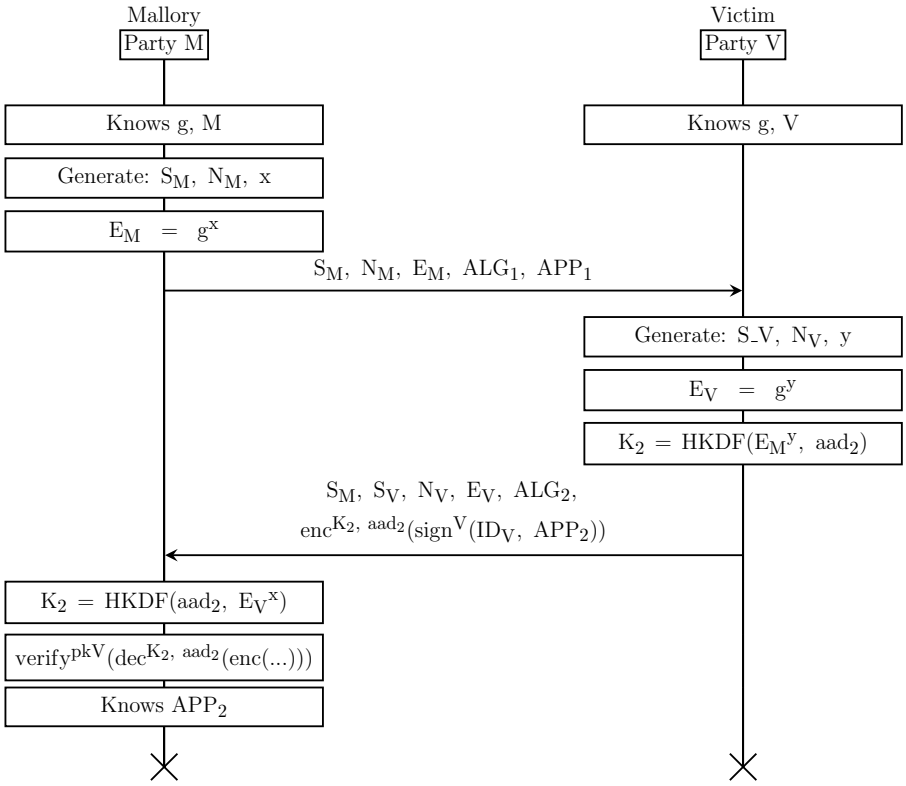


Figure 5.1: Attack on secrecy

The attack is carried out the following way. The attacker, `Mallory`, initiates the protocol by sending `message_1` with a his own key, E_M. At this point, `Victim` has no way of verifying the identity of `Mallory`. `Victim` creates a his own key, E_V. `message_2` is then encrypted with a key derived from the shared secret of E_M and E_V and aad_2. The message will be signed with the secret key of `Victim`, however as the public key is publicly available, this is not an issue. `Mallory` will now be able to decrypt the message and access the application data APP_2. He won't however be able to create a valid third message. The protocol will therefore be discontinued.

As the attacker does not depend upon compromise of long term keys in order to create this attack, the consequences of this attack is severe. The application data of the second message, APP_2, cannot be considered secure until the protocol has completed. As the application data can never be secured at the time of sending, it is impossible to send confidential data securely using APP_2.

The attack on Symmetric EDHOC

The attack on the symmetric variant has some prerequisites. First of all, this attack requires that long term keys are compromised. However, in order to be able to read the data, an active attack is required prior to the leakage. This means that this exploit requires more work beforehand compared to the exploit of the asymmetric variant. The following is a description of the attack on the symmetric variant of EDHOC. The attack is modelled in Figure 5.2.

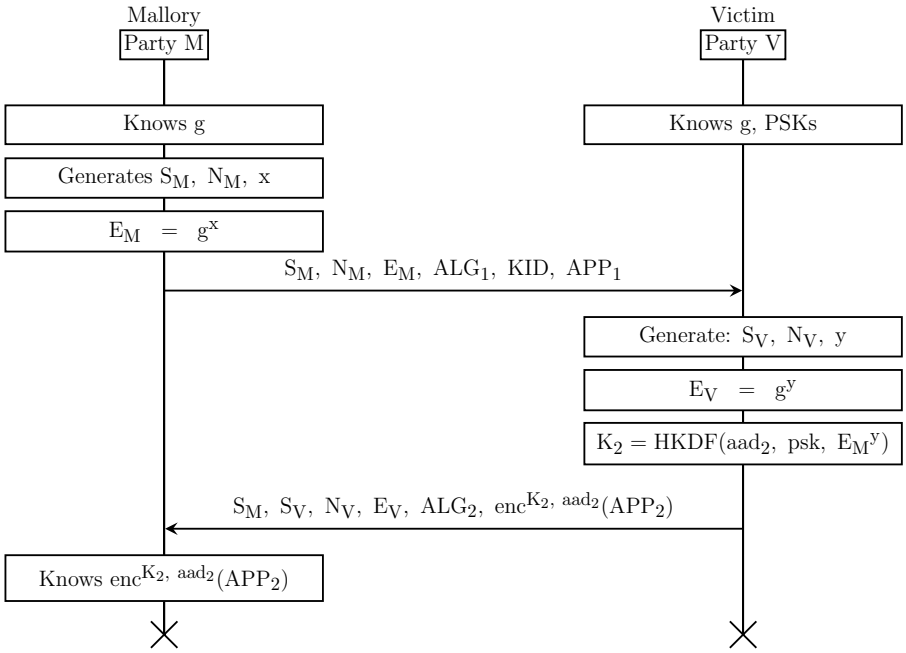


Figure 5.2: Exploitation of Symmetric EDHOC

The attacker, `Mallory`, initiates the protocol by sending `message_1` with his own key E_M. Similar to the case of the asymmetric variant, `Victim` has no way of identifying his counterpart. `Victim` will send back `message_2` containing data encrypted with a key derived from shared secret (E_M and E_V), `psk` and aad_2. The encrypted data of the message sent back will not be readable straight away because the attacker does not know `psk`. At this time `Mallory` is unable to create a valid third message and the session will terminate. If however the `psk` is compromised later, `Mallory` will be able to derive the key used for the encryption and thereby access APP_2.

Given this attack, it is clear that APP_2 should not be considered secure, as `Forward Secrecy` does not hold.

6 Improvements

As it is clear that the protocol has flaws, we propose changes in order to secure the protocol. The flaws described in section 5 enables an attacker to get a hold of encrypted application data. This means that APP_2 cannot be considered secure in all cases. The idea behind having APP_2 encrypted, is to be able to start processing data before the protocol has completed. However, as the data is first authenticated and secure after the protocol is completed, the data cannot securely be processed before.

The fact that APP_2 cannot be considered secure, before the protocol has completed, practically defeats the purpose of the encryption. There cannot be sent sensitive information in this package, as confidentiality cannot be guaranteed at the time of sending. As a result APP_2 poses a security risk without improving the protocol. Note that since the symmetric variant requires the leakage of the long term keys, it can be considered more secure than the asymmetric variant. However, it too cannot be considered secure.

As the encryption is flawed, the only use-case for APP_2 is as an unencrypted payload with the same properties as APP_1. Therefore, we suggest that APP_2 is moved out from the encryption, so that it too, is sent as plain text. Thus, in this way APP_2 will be flagged correctly regarding the properties that we can ensure.

We have modified the models in order to verify that this will indeed solve the

problems of `Confidentiality` as well as `Forward Secrecy`. The models, as well as output of the modified models, can be seen in Appendix C. As a result of the modification, ProVerif do no longer detect any attack traces. Thus, the modification effectively removes the vulnerabilities. Removing APP_2 from inside the encryption, EDHOC ensures all four security properties.

7 Conclusion

The successful modelling of EDHOC in ProVerif has been created and based upon the documentation and a given set of assumptions and simplifications. The model helped us discover two flaws of the protocol. Both these concerns the application data sent in the second message of both variants of the protocol.

In the asymmetric variant the flaw lies in the property of `Confidentiality` for APP_2. The property only holds on the condition that the protocol finishes. In other words, the property can never be insured at the time of sending the message. As a consequence, an attacker can initialize the protocol and obtain the encrypted information in `message_2`. The property of `Forward Secrecy` is inherently flawed.

In the symmetric variant the flaw lies in the property of `Forward Secrecy` for APP_2. The property cannot be considered secure, given the fact that an attacker can gain access to the application data of `message_2` if he has succeeded an active attack and the long term keys are compromised hereafter.

Given these flaws we recommend that the application data being sent in `message_2` is sent unencrypted with the same properties as the application data of `message_1`. This makes sure that a developer does not think APP_2 is secure.

To summarize our findings, the properties of `Injective Agreement` and `Privacy (Asymmetric only)` holds. `Confidentiality` holds for the symmetric variant, however is critically flawed for the asymmetric variant. `Forward Secrecy` is flawed for both variants.

Bibliography

[1] Bruno Blanchet et al. Proverif: Cryptographic protocol verifier in the formal model. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/. [Online; accessed 10 May. 2018].

[2] Francesca Palombini Göran Selander, John Mattsson. Ephemeral diffie-hellman over cose (edhoc). https://tools.ietf.org/html/draft-selander-ace-cose-ecdhe-08, March 05 2018. [Online; accessed 10 May. 2018].

[3] Pasi Eronen Hugo Krawczyk. Hmac-based extract-and-expand key derivation function (hkdf). https://tools.ietf.org/html/rfc5869, May 2010. [Online; accessed 10 May. 2018].

[4] Nadim Kobeissi Karthikeyan Bhargavan, Bruno Blanchet. Verified models and reference implementations for the tls 1.3 standard candidate. http://prosecco.gforge.inria.fr/personal/bblanche/publications/BhargavanBlanchetKobeissiSP2017.pdf, 2017. [Online; accessed 10 May. 2018].

[5] Gavin Lowe. A hierarchy of authentication specifications. https://pdfs.semanticscholar.org/db04/4d703941fefd5b4609034a3730aebf6ebcd1.pdf. [Online; accessed 10 May. 2018].

Appendices

A

Example of Model

The following is a modelling of the asymmetric variant querying for confidentiality. All models are available at Github[1]

```
free c: channel.

type host. (*U, V*)

type pkey. (*E_U, E_V*)
type skey. (*skU, skV*)
type derivedKey. (*K_i*)
type nonce. (*N_U & N_V*)
type G. (*Commonly known g*)
type exponent. (*x & y*)
type pkID.(*pkIdU & pkIdV*)

(*Message Types*)
free MSG_TYPE_1 : bitstring.
free MSG_TYPE_2 : bitstring.
free MSG_TYPE_3 : bitstring.

(*Creating public keys*)
fun pk(skey): pkey.

(*Shared Key Encryption*)
fun encrypt(bitstring, derivedKey, bitstring): bitstring.
reduc forall x: bitstring, y: derivedKey, aad: bitstring; decrypt(encrypt(x, y, aad), y,
     aad) = x.

(*Signing*)
fun sign(bitstring, skey): bitstring.
```

[1] https://github.com/theisgroenbech/edhoc-proverif

```
reduc forall x: bitstring, y: skey; verify(sign(x, y), pk(y)) = x.

(*Key derivation function*)
fun HKDF(G, bitstring): derivedKey.

(*Identifier function for the public keys*)
fun identifyPK(skey, pkey) : pkID.

(*Hash function*)
fun hash(bitstring): bitstring.

(*Diffie Hellman*)
(*From ProVerif examples - DiffieHellMan-active.pv*)
const g: G.
fun exp(G, exponent): G.
equation forall x: exponent, y: exponent; exp(exp(g, x), y) = exp(exp(g, y), x).

(*Confidentiality*)
query attacker(new APP_2);
      attacker(new APP_3).

(*Privacy*)
query attacker(identifyPK(new skV, pk(new skV)));
      attacker(identifyPK(new skU, pk(new skU))).

(*
Initiator process (U)
U & V = The two hosts in the protocol
skU = The long-term secret key of U
pkU & pkV = The long-term public keys of U and V
pkIdV = The identifier for the public key of V
*)
let initiator(U: host, V: host, skU : skey, pkU : pkey, pkV : pkey, pkIdV : pkID) =
  (*Generate exponent for g^x*)
  new x : exponent;
  (*Generate first part of the Diffie HellMan shared secret*)
  let E_U = exp(g, x) in
    (*Session identifier*)
    new S_U : bitstring;
    (*Random Nonce*)
    new N_U : nonce;
    (*Opaque application data*)
    new APP_1 : bitstring;
    (*Generate contents of message 1*)
    let message_1 : bitstring = (MSG_TYPE_1, S_U, N_U, E_U, APP_1) in
      (*Send message 1*)
```

```
      out(c, message_1);
      (*Recieve message 2*)
      in(c, message_2 : bitstring);
      (*Split message 2 into raw data and the encryption*)
      let (data_2 : bitstring, COSE_ENC_2 : bitstring) = message_2 in
        (*Get contents of the data*)
        let (=MSG_TYPE_2, =S_U, xS_V: bitstring, N_V: nonce, xE_V: G) = data_2 in
          (*Compute hash to message 2*)
          let aad_2 : bitstring = hash((message_1, data_2)) in
            (*Compute shared secret g^y^x*)
            let K : G = exp(xE_V, x) in
              (*Derive K_2 with the Diffie Hellman shared secret and the hash*)
              let K_2 : derivedKey = HKDF(K, aad_2) in
                (*Decrypt the encryption. Save and validate the contents*)
                let signature_2: bitstring = decrypt(COSE_ENC_2, K_2, aad_2) in
                  (*Open the signature and validate the contents*)
                  let (=pkIdV, =aad_2, APP_2 : bitstring) = verify(signature_2, pkV) in
                    (*Application data*)
                    new APP_3 : bitstring;
                    (*Generate data payload of message 3*)
                    let data_3 : bitstring = (MSG_TYPE_3, xS_V) in
                      (*Compute hash to message 3*)
                      let aad_3 : bitstring = hash((hash((message_1, message_2)), data_3)) in
                        (*Create the signature for message 3*)
                        let signature_3 : bitstring = sign((identifyPK(skU, pkU), aad_3, APP_3
                             ), skU) in
                          (*Derive K_3 with the Diffie Hellman shared secret and the hash*)
                          let K_3 : derivedKey = HKDF(K, aad_3) in
                            (*Encrypt the signature and the hash*)
                            let COSE_ENC_3 : bitstring = encrypt(signature_3, K_3, aad_3) in
                              (*Generate contents of message 3*)
                              let message_3 : bitstring = (data_3, COSE_ENC_3) in
                                (*Send message 3*)
                                out(c, message_3).

(*
Responder process (V)
V & U = The two hosts in the protocol
skV = The long-term secret key of V
pkV & pkU = The long-term public keys of V and U
pkIdU = The identifier for the public key of U
*)
let responder(V: host, U: host, skV : skey, pkV : pkey, pkU : pkey, pkIdU : pkID) =
  (*Generate exponent for g^y*)
  new y : exponent;
  (*Generate first part of the Diffie HellMan shared secret*)
```

```
let E_V : G = exp(g, y) in
  (*Session identifier*)
  new S_V : bitstring;
  (*Message 1*)
  in(c, message_1 : bitstring);
  (*Get contents of the message 1*)
  let (xMSG_TYPE_1 : bitstring, xS_U : bitstring, xN_U : nonce, xE_U: G, APP_1 : bitstring
      ) = message_1 in
    (*Random Nonce*)
    new N_V: nonce;
    (*Application data*)
    new APP_2 : bitstring;
    (*Generate data payload of message 2*)
    let data_2 : bitstring = (MSG_TYPE_2, xS_U, S_V, N_V, E_V) in
      (*Compute hash to message 2*)
      let aad_2 : bitstring = hash((message_1, data_2)) in
        (*Create the signature for message 2*)
        let signature_2 : bitstring = sign((identifyPK(skV, pkV), aad_2, APP_2), skV) in
          (*Compute shared secret g^x^y*)
          let K : G = exp(xE_U, y) in
            (*Derive K_2 with the Diffie Hellman shared secret and the hash*)
            let K_2 : derivedKey = HKDF(K, aad_2) in
              (*Encrypt the signature and the hash*)
              let COSE_ENC_2 : bitstring = encrypt(signature_2, K_2, aad_2) in
                (*Generate contents of message 2*)
                let message_2 : bitstring = (data_2, COSE_ENC_2) in
                  (*Send message 2*)
                  out(c, message_2);
                  (*Recieve message 3*)
                  in(c, message_3 : bitstring);
                  (*Split message 3 into raw data and the encryption*)
                  let (data_3 : bitstring, COSE_ENC_3 : bitstring) = message_3 in
                    (*Validate the contents of the data*)
                    let (=MSG_TYPE_3, =S_V) = data_3 in
                      (*Compute hash to message 3*)
                      let aad_3 : bitstring = hash((hash((message_1, message_2)), data_3))
                          in
                        (*Derive K_3 with the Diffie Hellman shared secret and the hash*)
                        let K_3 : derivedKey = HKDF(K, aad_3) in
                          (*Decrypt the encryption. Save and validate the contents*)
                          let signature_3 : bitstring = decrypt(COSE_ENC_3, K_3, aad_3) in
                            (*Open the signature and validate the contents*)
                            let (=pkIdU, =aad_3, APP_3 : bitstring) = verify(signature_3,
                                pkU) in
                              0.
```

```
process
  (*Creating hosts*)
  new U: host; new V: host;

  (*Generate public and private key*)
  new skU : skey; new skV : skey;
  let pkU = pk(skU) in let pkV = pk(skV) in

  (*ID of public keys*)
  let pkIdU = identifyPK(skU,pk(skU)) in let pkIdV = identifyPK(skV, pk(skV)) in

  (*Public keys*)
  out(c, pkU);
  out(c, pkV);

  ( !initiator(U, V, skU, pkU, pkV, pkIdV)
  | !responder(V, U, skV, pkV, pkU, pkIdU))
```

B
Results

Asymmetric injective agreement

```
Linear part:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Completing equations...
Completed equations:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new W: host;
{4}new skU: skey;
{5}new skV: skey;
{6}new skW: skey;
{7}let pkU: pkey = pk(skU) in
{8}let pkV: pkey = pk(skV) in
{9}let pkW: pkey = pk(skW) in
{10}let pkIdU: pkID = identifyPK(skU,pk(skU)) in
{11}let pkIdV: pkID = identifyPK(skV,pk(skV)) in
{12}let pkIdW: pkID = identifyPK(skW,pk(skW)) in
{13}out(c, pkU);
{14}out(c, pkV);
{15}out(c, pkW);
(
    {16}!
    {17}new x_63: exponent;
    {18}let E_U: G = exp(g,x_63) in
    {19}new S_U: bitstring;
    {20}event startInitiator(U,V,S_U);
    {21}new N_U: nonce;
```

```
    {22}new APP_64: bitstring;
    {23}let message_1_65: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,APP_64) in
    {24}out(c, message_1_65);
    {25}in(c, message_2_66: bitstring);
    {26}let (data_2_67: bitstring,COSE_ENC_2_68: bitstring) = message_2_66 in
    {27}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,N_V: nonce,xE_V: G) = data_2_67 in
    {28}let aad_2_69: bitstring = hash((message_1_65,data_2_67)) in
    {29}let K: G = exp(xE_V,x_63) in
    {30}let K_2_70: derivedKey = HKDF(K,aad_2_69) in
    {31}let signature_2_71: bitstring = decrypt(COSE_ENC_2_68,K_2_70,aad_2_69) in
    {32}let (=pkIdV,=aad_2_69,APP_2_72: bitstring) = verify(signature_2_71,pkV) in
    {33}new APP_73: bitstring;
    {34}let data_3_74: bitstring = (MSG_TYPE_3,xS_V) in
    {35}let aad_3_75: bitstring = hash((hash((message_1_65,message_2_66)),data_3_74)) in
    {36}let signature_3_76: bitstring = sign((identifyPK(skU,pkU),aad_3_75,APP_73),skU) in
    {37}let K_3_77: derivedKey = HKDF(K,aad_3_75) in
    {38}let COSE_ENC_3_78: bitstring = encrypt(signature_3_76,K_3_77,aad_3_75) in
    {39}let message_3_79: bitstring = (data_3_74,COSE_ENC_3_78) in
    {40}out(c, message_3_79);
    {41}event endInitiator(U,V,S_U,xS_V)
) | (
    {42}!
    {43}new y_80: exponent;
    {44}let E_V: G = exp(g,y_80) in
    {45}new S_V: bitstring;
    {46}event startResponder(U,V,S_V);
    {47}in(c, message_1_81: bitstring);
    {48}let (xMSG_TYPE_1_82: bitstring,xS_U: bitstring,xN_U: nonce,xE_U: G,APP_1_83:
        bitstring) = message_1_81 in
    {49}new N_V_84: nonce;
    {50}new APP_85: bitstring;
    {51}let data_2_86: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V_84,E_V) in
    {52}let aad_2_87: bitstring = hash((message_1_81,data_2_86)) in
    {53}let signature_2_88: bitstring = sign((identifyPK(skV,pkV),aad_2_87,APP_85),skV) in
    {54}let K_89: G = exp(xE_U,y_80) in
    {55}let K_2_90: derivedKey = HKDF(K_89,aad_2_87) in
    {56}let COSE_ENC_2_91: bitstring = encrypt(signature_2_88,K_2_90,aad_2_87) in
    {57}let message_2_92: bitstring = (data_2_86,COSE_ENC_2_91) in
    {58}out(c, message_2_92);
    {59}in(c, message_3_93: bitstring);
    {60}let (data_3_94: bitstring,COSE_ENC_3_95: bitstring) = message_3_93 in
    {61}let (=MSG_TYPE_3,=S_V) = data_3_94 in
    {62}let aad_3_96: bitstring = hash((hash((message_1_81,message_2_92)),data_3_94)) in
    {63}let K_3_97: derivedKey = HKDF(K_89,aad_3_96) in
    {64}let signature_3_98: bitstring = decrypt(COSE_ENC_3_95,K_3_97,aad_3_96) in
    {65}let (=pkIdU,=aad_3_96,APP_3_99: bitstring) = verify(signature_3_98,pkU) in
```

```
    {66}event endResponder(U,V,xS_U,S_V)
) | (
    {67}!
    {68}new x_100: exponent;
    {69}let E_U_101: G = exp(g,x_100) in
    {70}new S_U_102: bitstring;
    {71}event startInitiator(U,W,S_U_102);
    {72}new N_U_103: nonce;
    {73}new APP_104: bitstring;
    {74}let message_1_105: bitstring = (MSG_TYPE_1,S_U_102,N_U_103,E_U_101,APP_104) in
    {75}out(c, message_1_105);
    {76}in(c, message_2_106: bitstring);
    {77}let (data_2_107: bitstring,COSE_ENC_2_108: bitstring) = message_2_106 in
    {78}let (=MSG_TYPE_2,=S_U_102,xS_V_109: bitstring,N_V_110: nonce,xE_V_111: G) =
        data_2_107 in
    {79}let aad_2_112: bitstring = hash((message_1_105,data_2_107)) in
    {80}let K_113: G = exp(xE_V_111,x_100) in
    {81}let K_2_114: derivedKey = HKDF(K_113,aad_2_112) in
    {82}let signature_2_115: bitstring = decrypt(COSE_ENC_2_108,K_2_114,aad_2_112) in
    {83}let (=pkIdW,=aad_2_112,APP_2_116: bitstring) = verify(signature_2_115,pkW) in
    {84}new APP_117: bitstring;
    {85}let data_3_118: bitstring = (MSG_TYPE_3,xS_V_109) in
    {86}let aad_3_119: bitstring = hash((hash((message_1_105,message_2_106)),data_3_118)) in
    {87}let signature_3_120: bitstring = sign((identifyPK(skU,pkU),aad_3_119,APP_117),skU)
        in
    {88}let K_3_121: derivedKey = HKDF(K_113,aad_3_119) in
    {89}let COSE_ENC_3_122: bitstring = encrypt(signature_3_120,K_3_121,aad_3_119) in
    {90}let message_3_123: bitstring = (data_3_118,COSE_ENC_3_122) in
    {91}out(c, message_3_123);
    {92}event endInitiator(U,W,S_U_102,xS_V_109)
) | (
    {93}!
    {94}new y_124: exponent;
    {95}let E_V_125: G = exp(g,y_124) in
    {96}new S_V_126: bitstring;
    {97}event startResponder(U,W,S_V_126);
    {98}in(c, message_1_127: bitstring);
    {99}let (xMSG_TYPE_1_128: bitstring,xS_U_129: bitstring,xN_U_130: nonce,xE_U_131: G,
        APP_1_132: bitstring) = message_1_127 in
    {100}new N_V_133: nonce;
    {101}new APP_134: bitstring;
    {102}let data_2_135: bitstring = (MSG_TYPE_2,xS_U_129,S_V_126,N_V_133,E_V_125) in
    {103}let aad_2_136: bitstring = hash((message_1_127,data_2_135)) in
    {104}let signature_2_137: bitstring = sign((identifyPK(skW,pkW),aad_2_136,APP_134),skW)
        in
    {105}let K_138: G = exp(xE_U_131,y_124) in
```

```
    {106}let K_2_139: derivedKey = HKDF(K_138,aad_2_136) in
    {107}let COSE_ENC_2_140: bitstring = encrypt(signature_2_137,K_2_139,aad_2_136) in
    {108}let message_2_141: bitstring = (data_2_135,COSE_ENC_2_140) in
    {109}out(c, message_2_141);
    {110}in(c, message_3_142: bitstring);
    {111}let (data_3_143: bitstring,COSE_ENC_3_144: bitstring) = message_3_142 in
    {112}let (=MSG_TYPE_3,=S_V_126) = data_3_143 in
    {113}let aad_3_145: bitstring = hash((hash((message_1_127,message_2_141)),data_3_143))
         in
    {114}let K_3_146: derivedKey = HKDF(K_138,aad_3_145) in
    {115}let signature_3_147: bitstring = decrypt(COSE_ENC_3_144,K_3_146,aad_3_145) in
    {116}let (=pkIdU,=aad_3_145,APP_3_148: bitstring) = verify(signature_3_147,pkU) in
    {117}event endResponder(U,W,xS_U_129,S_V_126)
)

-- Query event(endResponder(U_149,V_150,S_U_151,S_V_152)) ==> event(startInitiator(U_149,
     V_150,S_U_151))
Completing...
200 rules inserted. The rule base contains 156 rules. 45 rules in the queue.
400 rules inserted. The rule base contains 218 rules. 14 rules in the queue.
Starting query event(endResponder(U_149,V_150,S_U_151,S_V_152)) ==> event(startInitiator(
     U_149,V_150,S_U_151))
goal reachable: begin(startInitiator(U[],W[],S_U_102[!1 = @sid_7500])) -> end(endResponder(
     U[],W[],S_U_102[!1 = @sid_7500],S_V_126[!1 = @sid_7501]))
goal reachable: begin(startInitiator(U[],V[],S_U[!1 = @sid_7504])) -> end(endResponder(U[],
     V[],S_U[!1 = @sid_7504],S_V[!1 = @sid_7505]))
RESULT event(endResponder(U_149,V_150,S_U_151,S_V_152)) ==> event(startInitiator(U_149,
     V_150,S_U_151)) is true.
-- Query event(endInitiator(U_7512,V_7513,S_U_7514,S_V_7515)) ==> event(startResponder(
     U_7512,V_7513,S_V_7515))
Completing...
200 rules inserted. The rule base contains 146 rules. 36 rules in the queue.
400 rules inserted. The rule base contains 230 rules. 8 rules in the queue.
Starting query event(endInitiator(U_7512,V_7513,S_U_7514,S_V_7515)) ==> event(
     startResponder(U_7512,V_7513,S_V_7515))
goal reachable: begin(startResponder(U[],W[],S_V_126[!1 = @sid_14476])) -> end(endInitiator
     (U[],W[],S_U_102[!1 = @sid_14477],S_V_126[!1 = @sid_14476]))
goal reachable: begin(startResponder(U[],V[],S_V[!1 = @sid_14480])) -> end(endInitiator(U
     [],V[],S_U[!1 = @sid_14481],S_V[!1 = @sid_14480]))
RESULT event(endInitiator(U_7512,V_7513,S_U_7514,S_V_7515)) ==> event(startResponder(U_7512
     ,V_7513,S_V_7515)) is true.
```

Asymmetric confidentiality

```
Linear part:
```

```
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Completing equations...
Completed equations:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new skU: skey;
{4}new skV: skey;
{5}let pkU: pkey = pk(skU) in
{6}let pkV: pkey = pk(skV) in
{7}let pkIdU: pkID = identifyPK(skU,pk(skU)) in
{8}let pkIdV: pkID = identifyPK(skV,pk(skV)) in
{9}out(c, pkU);
{10}out(c, pkV);
(
    {11}!
    {12}new x_63: exponent;
    {13}let E_U: G = exp(g,x_63) in
    {14}new S_U: bitstring;
    {15}new N_U: nonce;
    {16}new APP_64: bitstring;
    {17}let message_1_65: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,APP_64) in
    {18}out(c, message_1_65);
    {19}in(c, message_2_66: bitstring);
    {20}let (data_2_67: bitstring,COSE_ENC_2_68: bitstring) = message_2_66 in
    {21}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,N_V: nonce,xE_V: G) = data_2_67 in
    {22}let aad_2_69: bitstring = hash((message_1_65,data_2_67)) in
    {23}let K: G = exp(xE_V,x_63) in
    {24}let K_2_70: derivedKey = HKDF(K,aad_2_69) in
    {25}let signature_2_71: bitstring = decrypt(COSE_ENC_2_68,K_2_70,aad_2_69) in
    {26}let (=pkIdV,=aad_2_69,APP_2_72: bitstring) = verify(signature_2_71,pkV) in
    {27}new APP_73: bitstring;
    {28}let data_3_74: bitstring = (MSG_TYPE_3,xS_V) in
    {29}let aad_3_75: bitstring = hash((hash((message_1_65,message_2_66)),data_3_74)) in
    {30}let signature_3_76: bitstring = sign((identifyPK(skU,pkU),aad_3_75,APP_73),skU) in
    {31}let K_3_77: derivedKey = HKDF(K,aad_3_75) in
    {32}let COSE_ENC_3_78: bitstring = encrypt(signature_3_76,K_3_77,aad_3_75) in
    {33}let message_3_79: bitstring = (data_3_74,COSE_ENC_3_78) in
    {34}out(c, message_3_79)
) | (
    {35}!
    {36}new y_80: exponent;
```

```
    {37}let E_V: G = exp(g,y_80) in
    {38}new S_V: bitstring;
    {39}in(c, message_1_81: bitstring);
    {40}let (xMSG_TYPE_1_82: bitstring,xS_U: bitstring,xN_U: nonce,xE_U: G,APP_1_83:
        bitstring) = message_1_81 in
    {41}new N_V_84: nonce;
    {42}new APP_85: bitstring;
    {43}let data_2_86: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V_84,E_V) in
    {44}let aad_2_87: bitstring = hash((message_1_81,data_2_86)) in
    {45}let signature_2_88: bitstring = sign((identifyPK(skV,pkV),aad_2_87,APP_85),skV) in
    {46}let K_89: G = exp(xE_U,y_80) in
    {47}let K_2_90: derivedKey = HKDF(K_89,aad_2_87) in
    {48}let COSE_ENC_2_91: bitstring = encrypt(signature_2_88,K_2_90,aad_2_87) in
    {49}let message_2_92: bitstring = (data_2_86,COSE_ENC_2_91) in
    {50}out(c, message_2_92);
    {51}in(c, message_3_93: bitstring);
    {52}let (data_3_94: bitstring,COSE_ENC_3_95: bitstring) = message_3_93 in
    {53}let (=MSG_TYPE_3,=S_V) = data_3_94 in
    {54}let aad_3_96: bitstring = hash((hash((message_1_81,message_2_92)),data_3_94)) in
    {55}let K_3_97: derivedKey = HKDF(K_89,aad_3_96) in
    {56}let signature_3_98: bitstring = decrypt(COSE_ENC_3_95,K_3_97,aad_3_96) in
    {57}let (=pkIdU,=aad_3_96,APP_3_99: bitstring) = verify(signature_3_98,pkU) in
    0
)

-- Query not attacker(identifyPK(skV[],pk(skV[]))); not attacker(identifyPK(skU[],pk(skU[])
    ))
Completing...
Starting query not attacker(identifyPK(skV[],pk(skV[])))
goal reachable: attacker(identifyPK(skV[],pk(skV[])))
Abbreviations:
S = S_V[!1 = @sid_2516]
N_V_2527 = N_V_84[message_1_81 = (xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),!1 =
    @sid_2516]
y_2528 = y_80[!1 = @sid_2516]
APP = APP_85[message_1_81 = (xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),!1 =
    @sid_2516]
N_V_2529 = N_V_84[message_1_81 = (xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428
    ),!1 = @sid_2516]
APP_2530 = APP_85[message_1_81 = (xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428
    ),!1 = @sid_2516]
N_V_2531 = N_V_84[message_1_81 = (xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482
    ),!1 = @sid_2516]
APP_2532 = APP_85[message_1_81 = (xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482
    ),!1 = @sid_2516]
```

```
1. The message pk(skV[]) may be sent to the attacker at output {10}.
attacker(pk(skV[])).

2. The attacker has some term APP_1_2482.
attacker(APP_1_2482).

3. The attacker has some term xE_U_2481.
attacker(xE_U_2481).

4. The attacker has some term xN_U_2480.
attacker(xN_U_2480).

5. The attacker has some term xS_U_2479.
attacker(xS_U_2479).

6. The attacker has some term xMSG_TYPE_1_2478.
attacker(xMSG_TYPE_1_2478).

7. By 6, the attacker may know xMSG_TYPE_1_2478.
By 5, the attacker may know xS_U_2479.
By 4, the attacker may know xN_U_2480.
By 3, the attacker may know xE_U_2481.
By 2, the attacker may know APP_1_2482.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,
     xE_U_2481,APP_1_2482).
attacker((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482)).

8. The message (xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482) that the
     attacker may have by 7 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)),encrypt(sign((identifyPK(
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482),(
     MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)))),APP_2532),skV[]),HKDF(exp(xE_U_2481,
     y_2528),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482),(MSG_TYPE_2
     [],xS_U_2479,S,N_V_2531,exp(g,y_2528)))),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,
     xE_U_2481,APP_1_2482),(MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)))))) may be sent
      to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)),encrypt(sign((identifyPK(skV[],
     pk(skV[])),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482),(
     MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)))),APP_2532),skV[]),HKDF(exp(xE_U_2481,
     y_2528),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482),(MSG_TYPE_2
     [],xS_U_2479,S,N_V_2531,exp(g,y_2528)))),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,
     xE_U_2481,APP_1_2482),(MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)))))).

9. By 8, the attacker may know ((MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)),encrypt(
     sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481
     ,APP_1_2482),(MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)))),APP_2532),skV[]),HKDF(
```

```
     exp(xE_U_2481,y_2528),hash(((xMSG_TYPE_1_2478,xS_U_2479,xN_U_2480,xE_U_2481,APP_1_2482)
     ,(MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)))))),hash(((xMSG_TYPE_1_2478,xS_U_2479
     ,xN_U_2480,xE_U_2481,APP_1_2482),(MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528))))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_2479,S,
     N_V_2531,exp(g,y_2528)).
attacker((MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528))).

10. By 9, the attacker may know (MSG_TYPE_2[],xS_U_2479,S,N_V_2531,exp(g,y_2528)).
Using the function 5-proj-5-tuple the attacker may obtain exp(g,y_2528).
attacker(exp(g,y_2528)).

11. The attacker has some term APP_1_2515.
attacker(APP_1_2515).

12. Using the function g the attacker may obtain g.
attacker(g).

13. The attacker has some term xN_U_2513.
attacker(xN_U_2513).

14. The attacker has some term xS_U_2512.
attacker(xS_U_2512).

15. The attacker has some term xMSG_TYPE_1_2511.
attacker(xMSG_TYPE_1_2511).

16. By 15, the attacker may know xMSG_TYPE_1_2511.
By 14, the attacker may know xS_U_2512.
By 13, the attacker may know xN_U_2513.
By 12, the attacker may know g.
By 11, the attacker may know APP_1_2515.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,
     APP_1_2515).
attacker((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515)).

17. The message (xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515) that the attacker may
     have by 16 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)),encrypt(sign((identifyPK(
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2
     [],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP),skV[]),HKDF(exp(g,y_2528),hash(((
     xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,
     exp(g,y_2528))))),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2
     [],xS_U_2512,S,N_V_2527,exp(g,y_2528)))))) may be sent to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)),encrypt(sign((identifyPK(skV[],
     pk(skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],
     xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP),skV[]),HKDF(exp(g,y_2528),hash(((
```

```
     xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,
     exp(g,y_2528))))),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2
     [],xS_U_2512,S,N_V_2527,exp(g,y_2528))))))).

18. By 17, the attacker may know ((MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)),encrypt
     (sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,
     APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP),skV[]),HKDF(exp(g,
     y_2528),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],
     xS_U_2512,S,N_V_2527,exp(g,y_2528))))),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,
     APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_2512,S,
     N_V_2527,exp(g,y_2528)).
attacker((MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))).

19. By 18, the attacker may know (MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)).
Using the function 4-proj-5-tuple the attacker may obtain N_V_2527.
attacker(N_V_2527).

20. The attacker has some term APP_1_2428.
attacker(APP_1_2428).

21. The attacker has some term xE_U_2427.
attacker(xE_U_2427).

22. The attacker has some term xN_U_2426.
attacker(xN_U_2426).

23. The attacker has some term xS_U_2425.
attacker(xS_U_2425).

24. The attacker has some term xMSG_TYPE_1_2424.
attacker(xMSG_TYPE_1_2424).

25. By 24, the attacker may know xMSG_TYPE_1_2424.
By 23, the attacker may know xS_U_2425.
By 22, the attacker may know xN_U_2426.
By 21, the attacker may know xE_U_2427.
By 20, the attacker may know APP_1_2428.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,
     xE_U_2427,APP_1_2428).
attacker((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428)).

26. The message (xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428) that the
     attacker may have by 25 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)),encrypt(sign((identifyPK(
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428),(
```

```
     MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)))),APP_2530),skV[]),HKDF(exp(xE_U_2427,
     y_2528),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428),(MSG_TYPE_2
     [],xS_U_2425,S,N_V_2529,exp(g,y_2528)))),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,
     xE_U_2427,APP_1_2428),(MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)))))) may be sent
      to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)),encrypt(sign((identifyPK(skV[],
     pk(skV[])),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428),(
     MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)))),APP_2530),skV[]),HKDF(exp(xE_U_2427,
     y_2528),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428),(MSG_TYPE_2
     [],xS_U_2425,S,N_V_2529,exp(g,y_2528)))),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,
     xE_U_2427,APP_1_2428),(MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)))))))).

27. By 26, the attacker may know ((MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)),encrypt
     (sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,
     xE_U_2427,APP_1_2428),(MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)))),APP_2530),skV
     []),HKDF(exp(xE_U_2427,y_2528),hash(((xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,
     APP_1_2428),(MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)))),hash(((
     xMSG_TYPE_1_2424,xS_U_2425,xN_U_2426,xE_U_2427,APP_1_2428),(MSG_TYPE_2[],xS_U_2425,S,
     N_V_2529,exp(g,y_2528)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_2425,S,
     N_V_2529,exp(g,y_2528)).
attacker((MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528))).

28. By 27, the attacker may know (MSG_TYPE_2[],xS_U_2425,S,N_V_2529,exp(g,y_2528)).
Using the function 3-proj-5-tuple the attacker may obtain S.
attacker(S).

29. The attacker initially knows MSG_TYPE_2[].
attacker(MSG_TYPE_2[]).

30. By 29, the attacker may know MSG_TYPE_2[].
By 14, the attacker may know xS_U_2512.
By 28, the attacker may know S.
By 19, the attacker may know N_V_2527.
By 10, the attacker may know exp(g,y_2528).
Using the function 5-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g
     ,y_2528)).
attacker((MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))).

31. By 16, the attacker may know (xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515).
By 30, the attacker may know (MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)).
Using the function 2-tuple the attacker may obtain ((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g
     ,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))).
attacker(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,
     N_V_2527,exp(g,y_2528)))).
```

```
32. By 31, the attacker may know ((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(
     MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))).
Using the function hash the attacker may obtain hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513
     ,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))).
attacker(hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,
     S,N_V_2527,exp(g,y_2528))))).

33. By 10, the attacker may know exp(g,y_2528).
By 32, the attacker may know hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(
     MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))).
Using the function HKDF the attacker may obtain HKDF(exp(g,y_2528),hash(((xMSG_TYPE_1_2511,
     xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))))).
attacker(HKDF(exp(g,y_2528),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(
     MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))))).

34. By 17, the attacker may know ((MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)),encrypt
     (sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,
     APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP),skV[]),HKDF(exp(g,
     y_2528),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],
     xS_U_2512,S,N_V_2527,exp(g,y_2528))))),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,
     APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))))).
Using the function 2-proj-2-tuple the attacker may obtain encrypt(sign((identifyPK(skV[],pk
     (skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],
     xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP),skV[]),HKDF(exp(g,y_2528),hash(((
     xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,
     exp(g,y_2528))))),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2
     [],xS_U_2512,S,N_V_2527,exp(g,y_2528))))).
attacker(encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,
     xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP),skV[])
     ,HKDF(exp(g,y_2528),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(
     MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))))),hash(((xMSG_TYPE_1_2511,xS_U_2512,
     xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))))).

35. By 34, the attacker may know encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((
     xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,
     exp(g,y_2528)))),APP),skV[]),HKDF(exp(g,y_2528),hash(((xMSG_TYPE_1_2511,xS_U_2512,
     xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))))),hash(((
     xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,
     exp(g,y_2528))))).
By 33, the attacker may know HKDF(exp(g,y_2528),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513
     ,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528))))).
By 32, the attacker may know hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(
     MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))).
Using the function decrypt the attacker may obtain sign((identifyPK(skV[],pk(skV[])),hash
     (((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,
     N_V_2527,exp(g,y_2528)))),APP),skV[]).
```

```
attacker(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,
     APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP),skV[])).

36. By 35, the attacker may know sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,
     xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),
     APP),skV[]).
By 1, the attacker may know pk(skV[]).
Using the function verify the attacker may obtain (identifyPK(skV[],pk(skV[])),hash(((
     xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,
     exp(g,y_2528)))),APP).
attacker((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,xS_U_2512,xN_U_2513,g,
     APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),APP)).

37. By 36, the attacker may know (identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_2511,
     xS_U_2512,xN_U_2513,g,APP_1_2515),(MSG_TYPE_2[],xS_U_2512,S,N_V_2527,exp(g,y_2528)))),
     APP).
Using the function 1-proj-3-tuple the attacker may obtain identifyPK(skV[],pk(skV[])).
attacker(identifyPK(skV[],pk(skV[]))).


Unified xMSG_TYPE_1_2511 with xMSG_TYPE_1_2424
Unified xS_U_2512 with xS_U_2425
Unified xN_U_2513 with xN_U_2426
Unified xE_U_2427 with g
Unified APP_1_2515 with APP_1_2428
Unified xMSG_TYPE_1_2478 with xMSG_TYPE_1_2424
Unified xS_U_2479 with xS_U_2425
Unified xN_U_2480 with xN_U_2426
Unified xE_U_2481 with g
Unified APP_1_2482 with APP_1_2428
Iterating unifyDerivation.
Fixpoint reached: nothing more to unify.
The clause after unifyDerivation is
attacker(identifyPK(skV[],pk(skV[])))
This clause still contradicts the query.
A more detailed output of the traces is available with
  set traceDisplay = long.

new U: host creating U_2772 at {1}

new V: host creating V_2773 at {2}

new skU: skey creating skU_2774 at {3}

new skV: skey creating skV_2695 at {4}
```

```
out(c, ~M_2941) with ~M_2941 = pk(skU_2774) at {9}

out(c, ~M_2943) with ~M_2943 = pk(skV_2695) at {10}

new y_80: exponent creating y_2698 at {36} in copy a_2694

new S_V: bitstring creating S_V_2696 at {38} in copy a_2694

in(c, (a_2690,a_2691,a_2692,g,a_2693)) at {39} in copy a_2694

new N_V_84: nonce creating N_V_2697 at {41} in copy a_2694

new APP_85: bitstring creating APP_2699 at {42} in copy a_2694

out(c, ((~M_3106,~M_3107,~M_3108,~M_3109,~M_3110),~M_3105)) with ~M_3106 = MSG_TYPE_2, ~
     M_3107 = a_2691, ~M_3108 = S_V_2696, ~M_3109 = N_V_2697, ~M_3110 = exp(g,y_2698), ~
     M_3105 = encrypt(sign((identifyPK(skV_2695,pk(skV_2695)),hash(((a_2690,a_2691,a_2692,g,
     a_2693),(MSG_TYPE_2,a_2691,S_V_2696,N_V_2697,exp(g,y_2698)))),APP_2699),skV_2695),HKDF(
     exp(g,y_2698),hash(((a_2690,a_2691,a_2692,g,a_2693),(MSG_TYPE_2,a_2691,S_V_2696,
     N_V_2697,exp(g,y_2698)))),hash(((a_2690,a_2691,a_2692,g,a_2693),(MSG_TYPE_2,a_2691,
     S_V_2696,N_V_2697,exp(g,y_2698)))) at {50} in copy a_2694

The attacker has the message 1-proj-3-tuple(verify(decrypt(~M_3105,HKDF(~M_3110,hash(((
     a_2690,a_2691,a_2692,g,a_2693),(MSG_TYPE_2,a_2691,~M_3108,~M_3109,~M_3110)))),hash(((
     a_2690,a_2691,a_2692,g,a_2693),(MSG_TYPE_2,a_2691,~M_3108,~M_3109,~M_3110)))),~M_2943))
      = identifyPK(skV_2695,pk(skV_2695)).
A trace has been found.
The previous trace falsifies the query, because the query is
simple and the trace corresponds to the derivation.
RESULT not attacker(identifyPK(skV[],pk(skV[]))) is false.
Starting query not attacker(identifyPK(skU[],pk(skU[])))
RESULT not attacker(identifyPK(skU[],pk(skU[]))) is true.
-- Query not attacker(APP_85[message_1_81 = v_3113,!1 = v_3114]); not attacker(APP_73[
     message_2_66 = v_3111,!1 = v_3112])
Completing...
Starting query not attacker(APP_85[message_1_81 = v_3113,!1 = v_3114])
goal reachable: attacker(xMSG_TYPE_1_5337) && attacker(xS_U_5338) && attacker(xN_U_5339) &&
      attacker(y_5340) && attacker(APP_1_5341) -> attacker(APP_85[message_1_81 = (
     xMSG_TYPE_1_5337,xS_U_5338,xN_U_5339,exp(g,y_5340),APP_1_5341),!1 = @sid_5342])
Abbreviations:
APP_5464 = APP_85[message_1_81 = (xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),
     APP_1_5450),!1 = @sid_5451]
S_5465 = S_V[!1 = @sid_5451]
N_V_5466 = N_V_84[message_1_81 = (xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),
     APP_1_5450),!1 = @sid_5451]
y_5467 = y_80[!1 = @sid_5451]
```

```
N_V_5468 = N_V_84[message_1_81 = (xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359
     ),!1 = @sid_5451]
APP_5469 = APP_85[message_1_81 = (xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359
     ),!1 = @sid_5451]
N_V_5470 = N_V_84[message_1_81 = (xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417
     ),!1 = @sid_5451]
APP_5471 = APP_85[message_1_81 = (xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417
     ),!1 = @sid_5451]

1. The message pk(skV[]) may be sent to the attacker at output {10}.
attacker(pk(skV[])).

2. The attacker has some term APP_1_5417.
attacker(APP_1_5417).

3. The attacker has some term xE_U_5416.
attacker(xE_U_5416).

4. The attacker has some term xN_U_5415.
attacker(xN_U_5415).

5. The attacker has some term xS_U_5414.
attacker(xS_U_5414).

6. The attacker has some term xMSG_TYPE_1_5413.
attacker(xMSG_TYPE_1_5413).

7. By 6, the attacker may know xMSG_TYPE_1_5413.
By 5, the attacker may know xS_U_5414.
By 4, the attacker may know xN_U_5415.
By 3, the attacker may know xE_U_5416.
By 2, the attacker may know APP_1_5417.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,
     xE_U_5416,APP_1_5417).
attacker((xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417)).

8. The message (xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417) that the
     attacker may have by 7 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)),encrypt(sign((
     identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,
     APP_1_5417),(MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)))),APP_5471),skV[]),
     HKDF(exp(xE_U_5416,y_5467),hash(((xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,
     APP_1_5417),(MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467))))),hash(((
     xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417),(MSG_TYPE_2[],xS_U_5414,
     S_5465,N_V_5470,exp(g,y_5467)))))) may be sent to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)),encrypt(sign((identifyPK(
```

```
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417),(
     MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)))),APP_5471),skV[]),HKDF(exp(
     xE_U_5416,y_5467),hash(((xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417),(
     MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)))),hash(((xMSG_TYPE_1_5413,
     xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417),(MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g
     ,y_5467))))))).

9. By 8, the attacker may know ((MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)),
     encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,
     xE_U_5416,APP_1_5417),(MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)))),APP_5471
     ),skV[]),HKDF(exp(xE_U_5416,y_5467),hash(((xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,
     xE_U_5416,APP_1_5417),(MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)))),hash(((
     xMSG_TYPE_1_5413,xS_U_5414,xN_U_5415,xE_U_5416,APP_1_5417),(MSG_TYPE_2[],xS_U_5414,
     S_5465,N_V_5470,exp(g,y_5467))))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5414,S_5465,
     N_V_5470,exp(g,y_5467)).
attacker((MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467))).

10. By 9, the attacker may know (MSG_TYPE_2[],xS_U_5414,S_5465,N_V_5470,exp(g,y_5467)).
Using the function 5-proj-5-tuple the attacker may obtain exp(g,y_5467).
attacker(exp(g,y_5467)).

11. We assume as hypothesis that
attacker(APP_1_5450).

12. We assume as hypothesis that
attacker(y_5407).

13. Using the function g the attacker may obtain g.
attacker(g).

14. By 13, the attacker may know g.
By 12, the attacker may know y_5407.
Using the function exp the attacker may obtain exp(g,y_5407).
attacker(exp(g,y_5407)).

15. We assume as hypothesis that
attacker(xN_U_5448).

16. We assume as hypothesis that
attacker(xS_U_5447).

17. We assume as hypothesis that
attacker(xMSG_TYPE_1_5446).

18. By 17, the attacker may know xMSG_TYPE_1_5446.
```

```
By 16, the attacker may know xS_U_5447.
By 15, the attacker may know xN_U_5448.
By 14, the attacker may know exp(g,y_5407).
By 11, the attacker may know APP_1_5450.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,
     exp(g,y_5407),APP_1_5450).
attacker((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450)).

19. The message (xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450) that the
     attacker may have by 18 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)),encrypt(sign((
     identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),
     APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))),APP_5464),skV[]),
     HKDF(exp(exp(g,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407
     ),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))),hash(((
     xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,
     S_5465,N_V_5466,exp(g,y_5467)))))) may be sent to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)),encrypt(sign((identifyPK(
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450)
     ,(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))),APP_5464),skV[]),HKDF(exp(exp
     (g,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450
     ),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))),hash(((xMSG_TYPE_1_5446,
     xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,
     exp(g,y_5467))))))).

20. By 19, the attacker may know ((MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)),
     encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,
     exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))),
     APP_5464),skV[]),HKDF(exp(exp(g,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,xS_U_5447,
     xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,
     y_5467))))),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(
     MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5447,S_5465,
     N_V_5466,exp(g,y_5467)).
attacker((MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))).

21. By 20, the attacker may know (MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)).
Using the function 4-proj-5-tuple the attacker may obtain N_V_5466.
attacker(N_V_5466).

22. The attacker has some term APP_1_5359.
attacker(APP_1_5359).

23. The attacker has some term xE_U_5358.
attacker(xE_U_5358).
```

```
24. The attacker has some term xN_U_5357.
attacker(xN_U_5357).

25. The attacker has some term xS_U_5356.
attacker(xS_U_5356).

26. The attacker has some term xMSG_TYPE_1_5355.
attacker(xMSG_TYPE_1_5355).

27. By 26, the attacker may know xMSG_TYPE_1_5355.
By 25, the attacker may know xS_U_5356.
By 24, the attacker may know xN_U_5357.
By 23, the attacker may know xE_U_5358.
By 22, the attacker may know APP_1_5359.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,
    xE_U_5358,APP_1_5359).
attacker((xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359)).

28. The message (xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359) that the
    attacker may have by 27 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467)),encrypt(sign((
    identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,
    APP_1_5359),(MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467)))),APP_5469),skV[]),
    HKDF(exp(xE_U_5358,y_5467),hash(((xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,
    APP_1_5359),(MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467))))),hash(((
    xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359),(MSG_TYPE_2[],xS_U_5356,
    S_5465,N_V_5468,exp(g,y_5467)))))) may be sent to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467)),encrypt(sign((identifyPK(
    skV[],pk(skV[])),hash(((xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359),(
    MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467)))),APP_5469),skV[]),HKDF(exp(
    xE_U_5358,y_5467),hash(((xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359),(
    MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467))))),hash(((xMSG_TYPE_1_5355,
    xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359),(MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g
    ,y_5467)))))))).

29. By 28, the attacker may know ((MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467)),
    encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,
    xE_U_5358,APP_1_5359),(MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467)))),APP_5469
    ),skV[]),HKDF(exp(xE_U_5358,y_5467),hash(((xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,
    xE_U_5358,APP_1_5359),(MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467))))),hash(((
    xMSG_TYPE_1_5355,xS_U_5356,xN_U_5357,xE_U_5358,APP_1_5359),(MSG_TYPE_2[],xS_U_5356,
    S_5465,N_V_5468,exp(g,y_5467))))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5356,S_5465,
    N_V_5468,exp(g,y_5467)).
attacker((MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467))).
```

```
30. By 29, the attacker may know (MSG_TYPE_2[],xS_U_5356,S_5465,N_V_5468,exp(g,y_5467)).
Using the function 3-proj-5-tuple the attacker may obtain S_5465.
attacker(S_5465).

31. The attacker initially knows MSG_TYPE_2[].
attacker(MSG_TYPE_2[]).

32. By 31, the attacker may know MSG_TYPE_2[].
By 16, the attacker may know xS_U_5447.
By 30, the attacker may know S_5465.
By 21, the attacker may know N_V_5466.
By 10, the attacker may know exp(g,y_5467).
Using the function 5-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,
     exp(g,y_5467)).
attacker((MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))).

33. By 18, the attacker may know (xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),
     APP_1_5450).
By 32, the attacker may know (MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)).
Using the function 2-tuple the attacker may obtain ((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,
     exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))).
attacker(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],
     xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))).

34. By 33, the attacker may know ((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),
     APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))).
Using the function hash the attacker may obtain hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448
     ,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))).
attacker(hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2
     [],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))).

35. By 10, the attacker may know exp(g,y_5467).
By 12, the attacker may know y_5407.
Using the function exp the attacker may obtain exp(exp(g,y_5407),y_5467).
attacker(exp(exp(g,y_5407),y_5467)).

36. By 35, the attacker may know exp(exp(g,y_5407),y_5467).
By 34, the attacker may know hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),
     APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))).
Using the function HKDF the attacker may obtain HKDF(exp(exp(g,y_5407),y_5467),hash(((
     xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,
     S_5465,N_V_5466,exp(g,y_5467))))).
attacker(HKDF(exp(exp(g,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,
     y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))))).

37. By 19, the attacker may know ((MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)),
```

```
      encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,
      exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))),
      APP_5464),skV[]),HKDF(exp(exp(g,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,xS_U_5447,
      xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,
      y_5467))))),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(
      MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))))).
Using the function 2-proj-2-tuple the attacker may obtain encrypt(sign((identifyPK(skV[],pk
      (skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(
      MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))),APP_5464),skV[]),HKDF(exp(exp(g
      ,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450)
      ,(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))),hash(((xMSG_TYPE_1_5446,
      xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,
      exp(g,y_5467))))).
attacker(encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,
      xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,
      y_5467)))),APP_5464),skV[]),HKDF(exp(exp(g,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,
      xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,
      exp(g,y_5467))))),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450)
      ,(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))))).

38. By 37, the attacker may know encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((
      xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,
      S_5465,N_V_5466,exp(g,y_5467)))),APP_5464),skV[]),HKDF(exp(exp(g,y_5407),y_5467),hash
      (((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],
      xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448
      ,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))).
By 36, the attacker may know HKDF(exp(exp(g,y_5407),y_5467),hash(((xMSG_TYPE_1_5446,
      xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,
      exp(g,y_5467))))).
By 34, the attacker may know hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),
      APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))).
Using the function decrypt the attacker may obtain sign((identifyPK(skV[],pk(skV[])),hash
      (((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],
      xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))),APP_5464),skV[]).
attacker(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(
      g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467)))),APP_5464
      ),skV[])).

39. By 38, the attacker may know sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,
      xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,
      exp(g,y_5467)))),APP_5464),skV[]).
By 1, the attacker may know pk(skV[]).
Using the function verify the attacker may obtain (identifyPK(skV[],pk(skV[])),hash(((
      xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,
      S_5465,N_V_5466,exp(g,y_5467)))),APP_5464).
attacker((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,xS_U_5447,xN_U_5448,exp(g,
```

```
     y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,exp(g,y_5467))))),APP_5464))
     .

40. By 39, the attacker may know (identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5446,
     xS_U_5447,xN_U_5448,exp(g,y_5407),APP_1_5450),(MSG_TYPE_2[],xS_U_5447,S_5465,N_V_5466,
     exp(g,y_5467)))),APP_5464).
Using the function 3-proj-3-tuple the attacker may obtain APP_5464.
attacker(APP_5464).


Unified xMSG_TYPE_1_5446 with xMSG_TYPE_1_5355
Unified xS_U_5447 with xS_U_5356
Unified xN_U_5448 with xN_U_5357
Unified xE_U_5358 with exp(g,y_5407)
Unified APP_1_5450 with APP_1_5359
Unified xMSG_TYPE_1_5413 with xMSG_TYPE_1_5355
Unified xS_U_5414 with xS_U_5356
Unified xN_U_5415 with xN_U_5357
Unified xE_U_5416 with exp(g,y_5407)
Unified APP_1_5417 with APP_1_5359
Iterating unifyDerivation.
Fixpoint reached: nothing more to unify.
The clause after unifyDerivation is
attacker(APP_1_5722) && attacker(xN_U_5720) && attacker(xS_U_5719) && attacker(
     xMSG_TYPE_1_5718) && attacker(y_5721) -> attacker(APP_85[message_1_81 = (
     xMSG_TYPE_1_5718,xS_U_5719,xN_U_5720,exp(g,y_5721),APP_1_5722),!1 = @sid_5723])
This clause still contradicts the query.
A more detailed output of the traces is available with
  set traceDisplay = long.

new U: host creating U_5817 at {1}

new V: host creating V_5818 at {2}

new skU: skey creating skU_5819 at {3}

new skV: skey creating skV_5731 at {4}

out(c, ~M_5987) with ~M_5987 = pk(skU_5819) at {9}

out(c, ~M_5989) with ~M_5989 = pk(skV_5731) at {10}

new y_80: exponent creating y_5734 at {36} in copy a_5729

new S_V: bitstring creating S_V_5732 at {38} in copy a_5729
```

```
in(c, (a_5724,a_5725,a_5726,exp(g,a_5727),a_5728)) at {39} in copy a_5729

new N_V_84: nonce creating N_V_5733 at {41} in copy a_5729

new APP_85: bitstring creating APP_5730 at {42} in copy a_5729

out(c, ((~M_6153,~M_6154,~M_6155,~M_6156,~M_6157),~M_6152)) with ~M_6153 = MSG_TYPE_2, ~
     M_6154 = a_5725, ~M_6155 = S_V_5732, ~M_6156 = N_V_5733, ~M_6157 = exp(g,y_5734), ~
     M_6152 = encrypt(sign((identifyPK(skV_5731,pk(skV_5731)),hash(((a_5724,a_5725,a_5726,
     exp(g,a_5727),a_5728),(MSG_TYPE_2,a_5725,S_V_5732,N_V_5733,exp(g,y_5734)))),APP_5730),
     skV_5731),HKDF(exp(exp(g,a_5727),y_5734),hash(((a_5724,a_5725,a_5726,exp(g,a_5727),
     a_5728),(MSG_TYPE_2,a_5725,S_V_5732,N_V_5733,exp(g,y_5734))))),hash(((a_5724,a_5725,
     a_5726,exp(g,a_5727),a_5728),(MSG_TYPE_2,a_5725,S_V_5732,N_V_5733,exp(g,y_5734))))) at
     {50} in copy a_5729

The attacker has the message 3-proj-3-tuple(verify(decrypt(~M_6152,HKDF(exp(~M_6157,a_5727)
     ,hash(((a_5724,a_5725,a_5726,exp(g,a_5727),a_5728),(MSG_TYPE_2,a_5725,~M_6155,~M_6156,~
     M_6157)))),hash(((a_5724,a_5725,a_5726,exp(g,a_5727),a_5728),(MSG_TYPE_2,a_5725,~M_6155
     ,~M_6156,~M_6157)))),~M_5989)) = APP_5730.
A trace has been found.
The previous trace falsifies the query, because the query is
simple and the trace corresponds to the derivation.
RESULT not attacker(APP_85[message_1_81 = v_3113,!1 = v_3114]) is false.
Starting query not attacker(APP_73[message_2_66 = v_3111,!1 = v_3112])
RESULT not attacker(APP_73[message_2_66 = v_3111,!1 = v_3112]) is true.
```

Asymmetric forward secrecy

```
Linear part:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Completing equations...
Completed equations:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new skU: skey;
{4}new skV: skey;
{5}let pkU: pkey = pk(skU) in
{6}let pkV: pkey = pk(skV) in
{7}let pkIdU: pkID = identifyPK(skU,pk(skU)) in
{8}let pkIdV: pkID = identifyPK(skV,pk(skV)) in
```

```
{9}out(c, pkU);
{10}out(c, pkV);
(
    {11}!
    {12}new x_63: exponent;
    {13}let E_U: G = exp(g,x_63) in
    {14}new S_U: bitstring;
    {15}new N_U: nonce;
    {16}new APP_64: bitstring;
    {17}let message_1_65: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,APP_64) in
    {18}out(c, message_1_65);
    {19}in(c, message_2_66: bitstring);
    {20}let (data_2_67: bitstring,COSE_ENC_2_68: bitstring) = message_2_66 in
    {21}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,N_V: nonce,xE_V: G) = data_2_67 in
    {22}let aad_2_69: bitstring = hash((message_1_65,data_2_67)) in
    {23}let K: G = exp(xE_V,x_63) in
    {24}let K_2_70: derivedKey = HKDF(K,aad_2_69) in
    {25}let signature_2_71: bitstring = decrypt(COSE_ENC_2_68,K_2_70,aad_2_69) in
    {26}let (=pkIdV,=aad_2_69,APP_2_72: bitstring) = verify(signature_2_71,pkV) in
    {27}new APP_73: bitstring;
    {28}let data_3_74: bitstring = (MSG_TYPE_3,xS_V) in
    {29}let aad_3_75: bitstring = hash((hash((message_1_65,message_2_66)),data_3_74)) in
    {30}let signature_3_76: bitstring = sign((identifyPK(skU,pkU),aad_3_75,APP_73),skU) in
    {31}let K_3_77: derivedKey = HKDF(K,aad_3_75) in
    {32}let COSE_ENC_3_78: bitstring = encrypt(signature_3_76,K_3_77,aad_3_75) in
    {33}let message_3_79: bitstring = (data_3_74,COSE_ENC_3_78) in
    {34}out(c, message_3_79)
) | (
    {35}!
    {36}new y_80: exponent;
    {37}let E_V: G = exp(g,y_80) in
    {38}new S_V: bitstring;
    {39}in(c, message_1_81: bitstring);
    {40}let (xMSG_TYPE_1_82: bitstring,xS_U: bitstring,xN_U: nonce,xE_U: G,APP_1_83:
         bitstring) = message_1_81 in
    {41}new N_V_84: nonce;
    {42}new APP_85: bitstring;
    {43}let data_2_86: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V_84,E_V) in
    {44}let aad_2_87: bitstring = hash((message_1_81,data_2_86)) in
    {45}let signature_2_88: bitstring = sign((identifyPK(skV,pkV),aad_2_87,APP_85),skV) in
    {46}let K_89: G = exp(xE_U,y_80) in
    {47}let K_2_90: derivedKey = HKDF(K_89,aad_2_87) in
    {48}let COSE_ENC_2_91: bitstring = encrypt(signature_2_88,K_2_90,aad_2_87) in
    {49}let message_2_92: bitstring = (data_2_86,COSE_ENC_2_91) in
    {50}out(c, message_2_92);
    {51}in(c, message_3_93: bitstring);
```

```
    {52}let (data_3_94: bitstring,COSE_ENC_3_95: bitstring) = message_3_93 in
    {53}let (=MSG_TYPE_3,=S_V) = data_3_94 in
    {54}let aad_3_96: bitstring = hash((hash((message_1_81,message_2_92)),data_3_94)) in
    {55}let K_3_97: derivedKey = HKDF(K_89,aad_3_96) in
    {56}let signature_3_98: bitstring = decrypt(COSE_ENC_3_95,K_3_97,aad_3_96) in
    {57}let (=pkIdU,=aad_3_96,APP_3_99: bitstring) = verify(signature_3_98,pkU) in
    0
) | (
    {58}phase 1;
    {59}out(c, skV);
    {60}out(c, skU)
)

-- Query not attacker_p1(APP_85[message_1_81 = v_1346,!1 = v_1347]); not attacker_p1(APP_73
     [message_2_66 = v_1348,!1 = v_1349])
Completing...
200 rules inserted. The rule base contains 158 rules. 31 rules in the queue.
400 rules inserted. The rule base contains 209 rules. 22 rules in the queue.
Starting query not attacker_p1(APP_85[message_1_81 = v_1346,!1 = v_1347])
goal reachable: attacker(xMSG_TYPE_1_5012) && attacker(xS_U_5013) && attacker(xN_U_5014) &&
      attacker(y_5015) && attacker(APP_1_5016) -> attacker_p1(APP_85[message_1_81 = (
     xMSG_TYPE_1_5012,xS_U_5013,xN_U_5014,exp(g,y_5015),APP_1_5016),!1 = @sid_5017])
Abbreviations:
APP = APP_85[message_1_81 = (xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125)
     ,!1 = @sid_5126]
S = S_V[!1 = @sid_5126]
N_V_5140 = N_V_84[message_1_81 = (xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),
     APP_1_5125),!1 = @sid_5126]
y_5141 = y_80[!1 = @sid_5126]
N_V_5142 = N_V_84[message_1_81 = (xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034
     ),!1 = @sid_5126]
APP_5143 = APP_85[message_1_81 = (xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034
     ),!1 = @sid_5126]
N_V_5144 = N_V_84[message_1_81 = (xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092
     ),!1 = @sid_5126]
APP_5145 = APP_85[message_1_81 = (xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092
     ),!1 = @sid_5126]

1. The message pk(skV[]) may be sent to the attacker at output {10}.
attacker(pk(skV[])).

2. The attacker has some term APP_1_5092.
attacker(APP_1_5092).

3. The attacker has some term xE_U_5091.
attacker(xE_U_5091).
```

```
4. The attacker has some term xN_U_5090.
attacker(xN_U_5090).

5. The attacker has some term xS_U_5089.
attacker(xS_U_5089).

6. The attacker has some term xMSG_TYPE_1_5088.
attacker(xMSG_TYPE_1_5088).

7. By 6, the attacker may know xMSG_TYPE_1_5088.
By 5, the attacker may know xS_U_5089.
By 4, the attacker may know xN_U_5090.
By 3, the attacker may know xE_U_5091.
By 2, the attacker may know APP_1_5092.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,
     xE_U_5091,APP_1_5092).
attacker((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092)).

8. The message (xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092) that the
     attacker may have by 7 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)),encrypt(sign((identifyPK(
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092),(
     MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)))),APP_5145),skV[]),HKDF(exp(xE_U_5091,
     y_5141),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092),(MSG_TYPE_2
     [],xS_U_5089,S,N_V_5144,exp(g,y_5141)))))),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,
     xE_U_5091,APP_1_5092),(MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)))))) may be sent
      to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)),encrypt(sign((identifyPK(skV[],
     pk(skV[])),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092),(
     MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)))),APP_5145),skV[]),HKDF(exp(xE_U_5091,
     y_5141),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092),(MSG_TYPE_2
     [],xS_U_5089,S,N_V_5144,exp(g,y_5141)))))),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,
     xE_U_5091,APP_1_5092),(MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)))))))).

9. By 8, the attacker may know ((MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)),encrypt(
     sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091
     ,APP_1_5092),(MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)))),APP_5145),skV[]),HKDF(
     exp(xE_U_5091,y_5141),hash(((xMSG_TYPE_1_5088,xS_U_5089,xN_U_5090,xE_U_5091,APP_1_5092)
     ,(MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)))))),hash(((xMSG_TYPE_1_5088,xS_U_5089
     ,xN_U_5090,xE_U_5091,APP_1_5092),(MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5089,S,
     N_V_5144,exp(g,y_5141)).
attacker((MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141))).

10. By 9, the attacker may know (MSG_TYPE_2[],xS_U_5089,S,N_V_5144,exp(g,y_5141)).
```

```
Using the function 5-proj-5-tuple the attacker may obtain exp(g,y_5141).
attacker(exp(g,y_5141)).

11. We assume as hypothesis that
attacker(APP_1_5125).

12. We assume as hypothesis that
attacker(y_5082).

13. Using the function g the attacker may obtain g.
attacker(g).

14. By 13, the attacker may know g.
By 12, the attacker may know y_5082.
Using the function exp the attacker may obtain exp(g,y_5082).
attacker(exp(g,y_5082)).

15. We assume as hypothesis that
attacker(xN_U_5123).

16. We assume as hypothesis that
attacker(xS_U_5122).

17. We assume as hypothesis that
attacker(xMSG_TYPE_1_5121).

18. By 17, the attacker may know xMSG_TYPE_1_5121.
By 16, the attacker may know xS_U_5122.
By 15, the attacker may know xN_U_5123.
By 14, the attacker may know exp(g,y_5082).
By 11, the attacker may know APP_1_5125.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,
     exp(g,y_5082),APP_1_5125).
attacker((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125)).

19. The message (xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125) that the
     attacker may have by 18 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)),encrypt(sign((identifyPK(
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125)
     ,(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP),skV[]),HKDF(exp(exp(g,y_5082)
     ,y_5141),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(
     MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))),hash(((xMSG_TYPE_1_5121,xS_U_5122,
     xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))
     )) may be sent to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)),encrypt(sign((identifyPK(skV[],
     pk(skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(
```

```
     MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP),skV[]),HKDF(exp(exp(g,y_5082),
     y_5141),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(
     MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),hash(((xMSG_TYPE_1_5121,xS_U_5122,
     xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))
     ))).

20. By 19, the attacker may know ((MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)),encrypt
     (sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,
     y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP),skV[]),
     HKDF(exp(exp(g,y_5082),y_5141),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082
     ),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),hash(((
     xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,
     S,N_V_5140,exp(g,y_5141)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5122,S,
     N_V_5140,exp(g,y_5141)).
attacker((MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))).

21. By 20, the attacker may know (MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)).
Using the function 4-proj-5-tuple the attacker may obtain N_V_5140.
attacker(N_V_5140).

22. The attacker has some term APP_1_5034.
attacker(APP_1_5034).

23. The attacker has some term xE_U_5033.
attacker(xE_U_5033).

24. The attacker has some term xN_U_5032.
attacker(xN_U_5032).

25. The attacker has some term xS_U_5031.
attacker(xS_U_5031).

26. The attacker has some term xMSG_TYPE_1_5030.
attacker(xMSG_TYPE_1_5030).

27. By 26, the attacker may know xMSG_TYPE_1_5030.
By 25, the attacker may know xS_U_5031.
By 24, the attacker may know xN_U_5032.
By 23, the attacker may know xE_U_5033.
By 22, the attacker may know APP_1_5034.
Using the function 5-tuple the attacker may obtain (xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,
     xE_U_5033,APP_1_5034).
attacker((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034)).

28. The message (xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034) that the
```

```
     attacker may have by 27 may be received at input {39}.
So the message ((MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)),encrypt(sign((identifyPK(
     skV[],pk(skV[])),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034),(
     MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)))),APP_5143),skV[]),HKDF(exp(xE_U_5033,
     y_5141),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034),(MSG_TYPE_2
     [],xS_U_5031,S,N_V_5142,exp(g,y_5141))))),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,
     xE_U_5033,APP_1_5034),(MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)))))) may be sent
      to the attacker at output {50}.
attacker(((MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)),encrypt(sign((identifyPK(skV[],
     pk(skV[])),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034),(
     MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)))),APP_5143),skV[]),HKDF(exp(xE_U_5033,
     y_5141),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034),(MSG_TYPE_2
     [],xS_U_5031,S,N_V_5142,exp(g,y_5141))))),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,
     xE_U_5033,APP_1_5034),(MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)))))).

29. By 28, the attacker may know ((MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)),encrypt
     (sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,
     xE_U_5033,APP_1_5034),(MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)))),APP_5143),skV
     []),HKDF(exp(xE_U_5033,y_5141),hash(((xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,
     APP_1_5034),(MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141))))),hash(((
     xMSG_TYPE_1_5030,xS_U_5031,xN_U_5032,xE_U_5033,APP_1_5034),(MSG_TYPE_2[],xS_U_5031,S,
     N_V_5142,exp(g,y_5141)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5031,S,
     N_V_5142,exp(g,y_5141)).
attacker((MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141))).

30. By 29, the attacker may know (MSG_TYPE_2[],xS_U_5031,S,N_V_5142,exp(g,y_5141)).
Using the function 3-proj-5-tuple the attacker may obtain S.
attacker(S).

31. The attacker initially knows MSG_TYPE_2[].
attacker(MSG_TYPE_2[]).

32. By 31, the attacker may know MSG_TYPE_2[].
By 16, the attacker may know xS_U_5122.
By 30, the attacker may know S.
By 21, the attacker may know N_V_5140.
By 10, the attacker may know exp(g,y_5141).
Using the function 5-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g
     ,y_5141)).
attacker((MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))).

33. By 18, the attacker may know (xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),
     APP_1_5125).
By 32, the attacker may know (MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)).
Using the function 2-tuple the attacker may obtain ((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,
```

```
    exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))).
attacker(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],
    xS_U_5122,S,N_V_5140,exp(g,y_5141)))).

34. By 33, the attacker may know ((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),
    APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))).
Using the function hash the attacker may obtain hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123
    ,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))).
attacker(hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2
    [],xS_U_5122,S,N_V_5140,exp(g,y_5141))))).

35. By 10, the attacker may know exp(g,y_5141).
By 12, the attacker may know y_5082.
Using the function exp the attacker may obtain exp(exp(g,y_5082),y_5141).
attacker(exp(exp(g,y_5082),y_5141)).

36. By 35, the attacker may know exp(exp(g,y_5082),y_5141).
By 34, the attacker may know hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),
    APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))).
Using the function HKDF the attacker may obtain HKDF(exp(exp(g,y_5082),y_5141),hash(((
    xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,
    S,N_V_5140,exp(g,y_5141))))).
attacker(HKDF(exp(exp(g,y_5082),y_5141),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,
    y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))))).

37. By 19, the attacker may know ((MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)),encrypt
    (sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,
    y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP),skV[]),
    HKDF(exp(exp(g,y_5082),y_5141),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082
    ),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))),hash(((
    xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,
    S,N_V_5140,exp(g,y_5141)))))).
Using the function 2-proj-2-tuple the attacker may obtain encrypt(sign((identifyPK(skV[],pk
    (skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(
    MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP),skV[]),HKDF(exp(exp(g,y_5082),
    y_5141),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(
    MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))),hash(((xMSG_TYPE_1_5121,xS_U_5122,
    xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))
    ).
attacker(encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,
    xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))
    ,APP),skV[]),HKDF(exp(exp(g,y_5082),y_5141),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123
    ,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))),hash(((
    xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,
    S,N_V_5140,exp(g,y_5141)))))).
```

```
38. By 37, the attacker may know encrypt(sign((identifyPK(skV[],pk(skV[])),hash(((
    xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,
    S,N_V_5140,exp(g,y_5141)))),APP),skV[]),HKDF(exp(exp(g,y_5082),y_5141),hash(((
    xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,
    S,N_V_5140,exp(g,y_5141))))),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),
    APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141))))).
By 36, the attacker may know HKDF(exp(exp(g,y_5082),y_5141),hash(((xMSG_TYPE_1_5121,
    xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,
    y_5141))))).
By 34, the attacker may know hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),
    APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))).
Using the function decrypt the attacker may obtain sign((identifyPK(skV[],pk(skV[])),hash
    (((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],
    xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP),skV[]).
attacker(sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(
    g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP),skV[])).

39. By 38, the attacker may know sign((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,
    xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,
    y_5141)))),APP),skV[]).
By 1, the attacker may know pk(skV[]).
Using the function verify the attacker may obtain (identifyPK(skV[],pk(skV[])),hash(((
    xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,
    S,N_V_5140,exp(g,y_5141)))),APP).
attacker((identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,xS_U_5122,xN_U_5123,exp(g,
    y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,y_5141)))),APP)).

40. By 39, the attacker may know (identifyPK(skV[],pk(skV[])),hash(((xMSG_TYPE_1_5121,
    xS_U_5122,xN_U_5123,exp(g,y_5082),APP_1_5125),(MSG_TYPE_2[],xS_U_5122,S,N_V_5140,exp(g,
    y_5141)))),APP).
Using the function 3-proj-3-tuple the attacker may obtain APP.
attacker(APP).

41. By 40, the attacker may know APP.
So the attacker may know APP in phase 1.
attacker_p1(APP).


Unified xMSG_TYPE_1_5121 with xMSG_TYPE_1_5030
Unified xS_U_5122 with xS_U_5031
Unified xN_U_5123 with xN_U_5032
Unified xE_U_5033 with exp(g,y_5082)
Unified APP_1_5125 with APP_1_5034
Unified xMSG_TYPE_1_5088 with xMSG_TYPE_1_5030
Unified xS_U_5089 with xS_U_5031
Unified xN_U_5090 with xN_U_5032
```

```
Unified xE_U_5091 with exp(g,y_5082)
Unified APP_1_5092 with APP_1_5034
Iterating unifyDerivation.
Fixpoint reached: nothing more to unify.
The clause after unifyDerivation is
attacker(APP_1_5395) && attacker(xN_U_5393) && attacker(xS_U_5392) && attacker(
     xMSG_TYPE_1_5391) && attacker(y_5394) -> attacker_p1(APP_85[message_1_81 = (
     xMSG_TYPE_1_5391,xS_U_5392,xN_U_5393,exp(g,y_5394),APP_1_5395),!1 = @sid_5396])
This clause still contradicts the query.
A more detailed output of the traces is available with
  set traceDisplay = long.

new U: host creating U_5488 at {1}

new V: host creating V_5489 at {2}

new skU: skey creating skU_5490 at {3}

new skV: skey creating skV_5404 at {4}

out(c, ~M_5657) with ~M_5657 = pk(skU_5490) at {9}

out(c, ~M_5659) with ~M_5659 = pk(skV_5404) at {10}

new y_80: exponent creating y_5407 at {36} in copy a_5402

new S_V: bitstring creating S_V_5405 at {38} in copy a_5402

in(c, (a_5397,a_5398,a_5399,exp(g,a_5400),a_5401)) at {39} in copy a_5402

new N_V_84: nonce creating N_V_5406 at {41} in copy a_5402

new APP_85: bitstring creating APP_5403 at {42} in copy a_5402

out(c, ((~M_5822,~M_5823,~M_5824,~M_5825,~M_5826),~M_5821)) with ~M_5822 = MSG_TYPE_2, ~
     M_5823 = a_5398, ~M_5824 = S_V_5405, ~M_5825 = N_V_5406, ~M_5826 = exp(g,y_5407), ~
     M_5821 = encrypt(sign((identifyPK(skV_5404,pk(skV_5404)),hash(((a_5397,a_5398,a_5399,
     exp(g,a_5400),a_5401),(MSG_TYPE_2,a_5398,S_V_5405,N_V_5406,exp(g,y_5407)))),APP_5403),
     skV_5404),HKDF(exp(exp(g,a_5400),y_5407),hash(((a_5397,a_5398,a_5399,exp(g,a_5400),
     a_5401),(MSG_TYPE_2,a_5398,S_V_5405,N_V_5406,exp(g,y_5407))))),hash(((a_5397,a_5398,
     a_5399,exp(g,a_5400),a_5401),(MSG_TYPE_2,a_5398,S_V_5405,N_V_5406,exp(g,y_5407))))) at
     {50} in copy a_5402

The attacker has the message 3-proj-3-tuple(verify(decrypt(~M_5821,HKDF(exp(~M_5826,a_5400)
     ,hash(((a_5397,a_5398,a_5399,exp(g,a_5400),a_5401),(MSG_TYPE_2,a_5398,~M_5824,~M_5825,~
     M_5826)))),hash(((a_5397,a_5398,a_5399,exp(g,a_5400),a_5401),(MSG_TYPE_2,a_5398,~M_5824
```

```
     ,~M_5825,~M_5826)))),~M_5659)) = APP_5403 in phase 1.
A trace has been found.
The previous trace falsifies the query, because the query is
simple and the trace corresponds to the derivation.
RESULT not attacker_p1(APP_85[message_1_81 = v_1346,!1 = v_1347]) is false.
Starting query not attacker_p1(APP_73[message_2_66 = v_1348,!1 = v_1349])
RESULT not attacker_p1(APP_73[message_2_66 = v_1348,!1 = v_1349]) is true.
```

Symmetric injective agreement

```
Linear part:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Completing equations...
Completed equations:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new W: host;
{4}new PSKs_UV: bitstring;
{5}new PSKs_UW: bitstring;
(
    {6}!
    {7}new x_49: exponent;
    {8}let E_U: G = exp(g,x_49) in
    {9}new S_U: bitstring;
    {10}event startInitiator(U,V,S_U);
    {11}new N_U: nonce;
    {12}new KID: bitstring;
    {13}new APP_50: bitstring;
    {14}let PSK: preSharedkey = identifyPreSharedKey(KID,PSKs_UV) in
    {15}let message_1_51: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,KID,APP_50) in
    {16}out(c, message_1_51);
    {17}in(c, message_2_52: bitstring);
    {18}let (data_2_53: bitstring,COSE_ENC_2_54: bitstring) = message_2_52 in
    {19}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,xN_V: nonce,xE_V: G) = data_2_53 in
    {20}let aad_2_55: bitstring = hash((message_1_51,data_2_53)) in
    {21}let K: G = exp(xE_V,x_49) in
    {22}let K_2_56: derivedKey = HKDF(K,aad_2_55,PSK) in
    {23}let APP_2_57: bitstring = sharedDecrypt(COSE_ENC_2_54,K_2_56,aad_2_55) in
    {24}let data_3_58: bitstring = (MSG_TYPE_3,xS_V) in
    {25}let aad_3_59: bitstring = hash((hash((message_1_51,message_2_52)),data_3_58)) in
```

```
    {26}let K_3_60: derivedKey = HKDF(K,aad_3_59,PSK) in
    {27}new APP_61: bitstring;
    {28}let message_3_62: bitstring = (data_3_58,sharedEncrypt(APP_61,K_3_60,aad_3_59)) in
    {29}out(c, message_3_62);
    {30}event endInitiator(U,V,S_U,xS_V)
) | (
    {31}!
    {32}new y_63: exponent;
    {33}let E_V: G = exp(g,y_63) in
    {34}new S_V: bitstring;
    {35}event startResponder(U,V,S_V);
    {36}in(c, message_1_64: bitstring);
    {37}let (=MSG_TYPE_1,xS_U: bitstring,N_U_65: nonce,xE_U: G,xKID: bitstring,APP_1_66:
        bitstring) = message_1_64 in
    {38}new N_V: nonce;
    {39}new APP_67: bitstring;
    {40}let PSK_68: preSharedkey = identifyPreSharedKey(xKID,PSKs_UV) in
    {41}let data_2_69: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V,E_V) in
    {42}let aad_2_70: bitstring = hash((message_1_64,data_2_69)) in
    {43}let K_71: G = exp(xE_U,y_63) in
    {44}let K_2_72: derivedKey = HKDF(K_71,aad_2_70,PSK_68) in
    {45}let message_2_73: bitstring = (data_2_69,sharedEncrypt(APP_67,K_2_72,aad_2_70)) in
    {46}out(c, message_2_73);
    {47}in(c, message_3_74: bitstring);
    {48}let (data_3_75: bitstring,COSE_ENC_3_76: bitstring) = message_3_74 in
    {49}let (=MSG_TYPE_3,=S_V) = data_3_75 in
    {50}let aad_3_77: bitstring = hash((hash((message_1_64,message_2_73)),data_3_75)) in
    {51}let K_3_78: derivedKey = HKDF(K_71,aad_3_77,PSK_68) in
    {52}let APP_3_79: bitstring = sharedDecrypt(COSE_ENC_3_76,K_3_78,aad_3_77) in
    {53}event endResponder(U,V,S_V,xS_U)
) | (
    {54}!
    {55}new x_80: exponent;
    {56}let E_U_81: G = exp(g,x_80) in
    {57}new S_U_82: bitstring;
    {58}event startInitiator(U,W,S_U_82);
    {59}new N_U_83: nonce;
    {60}new KID_84: bitstring;
    {61}new APP_85: bitstring;
    {62}let PSK_86: preSharedkey = identifyPreSharedKey(KID_84,PSKs_UW) in
    {63}let message_1_87: bitstring = (MSG_TYPE_1,S_U_82,N_U_83,E_U_81,KID_84,APP_85) in
    {64}out(c, message_1_87);
    {65}in(c, message_2_88: bitstring);
    {66}let (data_2_89: bitstring,COSE_ENC_2_90: bitstring) = message_2_88 in
    {67}let (=MSG_TYPE_2,=S_U_82,xS_V_91: bitstring,xN_V_92: nonce,xE_V_93: G) = data_2_89
        in
```

```
    {68}let aad_2_94: bitstring = hash((message_1_87,data_2_89)) in
    {69}let K_95: G = exp(xE_V_93,x_80) in
    {70}let K_2_96: derivedKey = HKDF(K_95,aad_2_94,PSK_86) in
    {71}let APP_2_97: bitstring = sharedDecrypt(COSE_ENC_2_90,K_2_96,aad_2_94) in
    {72}let data_3_98: bitstring = (MSG_TYPE_3,xS_V_91) in
    {73}let aad_3_99: bitstring = hash((hash((message_1_87,message_2_88)),data_3_98)) in
    {74}let K_3_100: derivedKey = HKDF(K_95,aad_3_99,PSK_86) in
    {75}new APP_101: bitstring;
    {76}let message_3_102: bitstring = (data_3_98,sharedEncrypt(APP_101,K_3_100,aad_3_99))
         in
    {77}out(c, message_3_102);
    {78}event endInitiator(U,W,S_U_82,xS_V_91)
) | (
    {79}!
    {80}new y_103: exponent;
    {81}let E_V_104: G = exp(g,y_103) in
    {82}new S_V_105: bitstring;
    {83}event startResponder(U,W,S_V_105);
    {84}in(c, message_1_106: bitstring);
    {85}let (=MSG_TYPE_1,xS_U_107: bitstring,N_U_108: nonce,xE_U_109: G,xKID_110: bitstring,
         APP_1_111: bitstring) = message_1_106 in
    {86}new N_V_112: nonce;
    {87}new APP_113: bitstring;
    {88}let PSK_114: preSharedkey = identifyPreSharedKey(xKID_110,PSKs_UW) in
    {89}let data_2_115: bitstring = (MSG_TYPE_2,xS_U_107,S_V_105,N_V_112,E_V_104) in
    {90}let aad_2_116: bitstring = hash((message_1_106,data_2_115)) in
    {91}let K_117: G = exp(xE_U_109,y_103) in
    {92}let K_2_118: derivedKey = HKDF(K_117,aad_2_116,PSK_114) in
    {93}let message_2_119: bitstring = (data_2_115,sharedEncrypt(APP_113,K_2_118,aad_2_116))
          in
    {94}out(c, message_2_119);
    {95}in(c, message_3_120: bitstring);
    {96}let (data_3_121: bitstring,COSE_ENC_3_122: bitstring) = message_3_120 in
    {97}let (=MSG_TYPE_3,=S_V_105) = data_3_121 in
    {98}let aad_3_123: bitstring = hash((hash((message_1_106,message_2_119)),data_3_121)) in
    {99}let K_3_124: derivedKey = HKDF(K_117,aad_3_123,PSK_114) in
    {100}let APP_3_125: bitstring = sharedDecrypt(COSE_ENC_3_122,K_3_124,aad_3_123) in
    {101}event endResponder(U,W,S_V_105,xS_U_107)
)

-- Query event(endResponder(U_126,V_127,S_V_129,S_U_128)) ==> event(startInitiator(U_126,
     V_127,S_U_128))
Completing...
200 rules inserted. The rule base contains 140 rules. 48 rules in the queue.
400 rules inserted. The rule base contains 259 rules. 8 rules in the queue.
Starting query event(endResponder(U_126,V_127,S_V_129,S_U_128)) ==> event(startInitiator(
```

```
    U_126,V_127,S_U_128))
goal reachable: begin(startInitiator(U[],W[],S_U_82[!1 = @sid_6294])) -> end(endResponder(U
    [],W[],S_V_105[!1 = @sid_6295],S_U_82[!1 = @sid_6294]))
goal reachable: begin(startInitiator(U[],V[],S_U[!1 = @sid_6298])) -> end(endResponder(U[],
    V[],S_V[!1 = @sid_6299],S_U[!1 = @sid_6298]))
RESULT event(endResponder(U_126,V_127,S_V_129,S_U_128)) ==> event(startInitiator(U_126,
    V_127,S_U_128)) is true.
-- Query event(endInitiator(U_6306,V_6307,S_U_6308,S_V_6309)) ==> event(startResponder(
    U_6306,V_6307,S_V_6309))
Completing...
200 rules inserted. The rule base contains 146 rules. 41 rules in the queue.
400 rules inserted. The rule base contains 266 rules. 5 rules in the queue.
Starting query event(endInitiator(U_6306,V_6307,S_U_6308,S_V_6309)) ==> event(
    startResponder(U_6306,V_6307,S_V_6309))
goal reachable: begin(startResponder(U[],W[],S_V_105[!1 = @sid_12045])) -> end(endInitiator
    (U[],W[],S_U_82[!1 = @sid_12046],S_V_105[!1 = @sid_12045]))
goal reachable: begin(startResponder(U[],V[],S_V[!1 = @sid_12049])) -> end(endInitiator(U
    [],V[],S_U[!1 = @sid_12050],S_V[!1 = @sid_12049]))
RESULT event(endInitiator(U_6306,V_6307,S_U_6308,S_V_6309)) ==> event(startResponder(U_6306
    ,V_6307,S_V_6309)) is true.
```

Symmetric confidentiality

```
Linear part:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Completing equations...
Completed equations:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new PSKs: bitstring;
(
   {4}!
   {5}new x_49: exponent;
   {6}let E_U: G = exp(g,x_49) in
   {7}new S_U: bitstring;
   {8}new N_U: nonce;
   {9}new KID: bitstring;
   {10}new APP_50: bitstring;
   {11}let PSK: preSharedkey = identifyPreSharedKey(KID,PSKs) in
   {12}let message_1_51: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,KID,APP_50) in
```

```
    {13}out(c, message_1_51);
    {14}in(c, message_2_52: bitstring);
    {15}let (data_2_53: bitstring,COSE_ENC_2_54: bitstring) = message_2_52 in
    {16}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,xN_V: nonce,xE_V: G) = data_2_53 in
    {17}let aad_2_55: bitstring = hash((message_1_51,data_2_53)) in
    {18}let K: G = exp(xE_V,x_49) in
    {19}let K_2_56: derivedKey = HKDF(K,aad_2_55,PSK) in
    {20}let APP_2_57: bitstring = sharedDecrypt(COSE_ENC_2_54,K_2_56,aad_2_55) in
    {21}let data_3_58: bitstring = (MSG_TYPE_3,xS_V) in
    {22}let aad_3_59: bitstring = hash((hash((message_1_51,message_2_52)),data_3_58)) in
    {23}let K_3_60: derivedKey = HKDF(K,aad_3_59,PSK) in
    {24}new APP_61: bitstring;
    {25}let message_3_62: bitstring = (data_3_58,sharedEncrypt(APP_61,K_3_60,aad_3_59)) in
    {26}out(c, message_3_62)
) | (
    {27}!
    {28}new y_63: exponent;
    {29}let E_V: G = exp(g,y_63) in
    {30}new S_V: bitstring;
    {31}in(c, message_1_64: bitstring);
    {32}let (=MSG_TYPE_1,xS_U: bitstring,N_U_65: nonce,xE_U: G,xKID: bitstring,APP_1_66:
        bitstring) = message_1_64 in
    {33}new N_V: nonce;
    {34}new APP_67: bitstring;
    {35}let PSK_68: preSharedkey = identifyPreSharedKey(xKID,PSKs) in
    {36}let data_2_69: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V,E_V) in
    {37}let aad_2_70: bitstring = hash((message_1_64,data_2_69)) in
    {38}let K_71: G = exp(xE_U,y_63) in
    {39}let K_2_72: derivedKey = HKDF(K_71,aad_2_70,PSK_68) in
    {40}let message_2_73: bitstring = (data_2_69,sharedEncrypt(APP_67,K_2_72,aad_2_70)) in
    {41}out(c, message_2_73);
    {42}in(c, message_3_74: bitstring);
    {43}let (data_3_75: bitstring,COSE_ENC_3_76: bitstring) = message_3_74 in
    {44}let (=MSG_TYPE_3,=S_V) = data_3_75 in
    {45}let aad_3_77: bitstring = hash((hash((message_1_64,message_2_73)),data_3_75)) in
    {46}let K_3_78: derivedKey = HKDF(K_71,aad_3_77,PSK_68) in
    {47}let APP_3_79: bitstring = sharedDecrypt(COSE_ENC_3_76,K_3_78,aad_3_77) in
    0
)

-- Query not attacker(APP_67[message_1_64 = v_1141,!1 = v_1142]); not attacker(APP_61[
     message_2_52 = v_1143,!1 = v_1144])
Completing...
Starting query not attacker(APP_67[message_1_64 = v_1141,!1 = v_1142])
RESULT not attacker(APP_67[message_1_64 = v_1141,!1 = v_1142]) is true.
Starting query not attacker(APP_61[message_2_52 = v_1143,!1 = v_1144])
```

```
RESULT not attacker(APP_61[message_2_52 = v_1143,!1 = v_1144]) is true.
```

Symmetric forward secrecy

```
Linear part:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Completing equations...
Completed equations:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new PSKs: bitstring;
(
    {4}!
    {5}new x_49: exponent;
    {6}let E_U: G = exp(g,x_49) in
    {7}new S_U: bitstring;
    {8}new N_U: nonce;
    {9}new KID: bitstring;
    {10}new APP_50: bitstring;
    {11}let PSK: preSharedkey = identifyPreSharedKey(KID,PSKs) in
    {12}let message_1_51: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,KID,APP_50) in
    {13}out(c, message_1_51);
    {14}in(c, message_2_52: bitstring);
    {15}let (data_2_53: bitstring,COSE_ENC_2_54: bitstring) = message_2_52 in
    {16}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,xN_V: nonce,xE_V: G) = data_2_53 in
    {17}let aad_2_55: bitstring = hash((message_1_51,data_2_53)) in
    {18}let K: G = exp(xE_V,x_49) in
    {19}let K_2_56: derivedKey = HKDF(K,aad_2_55,PSK) in
    {20}let APP_2_57: bitstring = sharedDecrypt(COSE_ENC_2_54,K_2_56,aad_2_55) in
    {21}let data_3_58: bitstring = (MSG_TYPE_3,xS_V) in
    {22}let aad_3_59: bitstring = hash((hash((message_1_51,message_2_52)),data_3_58)) in
    {23}let K_3_60: derivedKey = HKDF(K,aad_3_59,PSK) in
    {24}new APP_61: bitstring;
    {25}let message_3_62: bitstring = (data_3_58,sharedEncrypt(APP_61,K_3_60,aad_3_59)) in
    {26}out(c, message_3_62)
) | (
    {27}!
    {28}new y_63: exponent;
    {29}let E_V: G = exp(g,y_63) in
    {30}new S_V: bitstring;
```

```
    {31}in(c, message_1_64: bitstring);
    {32}let (=MSG_TYPE_1,xS_U: bitstring,N_U_65: nonce,xE_U: G,xKID: bitstring,APP_1_66:
        bitstring) = message_1_64 in
    {33}new N_V: nonce;
    {34}new APP_67: bitstring;
    {35}let PSK_68: preSharedkey = identifyPreSharedKey(xKID,PSKs) in
    {36}let data_2_69: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V,E_V) in
    {37}let aad_2_70: bitstring = hash((message_1_64,data_2_69)) in
    {38}let K_71: G = exp(xE_U,y_63) in
    {39}let K_2_72: derivedKey = HKDF(K_71,aad_2_70,PSK_68) in
    {40}let message_2_73: bitstring = (data_2_69,sharedEncrypt(APP_67,K_2_72,aad_2_70)) in
    {41}out(c, message_2_73);
    {42}in(c, message_3_74: bitstring);
    {43}let (data_3_75: bitstring,COSE_ENC_3_76: bitstring) = message_3_74 in
    {44}let (=MSG_TYPE_3,=S_V) = data_3_75 in
    {45}let aad_3_77: bitstring = hash((hash((message_1_64,message_2_73)),data_3_75)) in
    {46}let K_3_78: derivedKey = HKDF(K_71,aad_3_77,PSK_68) in
    {47}let APP_3_79: bitstring = sharedDecrypt(COSE_ENC_3_76,K_3_78,aad_3_77) in
    0
) | (
    {48}phase 1;
    {49}out(c, PSKs)
)

-- Query not attacker_p1(APP_67[message_1_64 = v_1316,!1 = v_1317]); not attacker_p1(APP_61
     [message_2_52 = v_1318,!1 = v_1319])
Completing...
200 rules inserted. The rule base contains 157 rules. 21 rules in the queue.
400 rules inserted. The rule base contains 270 rules. 19 rules in the queue.
600 rules inserted. The rule base contains 366 rules. 26 rules in the queue.
Starting query not attacker_p1(APP_67[message_1_64 = v_1316,!1 = v_1317])
goal reachable: attacker_p1(xKID_4933) && attacker(xS_U_4934) && attacker(N_U_4935) &&
     attacker(y_4936) && attacker(xKID_4933) && attacker(APP_1_4937) -> attacker_p1(APP_67[
     message_1_64 = (MSG_TYPE_1[],xS_U_4934,N_U_4935,exp(g,y_4936),xKID_4933,APP_1_4937),!1
     = @sid_4938])
Abbreviations:
APP = APP_67[message_1_64 = (MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,
     APP_1_5059),!1 = @sid_5060]
y_5069 = y_63[!1 = @sid_5060]
S = S_V[!1 = @sid_5060]
N = N_V[message_1_64 = (MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059)
     ,!1 = @sid_5060]
N_5070 = N_V[message_1_64 = (MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,APP_1_4958
     ),!1 = @sid_5060]
APP_5071 = APP_67[message_1_64 = (MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,
     APP_1_4958),!1 = @sid_5060]
```

```
N_5072 = N_V[message_1_64 = (MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,APP_1_5003
    ),!1 = @sid_5060]
APP_5073 = APP_67[message_1_64 = (MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,
    APP_1_5003),!1 = @sid_5060]

1. The attacker has some term APP_1_5003.
attacker(APP_1_5003).

2. The attacker has some term xKID_5002.
attacker(xKID_5002).

3. The attacker has some term xE_U_5001.
attacker(xE_U_5001).

4. The attacker has some term N_U_5000.
attacker(N_U_5000).

5. The attacker has some term xS_U_4999.
attacker(xS_U_4999).

6. The attacker initially knows MSG_TYPE_1[].
attacker(MSG_TYPE_1[]).

7. By 6, the attacker may know MSG_TYPE_1[].
By 5, the attacker may know xS_U_4999.
By 4, the attacker may know N_U_5000.
By 3, the attacker may know xE_U_5001.
By 2, the attacker may know xKID_5002.
By 1, the attacker may know APP_1_5003.
Using the function 6-tuple the attacker may obtain (MSG_TYPE_1[],xS_U_4999,N_U_5000,
    xE_U_5001,xKID_5002,APP_1_5003).
attacker((MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,APP_1_5003)).

8. The message (MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,APP_1_5003) that the
    attacker may have by 7 may be received at input {31}.
So the message ((MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)),sharedEncrypt(APP_5073,HKDF
    (exp(xE_U_5001,y_5069),hash(((MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,
    APP_1_5003),(MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)))),identifyPreSharedKey(
    xKID_5002,PSKs[])),hash(((MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,
    APP_1_5003),(MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)))))) may be sent to the
    attacker at output {41}.
attacker(((MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)),sharedEncrypt(APP_5073,HKDF(exp(
    xE_U_5001,y_5069),hash(((MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,APP_1_5003
    ),(MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)))),identifyPreSharedKey(xKID_5002,PSKs
    [])),hash(((MSG_TYPE_1[],xS_U_4999,N_U_5000,xE_U_5001,xKID_5002,APP_1_5003),(MSG_TYPE_2
    [],xS_U_4999,S,N_5072,exp(g,y_5069)))))))).
```

```
9. By 8, the attacker may know ((MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)),
     sharedEncrypt(APP_5073,HKDF(exp(xE_U_5001,y_5069),hash(((MSG_TYPE_1[],xS_U_4999,
     N_U_5000,xE_U_5001,xKID_5002,APP_1_5003),(MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)
     ))),identifyPreSharedKey(xKID_5002,PSKs[])),hash(((MSG_TYPE_1[],xS_U_4999,N_U_5000,
     xE_U_5001,xKID_5002,APP_1_5003),(MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069))))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_4999,S,N_5072,
     exp(g,y_5069)).
attacker((MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069))).

10. By 9, the attacker may know (MSG_TYPE_2[],xS_U_4999,S,N_5072,exp(g,y_5069)).
Using the function 5-proj-5-tuple the attacker may obtain exp(g,y_5069).
attacker(exp(g,y_5069)).

11. We assume as hypothesis that
attacker(APP_1_5059).

12. We assume as hypothesis that
attacker(xKID_5058).

13. We assume as hypothesis that
attacker(y_4991).

14. Using the function g the attacker may obtain g.
attacker(g).

15. By 14, the attacker may know g.
By 13, the attacker may know y_4991.
Using the function exp the attacker may obtain exp(g,y_4991).
attacker(exp(g,y_4991)).

16. We assume as hypothesis that
attacker(N_U_5056).

17. We assume as hypothesis that
attacker(xS_U_5055).

18. By 6, the attacker may know MSG_TYPE_1[].
By 17, the attacker may know xS_U_5055.
By 16, the attacker may know N_U_5056.
By 15, the attacker may know exp(g,y_4991).
By 12, the attacker may know xKID_5058.
By 11, the attacker may know APP_1_5059.
Using the function 6-tuple the attacker may obtain (MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,
     y_4991),xKID_5058,APP_1_5059).
attacker((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059)).
```

```
19. The message (MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059) that
     the attacker may have by 18 may be received at input {31}.
So the message ((MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)),sharedEncrypt(APP,HKDF(exp(exp(g
     ,y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,
     APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))),identifyPreSharedKey(xKID_5058
     ,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(
     MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))))) may be sent to the attacker at output
     {41}.
attacker(((MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)),sharedEncrypt(APP,HKDF(exp(exp(g,
     y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,
     APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))),identifyPreSharedKey(xKID_5058
     ,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(
     MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))))))).

20. By 19, the attacker may know ((MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)),sharedEncrypt(
     APP,HKDF(exp(exp(g,y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991)
     ,xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))),
     identifyPreSharedKey(xKID_5058,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,
     y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5055,S,N,exp(g
     ,y_5069)).
attacker((MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))).

21. By 20, the attacker may know (MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)).
Using the function 4-proj-5-tuple the attacker may obtain N.
attacker(N).

22. The attacker has some term APP_1_4958.
attacker(APP_1_4958).

23. The attacker has some term xKID_4957.
attacker(xKID_4957).

24. The attacker has some term xE_U_4956.
attacker(xE_U_4956).

25. The attacker has some term N_U_4955.
attacker(N_U_4955).

26. The attacker has some term xS_U_4954.
attacker(xS_U_4954).

27. By 6, the attacker may know MSG_TYPE_1[].
By 26, the attacker may know xS_U_4954.
By 25, the attacker may know N_U_4955.
```

```
By 24, the attacker may know xE_U_4956.
By 23, the attacker may know xKID_4957.
By 22, the attacker may know APP_1_4958.
Using the function 6-tuple the attacker may obtain (MSG_TYPE_1[],xS_U_4954,N_U_4955,
     xE_U_4956,xKID_4957,APP_1_4958).
attacker((MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,APP_1_4958)).

28. The message (MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,APP_1_4958) that the
     attacker may have by 27 may be received at input {31}.
So the message ((MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)),sharedEncrypt(APP_5071,HKDF
     (exp(xE_U_4956,y_5069),hash(((MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,
     APP_1_4958),(MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)))),identifyPreSharedKey(
     xKID_4957,PSKs[])),hash(((MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,
     APP_1_4958),(MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)))))) may be sent to the
     attacker at output {41}.
attacker(((MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)),sharedEncrypt(APP_5071,HKDF(exp(
     xE_U_4956,y_5069),hash(((MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,APP_1_4958
     ),(MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)))),identifyPreSharedKey(xKID_4957,PSKs
     [])),hash(((MSG_TYPE_1[],xS_U_4954,N_U_4955,xE_U_4956,xKID_4957,APP_1_4958),(MSG_TYPE_2
     [],xS_U_4954,S,N_5070,exp(g,y_5069)))))).

29. By 28, the attacker may know ((MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)),
     sharedEncrypt(APP_5071,HKDF(exp(xE_U_4956,y_5069),hash(((MSG_TYPE_1[],xS_U_4954,
     N_U_4955,xE_U_4956,xKID_4957,APP_1_4958),(MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)
     ))),identifyPreSharedKey(xKID_4957,PSKs[])),hash(((MSG_TYPE_1[],xS_U_4954,N_U_4955,
     xE_U_4956,xKID_4957,APP_1_4958),(MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)))))).
Using the function 1-proj-2-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_4954,S,N_5070,
     exp(g,y_5069)).
attacker((MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069))).

30. By 29, the attacker may know (MSG_TYPE_2[],xS_U_4954,S,N_5070,exp(g,y_5069)).
Using the function 3-proj-5-tuple the attacker may obtain S.
attacker(S).

31. The attacker initially knows MSG_TYPE_2[].
attacker(MSG_TYPE_2[]).

32. By 31, the attacker may know MSG_TYPE_2[].
By 17, the attacker may know xS_U_5055.
By 30, the attacker may know S.
By 21, the attacker may know N.
By 10, the attacker may know exp(g,y_5069).
Using the function 5-tuple the attacker may obtain (MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069
     )).
attacker((MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))).
```

```
33. By 18, the attacker may know (MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,
     APP_1_5059).
By 32, the attacker may know (MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)).
Using the function 2-tuple the attacker may obtain ((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,
     y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))).
attacker(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2
     [],xS_U_5055,S,N,exp(g,y_5069)))).

34. By 33, the attacker may know ((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,
     APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))).
Using the function hash the attacker may obtain hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(
     g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))).
attacker(hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(
     MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))))).

35. By 34, the attacker may know hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),
     xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))).
So the attacker may know hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,
     APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))) in phase 1.
attacker_p1(hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(
     MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))))).

36. The message PSKs[] may be sent to the attacker in phase 1 at output {49}.
attacker_p1(PSKs[]).

37. We assume as hypothesis that
attacker_p1(xKID_5058).

38. By 37, the attacker may know xKID_5058 in phase 1.
By 36, the attacker may know PSKs[] in phase 1.
Using the function identifyPreSharedKey the attacker may obtain identifyPreSharedKey(
     xKID_5058,PSKs[]) in phase 1.
attacker_p1(identifyPreSharedKey(xKID_5058,PSKs[])).

39. By 10, the attacker may know exp(g,y_5069).
By 13, the attacker may know y_4991.
Using the function exp the attacker may obtain exp(exp(g,y_4991),y_5069).
attacker(exp(exp(g,y_4991),y_5069)).

40. By 39, the attacker may know exp(exp(g,y_4991),y_5069).
So the attacker may know exp(exp(g,y_4991),y_5069) in phase 1.
attacker_p1(exp(exp(g,y_4991),y_5069)).

41. By 40, the attacker may know exp(exp(g,y_4991),y_5069) in phase 1.
By 35, the attacker may know hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058
     ,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))) in phase 1.
```

```
By 38, the attacker may know identifyPreSharedKey(xKID_5058,PSKs[]) in phase 1.
Using the function HKDF the attacker may obtain HKDF(exp(exp(g,y_4991),y_5069),hash(((
     MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],
     xS_U_5055,S,N,exp(g,y_5069)))),identifyPreSharedKey(xKID_5058,PSKs[])) in phase 1.
attacker_p1(HKDF(exp(exp(g,y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,
     y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))),
     identifyPreSharedKey(xKID_5058,PSKs[]))).

42. By 19, the attacker may know ((MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)),sharedEncrypt(
     APP,HKDF(exp(exp(g,y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991)
     ,xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))),
     identifyPreSharedKey(xKID_5058,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,
     y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))))))).
Using the function 2-proj-2-tuple the attacker may obtain sharedEncrypt(APP,HKDF(exp(exp(g,
     y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,
     APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))),identifyPreSharedKey(xKID_5058
     ,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(
     MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))))).
attacker(sharedEncrypt(APP,HKDF(exp(exp(g,y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,
     N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))
     )),identifyPreSharedKey(xKID_5058,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g
     ,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))))))).

43. By 42, the attacker may know sharedEncrypt(APP,HKDF(exp(exp(g,y_4991),y_5069),hash(((
     MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],
     xS_U_5055,S,N,exp(g,y_5069)))),identifyPreSharedKey(xKID_5058,PSKs[])),hash(((
     MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],
     xS_U_5055,S,N,exp(g,y_5069))))).
So the attacker may know sharedEncrypt(APP,HKDF(exp(exp(g,y_4991),y_5069),hash(((MSG_TYPE_1
     [],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,
     exp(g,y_5069)))),identifyPreSharedKey(xKID_5058,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,
     N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))
     ))) in phase 1.
attacker_p1(sharedEncrypt(APP,HKDF(exp(exp(g,y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,
     N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))
     )),identifyPreSharedKey(xKID_5058,PSKs[])),hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g
     ,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))))))).

44. By 43, the attacker may know sharedEncrypt(APP,HKDF(exp(exp(g,y_4991),y_5069),hash(((
     MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],
     xS_U_5055,S,N,exp(g,y_5069)))),identifyPreSharedKey(xKID_5058,PSKs[])),hash(((
     MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],
     xS_U_5055,S,N,exp(g,y_5069))))) in phase 1.
By 41, the attacker may know HKDF(exp(exp(g,y_4991),y_5069),hash(((MSG_TYPE_1[],xS_U_5055,
     N_U_5056,exp(g,y_4991),xKID_5058,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069))
     )),identifyPreSharedKey(xKID_5058,PSKs[])) in phase 1.
```

```
By 35, the attacker may know hash(((MSG_TYPE_1[],xS_U_5055,N_U_5056,exp(g,y_4991),xKID_5058
     ,APP_1_5059),(MSG_TYPE_2[],xS_U_5055,S,N,exp(g,y_5069)))) in phase 1.
Using the function sharedDecrypt the attacker may obtain APP in phase 1.
attacker_p1(APP).


Unified xS_U_5055 with xS_U_4954
Unified N_U_5056 with N_U_4955
Unified xE_U_4956 with exp(g,y_4991)
Unified xKID_5058 with xKID_4957
Unified APP_1_5059 with APP_1_4958
Unified xS_U_4999 with xS_U_4954
Unified N_U_5000 with N_U_4955
Unified xE_U_5001 with exp(g,y_4991)
Unified xKID_5002 with xKID_4957
Unified APP_1_5003 with APP_1_4958
Iterating unifyDerivation.
Fixpoint reached: nothing more to unify.
The clause after unifyDerivation is
attacker(APP_1_5263) && attacker(xKID_5262) && attacker(N_U_5260) && attacker(xS_U_5259) &&
      attacker(y_5261) && attacker_p1(xKID_5262) -> attacker_p1(APP_67[message_1_64 = (
     MSG_TYPE_1[],xS_U_5259,N_U_5260,exp(g,y_5261),xKID_5262,APP_1_5263),!1 = @sid_5264])
This clause still contradicts the query.
A more detailed output of the traces is available with
  set traceDisplay = long.

new U: host creating U_5354 at {1}

new V: host creating V_5355 at {2}

new PSKs: bitstring creating PSKs_5275 at {3}

new y_63: exponent creating y_5272 at {28} in copy a_5270

new S_V: bitstring creating S_V_5273 at {30} in copy a_5270

in(c, (MSG_TYPE_1,a_5265,a_5266,exp(g,a_5267),a_5268,a_5269)) at {31} in copy a_5270

new N_V: nonce creating N_V_5274 at {33} in copy a_5270

new APP_67: bitstring creating APP_5271 at {34} in copy a_5270

out(c, ((~M_5492,~M_5493,~M_5494,~M_5495,~M_5496),~M_5491)) with ~M_5492 = MSG_TYPE_2, ~
     M_5493 = a_5265, ~M_5494 = S_V_5273, ~M_5495 = N_V_5274, ~M_5496 = exp(g,y_5272), ~
     M_5491 = sharedEncrypt(APP_5271,HKDF(exp(exp(g,a_5267),y_5272),hash(((MSG_TYPE_1,a_5265
     ,a_5266,exp(g,a_5267),a_5268,a_5269),(MSG_TYPE_2,a_5265,S_V_5273,N_V_5274,exp(g,y_5272)
```

```
     ))),identifyPreSharedKey(a_5268,PSKs_5275)),hash(((MSG_TYPE_1,a_5265,a_5266,exp(g,
     a_5267),a_5268,a_5269),(MSG_TYPE_2,a_5265,S_V_5273,N_V_5274,exp(g,y_5272))))) at {41}
     in copy a_5270

out(c, ~M_5500) with ~M_5500 = PSKs_5275 at {49}

The attacker has the message sharedDecrypt(~M_5491,HKDF(exp(~M_5496,a_5267),hash(((
     MSG_TYPE_1,a_5265,a_5266,exp(g,a_5267),a_5268,a_5269),(MSG_TYPE_2,a_5265,~M_5494,~
     M_5495,~M_5496))),identifyPreSharedKey(a_5268,~M_5500)),hash(((MSG_TYPE_1,a_5265,a_5266
     ,exp(g,a_5267),a_5268,a_5269),(MSG_TYPE_2,a_5265,~M_5494,~M_5495,~M_5496)))) = APP_5271
      in phase 1.
A trace has been found.
The previous trace falsifies the query, because the query is
simple and the trace corresponds to the derivation.
RESULT not attacker_p1(APP_67[message_1_64 = v_1316,!1 = v_1317]) is false.
Starting query not attacker_p1(APP_61[message_2_52 = v_1318,!1 = v_1319])
RESULT not attacker_p1(APP_61[message_2_52 = v_1318,!1 = v_1319]) is true.
```

C

Improvements

Asymmetric injective agreement without encrypted APP_2

```
Linear part:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Completing equations...
Completed equations:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new W: host;
{4}new skU: skey;
{5}new skV: skey;
{6}new skW: skey;
{7}let pkU: pkey = pk(skU) in
{8}let pkV: pkey = pk(skV) in
{9}let pkW: pkey = pk(skW) in
{10}let pkIdU: pkID = identifyPK(skU,pk(skU)) in
{11}let pkIdV: pkID = identifyPK(skV,pk(skV)) in
{12}let pkIdW: pkID = identifyPK(skW,pk(skW)) in
{13}out(c, pkU);
{14}out(c, pkV);
{15}out(c, pkW);
(
    {16}!
    {17}new x_63: exponent;
    {18}let E_U: G = exp(g,x_63) in
    {19}new S_U: bitstring;
```

```
    {20}event startInitiator(U,V,S_U);
    {21}new N_U: nonce;
    {22}new APP_64: bitstring;
    {23}let message_1_65: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,APP_64) in
    {24}out(c, message_1_65);
    {25}in(c, message_2_66: bitstring);
    {26}let (data_2_67: bitstring,COSE_ENC_2_68: bitstring) = message_2_66 in
    {27}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,N_V: nonce,xE_V: G,xAPP_2_69: bitstring) =
        data_2_67 in
    {28}let aad_2_70: bitstring = hash((message_1_65,data_2_67)) in
    {29}let K: G = exp(xE_V,x_63) in
    {30}let K_2_71: derivedKey = HKDF(K,aad_2_70) in
    {31}let signature_2_72: bitstring = decrypt(COSE_ENC_2_68,K_2_71,aad_2_70) in
    {32}let (=pkIdV,=aad_2_70) = verify(signature_2_72,pkV) in
    {33}new APP_73: bitstring;
    {34}let data_3_74: bitstring = (MSG_TYPE_3,xS_V) in
    {35}let aad_3_75: bitstring = hash((hash((message_1_65,message_2_66)),data_3_74)) in
    {36}let signature_3_76: bitstring = sign((identifyPK(skU,pkU),aad_3_75,APP_73),skU) in
    {37}let K_3_77: derivedKey = HKDF(K,aad_3_75) in
    {38}let COSE_ENC_3_78: bitstring = encrypt(signature_3_76,K_3_77,aad_3_75) in
    {39}let message_3_79: bitstring = (data_3_74,COSE_ENC_3_78) in
    {40}out(c, message_3_79);
    {41}event endInitiator(U,V,S_U,xS_V)
) | (
    {42}!
    {43}new y_80: exponent;
    {44}let E_V: G = exp(g,y_80) in
    {45}new S_V: bitstring;
    {46}event startResponder(U,V,S_V);
    {47}in(c, message_1_81: bitstring);
    {48}let (xMSG_TYPE_1_82: bitstring,xS_U: bitstring,xN_U: nonce,xE_U: G,APP_1_83:
        bitstring) = message_1_81 in
    {49}new N_V_84: nonce;
    {50}new APP_85: bitstring;
    {51}let data_2_86: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V_84,E_V,APP_85) in
    {52}let aad_2_87: bitstring = hash((message_1_81,data_2_86)) in
    {53}let signature_2_88: bitstring = sign((identifyPK(skV,pkV),aad_2_87),skV) in
    {54}let K_89: G = exp(xE_U,y_80) in
    {55}let K_2_90: derivedKey = HKDF(K_89,aad_2_87) in
    {56}let COSE_ENC_2_91: bitstring = encrypt(signature_2_88,K_2_90,aad_2_87) in
    {57}let message_2_92: bitstring = (data_2_86,COSE_ENC_2_91) in
    {58}out(c, message_2_92);
    {59}in(c, message_3_93: bitstring);
    {60}let (data_3_94: bitstring,COSE_ENC_3_95: bitstring) = message_3_93 in
    {61}let (=MSG_TYPE_3,=S_V) = data_3_94 in
    {62}let aad_3_96: bitstring = hash((hash((message_1_81,message_2_92)),data_3_94)) in
```

```
    {63}let K_3_97: derivedKey = HKDF(K_89,aad_3_96) in
    {64}let signature_3_98: bitstring = decrypt(COSE_ENC_3_95,K_3_97,aad_3_96) in
    {65}let (=pkIdU,=aad_3_96,APP_3_99: bitstring) = verify(signature_3_98,pkU) in
    {66}event endResponder(U,V,xS_U,S_V)
) | (
    {67}!
    {68}new x_100: exponent;
    {69}let E_U_101: G = exp(g,x_100) in
    {70}new S_U_102: bitstring;
    {71}event startInitiator(U,W,S_U_102);
    {72}new N_U_103: nonce;
    {73}new APP_104: bitstring;
    {74}let message_1_105: bitstring = (MSG_TYPE_1,S_U_102,N_U_103,E_U_101,APP_104) in
    {75}out(c, message_1_105);
    {76}in(c, message_2_106: bitstring);
    {77}let (data_2_107: bitstring,COSE_ENC_2_108: bitstring) = message_2_106 in
    {78}let (=MSG_TYPE_2,=S_U_102,xS_V_109: bitstring,N_V_110: nonce,xE_V_111: G,xAPP_2_112:
          bitstring) = data_2_107 in
    {79}let aad_2_113: bitstring = hash((message_1_105,data_2_107)) in
    {80}let K_114: G = exp(xE_V_111,x_100) in
    {81}let K_2_115: derivedKey = HKDF(K_114,aad_2_113) in
    {82}let signature_2_116: bitstring = decrypt(COSE_ENC_2_108,K_2_115,aad_2_113) in
    {83}let (=pkIdW,=aad_2_113) = verify(signature_2_116,pkW) in
    {84}new APP_117: bitstring;
    {85}let data_3_118: bitstring = (MSG_TYPE_3,xS_V_109) in
    {86}let aad_3_119: bitstring = hash((hash((message_1_105,message_2_106)),data_3_118)) in
    {87}let signature_3_120: bitstring = sign((identifyPK(skU,pkU),aad_3_119,APP_117),skU)
         in
    {88}let K_3_121: derivedKey = HKDF(K_114,aad_3_119) in
    {89}let COSE_ENC_3_122: bitstring = encrypt(signature_3_120,K_3_121,aad_3_119) in
    {90}let message_3_123: bitstring = (data_3_118,COSE_ENC_3_122) in
    {91}out(c, message_3_123);
    {92}event endInitiator(U,W,S_U_102,xS_V_109)
) | (
    {93}!
    {94}new y_124: exponent;
    {95}let E_V_125: G = exp(g,y_124) in
    {96}new S_V_126: bitstring;
    {97}event startResponder(U,W,S_V_126);
    {98}in(c, message_1_127: bitstring);
    {99}let (xMSG_TYPE_1_128: bitstring,xS_U_129: bitstring,xN_U_130: nonce,xE_U_131: G,
         APP_1_132: bitstring) = message_1_127 in
    {100}new N_V_133: nonce;
    {101}new APP_134: bitstring;
    {102}let data_2_135: bitstring = (MSG_TYPE_2,xS_U_129,S_V_126,N_V_133,E_V_125,APP_134)
         in
```

```
    {103}let aad_2_136: bitstring = hash((message_1_127,data_2_135)) in
    {104}let signature_2_137: bitstring = sign((identifyPK(skW,pkW),aad_2_136),skW) in
    {105}let K_138: G = exp(xE_U_131,y_124) in
    {106}let K_2_139: derivedKey = HKDF(K_138,aad_2_136) in
    {107}let COSE_ENC_2_140: bitstring = encrypt(signature_2_137,K_2_139,aad_2_136) in
    {108}let message_2_141: bitstring = (data_2_135,COSE_ENC_2_140) in
    {109}out(c, message_2_141);
    {110}in(c, message_3_142: bitstring);
    {111}let (data_3_143: bitstring,COSE_ENC_3_144: bitstring) = message_3_142 in
    {112}let (=MSG_TYPE_3,=S_V_126) = data_3_143 in
    {113}let aad_3_145: bitstring = hash((hash((message_1_127,message_2_141)),data_3_143))
         in
    {114}let K_3_146: derivedKey = HKDF(K_138,aad_3_145) in
    {115}let signature_3_147: bitstring = decrypt(COSE_ENC_3_144,K_3_146,aad_3_145) in
    {116}let (=pkIdU,=aad_3_145,APP_3_148: bitstring) = verify(signature_3_147,pkU) in
    {117}event endResponder(U,W,xS_U_129,S_V_126)
)

-- Query event(endResponder(U_149,V_150,S_U_151,S_V_152)) ==> event(startInitiator(U_149,
     V_150,S_U_151))
Completing...
200 rules inserted. The rule base contains 156 rules. 38 rules in the queue.
400 rules inserted. The rule base contains 220 rules. 22 rules in the queue.
Starting query event(endResponder(U_149,V_150,S_U_151,S_V_152)) ==> event(startInitiator(
     U_149,V_150,S_U_151))
goal reachable: begin(startInitiator(U[],W[],S_U_102[!1 = @sid_10613])) -> end(endResponder
     (U[],W[],S_U_102[!1 = @sid_10613],S_V_126[!1 = @sid_10614]))
goal reachable: begin(startInitiator(U[],V[],S_U[!1 = @sid_10617])) -> end(endResponder(U
     [],V[],S_U[!1 = @sid_10617],S_V[!1 = @sid_10618]))
RESULT event(endResponder(U_149,V_150,S_U_151,S_V_152)) ==> event(startInitiator(U_149,
     V_150,S_U_151)) is true.
-- Query event(endInitiator(U_10625,V_10626,S_U_10627,S_V_10628)) ==> event(startResponder(
     U_10625,V_10626,S_V_10628))
Completing...
200 rules inserted. The rule base contains 158 rules. 36 rules in the queue.
400 rules inserted. The rule base contains 231 rules. 4 rules in the queue.
Starting query event(endInitiator(U_10625,V_10626,S_U_10627,S_V_10628)) ==> event(
     startResponder(U_10625,V_10626,S_V_10628))
goal reachable: begin(startResponder(U[],W[],S_V_126[!1 = @sid_20279])) -> end(endInitiator
     (U[],W[],S_U_102[!1 = @sid_20280],S_V_126[!1 = @sid_20279]))
goal reachable: begin(startResponder(U[],V[],S_V[!1 = @sid_20283])) -> end(endInitiator(U
     [],V[],S_U[!1 = @sid_20284],S_V[!1 = @sid_20283]))
RESULT event(endInitiator(U_10625,V_10626,S_U_10627,S_V_10628)) ==> event(startResponder(
     U_10625,V_10626,S_V_10628)) is true.
```

Asymmetric confidentiality without encrypted APP_2

```
Linear part:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Completing equations...
Completed equations:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new skU: skey;
{4}new skV: skey;
{5}let pkU: pkey = pk(skU) in
{6}let pkV: pkey = pk(skV) in
{7}let pkIdU: pkID = identifyPK(skU,pk(skU)) in
{8}let pkIdV: pkID = identifyPK(skV,pk(skV)) in
{9}out(c, pkU);
{10}out(c, pkV);
(
    {11}!
    {12}new x_63: exponent;
    {13}let E_U: G = exp(g,x_63) in
    {14}new S_U: bitstring;
    {15}new N_U: nonce;
    {16}new APP_64: bitstring;
    {17}let message_1_65: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,APP_64) in
    {18}out(c, message_1_65);
    {19}in(c, message_2_66: bitstring);
    {20}let (data_2_67: bitstring,COSE_ENC_2_68: bitstring) = message_2_66 in
    {21}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,N_V: nonce,xE_V: G,xAPP_2_69: bitstring) =
        data_2_67 in
    {22}let aad_2_70: bitstring = hash((message_1_65,data_2_67)) in
    {23}let K: G = exp(xE_V,x_63) in
    {24}let K_2_71: derivedKey = HKDF(K,aad_2_70) in
    {25}let signature_2_72: bitstring = decrypt(COSE_ENC_2_68,K_2_71,aad_2_70) in
    {26}let (=pkIdV,=aad_2_70) = verify(signature_2_72,pkV) in
    {27}new APP_73: bitstring;
    {28}let data_3_74: bitstring = (MSG_TYPE_3,xS_V) in
    {29}let aad_3_75: bitstring = hash((hash((message_1_65,message_2_66)),data_3_74)) in
    {30}let signature_3_76: bitstring = sign((identifyPK(skU,pkU),aad_3_75,APP_73),skU) in
```

```
    {31}let K_3_77: derivedKey = HKDF(K,aad_3_75) in
    {32}let COSE_ENC_3_78: bitstring = encrypt(signature_3_76,K_3_77,aad_3_75) in
    {33}let message_3_79: bitstring = (data_3_74,COSE_ENC_3_78) in
    {34}out(c, message_3_79)
) | (
    {35}!
    {36}new y_80: exponent;
    {37}let E_V: G = exp(g,y_80) in
    {38}new S_V: bitstring;
    {39}in(c, message_1_81: bitstring);
    {40}let (xMSG_TYPE_1_82: bitstring,xS_U: bitstring,xN_U: nonce,xE_U: G,APP_1_83:
        bitstring) = message_1_81 in
    {41}new N_V_84: nonce;
    {42}new APP_85: bitstring;
    {43}let data_2_86: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V_84,E_V,APP_85) in
    {44}let aad_2_87: bitstring = hash((message_1_81,data_2_86)) in
    {45}let signature_2_88: bitstring = sign((identifyPK(skV,pkV),aad_2_87),skV) in
    {46}let K_89: G = exp(xE_U,y_80) in
    {47}let K_2_90: derivedKey = HKDF(K_89,aad_2_87) in
    {48}let COSE_ENC_2_91: bitstring = encrypt(signature_2_88,K_2_90,aad_2_87) in
    {49}let message_2_92: bitstring = (data_2_86,COSE_ENC_2_91) in
    {50}out(c, message_2_92);
    {51}in(c, message_3_93: bitstring);
    {52}let (data_3_94: bitstring,COSE_ENC_3_95: bitstring) = message_3_93 in
    {53}let (=MSG_TYPE_3,=S_V) = data_3_94 in
    {54}let aad_3_96: bitstring = hash((hash((message_1_81,message_2_92)),data_3_94)) in
    {55}let K_3_97: derivedKey = HKDF(K_89,aad_3_96) in
    {56}let signature_3_98: bitstring = decrypt(COSE_ENC_3_95,K_3_97,aad_3_96) in
    {57}let (=pkIdU,=aad_3_96,APP_3_99: bitstring) = verify(signature_3_98,pkU) in
    0
)

-- Query not attacker(APP_73[message_2_66 = v_1314,!1 = v_1315])
Completing...
Starting query not attacker(APP_73[message_2_66 = v_1314,!1 = v_1315])
RESULT not attacker(APP_73[message_2_66 = v_1314,!1 = v_1315]) is true.
```

Asymmetric forward secrecy without encrypted APP_2

```
Linear part:
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Completing equations...
Completed equations:
```

```
exp(exp(g,x_24),y_25) = exp(exp(g,y_25),x_24)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new skU: skey;
{4}new skV: skey;
{5}let pkU: pkey = pk(skU) in
{6}let pkV: pkey = pk(skV) in
{7}let pkIdU: pkID = identifyPK(skU,pk(skU)) in
{8}let pkIdV: pkID = identifyPK(skV,pk(skV)) in
{9}out(c, pkU);
{10}out(c, pkV);
(
    {11}!
    {12}new x_63: exponent;
    {13}let E_U: G = exp(g,x_63) in
    {14}new S_U: bitstring;
    {15}new N_U: nonce;
    {16}new APP_64: bitstring;
    {17}let message_1_65: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,APP_64) in
    {18}out(c, message_1_65);
    {19}in(c, message_2_66: bitstring);
    {20}let (data_2_67: bitstring,COSE_ENC_2_68: bitstring) = message_2_66 in
    {21}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,N_V: nonce,xE_V: G,xAPP_2_69: bitstring) =
        data_2_67 in
    {22}let aad_2_70: bitstring = hash((message_1_65,data_2_67)) in
    {23}let K: G = exp(xE_V,x_63) in
    {24}let K_2_71: derivedKey = HKDF(K,aad_2_70) in
    {25}let signature_2_72: bitstring = decrypt(COSE_ENC_2_68,K_2_71,aad_2_70) in
    {26}let (=pkIdV,=aad_2_70) = verify(signature_2_72,pkV) in
    {27}new APP_73: bitstring;
    {28}let data_3_74: bitstring = (MSG_TYPE_3,xS_V) in
    {29}let aad_3_75: bitstring = hash((hash((message_1_65,message_2_66)),data_3_74)) in
    {30}let signature_3_76: bitstring = sign((identifyPK(skU,pkU),aad_3_75,APP_73),skU) in
    {31}let K_3_77: derivedKey = HKDF(K,aad_3_75) in
    {32}let COSE_ENC_3_78: bitstring = encrypt(signature_3_76,K_3_77,aad_3_75) in
    {33}let message_3_79: bitstring = (data_3_74,COSE_ENC_3_78) in
    {34}out(c, message_3_79)
) | (
    {35}!
    {36}new y_80: exponent;
    {37}let E_V: G = exp(g,y_80) in
    {38}new S_V: bitstring;
```

```
    {39}in(c, message_1_81: bitstring);
    {40}let (xMSG_TYPE_1_82: bitstring,xS_U: bitstring,xN_U: nonce,xE_U: G,APP_1_83:
        bitstring) = message_1_81 in
    {41}new N_V_84: nonce;
    {42}new APP_85: bitstring;
    {43}let data_2_86: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V_84,E_V,APP_85) in
    {44}let aad_2_87: bitstring = hash((message_1_81,data_2_86)) in
    {45}let signature_2_88: bitstring = sign((identifyPK(skV,pkV),aad_2_87),skV) in
    {46}let K_89: G = exp(xE_U,y_80) in
    {47}let K_2_90: derivedKey = HKDF(K_89,aad_2_87) in
    {48}let COSE_ENC_2_91: bitstring = encrypt(signature_2_88,K_2_90,aad_2_87) in
    {49}let message_2_92: bitstring = (data_2_86,COSE_ENC_2_91) in
    {50}out(c, message_2_92);
    {51}in(c, message_3_93: bitstring);
    {52}let (data_3_94: bitstring,COSE_ENC_3_95: bitstring) = message_3_93 in
    {53}let (=MSG_TYPE_3,=S_V) = data_3_94 in
    {54}let aad_3_96: bitstring = hash((hash((message_1_81,message_2_92)),data_3_94)) in
    {55}let K_3_97: derivedKey = HKDF(K_89,aad_3_96) in
    {56}let signature_3_98: bitstring = decrypt(COSE_ENC_3_95,K_3_97,aad_3_96) in
    {57}let (=pkIdU,=aad_3_96,APP_3_99: bitstring) = verify(signature_3_98,pkU) in
    0
) | (
    {58}phase 1;
    {59}out(c, skV);
    {60}out(c, skU)
)

-- Query not attacker_p1(APP_73[message_2_66 = v_1500,!1 = v_1501])
Completing...
200 rules inserted. The rule base contains 163 rules. 33 rules in the queue.
400 rules inserted. The rule base contains 201 rules. 9 rules in the queue.
Starting query not attacker_p1(APP_73[message_2_66 = v_1500,!1 = v_1501])
RESULT not attacker_p1(APP_73[message_2_66 = v_1500,!1 = v_1501]) is true.
```

Symmetric injective agreement without encrypted APP_2

```
Linear part:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Completing equations...
Completed equations:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Convergent part:
Completing equations...
```

```
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new W: host;
{4}new PSKs_UV: bitstring;
{5}new PSKs_UW: bitstring;
(
    {6}!
    {7}new x_49: exponent;
    {8}let E_U: G = exp(g,x_49) in
    {9}new S_U: bitstring;
    {10}event startInitiator(U,V,S_U);
    {11}new N_U: nonce;
    {12}new KID: bitstring;
    {13}new APP_50: bitstring;
    {14}let PSK: preSharedkey = identifyPreSharedKey(KID,PSKs_UV) in
    {15}let message_1_51: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,KID,APP_50) in
    {16}out(c, message_1_51);
    {17}in(c, message_2_52: bitstring);
    {18}let (data_2_53: bitstring,COSE_ENC_2_54: bitstring) = message_2_52 in
    {19}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,xN_V: nonce,xE_V: G,xAPP_2_55: bitstring) =
        data_2_53 in
    {20}let aad_2_56: bitstring = hash((message_1_51,data_2_53)) in
    {21}let K: G = exp(xE_V,x_49) in
    {22}let K_2_57: derivedKey = HKDF(K,aad_2_56,PSK) in
    {23}let () = sharedDecrypt(COSE_ENC_2_54,K_2_57,aad_2_56) in
    {24}let data_3_58: bitstring = (MSG_TYPE_3,xS_V) in
    {25}let aad_3_59: bitstring = hash((hash((message_1_51,message_2_52)),data_3_58)) in
    {26}let K_3_60: derivedKey = HKDF(K,aad_3_59,PSK) in
    {27}new APP_61: bitstring;
    {28}let message_3_62: bitstring = (data_3_58,sharedEncrypt(APP_61,K_3_60,aad_3_59)) in
    {29}out(c, message_3_62);
    {30}event endInitiator(U,V,S_U,xS_V)
) | (
    {31}!
    {32}new y_63: exponent;
    {33}let E_V: G = exp(g,y_63) in
    {34}new S_V: bitstring;
    {35}event startResponder(U,V,S_V);
    {36}in(c, message_1_64: bitstring);
    {37}let (=MSG_TYPE_1,xS_U: bitstring,N_U_65: nonce,xE_U: G,xKID: bitstring,APP_1_66:
        bitstring) = message_1_64 in
    {38}new N_V: nonce;
    {39}new APP_67: bitstring;
    {40}let PSK_68: preSharedkey = identifyPreSharedKey(xKID,PSKs_UV) in
```

```
    {41}let data_2_69: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V,E_V,APP_67) in
    {42}let aad_2_70: bitstring = hash((message_1_64,data_2_69)) in
    {43}let K_71: G = exp(xE_U,y_63) in
    {44}let K_2_72: derivedKey = HKDF(K_71,aad_2_70,PSK_68) in
    {45}let message_2_73: bitstring = (data_2_69,sharedEncrypt((),K_2_72,aad_2_70)) in
    {46}out(c, message_2_73);
    {47}in(c, message_3_74: bitstring);
    {48}let (data_3_75: bitstring,COSE_ENC_3_76: bitstring) = message_3_74 in
    {49}let (=MSG_TYPE_3,=S_V) = data_3_75 in
    {50}let aad_3_77: bitstring = hash((hash((message_1_64,message_2_73)),data_3_75)) in
    {51}let K_3_78: derivedKey = HKDF(K_71,aad_3_77,PSK_68) in
    {52}let APP_3_79: bitstring = sharedDecrypt(COSE_ENC_3_76,K_3_78,aad_3_77) in
    {53}event endResponder(U,V,S_V,xS_U)
) | (
    {54}!
    {55}new x_80: exponent;
    {56}let E_U_81: G = exp(g,x_80) in
    {57}new S_U_82: bitstring;
    {58}event startInitiator(U,W,S_U_82);
    {59}new N_U_83: nonce;
    {60}new KID_84: bitstring;
    {61}new APP_85: bitstring;
    {62}let PSK_86: preSharedkey = identifyPreSharedKey(KID_84,PSKs_UW) in
    {63}let message_1_87: bitstring = (MSG_TYPE_1,S_U_82,N_U_83,E_U_81,KID_84,APP_85) in
    {64}out(c, message_1_87);
    {65}in(c, message_2_88: bitstring);
    {66}let (data_2_89: bitstring,COSE_ENC_2_90: bitstring) = message_2_88 in
    {67}let (=MSG_TYPE_2,=S_U_82,xS_V_91: bitstring,xN_V_92: nonce,xE_V_93: G,xAPP_2_94:
        bitstring) = data_2_89 in
    {68}let aad_2_95: bitstring = hash((message_1_87,data_2_89)) in
    {69}let K_96: G = exp(xE_V_93,x_80) in
    {70}let K_2_97: derivedKey = HKDF(K_96,aad_2_95,PSK_86) in
    {71}let () = sharedDecrypt(COSE_ENC_2_90,K_2_97,aad_2_95) in
    {72}let data_3_98: bitstring = (MSG_TYPE_3,xS_V_91) in
    {73}let aad_3_99: bitstring = hash((hash((message_1_87,message_2_88)),data_3_98)) in
    {74}let K_3_100: derivedKey = HKDF(K_96,aad_3_99,PSK_86) in
    {75}new APP_101: bitstring;
    {76}let message_3_102: bitstring = (data_3_98,sharedEncrypt(APP_101,K_3_100,aad_3_99))
        in
    {77}out(c, message_3_102);
    {78}event endInitiator(U,W,S_U_82,xS_V_91)
) | (
    {79}!
    {80}new y_103: exponent;
    {81}let E_V_104: G = exp(g,y_103) in
    {82}new S_V_105: bitstring;
```

```
    {83}event startResponder(U,W,S_V_105);
    {84}in(c, message_1_106: bitstring);
    {85}let (=MSG_TYPE_1,xS_U_107: bitstring,N_U_108: nonce,xE_U_109: G,xKID_110: bitstring,
         APP_1_111: bitstring) = message_1_106 in
    {86}new N_V_112: nonce;
    {87}new APP_113: bitstring;
    {88}let PSK_114: preSharedkey = identifyPreSharedKey(xKID_110,PSKs_UW) in
    {89}let data_2_115: bitstring = (MSG_TYPE_2,xS_U_107,S_V_105,N_V_112,E_V_104,APP_113) in
    {90}let aad_2_116: bitstring = hash((message_1_106,data_2_115)) in
    {91}let K_117: G = exp(xE_U_109,y_103) in
    {92}let K_2_118: derivedKey = HKDF(K_117,aad_2_116,PSK_114) in
    {93}let message_2_119: bitstring = (data_2_115,sharedEncrypt((),K_2_118,aad_2_116)) in
    {94}out(c, message_2_119);
    {95}in(c, message_3_120: bitstring);
    {96}let (data_3_121: bitstring,COSE_ENC_3_122: bitstring) = message_3_120 in
    {97}let (=MSG_TYPE_3,=S_V_105) = data_3_121 in
    {98}let aad_3_123: bitstring = hash((hash((message_1_106,message_2_119)),data_3_121)) in
    {99}let K_3_124: derivedKey = HKDF(K_117,aad_3_123,PSK_114) in
    {100}let APP_3_125: bitstring = sharedDecrypt(COSE_ENC_3_122,K_3_124,aad_3_123) in
    {101}event endResponder(U,W,S_V_105,xS_U_107)
)

-- Query event(endResponder(U_126,V_127,S_V_129,S_U_128)) ==> event(startInitiator(U_126,
     V_127,S_U_128))
Completing...
200 rules inserted. The rule base contains 121 rules. 42 rules in the queue.
Starting query event(endResponder(U_126,V_127,S_V_129,S_U_128)) ==> event(startInitiator(
     U_126,V_127,S_U_128))
goal reachable: begin(startInitiator(U[],W[],S_U_82[!1 = @sid_7241])) -> end(endResponder(U
     [],W[],S_V_105[!1 = @sid_7242],S_U_82[!1 = @sid_7241]))
goal reachable: begin(startInitiator(U[],V[],S_U[!1 = @sid_7245])) -> end(endResponder(U[],
     V[],S_V[!1 = @sid_7246],S_U[!1 = @sid_7245]))
RESULT event(endResponder(U_126,V_127,S_V_129,S_U_128)) ==> event(startInitiator(U_126,
     V_127,S_U_128)) is true.
-- Query event(endInitiator(U_7253,V_7254,S_U_7255,S_V_7256)) ==> event(startResponder(
     U_7253,V_7254,S_V_7256))
Completing...
200 rules inserted. The rule base contains 134 rules. 35 rules in the queue.
Starting query event(endInitiator(U_7253,V_7254,S_U_7255,S_V_7256)) ==> event(
     startResponder(U_7253,V_7254,S_V_7256))
goal reachable: begin(startResponder(U[],W[],S_V_105[!1 = @sid_13501])) -> end(endInitiator
     (U[],W[],S_U_82[!1 = @sid_13502],S_V_105[!1 = @sid_13501]))
goal reachable: begin(startResponder(U[],V[],S_V[!1 = @sid_13505])) -> end(endInitiator(U
     [],V[],S_U[!1 = @sid_13506],S_V[!1 = @sid_13505]))
RESULT event(endInitiator(U_7253,V_7254,S_U_7255,S_V_7256)) ==> event(startResponder(U_7253
     ,V_7254,S_V_7256)) is true.
```

Symmetric confidentiality without encrypted APP_2

```
Linear part:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Completing equations...
Completed equations:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new PSKs: bitstring;
(
    {4}!
    {5}new x_49: exponent;
    {6}let E_U: G = exp(g,x_49) in
    {7}new S_U: bitstring;
    {8}new N_U: nonce;
    {9}new KID: bitstring;
    {10}new APP_50: bitstring;
    {11}let PSK: preSharedkey = identifyPreSharedKey(KID,PSKs) in
    {12}let message_1_51: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,KID,APP_50) in
    {13}out(c, message_1_51);
    {14}in(c, message_2_52: bitstring);
    {15}let (data_2_53: bitstring,COSE_ENC_2_54: bitstring) = message_2_52 in
    {16}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,xN_V: nonce,xE_V: G,xAPP_2_55: bitstring) =
        data_2_53 in
    {17}let aad_2_56: bitstring = hash((message_1_51,data_2_53)) in
    {18}let K: G = exp(xE_V,x_49) in
    {19}let K_2_57: derivedKey = HKDF(K,aad_2_56,PSK) in
    {20}let () = sharedDecrypt(COSE_ENC_2_54,K_2_57,aad_2_56) in
    {21}let data_3_58: bitstring = (MSG_TYPE_3,xS_V) in
    {22}let aad_3_59: bitstring = hash((hash((message_1_51,message_2_52)),data_3_58)) in
    {23}let K_3_60: derivedKey = HKDF(K,aad_3_59,PSK) in
    {24}new APP_61: bitstring;
    {25}let message_3_62: bitstring = (data_3_58,sharedEncrypt(APP_61,K_3_60,aad_3_59)) in
    {26}out(c, message_3_62)
) | (
    {27}!
    {28}new y_63: exponent;
    {29}let E_V: G = exp(g,y_63) in
```

```
    {30}new S_V: bitstring;
    {31}in(c, message_1_64: bitstring);
    {32}let (=MSG_TYPE_1,xS_U: bitstring,N_U_65: nonce,xE_U: G,xKID: bitstring,APP_1_66:
        bitstring) = message_1_64 in
    {33}new N_V: nonce;
    {34}new APP_67: bitstring;
    {35}let PSK_68: preSharedkey = identifyPreSharedKey(xKID,PSKs) in
    {36}let data_2_69: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V,E_V,APP_67) in
    {37}let aad_2_70: bitstring = hash((message_1_64,data_2_69)) in
    {38}let K_71: G = exp(xE_U,y_63) in
    {39}let K_2_72: derivedKey = HKDF(K_71,aad_2_70,PSK_68) in
    {40}let message_2_73: bitstring = (data_2_69,sharedEncrypt((),K_2_72,aad_2_70)) in
    {41}out(c, message_2_73);
    {42}in(c, message_3_74: bitstring);
    {43}let (data_3_75: bitstring,COSE_ENC_3_76: bitstring) = message_3_74 in
    {44}let (=MSG_TYPE_3,=S_V) = data_3_75 in
    {45}let aad_3_77: bitstring = hash((hash((message_1_64,message_2_73)),data_3_75)) in
    {46}let K_3_78: derivedKey = HKDF(K_71,aad_3_77,PSK_68) in
    {47}let APP_3_79: bitstring = sharedDecrypt(COSE_ENC_3_76,K_3_78,aad_3_77) in
    0
)

-- Query not attacker(APP_61[message_2_52 = v_1083,!1 = v_1084])
Completing...
Starting query not attacker(APP_61[message_2_52 = v_1083,!1 = v_1084])
RESULT not attacker(APP_61[message_2_52 = v_1083,!1 = v_1084]) is true.
```

Symmetric forward secrecy without encrypted APP_2

```
Linear part:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Completing equations...
Completed equations:
exp(exp(g,x_16),y_17) = exp(exp(g,y_17),x_16)
Convergent part:
Completing equations...
Completed equations:
Process:
{1}new U: host;
{2}new V: host;
{3}new PSKs: bitstring;
(
    {4}!
```

```
    {5}new x_49: exponent;
    {6}let E_U: G = exp(g,x_49) in
    {7}new S_U: bitstring;
    {8}new N_U: nonce;
    {9}new KID: bitstring;
    {10}new APP_50: bitstring;
    {11}let PSK: preSharedkey = identifyPreSharedKey(KID,PSKs) in
    {12}let message_1_51: bitstring = (MSG_TYPE_1,S_U,N_U,E_U,KID,APP_50) in
    {13}out(c, message_1_51);
    {14}in(c, message_2_52: bitstring);
    {15}let (data_2_53: bitstring,COSE_ENC_2_54: bitstring) = message_2_52 in
    {16}let (=MSG_TYPE_2,=S_U,xS_V: bitstring,xN_V: nonce,xE_V: G,xAPP_2_55: bitstring) =
        data_2_53 in
    {17}let aad_2_56: bitstring = hash((message_1_51,data_2_53)) in
    {18}let K: G = exp(xE_V,x_49) in
    {19}let K_2_57: derivedKey = HKDF(K,aad_2_56,PSK) in
    {20}let () = sharedDecrypt(COSE_ENC_2_54,K_2_57,aad_2_56) in
    {21}let data_3_58: bitstring = (MSG_TYPE_3,xS_V) in
    {22}let aad_3_59: bitstring = hash((hash((message_1_51,message_2_52)),data_3_58)) in
    {23}let K_3_60: derivedKey = HKDF(K,aad_3_59,PSK) in
    {24}new APP_61: bitstring;
    {25}let message_3_62: bitstring = (data_3_58,sharedEncrypt(APP_61,K_3_60,aad_3_59)) in
    {26}out(c, message_3_62)
) | (
    {27}!
    {28}new y_63: exponent;
    {29}let E_V: G = exp(g,y_63) in
    {30}new S_V: bitstring;
    {31}in(c, message_1_64: bitstring);
    {32}let (=MSG_TYPE_1,xS_U: bitstring,N_U_65: nonce,xE_U: G,xKID: bitstring,APP_1_66:
        bitstring) = message_1_64 in
    {33}new N_V: nonce;
    {34}new APP_67: bitstring;
    {35}let PSK_68: preSharedkey = identifyPreSharedKey(xKID,PSKs) in
    {36}let data_2_69: bitstring = (MSG_TYPE_2,xS_U,S_V,N_V,E_V,APP_67) in
    {37}let aad_2_70: bitstring = hash((message_1_64,data_2_69)) in
    {38}let K_71: G = exp(xE_U,y_63) in
    {39}let K_2_72: derivedKey = HKDF(K_71,aad_2_70,PSK_68) in
    {40}let message_2_73: bitstring = (data_2_69,sharedEncrypt((),K_2_72,aad_2_70)) in
    {41}out(c, message_2_73);
    {42}in(c, message_3_74: bitstring);
    {43}let (data_3_75: bitstring,COSE_ENC_3_76: bitstring) = message_3_74 in
    {44}let (=MSG_TYPE_3,=S_V) = data_3_75 in
    {45}let aad_3_77: bitstring = hash((hash((message_1_64,message_2_73)),data_3_75)) in
    {46}let K_3_78: derivedKey = HKDF(K_71,aad_3_77,PSK_68) in
    {47}let APP_3_79: bitstring = sharedDecrypt(COSE_ENC_3_76,K_3_78,aad_3_77) in
```

```
   0
) | (
   {48}phase 1;
   {49}out(c, PSKs)
)

-- Query not attacker_p1(APP_61[message_2_52 = v_1218,!1 = v_1219])
Completing...
200 rules inserted. The rule base contains 137 rules. 13 rules in the queue.
400 rules inserted. The rule base contains 232 rules. 29 rules in the queue.
Starting query not attacker_p1(APP_61[message_2_52 = v_1218,!1 = v_1219])
RESULT not attacker_p1(APP_61[message_2_52 = v_1218,!1 = v_1219]) is true.
```