Security analysis for IPsec with EasyCrypt

Jakob Nussbaumer
Student ID: 2435607

Born 5th August 1992 in Mödling, Austria

1 April 2019

Master Thesis Computer Science

Advisor: Dr. Michael Nüsken

Second Advisor: Prof. Dr. Michael Meier

Bonn-Aachen International Center for Information Technology

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Confirmation

I hereby confirm, that this master thesis was written independently, that none other than the specified sources and aids were used, and that any citations have been marked.

Date, Place

Signature

Contents

1. Introduction

Mathematical proofs are difficult to verify by a human and even those verifications are error prone. This issue is everlasting and problematic in the field of cryptography. For this reason automatic provers and computer-aided toolsets are on the rise to achieve irrevocable verifications. Our goal is to have a computer-checked security proof for IPsec. For TLS this was already done in a project called miTLS (see Microsoft, Inria, and the Joint Centre, 2014a). They have verified a security proof for the TLS handshake (see Bhargavan, Fournet, Kohlweiss, Pironti, Strub, and Zanella-Béguelin, 2014). Inspired by this we use the same computer-aided toolset, EasyCrypt (see Institute, Inria, and École Polytechnique, 2014), which is suited for cryptographic proofs.
We focus on a specific security aspect of the authenticated key exchange (`AKE`) model for IPsec, namely the real-or-random scenario. This means that we check if it is possible to distinguish between a randomly generated key and an exchanged key. We use the proof in progress by Heussen, Loebenberger, and Nüsken (2017), reformulate and implement it in EasyCrypt. This yields a computer checked proof for IPsec with more restrictions compared to a proof done manually. Furthermore this work presents general difficulties and limitations in the usage of EasyCrypt.

1.1. IPsec

IPsec is a network protocol suite that enables authentication and encryption of data sent over an internet protocol network. Currently it is mainly used in virtual private networks (VPN). Considering the Open System Interconnection model (OSI), the communication over the internet is divided into seven different layers. IPsec operates at the Network layer, whereas other widespread used internet security systems like TLS and SSH operate at the Application layer. Generally speaking, IPsec has an advantage at establishing site-to-site conncetions compared to TLS. This is due to the fact that IPsec is better suited to situations where a remote client should behave as if they were locally attached to the network.

1.2. Related Work

The AKE model is a security model, which is used to check if a protocol is able to establish a secure channel for communication between two parties. One of the most talked about protocols is TLS. Since it is similar to IPsec, we first look at the security proofs for TLS. Note that it was impossible to prove security by using well-established security models like AKE. This is due to the fact that the key that is authenticated is the same that is used later on for the communication. This leaks information about the key considering the AKE model. The first security analysis of the unmodified TLS was done by Jager, Kohlar, Schäge, and Schwenk (2013). They introduce the notation for authenticated and confidential channel establishment (ACCE), an extension of the AKE notation. It is needed to show practical security for TLS-like protocols. Within the ACCE model they prove the TLS-DHE ciphersuite to be secure. Building up on this, Kohlar, Schäge, and Schwenk (2013) shows that the rest of the ciphersuites are ACCE secure too. Apart from that they reformulate the ACCE security notation for sever-only authentication. They then show that this security property holds for all TLS handshake families. This is the first time that the complete protocol TLS has shown to be secure. Building upon that, Li, Schäge, Yang, Kohlar, and Schwenk (2015) give a definition of ACCE security for authentication protocols with pre-shared keys. They present a renewed version of the ACCE model from Jager et al. (2013) that includes most recent attacks. Since this is the newest work, we focus on this ACCE model to include every attack known up to their work.

Looking at the security model introduced by Li et al. (2015), we see 6 different oracles, used in two phases. The first phase establishes a common key, while the second uses it for communication. Since the ACCE model is just an extension of the AKE model, we remove those oracles that are used after the key exchange, to get back to the AKE model. We use this method to use all the benefits and specific changes in security notation that the ACCE model went through, to get an up-to-date AKE model, which covers modern attacks. By doing that we then are left with the 4 oracles "Send$^{\text{pre}}$", "RegisterParty", "RevealKey" and "Corrupt" needed during the establishment of the key. Those oracles and the AKE model are explained later on in more detail.

After a full proof for TLS was presented, Krawczyk, Paterson, and Wee (2014) modularises the protocol by extracting a key-encapsulation mechanism (KEM) on a variant of the ACCE notion of Jager et al. (2013). This allows to analyse TLS in a modular fashion, meaning that sufficient conditions on the KEM will lead to a ACCE secure TLS-protocol. Bhargavan, Fournet, Kohlweiss, Pironti, Strub, and Zanella-Béguelin (2014) use a slightly different KEM to show that TLS is secure. They also provide a reference implementation of TLS, which they use to prove security with EasyCrypt (see Microsoft, Inria, and the Joint Centre, 2014a).

1.3. Contribution

Ideally we will have the same procedure for IPsec as was done for TLS in the future. These are a full proof, modularisation and a computer-verification. Even though IPsec was analyzed in the past years, we are still missing a full security proof. Currently Heussen, Loebenberger, and Nüsken (2017) are working on such a proof. They use game-style language similar to Katz and Lindell (2016) and use the corresponding game-notation for `AKE` that fits the description of Li, Schäge, Yang, Kohlar, and Schwenk (2015). They prove that IPsec fulfills the security properties presented by the game `AKE`.
Building upon that this work focuses on the computer aided toolset EasyCrypt to generate a computer-verified proof for the real-or-random (`RORA`) scenario of IPsec. This computer-assisted proof relies heavily on the work of Heussen et al. (2017) and functions more as an improvement compared to a completely new proof. The implementation needs a lot of preparatory work in the theoretical framework. This involves a restructure of the `AKE` game and the protocol, simplifying the game hops, and introducing a remodeled proof for the `RORA` scenario. For simplicity, we assume that party-certification is mandatory. Since this property is not used in the proof at any point, it is still secure without this assumption.
By using a computer tool like EasyCrypt this work sets a foundation towards an automated proof for IPsec. Previous work from Microsoft, Inria, and the Joint Centre (2014a) cannot be used, since it mainly focuses on the KEM, is not documented, and cannot be run by using the current version of EasyCrypt. This is probably due to changed internal libraries from EasyCrypt. For more information about this implementation, see Microsoft, Inria, and the Joint Centre (2014b).

2. Preliminaries

We use games to define security properties (see Katz and Lindell, 2016). This is done by defining a game, where a successful attack towards the security property leads to winning the game. Then we define an advantage towards that game. If the advantage of a game is negligible, then this implies that the chance to successfully attack the security property has to be negligible too. The main reason to use games for security notation is to apply the Game Hopping Lemma. It allows to analyze one game through the properties of another game. For an exact definition see section 2.4. In this chapter we present all necessary games and security properties we need for the main theorem later on.

In the following we briefly recall the definitions and games needed in the upcoming proof and implementation. We call the one playing a game *adversary* and denote him with $\mathcal{A}$. The adversary has access to everything but the long-term stored secret knowledge. Further on we grant him access to *oracles*. They themselves use the secret knowledge without leaking it to the adversary. Typically an oracle runs a scheme or protocol, or presents an ability we want to assume the adversary has. A game consists of 3 phases. The first one prepares the environment and sets up the games structure. The second prepares the hidden information and the oracles, then calls the adversary. In the last phase the game checks if the adversary won the game. We denote the advantage of a game G played by $\mathcal{A}$ as $\mathrm{adv}^G(\mathcal{A})$. We present its exact definition at the end of every game. The advantage adv^G of a game is the maximum advantage over all probabilistic polynomial time adversaries. We only consider such adversaries from now on. If the advantage of a game is negligible, then we call it a *null-game*. We denote with $\in_R$ the uniformly random assignment and with $\perp$ the empty symbol, indicating that no information is passed.

2.1. Game: PRF

The game PRF prepares an oracle $\mathcal{O}^{\mathtt{PRF}}$ that on input x either returns the keyed function applied to x or randomly picks an output. The adversary has to guess after any number of queries if he always got some random output or if the keyed function was used. If he is not able to see any non-negligible difference, then the keyed function behaves similar to a truly random function, thus we call it a pseudorandom function. Consequently we define a pseudorandom function as a keyed function, such that the game PRF using this function has negligible advantage.

Game: PRF

Parameter: A keyed function $\mathtt{prf} : \mathcal{K} \times \mathcal{X} \to \mathcal{Z}$.

Input: Security parameters.

Output: **ACCEPT** or **REJECT**

1: Pick $k \in_R \mathcal{K}$.

2: Pick hidden bit $h^{\mathtt{PRF}} \in_R \{0,1\}$.

3: Construct oracle $\mathcal{O}^{\mathtt{PRF}}$.

> **Oracle:** $\mathcal{O}^{\mathtt{PRF}}$
>
> Input: $x \in \mathcal{X}$
>
> Output: $z \in \mathcal{Z}$
>
> 1: **If** input has been seen **then Return** old answer
>
> 2: **Else If** $h^{\mathtt{PRF}} = 0$ **then Return** $\mathtt{prf}(k, x) \in \mathcal{Z}$
>
> 3: **Else If** $h^{\mathtt{PRF}} = 1$ **then Return** $r \in_R \mathcal{Z}$

4: Call adversary $\mathcal{A}$ with input $\mathcal{O}^{\mathtt{PRF}}$. Await guess $h' \in \{0,1\}$.

5: **If** $h' = h^{\mathtt{PRF}}$ **then** **ACCEPT** **Else** **REJECT**

Definition: *We define the advantage of the game PRF for adversary $\mathcal{A}$ as*

$$adv^{PRF}(\mathcal{A}) = \left| \Pr[\mathcal{A} \text{ wins game } PRF] - \frac{1}{2} \right|.$$

2.2. Game: MP-PRF

The following game is the multi primitive version for pseudorandom functions. Its difference to PRF is the set of functions, where the adversary is able to choose from. The advantage for the game MP-PRF is bounded by the sum of the advantages for every function that the adversary chooses from. Consequently if every function is secure in the sense of the game PRF, we know that this game has negligible advantage.

Game: MP-PRF

Parameter: A set $\mathcal{F}$ of keyed functions $\mathtt{prf} : \mathcal{K}_{\mathtt{prf}} \times \mathcal{X}_{\mathtt{prf}} \to \mathcal{Z}_{\mathtt{prf}}$.

Input: Security parameters.

Output: **ACCEPT** or **REJECT**

1: **For each** function $\mathtt{prf} \in \mathcal{F}$ **do**
   Pick a key $k_{\mathtt{prf}} \in_R \mathcal{K}_{\mathtt{prf}}$.
2: **For each** function $\mathtt{prf} \in \mathcal{F}$ **do**
   Pick a hidden bit $h_{\mathtt{prf}}^{\mathtt{MP\text{-}PRF}} \in_R \{0,1\}$.
3: Construct oracle $\mathcal{O}^{\mathtt{MP\text{-}PRF}}$.

> **Oracle:** $\mathcal{O}^{\mathtt{MP\text{-}PRF}}$
>
> Input: $\mathtt{prf} \in \mathcal{F}, x \in \mathcal{X}_{\mathtt{prf}}$
>
> Output: $z \in \mathcal{Z}_{\mathtt{prf}}$
>
> 1: **If** input has been seen **then Return** old answer
> 2: **Else If** $h_{\mathtt{prf}}^{\mathtt{MP\text{-}PRF}} = 0$ **then Return** $\mathtt{prf}(k, w) \in \mathcal{Z}$
> 3: **Else If** $h_{\mathtt{prf}}^{\mathtt{MP\text{-}PRF}} = 1$ **then Return** $r \in_R \mathcal{Z}$

4: Call adversary $\mathcal{A}$ with input $\mathcal{O}^{\mathtt{MP\text{-}PRF}}$. Await guess $(h', \mathtt{prf}') \in \{0,1\} \times \mathcal{F}$.
5: **If** $h' = h_{\mathtt{prf}'}^{\mathtt{MP\text{-}PRF}}$ **then ACCEPT Else REJECT**

Definition: *We define the advantage of the game MP-PRF for adversary $\mathcal{A}$ as*

$$adv^{MP\text{-}PRF}(\mathcal{A}) = \left| \Pr[\mathcal{A} \text{ wins game MP-PRF}] - \frac{1}{2} \right|.$$

2.3. Game: MP-PRF*-OODH

The star-symbol in the name PRF* indicates that in contrary to the game PRF the input for the keyed function is swapped. Analogous to the previous game, we look at the multi-primitive version of a game called PRF*-OODH. The idea is to formulate security of a Diffie-Hellman key exchange, where the pseudorandom function is applied to the transmitted information. Typically the Diffie-Hellman key exchange depends on the discrete logarithm problem. In case of the game MP-PRF*-OODH it is more complicated and nowadays a commonly used security assumption. For more details towards this specifically, see Brendel, Fischlin, Günther, and Janson (2017).

Game: MP-PRF*-OODH

Parameter: A set $\mathcal{G}$ of group data $\Gamma = (G, g, q)$, where q is the order of g.
A set $\mathcal{F}$ of keyed functions $\mathtt{prf} : \mathcal{K}_{\mathtt{prf}} \times \mathcal{X}_{\mathtt{prf}} \to \mathcal{Z}_{\mathtt{prf}}$.

Input: Security parameters.

Output: **ACCEPT** or **REJECT**

1: **For each** $\Gamma = (G, g, q) \in \mathcal{G}$ **do**
Choose $a_\Gamma, b_\Gamma \in_R \mathbb{Z}_q$.
Compute $A_\Gamma \leftarrow g^{a_\Gamma}, B_\Gamma \leftarrow g^{b_\Gamma}$ and $\mathtt{DH}_\Gamma \leftarrow g^{a_\Gamma b_\Gamma}$.

2: **For each** $\Gamma \in \mathcal{G}$ and $\mathtt{prf} \in \mathcal{F}$ **do**
pick a hidden bit $h^{\text{MP-PRF*-OODH}}_{\Gamma,\mathtt{prf}} \in_R \{0, 1\}$

3: Construct oracle $\mathcal{O}^{\text{MP-PRF*-OODH}}_{\mathtt{PRF}}$.

Oracle: $\mathcal{O}^{\text{MP-PRF*-OODH}}_{\mathtt{PRF}}$

Input: $\mathtt{prf} \in \mathcal{F}, \Gamma \in \mathcal{G}, x \in \mathcal{X}_{\mathtt{prf}}$

Output: $z \in \mathcal{Z}_{\mathtt{prf}}$

1: **If** $G \notin \mathcal{K}_{\mathtt{prf}}$ **then Return** $\perp$

2: **If** input has been seen **then Return** old answer.

3: **If** $h^{\text{MP-PRF*-OODH}}_{\Gamma,\mathtt{prf}} = 0$ **then**
Return $z \leftarrow \mathtt{prf}(x, \mathtt{DH}_\Gamma)$.

4: **Else Return** $z \in_R \mathcal{Z}$.

4: Construct one-time oracle $\mathcal{O}_{\texttt{LODH}}^{\texttt{MP-PRF*-OODH}}$.

Oracle: $\mathcal{O}_{\texttt{LODH}}^{\texttt{MP-PRF*-OODH}}$

Input: $\texttt{prf} \in \mathcal{F}$, $Y \in \Gamma.G$, $x \in \mathcal{X}_{\texttt{prf}}$

Output: $z \in \mathcal{Z}_{\texttt{prf}}$

1: **If** $G \notin \mathcal{K}_{\texttt{prf}}$ **then Return** $\perp$

2: **If** called before **then Return** $\perp$

3: **If** $Y = B_\Gamma$ **then Return** $\perp$

4: **Return** $\texttt{prf}(x, Y^{a_\Gamma})$

5: Construct one-time oracle $\mathcal{O}_{\texttt{RODH}}^{\texttt{MP-PRF*-OODH}}$.

Oracle: $\mathcal{O}_{\texttt{RODH}}^{\texttt{MP-PRF*-OODH}}$

Input: $\texttt{prf} \in \mathcal{F}$, $X \in \Gamma.G$, $x \in \mathcal{X}_{\texttt{prf}}$

Output: $z \in \mathcal{Z}_{\texttt{prf}}$

1: **If** called before **then Return** $\perp$

2: **If** $X = A_\Gamma$ **then Return** $\perp$

3: **Return** $\texttt{prf}(x, X^{b_\Gamma})$

6: Call adversary $\mathcal{A}$ with $\mathcal{G}, \mathcal{F}, (A_\Gamma)_{\Gamma\in\mathcal{G}}, (B_\Gamma)_{\Gamma\in\mathcal{G}}$ and oracles $\mathcal{O}_{\texttt{PRF}}^{\texttt{MP-PRF*-OODH}}, \mathcal{O}_{\texttt{LODH}}^{\texttt{MP-PRF*-OODH}}, \mathcal{O}_{\texttt{RODH}}^{\texttt{MP-PRF*-OODH}}$.
Await guess $(\Gamma', \texttt{prf}', h') \in \mathcal{G} \times \mathcal{F} \times \{0,1\}$.

7: **If** $h' = h_{\Gamma',\texttt{prf}'}^{\texttt{MP-PRF*-OODH}}$ **then ACCEPT**

8: **Else REJECT**

Definition: *We define the advantage of the game MP-PRF*-OODH for adversary $\mathcal{A}$ as*

$$adv^{\textit{MP-PRF*-OODH}}(\mathcal{A}) = \left|\Pr[\mathcal{A} \textit{ wins game MP-PRF*-OODH}] - \frac{1}{2}\right|.$$

2.4. Game Hopping Lemma

Consider a sequence of games $[G_0, G_1, G_2, \ldots, G_n]$, where the advantages of each two consecutive games G_i, G_{i+1} differ at most by $\varepsilon_{i,i+1}$. Then the first and last game differ at most by the sum of ε:

$$\mathrm{adv}^{G} - \mathrm{adv}^{G_n} \leq \sum_{i=0,\ldots,n-1} \varepsilon_{i,i+1}$$

Assume that all those differences of the games are negligible and the last game, G_n is a null-game. Since the sum of negligible terms is again negligible, we know that the game G has negligible advantage. Finding a game sequence that fulfills these assumptions is a method to prove security notations in the game style language. Each two consecutive games G_i, G_{i+1} are called a game hop. With the Game Hopping Lemma we analyze the values $\varepsilon_{i,i+1}$ for such a game hop. We present this lemma as defined by Heussen et al. (2017) and omit cases not needed in the security proof.

Theorem 1: Game Hopping Lemma**.**

(i) ***Indistinguishability hop.*** *Let G_0 and G_1 be two games which call the same adversary $\mathcal{A}$. Let $\mathcal{D}$ be a distinguisher that plays another game called G, where G hides a uniformly random bit $h_G \in \{0,1\}$ from $\mathcal{D}$. We denote with $\mathcal{D} \circ \mathcal{A}$ the distinguisher $\mathcal{D}$ joining with adversary $\mathcal{A}$ to play the game G. Then the Game Hopping Lemma states, if*

$$\mathcal{A} \text{ wins } G_0 \quad \textit{iff} \quad \mathcal{D} \circ \mathcal{A} \text{ wins } G \text{ given } h_G = 0 \qquad \text{(left side)}$$

and

$$\mathcal{A} \text{ wins } G_1 \quad \textit{iff} \quad \mathcal{D} \circ \mathcal{A} \text{ loses } G \text{ given } h_G = 1 \qquad \text{(right side)}$$

are true, then

$$\Pr[A \text{ wins } G_0] - \Pr[A \text{ wins } G_1] \;\leq\; 2 \;\cdot\; adv^{G}(\mathcal{D} \circ \mathcal{A})$$

holds.

(ii) ***Large error hop.*** *Let S_0 and S_1 represent the success events of two games G_0 and G_1 respectively. Let E be a set that contains all cases where the two games differ, i.e. $S_0 \triangle S_1 \subseteq E$.*

If S_0 and E are independent and $\Pr[S_1|E] = 1/2$, then

$$\left|\Pr[S_0] - \frac{1}{2}\right| = \frac{1}{\Pr[\neg E]} \;\cdot\; \left|\Pr[S_1] - \frac{1}{2}\right|.$$

The first case, the indistinguishability hop, is the most commonly used one. Since it is difficult to show that the equations (left side) and (right side) hold on their own, the following lemma simplifies its usage.

Lemma 1: Simplification of Game Hopping Lemma (Heussen et al., 2017)**.**
In the situation of Game Hopping Lemma (i), the assumption on the events is implied if for each hidden bit $h^* \in \{0,1\}$ the following properties hold.

1. *The view of $\mathcal{A}$ is identical whether interacting with G_{h^*} or called by $\mathcal{D}$ playing game G. Formally speaking, for each transcript T possibly seen by $\mathcal{A}$ we have*

$$\begin{aligned} &\Pr[T \text{ is the transcript of } \mathcal{A} \text{ talking to } \mathcal{D} \text{ joined with } G \quad | \ h_G = h^*] \\ = &\Pr[T \text{ is the transcript of } \mathcal{A} \text{ talking to } G_{h^*} \quad | \ h_G = h^*]. \end{aligned}$$

2. *G accepts if and only if its hidden bit is guessed correctly.*
3. *For each transcript T of $\mathcal{A}$, regardless of its probability, the answer of $\mathcal{D}$ is 0 if and only if T wins G_{h^*}, and 1 otherwise.*

We use this lemma to show that we can apply the Game Hopping Lemma for the upcoming game hops.

2.5. Signatures

A signature scheme `SIG` consists of three algorithms. Namely the key generation `SIG.Keygen`, the signing `SIG.Sign` and the verification of a signature `SIG.Verify`. The algorithm `SIG.Keygen` randomly generates on input 1^κ a keypair $(\texttt{pk}, \texttt{sk})$ in correspondence to the security parameter κ. The key `pk` is used for public signature verification and the key `sk` is used for signing. The algorithm `SIG.Sign` generates on input $(\texttt{sk}, \texttt{m})$ the signature `s` of `m` using the signing key `sk`. The verification `SIG.Verify` returns on input $(\texttt{pk}, \texttt{m}, \texttt{s})$ true if `s` is a valid signature for `m` and false otherwise.
Typically a signature scheme has to be `EUF-CMA` secure, meaning that no signature can be forged. In our case we can omit this security assumption since we do not need it in the upcoming proof.

2.6. Authenticated Encryption

An authenticated symmetric encryption scheme $\texttt{AE}$ consists of three algorithms. Namely the key generation $\texttt{AE.Keygen}$, the encryption $\texttt{AE.Enc}$ and the decryption of an encrypted message $\texttt{AE.Dec}$. The algorithm $\texttt{AE.Keygen}$ randomly generates on input 1^κ a key $\texttt{k}$ in correspondence to the security parameter κ. Calling $\texttt{AE.Enc}$ on input $(\texttt{k}, \texttt{Hdr}, \texttt{m})$ produces a ciphertext $\texttt{c}$ from the message $\texttt{m}$ using the key $\texttt{k}$ and potentially using additional information $\texttt{Hdr}$. The algorithm $\texttt{AE.Dec}$ computes on input $(\texttt{k}, \texttt{Hdr}, \texttt{c})$ the decryption of $\texttt{c}$ using the key $\texttt{k}$ and checks the validity of it. This means if the decryption is not valid, then it returns a failure, otherwise it returns the decrypted message $\texttt{m}$.
Normally we want an encryption scheme to be $\texttt{IND-CCA}$ secure, meaning that even with encryption and decryption oracles no difference between two encrypted messages can be detected. In our case we can omit this security assumption since we do not need it in the upcoming proof.

2.7. Security Assumptions

We do not need any security property for signatures or authenticated encryption schemes.
We assume each function $\texttt{prf}$ and its corresponding function $\texttt{prf}^*$ to be a pseudorandom function. For a pseudorandom function $\texttt{prf}$, we denote a function $\texttt{prf+}$ as a method to obtain more output from $\texttt{prf}$. The exact definition is: $\texttt{prf+}(K, S) = T_1|T_2|\ldots|T_{255}$ with $T_0 = \texttt{prf}(K, S|0\ldots0|1)$ and $T_i = \texttt{prf}(K, T_{i-1}|S|i)$ for $i \in \{1, \ldots, 255\}$.
It is possible to show that as long as $\texttt{prf}$ is a pseudorandom function, $\texttt{prf+}$ is also a pseudorandom function. For simplicity, we assume that $\texttt{prf+}$ is a pseudorandom function on its own.

3. The Protocol: IPsec

In this section we go through each step during the key exchange of IPsec and explain those in detail. To do that, we first have to define neccessary termonologies. Note that everytime two parties use the protocol, for each a different protocol instance is created. Every protocol instance is assigned a role, which are either `Initiator` or `Responder`. The `Initiator` starts the communication and does not need any message as input. A protocol instance is assigned the role of `Responder` if it starts upon receiving a message. Since we need to differentiate between these two roles and want to check for the correctness of their communication, we split every variable into two. This is indicated by the ending of a variable, where `i` stands for the `Initiator` and `r` for the `Responder`. For example the shared key from Diffie-Hellman key exchange `DH` is therefore split into `DHi` and `DHr` for the corresponding roles. Everytime the protocol reads a message it is parsed. This checks for errors, like wrong message type or incorrect addressee. We denote that this happens with `parse` and if an error is found, the protocol aborts. Let π and π' be two protocol instances. We define that π and π' *communicate* with each other, if the following properties hold:

1. Both were instantiated, meaning that they already have a defined role and at least one message was sent.
2. Every message received by π was sent by π' and vice versa.

Following from this properties, we know that if π and π' communicate, then each variable that is used in the message was not altered.
Compared to (Heussen et al., 2017) we change the presentation of the protocol a bit. We split the steps 5 and 10 into the corresponding parts of the `Initiator` and `Responder` indicated by `i` and `r` respectively. This is done to explicitly compute everything as soon as possible, which results in a better structure and simpler implementation in EasyCrypt. The following is a stpe-by-step explanation of IKEv2 as used in IPsec.

IKE_SA_INIT_initiator_send

1.1 $\varrho \leftarrow$ `Initiator`

Set the role ϱ of the protocol instance to `Initiator`.

1.2 `SPIi` $\in_R \{0,1\}^{64} \setminus \{0\}$

The security parameter index `SPI` is a bitstring used in the header to identify the protocol instance. In the first message the index for the `Responder` is set to 0.

1.3 $a \in_R \mathbb{Z}_q$, `with` q `order of` g

Prepare the security association `SAi1`, which contains four lists of supported cryptographic algorithms. Here the initiator guesses from the list of groups for the Diffie-Hellman key exchange one and randomly picks an exponent a.

1.4 `KEi` $\leftarrow g^a$

Use the secret information a to compute g^a, which is used for the Diffie-Hellman key exchange.

1.5 `Ni` $\in_R \{0,1\}^{\lambda}$

Choose a random nonce of length λ, which is part of the security parameters.

1.6 `Send` $m_1 \leftarrow$ `Hdr`, `SAi1`, `KEi`, `Ni`

Send the first message containing the header, security association, part of Diffie Hellman key exchange and a nonce. Await a response.

IKE_SA_INIT_responder_accept

2.1 `parse` m_1

Read the given message and check for `parse` errors.

2.2 `If no variant is supported, then reject,`
`notify NO_PROPOSAL_CHOSEN, and abort.`

Choose algorithms from `SAi1` to use from now on. If the set `SAr1` containing four algorithms cannot be defined, `abort`. Note that this only happens if both sides support different algorithms.

2.3 `If group in KEi was guessed wrongly then reject,`
`notify INVALID_KE_PAYLOAD indicating the correct group, and abort.`

Check if the group for Diffie-Hellman key exchange was guessed correctly. If not, send a reject-message indicating the correct group to use.

IKE_SA_INIT_responder_send

3.1 $\varrho \leftarrow$ Responder

Set the role ϱ of the protocol instance to Responder.

3.2 $\mathtt{SPIr} \in_R \{0,1\}^{64} \setminus \{0\}$

Randomly choose the security parameter index.

3.3 $b \in_R \mathbb{Z}_q$, with q order of g

Choose the secret knowledge for Diffie-Hellman key exchange.

3.4 $\mathtt{KEr} \leftarrow g^b, \mathtt{DH} \leftarrow \mathtt{KEi}^b$

Compute KEr and execute the key exchange.

3.5 $\mathtt{Nr} \in_R \{0,1\}^{\lambda}$

Choose a random nonce.

3.6 Send $m_2 \leftarrow \mathtt{Hdr}, \mathtt{SAr1}, \mathtt{KEr}, \mathtt{Nr}, \mathtt{CERTREQ}$

Send the second message containing CERTREQ, which is requesting the other party to provide a certificate.

IKE_SA_INIT_keyderivation

5r.1 $\mathtt{SKEYSEED} \leftarrow \mathtt{prf}(\mathtt{Ni}|\mathtt{Nr}, \mathtt{DH})$

Use the shared key DH and both nonces to generate common keyseed.

5r.2 $\mathtt{SK} \leftarrow \mathtt{prf+}(\mathtt{SKEYSEED}, \mathtt{Ni}|\mathtt{Nr}|\mathtt{SPIi}|\mathtt{SPIr})$

Use the common keyseed and lengthen this bitstring with prf+ to generate a longer common key.

5r.3 $\mathtt{SK_d}|\mathtt{SK_{ai}}|\mathtt{SK_{ar}}|\mathtt{SK_{ei}}|\mathtt{SK_{er}}|\mathtt{SK_{pi}}|\mathtt{SK_{pr}} \leftarrow \mathtt{SK}$

Split the key into parts. $\mathtt{SK_d}$ is used to derive further keys, $\mathtt{SK_{ai}}$ and $\mathtt{SK_{ai}}$ are used for authenticating, $\mathtt{SK_{ei}}$ and $\mathtt{SK_{er}}$ for encrypting messages, and $\mathtt{SK_{pi}}$ and $\mathtt{SK_{pr}}$ for proving that both know the shared secret at the end.

IKE_SA_INIT_initiator_accept

4.1 parse m_2

Read the given message and check for parse errors.

4.2 If SAr1 was not part of SAi1 or the chosen group does not match the one used for KEi and KEr then abort.

Check if the sent chosen algorithms are supported. Further, if the group picked for the Diffie-Hellman key exchange is wrong, abort. Optionally inform the other side.

4.3 $\mathtt{DH} \leftarrow \mathtt{KEr}^a$

Apply the key exchange.

IKE_SA_INIT_keyderivation

Same as the key derivation on the side of the `Responder`.

$5^{\mathtt{i}}.1$ $\mathtt{SKEYSEED} \leftarrow \mathtt{prf}(\mathtt{Ni}|\mathtt{Nr}, \mathtt{DH})$

$5^{\mathtt{i}}.2$ $\mathtt{SK} \leftarrow \mathtt{prf+}(\mathtt{SKEYSEED}, \mathtt{Ni}|\mathtt{Nr}|\mathtt{SPIi}|\mathtt{SPIr})$

$5^{\mathtt{i}}.3$ $\mathtt{SK_d}|\mathtt{SK_{ai}}|\mathtt{SK_{ar}}|\mathtt{SK_{ei}}|\mathtt{SK_{er}}|\mathtt{SK_{pi}}|\mathtt{SK_{pr}} \leftarrow \mathtt{SK}$

IKE_AUTH_initiator_send

6.1 $\mathtt{Mi} \leftarrow m_1|\mathtt{Nr}|\mathtt{prf}(\mathtt{SK_{pi}}, \mathtt{IDi})$

Prepare the identity and certificate of the party that instantiated this protocol instance on the side `Initiator`. Note that m_1 was provided by this side of the protocol instance, where `Nr` is given through the message m_2.

6.2 $\mathtt{AUTHi} \leftarrow \mathtt{SIG.Sign}(\mathtt{sk_{IDi}}, \mathtt{Mi})$

Sign the concatenated information for authentication.

6.3 $t_3 \leftarrow \mathtt{IDi}, \mathtt{CERTi}, \mathtt{CERTREQ}, \mathtt{AUTHi}, \mathtt{SAi2}, \mathtt{TSi}, \mathtt{TSr}$

Prepare `SAi2`, `TSi` and `TSr` for establishing the full common security association `SA`. Here `TSi` and `TSr` are called "Traffic Selectors". They are specifying the traffic handled by `SA`. Together with the authentication, certificate, identity and the certificate-request build the to-be sent message.

6.4 $t_3^{\mathtt{enc}} \leftarrow \mathtt{AE.Enc}(\mathtt{SK_{ei}}|\mathtt{SK_{ai}}, \mathtt{Hdr}, t_3)$

Use the authenticated encryption algorithm that was agreed upon earlier to encrypt the message.

6.5 `Send` $m_3 \leftarrow \mathtt{Hdr}, t_3^{\mathtt{enc}}$

Send the header with the encrypted message.

IKE_AUTH_responder_accept

7.1 `parse` m_3

Read the given message and check for `parse` errors.

7.2 $t_3 \leftarrow \mathtt{AE.Dec}(\mathtt{SK_{ei}}|\mathtt{SK_{ai}}, \mathtt{Hdr}, t_3^{\mathtt{enc}})$

Use the agreed upon scheme to decrypt the message.

7.3 `If` $t_3 = \perp$ `then reject, notify AUTHENTICATION_FAILED and abort.`

If the authenticated decryption failed, `abort`.

7.4 `parse` t_3

If decryption worked, check for `parse` errors.

7.5 `parse CERTi for SIG.Vfy`, $\mathtt{pk_{IDi}}$

Prepare everything needed to be able to verify the authentication `Mi` of the message.

7.6 $\texttt{Mi} \leftarrow m_1|\texttt{Nr}|\texttt{prf}(\texttt{SK}_{\texttt{pi}}, \texttt{IDi})$

Compute Mi with own variables. Note that m_1 is the received message, Nr was picked, $\texttt{SK}_{\texttt{pi}}$ was computed with common shared knowledge, and IDi was received just now.

7.7 `If not valid`$(\texttt{CERTi}, \texttt{IDi}, \texttt{pk}_{\texttt{IDi}})$ `or not SIG.Vfy`$(\texttt{pk}_{\texttt{IDi}}, \texttt{Mi}, \texttt{AUTHi})$`, then reject, notify AUTHENTICATION_FAILED and abort.`

Check the validation of the given certificate and the authentication of Mi. If not, abort.

7.8 $\Lambda \leftarrow$ `accept`

Set the state Λ of the protocol instance to accept.

7.9 $\Pi \leftarrow \texttt{IDi}$

Set the partner Π of this protocol instance to the sent identity. Due to the correct signature this partner fits to the given public key $\texttt{pk}_{\texttt{IDi}}$.

IKE_AUTH_responder_send

8.1 $\texttt{Mr} \leftarrow m_2|\texttt{Ni}|\texttt{prf}(\texttt{SK}_{\texttt{pr}}, \texttt{IDr})$

Analogous to the other side compute everything that is signed later on.

8.2 $\texttt{AUTHr} \leftarrow \texttt{SIG.Sign}(\texttt{Mr})$

Sign to generate authentication for identity.

8.3 $t_4 \leftarrow \texttt{IDr}, \texttt{CERTr}, \texttt{AUTHr}, \texttt{SAr2}, \texttt{TSi}, \texttt{TSr}$

Generate the to-be sent message.

8.4 $t_4^{\texttt{enc}} \leftarrow \texttt{AE.Enc}(\texttt{SK}_{\texttt{er}}|\texttt{SK}_{\texttt{ar}}, \texttt{Hdr}, t_4)$

Encrypt the message.

8.5 `Send` $m_4 \leftarrow \texttt{Hdr}, t_4^{\texttt{enc}}$

Together with the header send the encrypted message.

IKE_AUTH_keyderivation

10.1 $\texttt{KEYMAT} \leftarrow \texttt{prf+}(\texttt{SK}_{\texttt{d}}, \texttt{Ni}|\texttt{Nr})$

If negotiation was successful, then prepare the keymaterial KEYMAT. Use this together with SAr2 to establish common algorithms and keys that will be used for communication from now on.

IKE_AUTH_initiator_accept

9.1 parse m_4

Read the given message and check for parse errors.

9.2 $t_4 \leftarrow \mathtt{AE.Dec}(\mathtt{SK_{er}}|\mathtt{SK_{ar}}, \mathtt{Hdr}, t_4^{\mathtt{enc}})$

Decrypt the given message.

9.3 If $t_4 = \perp$ then reject, notify AUTHENTICATION_FAILED and abort.

Check if the authenticated decryption failed.

9.4 parse t_4

Check for parse errors.

9.5 parse CERTr for $\mathtt{pk_{IDr}}$, SIG.Vfy

Prepare to verify the authentication.

9.6 $\mathtt{Mr} \leftarrow m_2|\mathtt{Ni}|\mathtt{prf}(\mathtt{SK_{pr}}, \mathtt{IDr})$

Generate what should be signed with previous knowledge of Mr, Ni and $\mathtt{SK_{pr}}$.

9.7 If not $\mathtt{valid}(\mathtt{CERTr}, \mathtt{IDr}, \mathtt{pk_{IDr}})$ or not $\mathtt{SIG.Vfy}(\mathtt{pk_{IDr}}, \mathtt{Mr}, \mathtt{AUTHr})$ then reject, notify AUTHENTICATION_FAILED and abort.

Check the validation of the given certificate and the authentication of Mr. If not, abort.

9.8 $\Lambda \leftarrow$accept

Set the state Λ of the protocol instance to accept.

9.9 $\Pi \leftarrow \mathtt{IDr}$

Set the partner Π of this protocol instance to the sent identity.

IKE_AUTH_keyderivation

$10^{\mathtt{i}}.1$ $\mathtt{KEYMAT} \leftarrow \mathtt{prf+}(\mathtt{SK_d}, \mathtt{Ni}|\mathtt{Nr})$

Analogous to the side of the Responder, compute everything needed for further communication.

3.1. Short Version

This section contains a shortened version of the protocol IPsec without details.

IKE_SA_INIT_initiator_send

1.1. $\varrho \leftarrow$ Initiator
1.2. SPIi $\in_R \{0,1\}^{64} \setminus \{0\}$
1.3. $a \in_R \mathbb{Z}_q$, with q order of g
1.4. KEi $\leftarrow g^a$
1.5. Ni $\in_R \{0,1\}^{\lambda}$
1.6. Send $m_1 \leftarrow$ Hdr, SAi1, KEi, Ni

IKE_SA_INIT_responder_accept

2.1. parse m_1
2.2. If no variant is supported, then reject, notify NO_PROPOSAL_CHOSEN, and abort.
2.3. If group in KEi was guessed wrongly then reject, notify INVALID_KE_PAYLOAD indicating the correct group, and abort.

IKE_SA_INIT_responder_send

3.1. $\varrho \leftarrow$ Responder
3.2. SPIr $\in_R \{0,1\}^{64} \setminus \{0\}$
3.3. $b \in_R \mathbb{Z}_q$, with q order of g
3.4. KEr $\leftarrow g^b$, DH $\leftarrow$ KEi^b
3.5. Nr $\in_R \{0,1\}^{\lambda}$
3.6. Send $m_2 \leftarrow$ Hdr, SAr1, KEr, Nr, CERTREQ

IKE_SA_INIT_keyderivation

5^{r}.1. SKEYSEED $\leftarrow$ prf(Ni|Nr, DH)
5^{r}.2. SK $\leftarrow$ prf+(SKEYSEED, Ni|Nr|SPIi|SPIr)
5^{r}.3. $\mathrm{SK_d}|\mathrm{SK_{ai}}|\mathrm{SK_{ar}}|\mathrm{SK_{ei}}|\mathrm{SK_{er}}|\mathrm{SK_{pi}}|\mathrm{SK_{pr}} \leftarrow$ SK

IKE_SA_INIT_initiator_accept

4.1. parse m_2
4.2. If SAr1 was not part of SAi1 or the chosen group does not match the one used for KEi and KEr then abort.
4.3. DH $\leftarrow$ KEr^a

IKE_SA_INIT_keyderivation

5^{i}.1. SKEYSEED $\leftarrow$ prf(Ni|Nr, DH)
5^{i}.2. SK $\leftarrow$ prf+(SKEYSEED, Ni|Nr|SPIi|SPIr)
5^{i}.3. $\mathrm{SK_d}|\mathrm{SK_{ai}}|\mathrm{SK_{ar}}|\mathrm{SK_{ei}}|\mathrm{SK_{er}}|\mathrm{SK_{pi}}|\mathrm{SK_{pr}} \leftarrow$ SK

IKE_AUTH_initiator_send

6.1. Mi $\leftarrow m_1|$Nr|prf($\mathrm{SK_{pi}}$, IDi)
6.2. AUTHi $\leftarrow$ SIG.Sign($\mathrm{sk_{IDi}}$, Mi)
6.3. $t_3 \leftarrow$ IDi, CERTi, CERTREQ, AUTHi, SAi2, TSi, TSr
6.4. $t_3^{\mathrm{enc}} \leftarrow$ AE.Enc($\mathrm{SK_{ei}}|\mathrm{SK_{ai}}$, Hdr, t_3)
6.5. Send $m_3 \leftarrow$ Hdr, t_3^{enc}

IKE_AUTH_responder_accept

7.1. parse m_3
7.2. $t_3 \leftarrow$ AE.Dec($\mathrm{SK_{ei}}|\mathrm{SK_{ai}}$, Hdr, t_3^{enc})
7.3. If $t_3 = \perp$ then reject, notify AUTHENTICATION_FAILED and abort.
7.4. parse t_3
7.5. parse CERTi for SIG.Vfy, $\mathrm{pk_{IDi}}$
7.6. Mi $\leftarrow m_1|$Nr|prf($\mathrm{SK_{pi}}$, IDi)
7.7. If not valid(CERTi, IDi, $\mathrm{pk_{IDi}}$) or not SIG.Vfy($\mathrm{pk_{IDi}}$, Mi, AUTHi) then reject, notify AUTHENTICATION_FAILED and abort.
7.8. $\Lambda \leftarrow$ accept
7.9. $\Pi \leftarrow$ IDi

IKE_AUTH_responder_send

8.1. Mr $\leftarrow m_2|$Ni|prf($\mathrm{SK_{pr}}$, IDr)
8.2. AUTHr $\leftarrow$ SIG.Sign(Mr)
8.3. $t_4 \leftarrow$ IDr, CERTr, AUTHr, SAr2, TSi, TSr
8.4. $t_4^{\mathrm{enc}} \leftarrow$ AE.Enc($\mathrm{SK_{er}}|\mathrm{SK_{ar}}$, Hdr, t_4)
8.5. Send $m_4 \leftarrow$ Hdr, t_4^{enc}

IKE_AUTH_keyderivation

10^{r}.1. KEYMAT $\leftarrow$ prf+($\mathrm{SK_d}$, Ni|Nr)

IKE_AUTH_initiator_accept

9.1. parse m_4
9.2. $t_4 \leftarrow$ AE.Dec($\mathrm{SK_{er}}|\mathrm{SK_{ar}}$, Hdr, t_4^{enc})
9.3. If $t_4 = \perp$ then reject, notify AUTHENTICATION_FAILED and abort.
9.4. parse t_4
9.5. parse CERTr for $\mathrm{pk_{IDr}}$, SIG.Vfy
9.6. Mr $\leftarrow m_2|$Ni|prf($\mathrm{SK_{pr}}$, IDr)
9.7. If not valid(CERTr, IDr, $\mathrm{pk_{IDr}}$) or not SIG.Vfy($\mathrm{pk_{IDr}}$, Mr, AUTHr) then reject, notify AUTHENTICATION_FAILED and abort.
9.8. $\Lambda \leftarrow$ accept
9.9. $\Pi \leftarrow$ IDr

IKE_AUTH_keyderivation

10^{i}.1. KEYMAT $\leftarrow$ prf+($\mathrm{SK_d}$, Ni|Nr)

4. The Game: AKE

Compared to Heussen et al. (2017) we change the presentation of the game `AKE` such that it fits better to the language of EasyCrypt. Those changes are minimal and do not alter the idea or meaning of the game. We quickly go through this remodeling and then present the full game `AKE` with its oracles.

While loop

Instead of using a loop to go through all protocol instances and check for an authentication break this way, we instead introduce a set. This set contains all candidates for an authentication break. Every time a protocol instance would be skipped during the loop, it is removed from this set. This restructure only modifies the view and does not alter the check for an authentication break at all.

Return statement

In EasyCrypt we are only allowed to use a single return statement. For this reason we move everything connected to the game's decision to the end. Note that if an authentication break was detected, the computations during the check of a real-or-random attack are omitted. Therefore nothing is changed by replacing the return statements.

4.1. Oracles

The game `AKE` consists of four oracles. Those are $\mathcal{O}_{\texttt{Send}}$, $\mathcal{O}_{\texttt{Reveal}}$, $\mathcal{O}_{\texttt{Corrupt}}$ and $\mathcal{O}_{\texttt{Test}}$. We go through the oracles and their purpose with respect to those defined by Li et al. (2015), namely "Send$^{\text{pre}}$", "RegisterParty", "RevealKey" and "Corrupt".

$\mathcal{O}_{\texttt{Send}}(P, \pi, m)$**:**
This oracle consists of two separate steps. First, if the protocol instance π is not initiated yet, it is connected to P and starts for the first time with input-message m. Second, it responds with message m', which is exactly the one sent according to the protocol specifications and the internal state of π. This allows the adversary to simulate any communication between two parties. The same kind of oracle in (Li et al., 2015) is called "Send$^{\text{pre}}$".

`Instantiate`

Input: party $P \in \mathcal{P}$, protocol instance $\pi \in \mathcal{Q}$

Output: nothing

1: Instantiate the protocol instance π according to the protocol. Attach party P to π. Give π a copy of party P's certificate $\texttt{pk}_{\texttt{CERT}}$, public signature key $\texttt{pk}_P$ and signing oracle $\texttt{SIG.Sign}(\texttt{sk}_P)$.

Oracle: $\mathcal{O}_{\texttt{Send}}$

Input: party $P \in \mathcal{P}$, protocol instance $\pi \in \mathcal{Q}$, message m

Output: response m'

1: **If** π is not instantiated **then** `Instantiate`(pi)

2: Act according to the protocol. Depending on the internal state of the protocol instance π and the message m it gets, apply the steps from the protocol.

3: Set m' to the response that π generates by processing message m. This response is empty if π is in the state accept or abort.

4: **Return** m'

$\mathcal{O}_{\texttt{Reveal}}(\pi)$**:**

Calling this oracle with a protocol instance π will output the session key if one was already established. This means that we allow the adversary to look at the keys that are exchanged by the protocol, just as "RevealKey" does.

Oracle: $\mathcal{O}_{\texttt{Reveal}}$

Input: protocol instance $\pi \in \mathcal{Q}$

Output: session key `KEYMAT` or $\perp$

1: **If** π has not accepted **then Return** $\perp$

2: **Else Return** session key `KEYMAT` of π

$\mathcal{O}_{\texttt{Corrupt}}(P)$**:**
A query on this oracle allows the adversary to gain access to the long-term stored secret of the party P. This is similar to "RegisterParty" and "Reveal" by (Li et al., 2015), where they either allow the adversary to fully establish a new party knowing all secrets, or give him access to some stored secrets. Since IPsec has two completely split phases, before and after key exchange, we only need to differentiate between keys that are needed for either phase. This oracle $\mathcal{O}_{\texttt{Corrupt}}$ is used to break the first phase before the key exchange is completed. To correctly formulate the AKE security, we need to be careful and check that this oracle is only used to gain information after a connection was established between two parties.

Oracle: $\mathcal{O}_{\texttt{Corrupt}}$

Input: a party $P \in \mathcal{P}$

Output: secret key $\texttt{sk}_P$

1: **Return** long-term private key $\texttt{sk}_P$ of the party P.

$\mathcal{O}_{\texttt{Test}}(\pi)$**:**
This is the only oracle that is not defined by Li et al. (2015). Here we present the challenge towards the adversary. Considering the hidden bit the oracle either returns the truly generated key by the protocol instance π, or a randomly picked key. If the adversary guesses the hidden bit correctly, he wins the game AKE. Of course we do not want to allow the adversary to just use the oracle $\mathcal{O}_{\texttt{Reveal}}$ to instantly know if the returned key is random or not. For this reason we have to check that in the game and forbid the adversary to gain advantage by doing that.

Oracle: $\mathcal{O}_{\texttt{Test}}$

Input: protocol instance $\pi \in \mathcal{Q}$

Output: potential session key KEY or $\perp$

1: **If** this oracle has already been called **then Return** $\perp$

2: **If** π has not accepted **then Return** $\perp$

3: Define $\texttt{KEY}_0 \leftarrow \pi.\texttt{KEYMAT}$

4: Define $\texttt{KEY}_1 \in_R \mathcal{K}$

5: **Return** $\texttt{KEY}_h$

4.2. Game

Considering the AKE model, there exist two major security aspects during the establishment of the common key. One is called the authentication break (AB). This means that the adversary is able to establish a connection that was not intended by the party. For example a man-in-the-middle attack leads to such a break. The second aspect, called real-or-random attack (RORA), is the indistinguishability between a random and the established key. This is the highest security aspect that we can show for a key exchange. It means that no adversary is able to get any information about the exchanged key. We formulate both possible attack scenarios in the game. The bit 0 and **REJECT** are equal, as are the bit 1 and **ACCEPT**.

Game: AKE

Input: Security parameters.

Output: **ACCEPT** or **REJECT**

▷ Prepare the environment.

1: Pick $\texttt{RandomDecision} \in_R \{0,1\}$.
2: Pick $\bar{\pi}' \in_R \mathcal{Q}, \hat{\pi}' \in_R \mathcal{Q} \setminus \{\bar{\pi}'\}$.
3: Generate keypair $(\texttt{pk}_{\texttt{CERT}}, \texttt{sk}_{\texttt{CERT}}) \leftarrow \texttt{SIG.Keygen}(1^\kappa)$ for certification authority and construct signing oracle $\mathcal{O}_{\texttt{Sign}_{\texttt{CERT}}}$ that produces signatures using $\texttt{sk}_{\texttt{CERT}}$.
4: **For** $P \in \mathcal{P}$ **do**
5: Equip party P with a keypair $(\texttt{pk}_P, \texttt{sk}_P) \leftarrow \texttt{SIG.Keygen}(1^\kappa)$ and define its signing oracle $\texttt{SIG.Sign}(\texttt{sk}_P, \cdot)$.
6: Equip party P with $\texttt{CERT}_P \leftarrow \mathcal{O}^{\texttt{EUF-CMA}}_{\texttt{Sign}_{\texttt{CERT}}}(\texttt{pk}_P, \texttt{ID}_P)$.

▷ Next, call the adversary.

7: Pick a hidden bit $h \in_R \{0,1\}$.
8: Construct oracles $\mathcal{O}_{\texttt{Send}}$, $\mathcal{O}_{\texttt{Reveal}}$, $\mathcal{O}_{\texttt{Corrupt}}$, and $\mathcal{O}_{\texttt{Test}}$.
9: Call adversary $\mathcal{A}$ with the four oracles $\mathcal{O}_{\texttt{Send}}$, $\mathcal{O}_{\texttt{Reveal}}$, $\mathcal{O}_{\texttt{Corrupt}}$, and $\mathcal{O}_{\texttt{Test}}$. Await guess $h' \in \{0,1\}$.

▷ First, check for authentication break.

10: Define $\mathcal{Q}' \leftarrow \mathcal{Q}$. ▷ Reduce this set instead of loop.
11: Reduce: $\mathcal{Q}' \leftarrow \{\pi' \in \mathcal{Q}' \mid \pi' \text{ has accepted}\}$
12: Reduce: $\mathcal{Q}' \leftarrow \mathcal{Q}' \setminus \{\pi' \in \mathcal{Q}' \mid \text{party } \pi'.P \text{ was corrupted before } \pi' \text{ was accepted}\}$
13: Reduce: $\mathcal{Q}' \leftarrow \mathcal{Q}' \setminus \{\pi' \in \mathcal{Q}' \mid \pi' \text{ communicates with one protocol instance}\}$
14: Detect authentication break:
15: **If** $|\mathcal{Q}'| > 0$ **then** detect authentication break with $\bar{\pi} \in_R \mathcal{Q}'$ and party $\bar{P} \leftarrow \bar{\pi}.P$.
16: **Else** there is no authentication break.

▷ Second, check for real-or-random attack.

If no authentication break was detected **then**

  If $\mathcal{O}_{\texttt{Test}}$ was not called **then break**.

  Define $\bar{\pi}'$ as the tested protocol instance.

  If $\mathcal{O}_{\texttt{Corrupt}}$ was used on $\bar{\pi}'.P$ before $\bar{\pi}'$ accepted **then break**.

  Define $\hat{\pi}'$ as the one protocol instance that communicates with $\bar{\pi}'$. Therefore they have opposite roles and all messages between $\hat{\pi}'$ and $\bar{\pi}'$ coincide.

  If $\mathcal{O}_{\texttt{Reveal}}$ was used on $\bar{\pi}'$ or on $\hat{\pi}'$ **then break**.

  Detect real-or-random attack.

Else No real-or-random attack detected.

▷ Last, return the game's decision

If authentication break was detected **then**

  Return ACCEPT. ▷ AB

If real-or-random attack was detected **then** ▷ RORA

  If $h = h'$ **then**

    Return ACCEPT.

  Else

    Return REJECT.

If no authentication break and no real-or-random attack was detected **then**

  Return `RandomDecision`. ▷ NOATTACK

Definition: *We define the advantage of the game* ***AKE*** *for adversary* $\mathcal{A}$ *as*

$$adv^{AKE}(\mathcal{A}) = \left| \Pr[\mathcal{A} \textit{ wins game } \textbf{AKE}] - \frac{1}{2} \right| .$$

Considering the three possible disjoint return statements NOATTACK, AB and RORA. We split the advantage into those three parts. To do that, we first define the games AB and RORA. The game RORA is the same as the game AKE, where we just replace the return of an authentication break with a random decision. Vice versa the game AB is defined. This leads to them only having non-zero advantage if an attack corresponding to their own notation occured. Then we get:

$$\begin{aligned}\mathrm{adv}^{\mathtt{RORA}}(\mathcal{A}) &= \left|\Pr[\mathcal{A} \text{ wins } \mathtt{RORA}] - \frac{1}{2}\Pr[\mathtt{RORA}]\right| \\ &= \left|\Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{RORA}] - \frac{1}{2}\right| \cdot \Pr[\mathtt{RORA}],\end{aligned}$$

$$\begin{aligned}\mathrm{adv}^{\mathtt{AB}}(\mathcal{A}) &= |\Pr[\mathcal{A} \text{ wins } \mathtt{AB}] - 0| \\ &= |\Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{AB}]| \cdot \Pr[\mathtt{AB}].\end{aligned}$$

Note that $\Pr[\mathtt{AB}] + \Pr[\mathtt{RORA}] + \Pr[\mathtt{NOATTACK}] = 1$. This leads to the split.

$$\begin{aligned}\mathrm{adv}^{\mathtt{AKE}}(\mathcal{A}) &= \Big|\Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{AB}] \cdot \Pr[\mathtt{AB}] \\ &\quad + \Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{RORA}] \cdot \Pr[\mathtt{RORA}] \\ &\quad + \Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{NOATTACK}] \cdot \Pr[\mathtt{NOATTACK}] - \frac{1}{2}\Big| \\ &= \Bigg|\left(\Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{AB}] - \frac{1}{2}\right) \cdot \Pr[\mathtt{AB}] \\ &\quad + \left(\Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{RORA}] - \frac{1}{2}\right) \cdot \Pr[\mathtt{RORA}] \\ &\quad + \left(\Pr[\mathcal{A} \text{ wins } \mathtt{AKE} \mid \mathtt{NOATTACK}] - \frac{1}{2}\right) \cdot \Pr[\mathtt{NOATTACK}]\Bigg| \\ &= \mathrm{adv}^{\mathtt{RORA}}(\mathcal{A}) + \mathrm{adv}^{\mathtt{AB}}(\mathcal{A})\end{aligned}$$

The authentication break can be split into two parts, one for each role of the broken protocol instance. Since we focus on the real-or-random attack only, we omit this.

Note that a real-or-random attack only happens if $\bar{\pi}'$ and $\hat{\pi}'$ communicate.

5. RORA-security

In this chapter we formulate the security for a real-or-random attack for IPsec. To show that this property holds, we use a game sequence of 7 games and apply the Game Hopping Lemma several times.

5.1. Theorem

The following theorem states that under the given security assumptions towards the cryptographic primitives used by IPsec, it is secure against real-or-random attacks.

Theorem 2: Real or random security for IKEv2 in the game `AKE`, (Heussen et al., 2017)**.** *Assume that*

(i) *All primitives are as secure as stated in section 2.7.*

(ii) *Every party uses a certificate for authentication (note: not needed in `RORA`-scenario).*

Then for every probabilistic polynomial time adversary $\mathcal{A}$ the advantage $adv^{RORA}(\mathcal{A})$ is at most negligible.

To prove this theorem we define a sequence of game hops and show that every two consecutive games have negligible difference. All of the upcoming proofs are implemented in EasyCrypt and implemented. The code needs some further assumptions, which we discuss later on. We present a game hop with two columns, where the left side is for remembering the original lines of the game, whereas the right side is the modification done to get to the new game.

5.2. Game hops

To prove theorem 2, we rely on the game sequence presented by Heussen et al. (2017). We remodel the existing proof such that it suits an implementation in EasyCrypt better. Building upon this, we introduce two new games `A5` and `A6`. We show that each game hop has at most negligible difference in the corresponding games. This is done by either applying the Game Hopping Lemma or showing the equality of both games.

5.2.1. Game hop: RORA → A1

As our first game hop, we introduce two new protocol instances $\bar{\pi}^*$ and $\hat{\pi}^*$, which have to be the same as $\bar{\pi}'$ and $\hat{\pi}'$. Note that the adversary picks $\bar{\pi}'$ by calling it to the oracle $\mathcal{O}_{\texttt{Test}}$, which means that he has to guess the randomly picked $\bar{\pi}^*$ correctly. Consequently he is only able to win the new game A1 in the real-or-random case, if he uses $\bar{\pi}^*$ as input for the oracle $\mathcal{O}_{\texttt{Test}}$ and $\bar{\pi}^*$ and $\hat{\pi}^*$ communicate.

RORA

2: Pick $\bar{\pi}' \in_R \mathcal{Q}, \hat{\pi}' \in_R \mathcal{Q} \setminus \{\bar{\pi}'\}$.

27: **If** real-or-random attack was detected **then**

28: **If** $h = h'$ **then**
29: **Return ACCEPT**.
30: **Else**
31: **Return REJECT**.

A1

2: Pick $\bar{\pi}' \in_R \mathcal{Q}, \hat{\pi}' \in_R \mathcal{Q} \setminus \{\bar{\pi}'\}$, and pick $\bar{\pi}^* \in_R \mathcal{Q}, \hat{\pi}^* \in_R \mathcal{Q} \setminus \{\bar{\pi}^*\}$.
27: **If** real-or-random attack was detected **then**
If $(\bar{\pi}^*, \hat{\pi}^*) \neq (\bar{\pi}', \hat{\pi}')$ **then**
Return RandomDecision
28: **If** $h = h'$ **then**
29: **Return ACCEPT**.
30: **Else**
31: **Return REJECT**.

Lemma 2: Game hop RORA → A1.
For every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$adv^{RORA}(\mathcal{A}) = q(q-1) \cdot adv^{A1}(\mathcal{A}),$$

where q is the number of protocol instances, meaning $q = |\mathcal{Q}|$.

Proof. We apply Game Hopping Lemma (ii) with

- $G_0 =$ RORA and its success event S_0,
- $G_1 =$ A1 and its success event S_1, and
- the error event E where $(\bar{\pi}^*, \hat{\pi}^*) \neq (\bar{\pi}', \hat{\pi}')$ holds.

For this, we show the following properties:

1. $S_0 \triangle S_1 \subseteq E$.

   The two games only differ by a single if-statement. This if-statement leads to a different outcome if E holds, otherwise the games are the same.

2. S_0 and E are independent.

   In the game RORA the variables $\bar{\pi}^*$ and $\hat{\pi}^*$ do not occur. Therefore the probabilities $\Pr[S_0 \mid E]$ and $\Pr[S_0 \mid \neg E]$ are equal, hence §$_0$ and E are independent.

3. $\Pr[S_1 \mid E] = 1/2$.

   If the error event E holds, then the resulting return command is a random decision, hence the probability for winning the game A1 is $1/2$.

We know that $\Pr[\neg E] = \frac{1}{\mathtt{q}*(\mathtt{q}-1)}$ because $\bar{\pi}^* \neq \hat{\pi}^* \in_R \mathcal{Q}$ and $|\mathcal{Q}| = \mathtt{q}$. Therefore by the large error hop of the Game Hopping Lemma we get:

$$\left|\Pr[S_0] - \frac{1}{2}\right| = \frac{1}{\Pr[\neg E]} \cdot \left|\Pr[S_1] - \frac{1}{2}\right|$$
$$\Rightarrow \left|\Pr[G_0 \text{ is won}] - \frac{1}{2}\right| = \mathtt{q} * (\mathtt{q}-1) \cdot \left|\Pr[G_1 \text{ is won}] - \frac{1}{2}\right|$$

Applying the definition of advantage proves the lemma. □

5.2.2. Game hop: A1 → A2

As our next step we want to replace the Diffie-Hellman key exchange with something truly random. For this reason we construct distinguisher $\mathcal{D}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}_{\mathtt{A1}\to\mathtt{A2}}$ playing the game MP-PRF*-OODH. We use him to swap the pseudorandom function that is applied to the information of the Diffie-Hellman key exchange with a truly random function. This leads to the game A2, which is exactly the game resulting by this swap. We then apply the Game Hopping Lemma to show that the difference between A1 and A2 is bounded by the game MP-PRF*-OODH. First, we define the distinguisher.

Distinguisher: $\mathcal{D}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}_{\mathtt{A1}\to\mathtt{A2}}$

Input: $(A_\Gamma)_{\Gamma\in\mathcal{G}}, (B_\Gamma)_{\Gamma\in\mathcal{G}}, \mathcal{O}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}_{\mathtt{PRF}}, \mathcal{O}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}_{\mathtt{LODH}}, \mathcal{O}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}_{\mathtt{RODH}}$

Start with $\mathtt{SKEYSEED}_{\mathtt{mod}} \leftarrow \mathtt{false}$. Embed the oracles in the protocols $\bar{\pi}^*$ and $\hat{\pi}^*$ in the following lines of game A1:

1.4. **If** either $\bar{\pi}^*$ or $\hat{\pi}^*$ has role Initiator **then**
  KEi $\leftarrow A_\Gamma$
  $a \leftarrow$ empty
  Else KEi $\leftarrow g^a$.

3.4. **If** either $\bar{\pi}^*$ or $\hat{\pi}^*$ has role Responder **then**
  KEr $\leftarrow B_\Gamma$
  $b \leftarrow$ empty
  Else KEi $\leftarrow g^b$, DH $\leftarrow$ KEi^b

5r.1. **If** $b =$ empty **then**
  If KEi $= A_\Gamma$ **then**
    $\mathcal{O}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}_{\mathtt{PRF}}(\mathtt{prf}, \mathtt{SAr1}.\Gamma, \mathtt{Ni}|\mathtt{Nr})$

Else $\texttt{SKEYSEED} \leftarrow \mathcal{O}_{\texttt{RODH}}^{\texttt{MP-PRF*-OODH}}(\texttt{prf}, \texttt{KEi}, \texttt{Ni}|\texttt{Nr})$

Else $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{DH})$

4.3. **If** $a \neq \texttt{empty}$ **then** $\texttt{DH} \leftarrow \texttt{KEr}^a$

5$^{\texttt{i}}$.1. **If** $a = \texttt{empty}$ **then**

If $\texttt{KEr} = B_\Gamma$ **then**

$\mathcal{O}_{\texttt{PRF}}^{\texttt{MP-PRF*-OODH}}(\texttt{prf}, \texttt{SAr1}.\Gamma, \texttt{Ni}|\texttt{Nr})$

Else $\texttt{SKEYSEED} \leftarrow \mathcal{O}_{\texttt{LODH}}^{\texttt{MP-PRF*-OODH}}(\texttt{prf}, \texttt{KEr}, \texttt{Ni}|\texttt{Nr})$

Else $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{DH})$

Call the adversary with the changed protocol to get its guess. If he wins, then return $h' \leftarrow 0$, else return $h' \leftarrow 1$ for game MP-PRF*-OODH.

Note that we differentiate between the roles Initiator and Responder. Due to our split between Initiator and Responder, we do not need to check for information of $\bar{\pi}^*$ and $\hat{\pi}^*$ at the same time as presented by Heussen et al. (2017). This allows us to focus on one protocol instance at a time, which simplifies the implementation.

A1	A2
	Initialize values only used for $\bar{\pi}^*$ and $\hat{\pi}^*$: For each allowed group Γ pick $a_\Gamma, b_\Gamma \in_R \mathbb{Z}_{q_\Gamma}$, set $A_\Gamma \leftarrow g_\Gamma^{a_\Gamma}$, $B_\Gamma \leftarrow g_\Gamma^{b_\Gamma}$, $\texttt{SKEYSEED}_{\texttt{mod}} \leftarrow \texttt{false}$. For each function prf pick $\texttt{trf}_{\texttt{prf}}^* \in_R \mathcal{F}$.
1.4. $\texttt{KEi} \leftarrow g^a$.	1.4. **If** either $\bar{\pi}^*$ or $\hat{\pi}^*$ has role Initiator **then** $\texttt{KEi} \leftarrow A_\Gamma$; $a \leftarrow \texttt{empty}$ **Else** $\texttt{KEi} \leftarrow g^a$.
3.4. $\texttt{KEi} \leftarrow g^b, \texttt{DH} \leftarrow \texttt{KEi}^b$	3.4. **If** either $\bar{\pi}^*$ or $\hat{\pi}^*$ has role Responder **then** $\texttt{KEr} \leftarrow B_\Gamma$; $b \leftarrow \texttt{empty}$ **Else** $\texttt{KEr} \leftarrow g^b, \texttt{DH} \leftarrow \texttt{KEi}^b$
5$^{\texttt{r}}$.1. $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{DH})$	5$^{\texttt{r}}$.1. **If** $b = \texttt{empty}$ **then** **If** $\texttt{KEi} = A_\Gamma$ **then** $\texttt{SKEYSEED} \leftarrow \texttt{trf}_{\texttt{prf}}^*(\texttt{Ni}, \texttt{Nr})$; $\texttt{SKEYSEED}_{\texttt{mod}} \leftarrow \texttt{true}$ **Else** $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{KEi}^{b_\Gamma})$ **Else** $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{DH})$
4.3. $\texttt{DH} \leftarrow \texttt{KEr}^a$	4.3. **If** $a \neq \texttt{empty}$ **then** $\texttt{DH} \leftarrow \texttt{KEr}^a$
5$^{\texttt{i}}$.1. $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{DH})$	5$^{\texttt{i}}$.1. **If** $a = \texttt{empty}$ **then** **If** $\texttt{KEr} = B_\Gamma$ **then** $\texttt{SKEYSEED} \leftarrow \texttt{trf}_{\texttt{prf}}^*(\texttt{Ni}, \texttt{Nr})$; $\texttt{SKEYSEED}_{\texttt{mod}} \leftarrow \texttt{true}$ **Else** $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{KEr}^{a_\Gamma})$ **Else** $\texttt{SKEYSEED} \leftarrow \texttt{prf}(\texttt{Ni}|\texttt{Nr}, \texttt{DH})$

Lemma 3: Game hop $\texttt{A1} \rightarrow \texttt{A2}$.
For every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$adv^{A1}(\mathcal{A}) \leq adv^{A2}(\mathcal{A}) + 2 \cdot adv^{MP\text{-}PRF^*\text{-}OODH}.$$

Proof. It is really important to check the substitution of a function thoroughly. The function has to be replaced if and only if the same key is used. If this does not hold, the adversary could be able to read differences between the left and right game, which is not covered by the game hop. Therefore we have to check the correctness of the key ourselves. Note that we only change the protocol for $\bar{\pi}^*$ and $\hat{\pi}^*$. Assume that $\bar{\pi}^*$ has the role `Initiator`. We only look at this protocol instance since $\hat{\pi}^*$ behaves in the same way and the case for the other role is analogous.
To prove this lemma we apply Game Hopping Lemma (i). We use $G_0 = \texttt{A1}$, $G_1 = \texttt{A2}$, $G = \texttt{MP-PRF*-OODH}$, and distinguisher $\mathcal{D} = \mathcal{D}^{\texttt{MP-PRF*-OODH}}_{\texttt{A1}\rightarrow\texttt{A2}}$. Note that the adversary cannot see the variable $\texttt{SKEYSEED}_{\texttt{mod}}$. To be able to apply the Game Hopping Lemma, we show that the properties of Lemma 1 hold.

1. *The view of the adversary is identical, whether interacting with G_{h^*} or $\mathcal{D}$.*
   If $\texttt{KEr} \neq B_\Gamma$, then the oracle $\mathcal{O}^{\texttt{MP-PRF*-OODH}}_{\texttt{LODH}}$ is called with `KEr`. This results in a normal Diffie-Hellman key exchange where we swapped `KEi` with $A_\Gamma = g^a_\Gamma$ and a with a_Γ. This leads to no difference and is independent of the hidden bit. Now consider the case where $\texttt{KEr} = B_\Gamma$ holds. Then we apply $\mathcal{O}^{\texttt{MP-PRF*-OODH}}_{\texttt{PRF}}$, which depends on the hidden bit. Separate this in the two following two cases.

   - left side, $h^{\texttt{MP-PRF*-OODH}}_{\Gamma,\texttt{prf}} = 0$

     In this case both, a and b, are swapped with a_Γ and b_Γ at every possible place. This is the same as the Diffie-Hellman key exchange, where the secret knowledge is just predefined by the game `MP-PRF*-OODH`. The adversary has the same knowledge about the game `A1`, therefore the view is identical.

   - right side, $h^{\texttt{MP-PRF*-OODH}}_{\Gamma,\texttt{prf}} = 1$

     Here, the oracle $\mathcal{O}^{\texttt{MP-PRF*-OODH}}_{\texttt{PRF}}$ applies a truly random function instead of the pseudorandom function `prf` with key $\texttt{DH}_\Gamma$. We have to check that every time `prf` with `DH` as secondary input is called, the whole function is replaced and only then. The same key is $\texttt{DH}_\Gamma = A^{b_\Gamma}_\Gamma = B^{a_\Gamma}_\Gamma$ by the distinguisher. The oracle $\mathcal{O}^{\texttt{MP-PRF*-OODH}}_{\texttt{PRF}}$ is only called if $a = \texttt{empty}$, which means $\texttt{KEi} = A_\Gamma$, and $\texttt{KEr} = B_\Gamma$. Therefore the Diffie-Hellman key is $\texttt{DH}_\Gamma$. Vice versa this holds for $\hat{\pi}^*$, where $b = \texttt{empty}$ and $\texttt{KEi} = A_\Gamma$. This means that we really swap to the truly random function only if the key is $\texttt{DH}_\Gamma$. Therefore

if the same pseudorandom function is used, we swap it with the same truly random function. This is exactly what we need to show, thus the adversary does not see any difference between the right side game of the distinguisher and the game A2.

2. *G accepts if and only if its hidden bit is guessed correctly.*
   This is true by definition of the game MP-PRF*-OODH.
3. *The answer of the distinguisher $\mathcal{D}$ is 0 if and only if $\mathcal{A}$ wins G_{h^*}, otherwise $\mathcal{D}$ returns 1.*
   This is true by definition of the returned guess of $\mathcal{D}^{\texttt{MP-PRF*-OODH}}_{\texttt{A1}\to\texttt{A2}}$.

Now by applying Lemma 1, we know that the assumptions for the Game Hopping Lemma hold. This proves the lemma.

□

5.2.3. Game hop: A2 → A3

Now we use the randomly assigned Diffie-Hellman key to further randomize the value of SKEYSEED. The game hop is defined through the following distinguisher playing the game MP-PRF.

Distinguisher: $\mathcal{D}^{\texttt{MP-PRF}}_{\texttt{A2}\to\texttt{A3}}$

Input: $\mathcal{O}^{\texttt{MP-PRF}}$

Start with $\texttt{SK}_{\texttt{mod}} \leftarrow \texttt{false}$. Initialize $\texttt{Ni}^\circ = \texttt{Ni}$, $\texttt{Nr}^\circ = \texttt{Nr}$, and $\texttt{prf}^\circ = \texttt{prf}$, where Ni and Nr are the nonces of the $\bar{\pi}^*$ or $\hat{\pi}^*$ and prf is the used pseudorandom function specified in the security association, depending on who gets the role Responder first. Embed the oracles in the following line of game A2 for Initiator and Responder respectively:

5.2. **If** $\texttt{SKEYSEED}_{\texttt{mod}} = \texttt{true}$,
$\texttt{Ni} = \texttt{Ni}^\circ$,
$\texttt{Nr} = \texttt{Nr}^\circ$, and
$\texttt{prf} = \texttt{prf}^\circ$ **then**
$\texttt{SK} \leftarrow (\mathcal{O}^{\texttt{MP-PRF}})\texttt{+}(\texttt{prf}, \texttt{Ni}|\texttt{Nr}|\texttt{SPIi}|\texttt{SPIr})$
Else $\texttt{SK} \leftarrow \texttt{prf+}(\texttt{SKEYSEED}, \texttt{Ni}|\texttt{Nr}|\texttt{SPIi}|\texttt{SPIr})$

Call the adversary with the changed protocol to get its guess. If he wins, then return $h' \leftarrow 0$, else return $h' \leftarrow 1$ for game MP-PRF*-OODH.

Note that $(\mathcal{O}^{\texttt{MP-PRF}})\texttt{+}$ means that the oracle is used in the same way as the pseudorandom function prf to produce prf+.

A2	A3
	Initialize $\mathtt{SK}_{\mathtt{mod}} \leftarrow \mathtt{false}$. Let $\mathtt{Ni}^{\times}$, $\mathtt{Nr}^{\times}$ and $\mathtt{prf}^{\times}$ be defined through first appearance of $\bar{\pi}^*$ or $\hat{\pi}^*$ with role `Responder`, set $\mathtt{Ni}^{\circ} \leftarrow \mathtt{Ni}^{\times}$, $\mathtt{Nr}^{\circ} \leftarrow \mathtt{Nr}^{\times}$, and $\mathtt{prf}^{\circ} \leftarrow \mathtt{prf}^{\times}$.
5.2. $\mathtt{SK} \leftarrow \mathtt{prf+}(\mathtt{SKEYSEED}, \mathtt{Ni}\|\mathtt{Nr}\|\mathtt{SPIi}\|\mathtt{SPIr})$	5.2. **If** $\mathtt{SKEYSEED}_{\mathtt{mod}} = \mathtt{true}$, $\mathtt{Ni} = \mathtt{Ni}^{\circ}$, $\mathtt{Nr} = \mathtt{Nr}^{\circ}$, and $\mathtt{prf} = \mathtt{prf}^{\circ}$ **then** $\mathtt{SK} \leftarrow \mathtt{trf}_{\mathtt{prf}}\mathtt{+}(\mathtt{Ni}\|\mathtt{Nr}\|\mathtt{SPIi}\|\mathtt{SPIr})$ $\mathtt{SK}_{\mathtt{mod}} \leftarrow \mathtt{true}$ **Else** $\mathtt{SK} \leftarrow$ $\mathtt{prf+}(\mathtt{SKEYSEED}, \mathtt{Ni}\|\mathtt{Nr}\|\mathtt{SPIi}\|\mathtt{SPIr})$

Lemma 4: Game hop $\mathtt{A2} \rightarrow \mathtt{A3}$**.**
For every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$adv^{\mathtt{A2}}(\mathcal{A}) \leq adv^{\mathtt{A3}}(\mathcal{A}) + 2 \cdot adv^{\mathtt{MP\text{-}PRF}}.$$

Proof. This proof is similar to the last one. The oracle $\mathcal{O}^{\mathtt{MP\text{-}PRF}}$ exactly either uses $\mathtt{prf}(\mathtt{SKEYSEED}, \mathtt{Ni}\|\mathtt{Nr}\|\mathtt{SPIi}\|\mathtt{SPIr})$ or $\mathtt{trf}_{\mathtt{prf+}}(\mathtt{Ni}\|\mathtt{Nr}\|\mathtt{SPIi}\|\mathtt{SPIr})$ for hidden bit 0 and 1 respectively. We need to check, that every time `prf` and `SKEYSEED` are the same for $\bar{\pi}^*$ and $\hat{\pi}^*$, they also get the same truly random function in game `A3`. Since the oracle $\mathcal{O}^{\mathtt{MP\text{-}PRF}}$ is only called if the value $\mathtt{SKEYSEED}_{\mathtt{mod}}$ is set to true and the nonces are the same, then we know that $\mathtt{SKEYSEED} = \mathtt{trf}^*_{\mathtt{prf}}(\mathtt{Ni}\|\mathtt{Nr})$ holds. Therefore as long as the nonces and the pseudorandom function are the same, which we define through the if-statement, the oracle is only called for a fixed and specific key `SKEYSEED`. □

5.2.4. Game hop: A3 → A4

As our third game hop, we apply a distinguisher playing `MP-PRF` to randomize the key material `KEYMAT`. If $\mathtt{SK}_{\mathtt{mod}}$ is set to true, we know by game `A3` that the nonces and security association have to be equal. Therefore we only need to check if $\mathtt{SK}_{\mathtt{mod}}$ is set to true and the equality of the security parameter indices `SPIi` and `SPIr` to ensure that the same variable $\mathtt{SK}_{\mathtt{d}}$ is used.

Distinguisher: $\mathcal{D}^{\texttt{MP-PRF}}_{\texttt{A3}\to\texttt{A4}}$

Input: $\mathcal{O}^{\texttt{MP-PRF}}$

Initialize $\texttt{SPIi}^\circ = \texttt{SPIi}$ and $\texttt{SPIr}^\circ = \texttt{SPIr}$, where SPIi and SPIr are the security parameter indices of $\bar{\pi}^*$ or $\hat{\pi}^*$, depending on who gets the role Responder first. Embed the oracles in the following line of game A3 for Initiator and Responder respectively:

10.1. **If** $\texttt{SK}_{\texttt{mod}} = \texttt{true}$ and $\texttt{SPIi}|\texttt{SPIr} = \texttt{SPIi}^\circ\,|\texttt{SPIr}^\circ$ **then**
$\texttt{KEYMAT} \leftarrow (\mathcal{O}^{\texttt{MP-PRF}})\texttt{+}(\texttt{prf}, \texttt{Ni}|\texttt{Nr})$
Else $\texttt{KEYMAT} \leftarrow \texttt{prf+}(\texttt{SK}_\texttt{d}, \texttt{Ni}|\texttt{Nr})$

Call the adversary with the changed protocol to get its guess. If he wins, then return $h' \leftarrow 0$, else return $h' \leftarrow 1$ for game MP-PRF*-OODH.

A3	A4
	Initialize $\texttt{KEYMAT}_{\texttt{mod}} \leftarrow \texttt{false}$. Let $\texttt{SPIi}^\times$ and $\texttt{SPIr}^\times$ be defined through first apperance of $\bar{\pi}^*$ or $\hat{\pi}^*$ with role Responder, set $\texttt{SPIi}^\circ \leftarrow \texttt{SPIi}^\times$ and $\texttt{SPIr}^\circ \leftarrow \texttt{SPIr}^\times$.
10.1. $\texttt{KEYMAT} \leftarrow \texttt{prf+}(\texttt{SK}_\texttt{d}, \texttt{Ni}\|\texttt{Nr})$	10.1. **If** $\texttt{SK}_{\texttt{mod}} = \texttt{true}$ and $\texttt{SPIi}\|\texttt{SPIr} = \texttt{SPIi}^\circ\,\|\texttt{SPIr}^\circ$ **then** $\texttt{KEYMAT} \leftarrow \texttt{trf}_{\texttt{prf}}\texttt{+}(\texttt{Ni}\|\texttt{Nr})$ $\texttt{KEYMAT}_{\texttt{mod}} \leftarrow \texttt{true}$ **Else** $\texttt{KEYMAT} \leftarrow \texttt{prf+}(\texttt{SK}_\texttt{d}, \texttt{Ni}\|\texttt{Nr})$

Lemma 5: Game hop A3 → A4.
For every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$adv^{A3}(\mathcal{A}) \leq adv^{A4}(\mathcal{A}) + 2 \cdot adv^{MP\text{-}PRF}.$$

Proof. This proof is analogous to the one before. Note that by the if-statement we check that $\texttt{SK}_\texttt{d}$ and prf are the same function every time the oracle is called. □

5.2.5. Game hop: A4 → A5

Up to this point the proof is similar to the one presented by Heussen et al. (2017). We use two additional games, which are equal to the game A4. This simplifies the last step, where we have to show that the last game of the sequence is a null-game.
This game hop changes the oracle $\mathcal{O}_{\texttt{Test}}$ in such a way that the hidden bit is not used in any of the relevant cases anymore. This leads to the adversary having no access to anything related with the hidden bit, which we show later on.

A4

Oracle $\mathcal{O}_{\texttt{Test}}$:
1: **If** this oracle as already been called **then Return** $\perp$
2: **If** π has not accepted **then Return** $\perp$
3: Define $\texttt{KEY}_0 \leftarrow \pi.\texttt{KEYMAT}$
4: Define $\texttt{KEY}_1 \in_R \mathcal{K}$
5: **Return** $\texttt{KEY}_h$

A5

Redefine the oracle $\mathcal{O}_{\texttt{Test}}$:
1: **If** this oracle as already been called **then Return** $\perp$
2: **If** π has not accepted **then Return** $\perp$
3: Define $\texttt{KEY}_0 \leftarrow \pi.\texttt{KEYMAT}$
4: Define $\texttt{KEY}_1 \in_R \mathcal{K}$
5: **If** π has $\texttt{KEYMAT}_{\texttt{mod}}$ set to `true` **then Return** $\texttt{KEY}_1$
6: **Else Return** $\texttt{KEY}_h$

Lemma 6: Game hop A4 → A5.
For every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$adv^{A4}(\mathcal{A}) = adv^{A5}(\mathcal{A}).$$

Proof. First, note that the adversary only gets information about the hidden bit from the oracle $\mathcal{O}_{\texttt{Test}}$. Let π be the protocol instance that is called to $\mathcal{O}_{\texttt{Test}}$. During the check for a real-or-random attack, $\bar{\pi}'$ is set to π. Consider its value $\texttt{KEYMAT}_{\texttt{mod}}$. If this is false, then nothing changes and the games are equal. Look at the other case where $\texttt{KEYMAT}_{\texttt{mod}} = \texttt{true}$ holds. We split this into three cases.

$\bar{\pi}' \neq \bar{\pi}^*$

By game A1 the adversary has to guess $(\bar{\pi}^*, \hat{\pi}^*) = (\bar{\pi}', \hat{\pi}')$ in order to have an impact on the outcome of the game. Since this is not true, both games return the same, a random decision.

$\bar{\pi}' = \bar{\pi}^*$ with $\bar{\pi}^*$ and $\hat{\pi}^*$ do not communicate

The definition of a real-or-random attack defined through game RORA states that $\hat{\pi}'$ is defined as the only communication partner for $\bar{\pi}'$. If $\bar{\pi}^*$ and $\hat{\pi}^*$ do not communicate, that means that $\hat{\pi}^* \neq \hat{\pi}'$ holds. Analogous to before, in this case the games are equal,

since the adversary has no impact on the decision anymore. Even if he could detect the change from game A4 to A5, at this point where he is able to detect it, he already called $\mathcal{O}_{\texttt{Test}}$. This means that he cannot change the outcome of the game anymore.

$\bar{\pi}' = \bar{\pi}^*$ with $\bar{\pi}^*$ and $\hat{\pi}^*$ communicate

This case means that $\bar{\pi}^*$ and $\hat{\pi}^*$ have $\texttt{KEYMAT}_{\texttt{mod}}$ set to true, since every variable they use are the same. Consequently by game A4, both protocol instances have $\texttt{trf}_{\texttt{prf}}\texttt{+(Ni|Nr)}$ as the key material. Because this value is assigned through a truly random function, it is the same as a random assignment. Thus we know that in this case $\texttt{KEY}_0$ and $\texttt{KEY}_1$ are both randomly picked. This is the same as always returning $\texttt{KEY}_1$. Therefore the adversary does not detect any difference at all, which means that in both games he has the same probability to win.

We know that in each possible case the adversary has the same winning chance in game A4 and game A5. Since the advantage is defined through the probability to win, the equality of winning chance already implies the equality of the advantages. This proves the lemma. □

5.2.6. Game hop: A5 → A6

This is the last game hop. It changes the oracle $\mathcal{O}_{\texttt{Test}}$ such that the hidden bit is not used anymore. We always return the random key $\texttt{KEY}_1$ to the adversary. This way he has no information anymore, therefore he is only able to guess.

A5

Oracle $\mathcal{O}_{\texttt{Test}}$:

1: **If** this oracle as already been called **then Return** $\perp$
2: **If** π has not accepted **then Return** $\perp$
3: Define $\texttt{KEY}_0 \leftarrow \pi.\texttt{KEYMAT}$
4: Define $\texttt{KEY}_1 \in_R \mathcal{K}$
5: **If** π has $\texttt{KEYMAT}_{\texttt{mod}}$ set to `true` **then Return** $\texttt{KEY}_1$
6: **Else Return** $\texttt{KEY}_h$

A6

Redefine the oracle $\mathcal{O}_{\texttt{Test}}$:

1: **If** this oracle as already been called **then Return** $\perp$
2: **If** π has not accepted **then Return** $\perp$
3: Define $\texttt{KEY}_0 \leftarrow \pi.\texttt{KEYMAT}$
4: Define $\texttt{KEY}_1 \in_R \mathcal{K}$
5: **Return** $\texttt{KEY}_1$

Lemma 7: Game hop A5 → A6**.**
For every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$adv^{A5}(\mathcal{A}) = adv^{A6}(\mathcal{A}).$$

Proof. Let $\pi = \bar{\pi}'$ be the protocol instance called to $\mathcal{O}_{\texttt{Test}}$. In game A5 the returned value of $\mathcal{O}_{\texttt{Test}}$ is already random if $\texttt{KEYMAT}_{\texttt{mod}} = \texttt{true}$ holds. Therefore we only need to consider the case where $\texttt{KEYMAT}_{\texttt{mod}}$ is set to false. By games A2 and A3 we see that $\texttt{KEYMAT}_{\texttt{mod}}$ is only possibly be set to true, if the corresponding protocol instance is either $\bar{\pi}^*$ or $\hat{\pi}^*$. Thus we split into the following cases.

$\bar{\pi}' \neq \bar{\pi}^*$

Analogous to before, this means that during the call to $\mathcal{O}_{\texttt{Test}}$ the adversary lost his impact to the game's decision. The output is random in both games, meaning that the games are equal in this case.

$\bar{\pi}' = \bar{\pi}^*$

In this case $\bar{\pi}^*$ and $\hat{\pi}^*$ do not communicate. Otherwise $\texttt{KEYMAT}_{\texttt{mod}}$ would have been set to true. This means that no real-or-random attack is detected, resulting in same winning chance in both games.

Since all winning probabilities are the same, this proves the lemma. □

5.2.7. Null-game: A6

We now have a game sequence, where we have an explicit bound for each two consecutive games. As the next step we have to show that the last game of the sequence has no advantage over randomly guessing. We formulate this as the following lemma.

Lemma 8: Null-game A6**.**
For every probabilistic polynomial time adversary $\mathcal{A}$ we have

$$adv^{\texttt{A6}}(\mathcal{A}) = 0$$

Proof. The hidden bit was only used in the oracle $\mathcal{O}_{\texttt{Test}}$ in game RORA. Since this is not the case in game A6 anymore, the adversary has no access to any information connected to the hidden bit. For this reason guessing is his only choice, therefore the advantage is 0. □

5.3. Proof of Theorem

We apply the sequence of game hops to prove the main theorem, Real or random security for IKEv2 in the game AKE, (Heussen et al., 2017).

Proof. Consider the game sequence [RORA, A1, A2, A3, A4, A5, A6]. This gives us the following bound for the advantage of the game RORA.

$$
\begin{aligned}
\mathrm{adv}^{\mathtt{RORA}}(\mathcal{A}) &\overset{\text{Lemma 2}}{=} \mathsf{q}\cdot(\mathsf{q}-1)\cdot\mathrm{adv}^{\mathtt{A1}} \\
&\leq \mathsf{q}^2\cdot\mathrm{adv}^{\mathtt{A1}} \\
&\overset{\text{Lemma 3}}{\leq} \mathsf{q}^2\cdot\left(2\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}+\mathrm{adv}^{\mathtt{A2}}\right) \\
&\overset{\text{Lemma 4}}{\leq} \mathsf{q}^2\cdot\left(2\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}+2\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF}}+\mathrm{adv}^{\mathtt{A3}}\right) \\
&\overset{\text{Lemma 5}}{\leq} \mathsf{q}^2\cdot\left(2\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}+4\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF}}+\mathrm{adv}^{\mathtt{A4}}\right) \\
&\overset{\text{Lemma 6}}{=} \mathsf{q}^2\cdot\left(2\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}+4\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF}}+\mathrm{adv}^{\mathtt{A5}}\right) \\
&\overset{\text{Lemma 7}}{=} \mathsf{q}^2\cdot\left(2\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}+4\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF}}+\mathrm{adv}^{\mathtt{A6}}\right) \\
&\overset{\text{Lemma 8}}{=} \mathsf{q}^2\cdot\left(2\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF^*\text{-}OODH}}+4\cdot\mathrm{adv}^{\mathtt{MP\text{-}PRF}}\right)
\end{aligned}
$$

Consider the assumptions of the security of the used primitives. Then the games MP-PRF*-OODH and MP-PRF have negligible advantage, therefore we know that the game RORA has negligible advantage too. □

6. Implementation

This chapter focuses towards everything associated with the implementation of the proof. First we go through the language of EasyCrypt and its capabilities. Afterwards we present the code and its structure. The subsequent section consists of all hard limitations due to EasyCrypt's engine. This is followed by the difficulties during implementation and its implied restrictions that were necessary.

6.1. EasyCrypt

EasyCrypt is a computer-aided toolset for cryptographic proofs. To get in-depth details, see Institute, Inria, and École Polytechnique (2014). Its engine is based on the programming language OCaml. EasyCrypt itself is used for interactively finding, constructing and machine-checking security proofs. It suits the language of games, therefore is close to the proof in the previous chapter. By itself it does not provide any automatic proof generation. It supports the automatic provers Alt-Ergo and Z3, which can be used for logical statements. EasyCrypt can be run in two different setups. One, through the command line, where it always checks the whole code at once. The other setup needs to use emacs and ProofGeneral, which allows to run EasyCrypt in an interactive mode. In the following we recall important terminologies of EasyCrypt and present an example for this used in the implementation.

Types

A `type` is a mathematical set with some restrictions, mostly due to first order logic. Just by declaring a `type`, it only exists and has an arbitrary number of elements. It does not hold any other property at the point of declaration. Building upon that, we give it a structure and can define mathematical constructs through this. For example, if we want to construct a group, we define the corresponding group operation to get this result. Even though it is a powerful tool, it has some limitations. One example is that we cannot generate a set of functions, a powerset, or, to be more precise, a set which contains sets.

Modules and procedures

To model games and oracles we use EasyCrypt's predefined object called `module`. It consists of global variables and procedures. A `procedure` has at least one type as input, which it uses to operate on to generate some output. We can grant the adversary access to a procedure, which we need to allow him to call the oracles. We use two modules per game in the implementation. One for all the oracles, where each oracle is implemented as a procedure. The other module consists of one procedure, which is the game the adversary plays itself. For simplicity we split some larger operations, like the game `AKE`, into several smaller parts and call those in the main procedure.

Axioms, lemmas and proofs

An `axiom` is a logic statement that EasyCrypt always assumes to be correct. It is not automatically used, we have to explicitly state its usage during a proof. Note that it is possible to define axioms that are contradicting each other. Therefore they should only be used with care. We use axioms to define uniform distributions, a truly random function, and term equalities. For example define two sets `type X.` and `type Y.`, and two different functions `f : X -> Y` and `g : Y -> X`. We then define the following axiom `axiom equality_f_g : forall(x), g(f(x))= x.` to state that `g` is the left-inverse of `f`. This is especially useful to specify the correctness of the signature scheme and authenticated encryption scheme without specifying it. Similar to the example given before, we define the correctness of the schemes directly as an axiom without further implementation.

Contrary to an axiom, a `lemma` is a logical statement in EasyCrypt, which is not assumed to be true. We first have to prove that a lemma is correct. Afterwards we can use it the same way as an axiom. Every game hop consists of assumptions that need to be shown. This results in a different lemma for each assumption.

Note that procedures of modules can be used as variables in a logical statement. This way we define the probability that a procedure, for example the game, returns a specific value. This feature is the reason why EasyCrypt is suited for cryptographic proofs, especially in the game language. We formulate the chance an adversary has at winning a game in the following way. Let `Game` be a module containing the procedure `main`, which uses no input and returns either `ACCEPT` or `REJECT`. Then we define that this procedure has a fifty percent chance to return `ACCEPT` in the following way:

```
lemma null-game &m : Pr[Game.main()@ &m : res = ACCEPT] = 1%r/2%r.
```

The variable `&m` is a memory-variable, which is used to call `res`, the returned value of the procedure `Game.main()`. The term `1%r/2%r` represents the rational number $1/2$. Instead of showing the equality of probabilities, we can also show the equalities of the probabilities of procedures. For this we have to replace the probability-expression `1%r/2%r` with the probability-notion of the procedure as shown on the left hand side. We use this concept to check the assumptions needed to apply the Game Hopping Lemma. This is done by formulating the equality for both games, appearing at the (left side) and (right side) of the Game Hopping Lemma respectively. In EasyCrypt a proof for a lemma is interactively constructed in its language. The statement of the lemma is transformed into a *goal*. Every command modifies this goal and can even divide it into several smaller goals. Only after each single goal has shown to be true, then the whole lemma is accepted and holds. If the lemma itself is false, then at least one of the goals will lead to a dead end with a false statement. The reverse conclusion is not true, meaning that not finding a proof in EasyCrypt does not mean that the lemma itself has to be false.

Abstract adversary

To formulate a full game, we need to call the adversary at some point. For this, we define him as an abstract module. It is a module for which the method of operation is not defined. EasyCrypt quantifies over the power of the abstract adversary, meaning that it only considers the view of the adversary to check similarities in different scenarios. The idea behind that is that we use the same adversary in two different games. If everything he gets from both sides is equally structured, then both games are played the same way and therefore equal from his point of view. This is the general idea behind EasyCrypt's modus operandi. We have to grant the adversary access to certain modules and restrict him, such that he is not able to call global variables from the modules. Further on, we need to specify which adversary is used in which game. This means that above the procedure has to be called with `Game(Adversary).main()`, where `Adversary` is the module of the adversary.

6.2. The Code

In the following we present the structure and important details of the implementation. The written code consists of about 9500 lines. Out of those 2500 lines are for formulating and defining the games, oracles and the protocol. This starts with several different types for the definition of a protocol instance and a party. Building upon that we define several functions to simplify the handling of the protocol list. After that the code consists of the oracles for the three games `MP-PRF*-OODH`, `MP-PRF` and `AKE`. In the game hops `A2` $\rightarrow$ `A3` and `A3` $\rightarrow$ `A4` the oracle `MP-PRF` is used to replace a pseudorandom function. These two instances of the usage

of the oracle $\mathcal{O}^{\texttt{MP-PRF}}$ differentiate in the type of the input of the pseudorandom function. In the first of these two game hops the input consists of the concatenation of two nonces and two security parameter indices. The second hops uses only the nonces for the pseudorandom function. Since EasyCrypt focuses on the equality of terms and types, this means that the oracle $\mathcal{O}^{\texttt{MP-PRF}}$ needs to be implemented twice. Once for each different input. The oracle $\mathcal{O}_{\texttt{Send}}$ is the one with the most changes through the game sequence. To overcome the issue of duplicating errors, instead of rewriting this oracle every time, one variable for every game hop is used. This variable is a boolean and combined with an if-statement directs to the changes. Those variables have the prefix `gs_`. As an example consider game hop `A2` $\rightarrow$ `A3`. This hop changes the step 5.2. of the protocol. If the variable `gs_3` is set to false, then the protocol is run at this step as normal. Otherwise the modification as described in game `A3` is used. The same is done for the combination of the distinguisher and the adversary. Variables for these combinations have the prefix `gs_oracle_`. To be precise, this example is implemented in the following way.

```
(* Step 5.2 *)
sSK <- prfplus(sSKEYSEED,(nNi,nNr,sSPIi,sSPIr)) (* else case *);
if(protocol_get_keyseed_mod(pi) = true /\
  nNi = star_Ni /\
  nNr = star_Nr)
   (* equal nonces => equal sSKEYSEED *)
 {
  if(gs_a3 = true) {
    sSK <- trf_prfplus(nNi,nNr,sSPIi,sSPIr);
    pi <- protocol_set_sk_mod(pi)(true);
  }
  if(gs_a3_oracle = true) {
    sSK <- Oracle_A2_A3.prf(pick_prf,(nNi,nNr,sSPIi,sSPIr));
    if(Oracle_A2_A3.h = true) pi <- protocol_set_sk_mod(pi)(true);
  }
}
```

A short explanation considering the naming convention: EasyCrypt only allows variables to start with a lowercase letter. Therefore every variable starting with an uppercase letter defined in the protocol IPsec (see section 3.1) has the same letter in lowercase appended in front. The variables `star_Ni` and `star_Nr` are the presentations of `Ni`$^\circ$ and `Nr`$^\circ$. The oracle `Oracle_A2_A3` is one of the implementations of the oracle $\mathcal{O}^{\texttt{MP-PRF}}$ as stated before, where `pick_prf` is as formalization to pick the pseudorandom function.

The games themselves are defined subsequently to to the oracles. This contains the games AKE, MP-PRF*-OODH, and MP-PRF, completing the first part. The second part, which is 7000 lines long, consists of the proofs. This includes every assumption needed to apply the Game Hopping Lemma used in chapter 5. The application itself is omitted. Out of all the game hops, only the last one, namely A5 → A6 is missing. Reason for this is stated below in section 6.4. The code of the complete state-machine of the protocol as presented in chapter 3 can be seen in appendix A.

Technical Aspects

On a virtual machine with one 3.5GHz kernel and roughly 10GB RAM the code needs about 30 minutes to run without interactive mode. This time increases rapidly for every time the run is stopped. Trying to run the code on a laptop with 4GB RAM in interactive-mode lead to a deadlock. It was not possible to process some steps in the proof section with this hardware. This is most likely due to the fact that EasyCrypt's complexity scales badly with increased number of used variables. Typically automatic provers have more than an exponential complexity of the amount of variables. No information about runtime or memory storage complexity is given by the developers of EasyCrypt yet.

6.3. Limitations

EasyCrypt has a lot of restriction due to its engine's logic and specific usage of variables. We take a look at the most important constraints and limitations for the implementation and discuss the resulting changes.

Set of sets

EasyCrypt has the option to use unknown types or type parameters indicated by ‘ in front of the type. This means that a function `op prf : k * x -> z.` can be declared as `op prf : ‘k * ‘x -> ‘z.` to have something similar to a template of a function. The variables k, x, and z all need to be defined, which means the function prf is fixed towards those variables. On the other hand ‘k, ‘x, and ‘z are unknown type variables. In the second version of the function prf is defined for every arbitrary set combination. Unknown types can only be used in type declarations, not in procedures. This leads to a problem in using the games MP-PRF and MP-PRF*-OODH. There we have a set of keyed functions $\mathcal{F}$, which EasyCrypt is not able to define through a set of sets. Each game and oracle has to be implemented as a module with procedures, which is not able to use unknown types. For that reason we can only implement a function for specific sets. Consequently we restrict every pseudorandom function to the same domain and codomain. The same problematic aspects

can be found in the security association, the set of possible algorithms and groups to choose from for the protocol. Every algorithm and group has to be defined and controlled. Due to this high complexity, we restrict the security association to a single element. Consequently we omit the implementation of the security association altogether. EasyCrypt still proves the same with or without it, since it only checks for equalities of terms. If the protocol allows to use different algorithms, we only have to check that the same ones are used. Even if we have a restriction on a larger scale, it has no impact on the implemented proof or its verification.

Layers of operation

Consider the module representing a game. It first initializes the oracles, then it invokes the adversary, who is able to call the oracles himself. This leads to a specific structure during a proof in EasyCrypt. In the first layer, denote it as the *game layer*, we have access to everything the game is able to view, define and operate on. At some point we have to call the adversary and check everything he is able to compute. This leads to a new layer, denote it as the *oracle layer*. During this step, we only have the view and knowledge of the adversary, meaning that we cannot access any variables of the game directly anymore. In contrast to the theoretical view, where the game controls the protocol, in EasyCrypt the protocol is called through an oracle by the adversary. This results in the protocol begin in the oracle layer instead of the game layer. It is possible to transfer information in the format of logical statements from the game layer to the oracle layer. Even then, this still leads to a high loss in information. Due to this we need to change several steps in the games and the protocol.

First, consider the variables a and b for the Diffie Hellman key exchange. During game `A2` those variables are swapped with a_Γ and b_Γ in the oracle layer. EasyCrypt considers a and a_Γ as equal if they are picked from the same distribution by the same command. Since a is defined in the oracle layer and a_Γ in the game layer, we do not see the command which was used to define a_Γ the moment a is picked. To formulate this as a logical statement and transfer it all the way through both layers is highly complex and may even be impossible. Therefore we use another option, which is to predefine the variables a and b in the game layer for $\bar{\pi}^*$ and $\hat{\pi}^*$. This implementation does not change the way the protocol operates, it only gives us a new perspective.

Second, consider the oracle $\mathcal{O}_{\texttt{Test}}$, which is changed in the game hop from game `A4` to `A5`. We split the proof for this game hop into the cases depending on if $\bar{\pi}^*$ and $\hat{\pi}^*$ communicate or not. This is something only the game is able to tell, since the adversary has no access to the protocols themselves. Consequently for the sake of the proof, we need $\mathcal{O}_{\texttt{Test}}$ to be in the game layer, while it is called by the adversary, meaning it is in the oracle layer. Therefore we

work around this by splitting the adversary into two phases. In the following we check that this modification is applicable and no information is lost this way. It is not possible to prove such a statement in EasyCrypt, since this toolset only allows us to formulate properties of one adversary used several times. We are not able to state the equality of different adversaries. Note that $\mathcal{O}_{\texttt{Test}}$ is a one-time oracle and if the adversary does not call it, no real-or-random attack is detected by default. Thus this case is not interesting for us, since the advantage here is always 0. Consider the case where the adversary calls the oracle $\mathcal{O}_{\texttt{Test}}$. Then we exchange this adversary $\mathcal{A}$ with two adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$. We define $\mathcal{A}_1$ as the same as $\mathcal{A}$ up to the point where $\mathcal{O}_{\texttt{Test}}$ would be called. The game gets from $\mathcal{A}_1$ the information which protocol instance he wants to call the oracle with. Then the game does this call and forwards its output to $\mathcal{A}_2$, who does the same as $\mathcal{A}$ after $\mathcal{O}_{\texttt{Test}}$ was called. As long as $\mathcal{A}_1$ and $\mathcal{A}_2$ have access to the same oracles as $\mathcal{A}$, besides the test oracle, this transformation leads to the equality of the games and corresponding adversaries. This split allows to implement the oracle $\mathcal{O}_{\texttt{Test}}$ at the game layer instead of the oracle layer.

Return old values

Consider the oracle $\mathcal{O}^{\texttt{MP-PRF}}$, which wants to return the old answer if it was seen before. To achieve that we would have to implement a list and counter to save every call that was done to this oracle. That leads to a structure which is complex to handle. Instead we use a truly random function for hidden bit 1 instead of a random pick from the codomain. This way we ignore the step "return old answer" and only need to call the same function as before to return the same output. The only reason other literature does not define it this way lies in the impossibility to define a truly random function in polynomial time.

6.4. Difficulties and Restrictions

In this section we take a look at some of the problems that occurred while implementing the proofs in EasyCrypt. Their main difference to the limitations above lies in the fact that it is possible to implement them, though either with high complexity or time consuming work. We work around some of the problems using theoretical arguments, which are not proven in EasyCrypt. The problems that are not fixed yet yield restrictions towards the verification of the proof, which we also discuss. These restrictions are not severe, such that the code is still usable to attest the security of IPsec. Note that the implementation itself does not show the security of the RORA-scenario of IPsec. It assists the verification of the proof in chapter 5. This means that the upcoming arguments and discussion need to validate this verification and not the proof by itself.

Last game hop $\texttt{A5} \rightarrow \texttt{A6}$

The general idea of EasyCrypt being able to show that a game is a null-game, is by removing the hidden bit completely. This removal is problematic in the last game hop, which also defines the null-game A6.

Consider the case where π, called to $\mathcal{O}_{\texttt{Test}}$, has $\texttt{SK}_{\texttt{mod}}$ set to `false`. Then we need to argue that whatever the adversary is doing after that point has no influence towards the game's decision anymore. This is not possible to state in the current implementation in EasyCrypt. Remember that the adversary is split into two phases such that $\mathcal{O}_{\texttt{Test}}$ is in the game layer. In the second phase of the adversary, he is still able to manipulate the list of protocols $\mathcal{Q}$, since he can call $\mathcal{O}_{\texttt{Send}}$. On the other hand, we need that he can only change protocol instances that are not important anymore. In theory the game does not even need to call the adversary a second time. This is something too complex to implement as logical statements in EasyCrypt. Therefore this game hop is not proven and verified in EasyCrypt. One possible solution towards this problem would be to restructure the protocols $\bar{\pi}^*$ and $\hat{\pi}^*$, such that they are separated entities and not considered as elements of the list of all protocols $\mathcal{Q}$. Then we only need to check that $\bar{\pi}^*$ and $\hat{\pi}^*$ are not changed, which is definitely possible.

Oracles $\mathcal{O}_{\texttt{LODH}}^{\texttt{MP-PRF*-OODH}}$ and $\mathcal{O}_{\texttt{RODH}}^{\texttt{MP-PRF*-OODH}}$

The left and right oracles of the game MP-PRF*-OODH are one-time oracles. Trying to implement that fact during game hop $\texttt{A1} \rightarrow \texttt{A2}$ resulted in the problem that it was impossible to check at which point they are called. At the same time the proof for this game hop needs this information. It should be possible to specify the one-time call with enough properties inserting into the step from game layer to oracle layer, though this would arise much more work than benefit. Instead we allow the oracles to be called several times. From theory we know that they only appear at most once during the game. Therefore the version for the code is not different to the theoretical view. Note that the one-time fact is also used in the proof, since we did not define what the protocol is going to do if the oracle is called another time.

Hidden bits of MP-PRF*-OODH

Considering the hidden bits of the game MP-PRF*-OODH, one is used for every pseudorandom function. In the game hops where we use this game, we only need to consider one single hidden bit. This means that instead of implementing a list of hidden bits, we can just use one single hidden bit and simplify the implementation this way. Note that this change is only applicable since we use the game MP-PRF*-OODH for a game hop and never consider it by itself. Otherwise we would need more than just one predefined hidden bit, since the oracle $\mathcal{O}_{\texttt{PRF}}^{\texttt{MP-PRF*-OODH}}$ can be called several times.

Truly random function

EasyCrypt by itself sets one variable in two different games to be equal if they are generated by the same assignment call. It is further possible to define a bijective map between two assignment calls to show their equality. If one variable is generated through an assignment call and another is the output of a function, we are not able to show their equality directly. This problem arises in the game hops `A2` $\rightarrow$ `A3` and `A3` $\rightarrow$ `A4`. There in the games themselves the pseudorandom function uses a key that is the output of a truly random function. On the other side, the distinguisher uses a randomly picked key. Those two different constructions of the key behave in the same way, therefore they are equal from the view of the adversary. In EasyCrypt we are not able to explain this besides axioms, which we have to use in this instance.

Pseudorandom function plus

Consider the definition of `prf+`. It is a method to obtain up to 255 times more output from the pseudorandom function `prf`. Currently we restrict that to `prf+` being a pseudorandom function by itself, severing its connections to `prf`. Computation and memory rises, probably at an exponential rate, with every variable. Since `prf+` would be presented as the combination of 255 times `prf` applied every time it is called, this would result in a complexity higher than possible practical purpose. The property that `prf+` is a pseudorandom function on its own is also used in the theoretical proof. Therefore this does not decrease the relevance of the proof verification.

Bitstrings

In practice, the information sent during IPsec is in the format of bitstrings. The type of a bitstring is already defined through EasyCrypt's library, though impractical in the usage of a large protocol due to its complicated definition. Since every bitstring is just an integer with base 2, the code consists of integers only.

Pointer to protocol instance

The set $\mathcal{Q}$ is implemented as a list. This means that every time a protocol instance is changed, we first have to remove it from the list and after changing, add it again. To pick a specific protocol instance, we use its ID to identify its position. By the structure of a list in EasyCrypt, it is difficult to tie a protocol instance to its ID. Therefore this correlation is often omitted in a proof. This abundance also impacts the last game hop, which we discuss now.

Reference Manual

EasyCrypt is a work in progress and has a fragmentary reference manual. Most commands are presented in the manual, though there are hidden options for some of those. As an example the command `seq n : (goal)` divides one statement about a game into two. The first one consists of the first n lines of the game and has to show that `(goal)` holds at the end of those lines. The second statement starts with `(goal)`, consists of the lines from $n+1$ to end and has to show that the original claim holds. This is also presented in the reference manual. After asking the developers of EasyCrypt they told me that the command `seq n : (prob)(goal)` is applicable to probability statements too, where `(prob)` consists of the probabilities for each new sub-statement that is generated (in this case more than 2). This shows that we can not be sure that every limitation stated above is a result of EasyCrypt's engine, or a flaw in the currently existing reference manual. It is possible that in the future most of the problems are going to be fixed.

Errors and bugs

During the process of implementation, sometimes an smt-error occured. The command `smt()` uses the automatic prover Alt-Ergo or Z3 for the current goal that we have to prove. Without additional parameters it automatically stops after a few seconds if it was not able to solve in this time frame. Some steps in the proof have an error using the smt-command. In the following we talk about this problem and present a solution. Furthermore consider this can also be a bug correlated to the virtual machine, its operating system or the automatic provers themselves, and not EasyCrypt itself. This error consists of breaking the limited time frame and, if run long enough, crashing the interactive mode. If the command was interrupted, it would shortly display an "anomaly"-error. After that the goal was proven. This error only happened at instances where there were a lot of used variables and the statement was true. Probably it only occurs if the number of variables goes beyond the capacity of the automatic provers. As an indicator for this theory of cause, consider removing variables in a statement. This either leads to a wrong claim or a smaller number of variables. At some points, where the statement still was true, the command `smt()` worked correctly and no error occured. To overcome this bug at the other instances, the statements are proven directly with EasyCrypt-commands only and not calling an automatic prover.

7. Conclusion

We have shown that the protocol IPsec, using IKEv2, is secure against a real-or-random attack. To prove this, we took the proof presented by Heussen et al. (2017) and implemented it in EasyCrypt. Since this lead to several problems, we had to remodel it. This resulted in a new proof, which we have shown to be true separately. The security against a real-or-random attack depends on two properties of the used pseudorandom functions. One, they all have to be a pseudorandom function if the input is swapped. Second, the multiple-primitive game `MP-PRF*-OODH` has to have negligible advantage. We also assumed that every party uses a certificate. This is only done to simplify the proof and its implementation. Since we do not use this property at any point, the proof should hold without the usage of certificates. We have to leave this to future work.

Even though we have to rely on several restrictions for the proof verification in EasyCrypt, most are dispensable from a theoretical standpoint. There are two challenges we did not address yet. First, EasyCrypt is still in development, therefore error-prone. This downside only decreases over time until enough proofs are checked thoroughly. Second, we have to show that the implementation of the proof completely corresponds to the given proof. Unless this equality holds, we do not gain any advantage. This is the same problem as verifying a proof, though not as complex. Those two problems are not solved yet.

To fully prove that IPsec is secure in the `AKE` model, we need to show that is is secure against an authentication break. Ideally this is also checked by some toolset or automatic prover in some future work. We also address the usage of EasyCrypt. As a promising toolset for game-based cryptographic proofs, it does not provide enough usability for general use. Although it fits nicely to the format of game sequences, it still lacks in several parts. The main issue is the difficulty of joining the layer of the game and the adversary. This can definitely be improved on in the future. Ideally speaking, a game based toolset should only need to construct one single game, where the others are derived from, instead of defining each game separately. This especially leads to unnecessary complications in larger protocols and schemes like IPsec.

Bibliography

Microsoft, Inria, and the Joint Centre. miTLS: A Verified Reference Implementation of TLS, 2014a. URL `https://mitls.org/`.

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the TLS Handshake Secure (as it is), 2014. URL `https://eprint.iacr.org/2014/182`.

IMDEA Software Institute, Inria, and École Polytechnique. EasyCrypt: Computer-Aided Cryptographic Proofs, 2014. URL `https://www.easycrypt.info/trac/`.

Michael Heussen, Daniel Loebenberger, and Michael Nüsken. IPSEC SECURITY, 2017.

Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the Security of TLS-DHE in the Standard Model, 2013. URL `https://eprint.iacr.org/2011/219`.

Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the Security of TLS-DH and TLS-RSA in the Standard Model, 2013. URL `https://eprint.iacr.org/2013/367`.

Yong Li, Sven Schäge, Zheng Yang, Florian Kohlar, and Jörg Schwenk. On the Security of the Pre-Shared Key Ciphersuites of TLS, 2015. URL `https://eprint.iacr.org/2014/037`.

Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the Security of the TLS Protocol: A Systematic Analysis, 2014. URL `https://eprint.iacr.org/2013/339`.

Jonathan Katz and Yehuda Lindell. Introduction to Modern Cryptography. Chapman & Hall/CRC Cryptography and Network Security Series. AAA, 2016. ISBN 9781466570269.

Microsoft, Inria, and the Joint Centre. mitls-flex: file KEM.ec, 2014b. URL `https://github.com/mitls/mitls-flex/tree/master/src/tls/interactive/easycrypt`.

Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, Instantiations, and Impossibility Results, 2017. URL `https://eprint.iacr.org/2017/517`.

A. Code of the states

A.1. First initiator phase

```
proc step1 (pi : protocol_instance, piid : protocol_instance_ID) :
    message_transfer * protocol_instance = {
  var m',sSPIi,sSPIr,a,kKEi,nNi,hHdr;

  pi <- protocol_set_role(pi)(initiator);
  sSPIi <$ distr_security_parameter;
  sSPIr <- 0;
  pi <- protocol_set_SPIs(pi)(sSPIi,sSPIr);
  a <$ FDistr.dt;
  kKEi <- g ^ a;

  if((piid = pick_barstar \/ piid = pick_hatstar) /\ star_initiator =
      false) {
    a <- a_gamma;
    star_initiator <- true;
    kKEi <- aA_gamma;
    pi <- protocol_set_gamma_empty(pi)(true);
  }

  pi <- protocol_set_key_exchange_secret_a(pi)(a);
  nNi <$ distr_nonce; pi <- protocol_set_noncei(pi)(nNi);
  hHdr <- (sSPIi,sSPIr,type_m1);
  m' <- (m_not_enc_of (hHdr,kKEi,nNi));
  pi <- protocol_set_m1(pi)( (hHdr,kKEi,nNi) );
  pi <- protocol_set_internal_state(pi)(expect_m2);
  return ((m',type_m1),pi);
}
```

A.2. First responder phase

```
proc step235 (pi : protocol_instance, piid : protocol_instance_ID, m1
    : message_unencrypted) : message_transfer * protocol_instance = {
  var m',hHdr,kKEi,nNi,nNr,sSPIi,sSPIr,b,kKEr,dDH;
  var sSKEYSEED,sSK;
```

```
4
5       pi <- protocol_set_m1(pi)(m1);
6       (hHdr,kKEi,nNi) <- m1;
7       pi <- protocol_set_role(pi)(responder);
8       sSPIi <- hHdr.`1;
9       sSPIr <$ distr_security_parameter;
10      pi <- protocol_set_SPIs(pi)(sSPIi,sSPIr);
11      pi <- protocol_set_noncei(pi)(nNi);
12      b <$ FDistr.dt;
13      if((piid = pick_barstar \/ piid = pick_hatstar) /\ star_responder =
            false) b <- b_gamma;
14      pi <- protocol_set_key_exchange_secret_b(pi)(b);
15      kKEr <- g ^ b;
16      dDH <- kKEi ^ b;
17      pi <- protocol_set_key_exchange(pi)(dDH);
18
19      if((piid = pick_barstar \/ piid = pick_hatstar) /\ star_responder =
            false) {
20        star_responder <- true;
21        kKEr <- bB_gamma;
22        pi <- protocol_set_gamma_empty(pi)(true);
23      }
24
25      nNr <$ distr_nonce; pi <- protocol_set_noncer(pi)(nNr);
26      hHdr <- (sSPIi,sSPIr,type_m2);
27      m' <- (m_not_enc_of (hHdr,kKEr,nNr));
28
29      sSKEYSEED <- F_prf_group(pick_prf)((nNi,nNr),dDH);
30      pi <- protocol_set_keyseed_mod(pi)(false);
31
32      if((piid = pick_barstar \/ piid = pick_hatstar)) {
33        if(gs_a2 = true /\ protocol_get_gamma_empty(pi) = true) {
34          if(kKEi = aA_gamma) {
35            sSKEYSEED <- F_trf(pick_trf_prf)(nNi,nNr);
36            if(hop_A1_A2 = false) pi <- protocol_set_keyseed_mod(pi)(true);
37          }
38          else {
39            sSKEYSEED <- F_prf_group(pick_prf)((nNi,nNr),kKEi ^ b_gamma);
40          }
41        }
42        if(gs_a2_oracle = true /\ protocol_get_gamma_empty(pi) = true) {
43          if(kKEi = aA_gamma) sSKEYSEED <- Oracle_A1_A2.prf_once((nNi,nNr))
              ;
44          else sSKEYSEED <- Oracle_A1_A2.rodh(pick_prf,kKEi,(nNi,nNr));
45        }
```

```
46      }
47      pi <- protocol_set_keyseed(pi)(sSKEYSEED);
48
49      sSK <- prfplus(sSKEYSEED,(nNi,nNr,sSPIi,sSPIr));
50
51      if(protocol_get_keyseed_mod(pi) = true) {
52        if(gs_a3 = true) sSK <- trf_prfplus(nNi,nNr,sSPIi,sSPIr);
53        if(gs_a3_oracle = true) sSK <- Oracle_A2_A3.prf(pick_prf,(nNi,nNr,
              sSPIi,sSPIr));
54      }
55
56      pi <- protocol_set_SK(pi)(sSK);
57      pi <- protocol_set_m2(pi)((hHdr,kKEr,nNr));
58      pi <- protocol_set_internal_state(pi)(expect_m3);
59      return ((m',type_m2),pi);
60    }
```

A.3. Second initiator phase

```
1   proc step456 (pi : protocol_instance, piid : protocol_instance_ID, m2
        : message_unencrypted) : message_transfer * protocol_instance = {
2     var m',dDH,a,hHdr,kKEr,nNr,nNi,sSPIi,sSPIr;
3     var sSKEYSEED,sSK,sSK_d,sSK_ai,sSK_ar,sSK_ei,sSK_er,sSK_pi,sSK_pr;
4     var mMi,iIDi,aAUTHi,cCERTi,t3_enc,t3,m1,skIDi;
5     var keys;
6
7     pi <- protocol_set_m2(pi)( m2 );
8     (hHdr,kKEr,nNr) <- m2;
9     pi <- protocol_set_noncer(pi)(nNr);
10    sSPIi <- protocol_get_SPIi(pi);
11    sSPIr <- hHdr.`2;
12    a <- protocol_get_key_exchange_secret_a(pi);
13    nNi <- protocol_get_noncei(pi);
14
15    dDH <- kKEr ^ a;
16    pi <- protocol_set_key_exchange(pi)(dDH);
17    sSKEYSEED <- F_prf_group(pick_prf)((nNi,nNr),dDH);
18    pi <- protocol_set_keyseed_mod(pi)(false);
19
20    if(piid = pick_barstar \/ piid = pick_hatstar) {
21      if(gs_a2 = true /\ protocol_get_gamma_empty(pi) = true) {
22        if(kKEr = bB_gamma) {
23          sSKEYSEED <- F_trf(pick_trf_prf)(nNi,nNr);
24          if(hop_A1_A2 = false) pi <- protocol_set_keyseed_mod(pi)(true);
```

```
25          }
26          else sSKEYSEED <- F_prf_group(pick_prf)((nNi,nNr),kKEr ^ a_gamma)
                ;
27        }
28        if(gs_a2_oracle = true /\ protocol_get_gamma_empty(pi) = true) {
29          if(kKEr = bB_gamma) sSKEYSEED <- Oracle_A1_A2.prf_once((nNi,nNr))
                ;
30          else sSKEYSEED <- Oracle_A1_A2.lodh(pick_prf,kKEr,(nNi,nNr));
31        }
32      }
33      pi <- protocol_set_keyseed(pi)(sSKEYSEED);
34
35      sSK <- prfplus(sSKEYSEED,(nNi,nNr,sSPIi,sSPIr));
36
37      if(protocol_get_keyseed_mod(pi) = true) {
38        if(gs_a3 = true) sSK <- trf_prfplus(nNi,nNr,sSPIi,sSPIr);
39        if(gs_a3_oracle = true) sSK <- Oracle_A2_A3.prf(pick_prf,(nNi,nNr,
              sSPIi,sSPIr));
40      }
41
42      pi <- protocol_set_SK(pi)(sSK);
43
44      (sSK_d,sSK_ai,sSK_ar,sSK_ei,sSK_er,sSK_pi,sSK_pr) <- sSK;
45      iIDi <- protocol_get_party_ID(pi);
46      cCERTi <- protocol_get_certificate(pi);
47      keys <- protocol_get_keypair(pi);
48      skIDi <- keys.`2;
49      m1 <- protocol_get_m1(pi);
50      mMi <- (m1,nNr, F_prf_keyseed_pid(pick_prf)(sSK_pi,iIDi) );
51      aAUTHi <- sign(skIDi,mMi);
52      t3 <- (iIDi,cCERTi,aAUTHi);
53      t3_enc <- AE_enc((sSK_ei,sSK_ai),hHdr,t3);
54      m' <- m_enc_of (hHdr,t3_enc);
55
56      pi <- protocol_set_internal_state(pi)(expect_m4);
57      pi <- protocol_set_m3(pi)((hHdr,t3_enc));
58      return ((m',type_m3),pi);
59    }
```

A.4. Second responder phase

```
1   proc step78 (pi : protocol_instance, piid : protocol_instance_ID, m3 :
        message_encrypted) : message_transfer * protocol_instance = {
2     var m',hHdr,nNr,nNi,sSPIi,sSPIr;
```

```
3     var sSK,sSK_d,sSK_ai,sSK_ar,sSK_ei,sSK_er,sSK_pi,sSK_pr;
4     var t3,t3_enc,mMi,mMr,iIDi,iIDr,aAUTHi,aAUTHr,cCERTi,cCERTr,m1,m2,
          skIDr,t4,t4_enc;
5     var keys;
6     var bool_vfy,bool_valid : bool;
7     var kKEYMAT,output78;
8
9     pi <- protocol_set_m3(pi)(m3);
10    kKEYMAT <- default_session_key;
11    (hHdr,t3_enc) <- m3;
12    iIDr <- protocol_get_party_ID(pi);
13    m1 <- protocol_get_m1(pi);
14    m2 <- protocol_get_m2(pi);
15    keys <- protocol_get_keypair(pi);
16    skIDr <- keys.‘2;
17    cCERTr <- protocol_get_certificate(pi);
18
19    sSPIi <- protocol_get_SPIi(pi);
20    sSPIr <- protocol_get_SPIr(pi);
21    nNi <- protocol_get_noncei(pi);
22    nNr <- protocol_get_noncer(pi);
23    sSK <- protocol_get_SK(pi);
24    (sSK_d,sSK_ai,sSK_ar,sSK_ei,sSK_er,sSK_pi,sSK_pr) <- sSK;
25
26    t3 <- AE_dec((sSK_ei,sSK_ai),hHdr,t3_enc);
27    (iIDi,cCERTi,aAUTHi) <- t3;
28    mMi <- (m1,nNr, F_prf_keyseed_pid(pick_prf)(sSK_pi,iIDi) );
29    bool_valid <- valid(cCERTi,iIDi,protocol_get_public_key(pi));
30    bool_vfy <- vfy(aAUTHi,mMi);
31    if( bool_valid = false \/ bool_vfy = false) {
32      pi <- protocol_set_accepted(pi)(ABORT);
33      output78 <- ((m_enc_of m3,type_special),pi);
34    }
35    else {
36      kKEYMAT <- prfplus2( sSK_d ,(nNi,nNr));
37
38      if(protocol_get_keyseed_mod(pi) = true) {
39        if(gs_a4 = true) kKEYMAT <- trf_prfplus2(nNi,nNr);
40        if(gs_a4_oracle = true) kKEYMAT <- Oracle_A3_A4.prf(pick_prf, (
              nNi,nNr));
41      }
42
43      pi <- protocol_set_accepted(pi)(YES);
44      pi <- protocol_set_partner_party_ID(pi)(iIDi);
45      pi <- protocol_set_session_key(pi)( kKEYMAT );
```

```
46
47        mMr <- (m2,nNi,F_prf_keyseed_pid(pick_prf)(sSK_pr,iIDr));
48        aAUTHr <- sign(skIDr,mMr);
49        t4 <- (iIDr,cCERTr,aAUTHr);
50        t4_enc <- AE_enc((sSK_er,sSK_ar),hHdr,t4);
51        m' <- (m_enc_of (hHdr,t4_enc));
52        output78 <- ((m',type_m4),pi);
53        pi <- protocol_set_m4(pi)(hHdr,t4_enc);
54      }
55      pi <- protocol_set_internal_state(pi)(finished);
56
57      return output78;
58    }
```

A.5. Third initiator phase

```
1   proc step910 (pi : protocol_instance, piid : protocol_instance_ID, m4
        : message_encrypted) : protocol_instance = {
2     var hHdr,nNr,nNi,sSPIi,sSPIr;
3     var sSK,sSK_d,sSK_ai,sSK_ar,sSK_ei,sSK_er,sSK_pi,sSK_pr;
4     var mMr,iIDr,aAUTHr,cCERTr,m1,m2,t4,t4_enc;
5     var bool_vfy,bool_valid : bool;
6     var kKEYMAT;
7
8     pi <- protocol_set_m4(pi)(m4);
9     kKEYMAT <- default_session_key;
10    (hHdr,t4_enc) <- m4;
11    m1 <- protocol_get_m1(pi);
12    m2 <- protocol_get_m2(pi);
13
14    sSPIi <- hHdr.`1;
15    sSPIr <- hHdr.`2;
16    nNi <- protocol_get_noncei(pi);
17    nNr <- protocol_get_noncer(pi);
18    sSK <- protocol_get_SK(pi);
19    (sSK_d,sSK_ai,sSK_ar,sSK_ei,sSK_er,sSK_pi,sSK_pr) <- sSK;
20
21    t4 <- AE_dec((sSK_er,sSK_ar),hHdr,t4_enc);
22    (iIDr,cCERTr,aAUTHr) <- t4;
23    mMr <- (m2,nNi, F_prf_keyseed_pid(pick_prf)(sSK_pr,iIDr) );
24    bool_valid <- valid(cCERTr,iIDr,protocol_get_public_key(pi));
25    bool_vfy <- vfy(aAUTHr,mMr);
26    if( bool_valid = false \/ bool_vfy = false) pi <-
          protocol_set_accepted(pi)(ABORT);
```

```
27      else {
28        kKEYMAT <- prfplus2( sSK_d ,(nNi,nNr));
29        if(protocol_get_keyseed_mod(pi) = true) {
30          if(gs_a4 = true) kKEYMAT <- trf_prfplus2(nNi,nNr);
31          if(gs_a4_oracle = true) kKEYMAT <- Oracle_A3_A4.prf(pick_prf, (
                nNi,nNr));
32        }
33        pi <- protocol_set_accepted(pi)(YES);
34        pi <- protocol_set_partner_party_ID(pi)(iIDr);
35        pi <- protocol_set_session_key(pi)( kKEYMAT );
36      }
37      pi <- protocol_set_internal_state(pi)(finished);
38
39      return pi;
40    }
```