# FINAL PROJECT

*Mini Mariya Thomas - 7057695*
*Areeba Tabassum Shoaib - 7069662*

# Project 1: IKEv2

a) Added the initialization of key pairs (sk,pk) and identities (A,B) in the rule `rule initialize_Key_identity.`

In the rule, We have mentioned identities as $ID which can take any identity like A and B. So both A and B will be able to play the role of the initiator and the responder as asked in the question.

b) Modelled the protocol as shown in the figure of project description in below rules:
Added comments in the *ikev2.spthy* file to describe the model.

```
rule step1_Send_INIT_A
rule step2_Send_INIT_B_Cookie
rule step3_Send_INIT_A_cookie
rule step4_Send_INIT_B_Final
rule step5_Send_AUTH_A
rule step6_Send_AUTH_B
rule step7_ReceiveAUTHB_A
```

We have run the model using the command *--derivcheck-timeout=0* because of the well formedness error for derivation check timeout
Also we have added restriction `NoSelfKeyExchange` to ensure that neither party is able to perform a key exchange with themselves. This restriction ensures that no party can perform a key exchange with itself. The condition *KeyExchangeAB(X, X) @i ==> F* means that if the same entity (*X*) appears on both sides of the exchange, it is not allowed. This prevents invalid sessions where a party tries to establish a connection with itself.

c) Added restriction `restriction OneSessionPerKeyPair` to only allow one single session per key pair.
This restriction ensures that each key pair is used for only one session. It enforces that when an entity `A` uses a public key `pkA` and an entity `B` uses a public key `pkB`, both must engage in the session at the same time. By requiring the timestamps of these key usages to be the same (`#i = #j`), the restriction prevents multiple sessions from being established with the

same key pair. This guarantees that no entity can initiate or participate in more than one session using the same key pair.

d) Added a lemma for the reachability to make sure that both parties are able to reach the end of an execution. Lemma is added as exists-trace by adding action facts of first and last rules representing first and last steps of the protocol.
   This lemma checks if both parties can complete the protocol successfully. It ensures that the key pairs are initialized, messages are exchanged, and authentication is completed by both `A` and `B`. The time order conditions (`#i < #j < #k < #l < #m`) confirm that the steps happen in the correct sequence. This acts as a basic check to verify that the protocol runs as expected from start to finish.
   This lemma is getting verified with 32 steps.

e) Modelled six different agreement lemmas. Added two reveal rules for secret key sk and diffie-helman secret key (x & y). So in all agreement lemmas, we have added reveal action facts to not reveal keys to the network.

**`lemma SendAuthMessageFromA`**

This lemma is to prove that if a message with Transcript and Auth is received by B at a time i, then the same message with Transcript and Auth should be sent by A before.
But this lemma is not getting verified when we prove.
Counter example:

- Using a random x attacker can calculate `k=(gxB^x)` (The x chosen by attacker and A might be same)
- Using the above k, the attacker can compute k1.
- Using k1, Attacker can encrypt `msg5=senc(<'AUTH',pkA,signA>,k1)` and send it to B.
- B receives the msg5 sent by the attacker.

**`lemma SendAuthMessageFromB`**

This lemma is to prove that if a message with Transcript and Auth is received by A at a time i, then the same message with Transcript and Auth should be sent by B before.

**`lemma signatureOfA_Agreement`**

This lemma is to prove that if B receives signA in the message from A at a time i, then A should compute signA before.
But this lemma is not getting verified when we prove.
Counter example:

- Attacker generate or learn a secret key of A (somehow)
- Some values for the TranscriptA can be learned by the attacker from the network when A sends values to the network. By hashing the values attacker can create TranscriptA

- Attacker compute `signA = sign(transcriptA,skA)` and send to B in the msg5
- When B receives msg5 in the rule `rule step6_Send_AUTH_B,` they also receive signA but it is actually sent by A.

**`lemma signatureOfB_Agreement`**

This lemma is to prove that if A receives signB in the message from B at a time i, then B should compute signB before.

**`lemma transcriptAgreementAandB`**

This lemma is to prove that if B receives TranscriptA in the message from A at a time i, then A should compute TranscriptA before.
But this lemma is not getting verified when we prove.
Counter example:

- Attacker generate or learn a secret key of A (somehow)
- Some values for the TranscriptA can be learned by the attacker from the network when A sends values to the network. By hashing the values attacker can create TranscriptA and send to B in the msg5
- When B receives msg5 in the rule `rule step6_Send_AUTH_B,` they also receive TranscriptA but it is actually sent by A.

**`lemma transcriptAgreementBandA`**

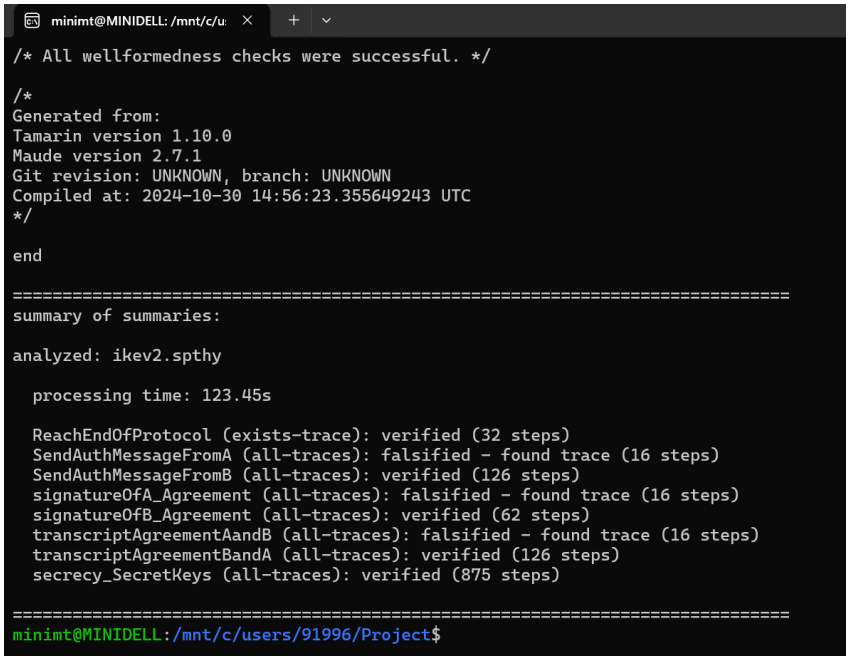
This lemma is to prove that if A receives TranscriptB in the message from B at a time i, then B should compute TranscriptB before.

f) Added **`lemma secrecy_SecretKeys`** to prove key secrecy for every key (x,y,k,k1,k2,k3,k4). In the lemma we are checking the secrecy of diffie-helmen secret keys x and y, shared secret key k and other keys k1,k2,k3,k4 used in IKEV2 protocol. A and B represent two entities participating in the protocol. Declares secret keys at different time points. Lemma implies that no adversary can learn any of the specified keys at a different time point. The lemma applies to all possible instances of secret keys and time indices.

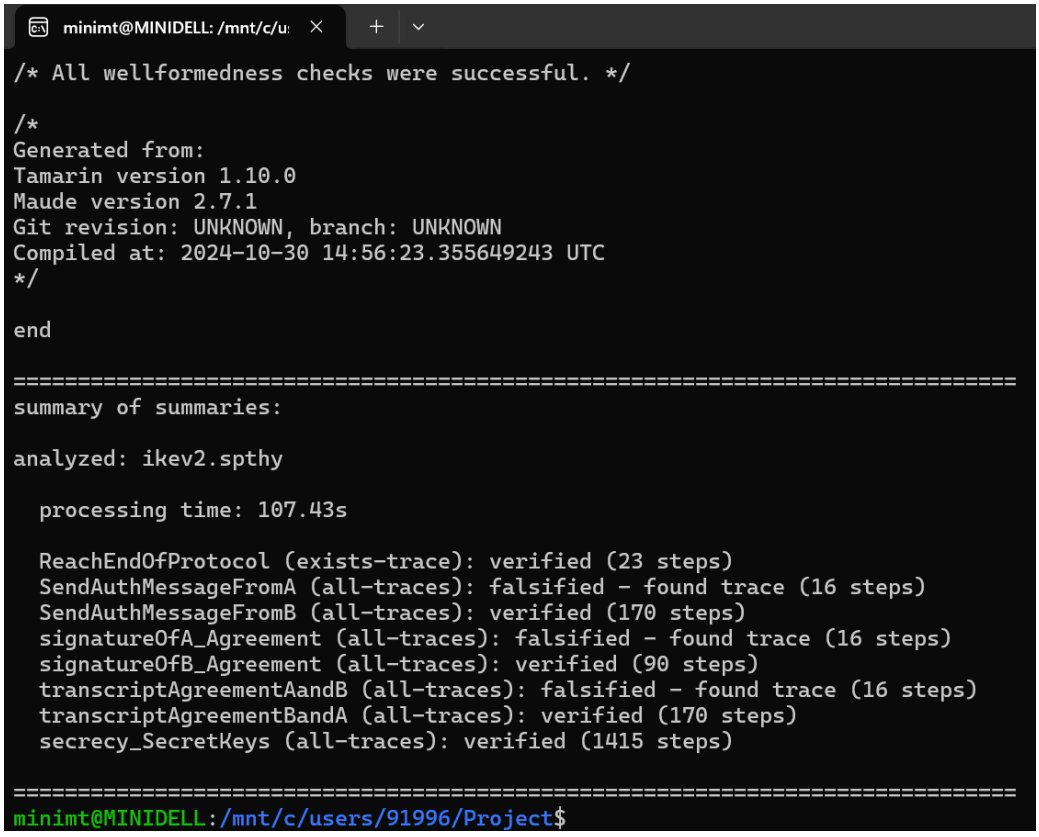
This lemma is getting verified with 875 steps.

g) We have removed the **`restriction OneSessionPerKeyPair`** and run the model.

Screenshot of summary with the **`restriction OneSessionPerKeyPair:`**




Screenshot of summary without the **`restriction OneSessionPerKeyPair:`**



Analysis status is the same for both cases. But steps of the lemma for below lemmas got changed for multiple sessions.

```
lemma ReachEndOfProtocol
lemma SendAuthMessageFromB
lemma signatureOfB_Agreement
lemma transcriptAgreementBandA
lemma secrecy_SecretKeys
```

# Project 3: SSH

- *The key exchange and service request phase*

Key exchange and service requests are handled in -

rule_key_exchange_client_1
rule_key_exchange_server
rule_key_exchange_client_2
rule_encryption_integrity_key_client
rule_encryption_integrity_key_server
rule_service_request_client
rule_service_reject_server
rule_service_accept_server
rule_client_send_data
rule_server_accept_data

- *Keyre-exchanges after an initial key was been established. Note, that these key exchanges use the established key for encryption of the messages.*

Keyre-exchanges implemented in -
rule_key_reexchange_server_1
rule_key_reexchange_client_1
rule_key_reexchange_server_2
rule_key_reexchange_client_2
rule_encryption_integrity_key_client_reexchange
rule_encryption_integrity_key_server_reexchange

- *Different algorithm choices for the encryption, the MAC, and the hash function that are negotiated during the key exchanges.*

Algorithm negotiation by client and server is implemented in-

rule_Client_Algorithm_proposals
rule_Server_Algorithm_Selection
rule_key_exchange_client_1 *(Client accepting server selected algorithms and initiate key exchange)*

- *Packet sequence numbers that are used for the data integrity layer starting with the initial key exchange.*

Packet sequence numbers and its incrementing mechanism implemented in-

`rule_client_send_data`
`rule_server_accept_data`

We tried to implement the sequence numbers below as described in the tamarin manual. But this results in an error *"`zero` is a reserved function name for builtins."*

```
functions:XOR/2
functions:zero/0

equations:x_XOR_zero_=_x
equations:x_XOR_x_=_zero
```

And set `sequence_number_=_zero` in the `rule_client_send_data`
So unfortunately we couldn't initialize the sequence number as zero. Instead initialized to 1 and incremented the value by 1 using the built-in ***natural-numbers***.

- *Additional adversarial capabilities like, for instance, key reveals.*

Different key reveal rules are implemented in -

```
rule reveal_dh_key
rule Reveal_shared_secretKey
rule reveal_encryption_key
rule reveal_integrity_key
```

- *Appropriate security properties against the different adversarial capabilities you identified*

`lemma secrecy` - The shared secret k remains confidential and is not leaked during the protocol execution. Hence, the lemma is verified.
`lemma algorithmSelection` - The protocol successfully ensures that both parties agree on the negotiated algorithm without interference or mismatch. Hence, the lemma is verified.
`lemma dataSendAgreement` - Agreement on sending data only if the server accepts the service request is not being proven. We have tried all possible ways but were unable to find the reason for its failure. If we had more time and enough resources, we might have been able to prove it. Hence, the lemma is getting killed.

`lemma keyReExchangeAgreement` – Agreement on initiating key re-exchange and then computing keys does not hold. The expected agreement is not achieved. Hence, the lemma is falsified.

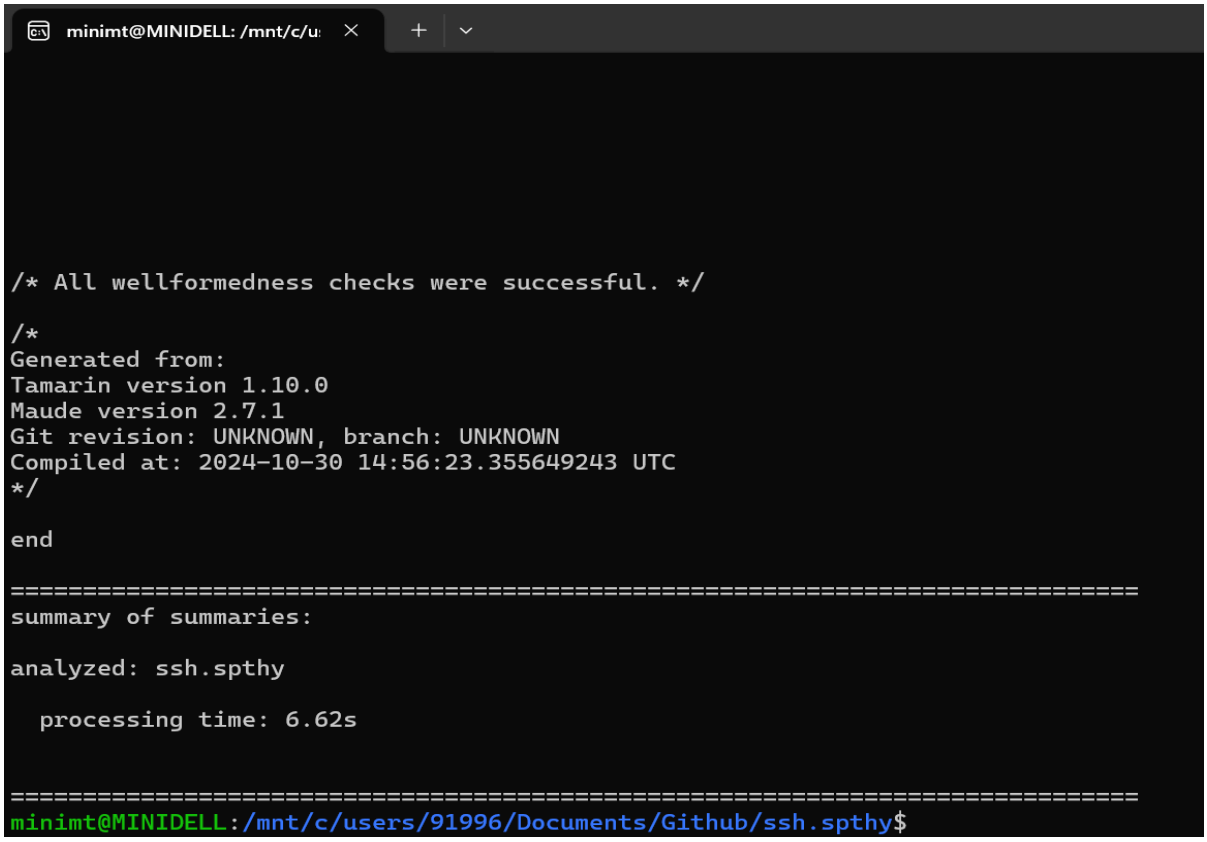
- *Accepting service request messages during re-key exchanges.*

Service request and accepting the service request messages are implemented in -
`rule service_request_reexchange_client`
`rule service_accept_reexchange_server`

- *More than one host key per identity. Instead, assume that there is only a single one.*

Host key generation of server is added in the `rule generate_server_host_keys` and `rule generate_client_host_keys`

We tried to run the model for checking well formedness errors without lemmas and below is the result.



# Project 2: SPDM (Security Protocol And Data Model)

(a) Write the protocol in Alice & Bob notation and model it in Tamarin.

## Initialize Identity:

A certificate is basically a unique key pair (public and private keys) that's signed by a trusted authority. This trusted authority is known as a Certificate Authority (CA).

To implement this, we defined some basic rules that govern how certificates are generated and issued.

### Rule: Certificate_Authority

This rule initializes the CA by generating a public-private key pair.

### Rule: IssuingCertificates

Now that we have a CA, we need a way to issue certificates to devices.

## Initialize Roles

In SPDM, communication always begins with an endpoint taking on the role of an Initiator, while the receiving endpoint acts as a Responder. A connection is established between these two roles. However, an endpoint is not restricted to just one role—it can act as both an Initiator and a Responder in different communications.

To define roles within our model, we introduced a rule that allows devices to be assigned an identity and function as either an Initiator or a Responder based on the communication context.

### Rule: CreatingRole

This rule ensures that each SPDM endpoint is assigned an identity and can take on the role of either an Initiator or a Responder, making secure connections possible.

## Version-Capabilities-Algorithms (VCA)

In our simplified SPDM model, each endpoint has a predefined set of capabilities, such as protocol version, which are shared in a trusted manner.

The communication process begins with both the Initiator and Responder exchanging versions and certificates securely. To model this, we assume that these exchanges occur over a trusted channel where attackers cannot interfere.

To implement this, we created a set of rules that define how the VCA process unfolds, from the initial request to response processing.

### Rule: VCA_Request_From_Alice

This rule allows an Initiator to send a VCA request to a Responder, including the protocol version and certificate.

### Rule: VCA_Response_From_Bob

This rule enables the Responder to process the request and respond with its own certificate, forming a complete VCA exchange.

### Rule: VCA_Response_Processed_By_Alice

This rule ensures that after receiving a response, the Initiator stores the Responder's certificate.

## Measurements

Measurements are device-specific values representing internal states, such as software versions. Once the VCA process is completed, an Initiator can send a GET_MEASUREMENTS request multiple times to a Responder. The request contains the protocol version and request code. The Responder then replies with a MEASUREMENTS message containing the local measurements.

These measurements are sent in the clear, meaning no encryption is applied during transmission.

To implement this, we created a set of rules defining how measurements are requested, responded to, and processed.

### Rule: Get_Measurements_Request_by_Alice

This rule allows the Initiator to request measurements from the Responder by sending a message containing the request code and Software version.

### Rule: Send_Measurements_Response_by_Bob

This rule ensures the Responder processes the request and sends back the corresponding measurements.

### Rule: Process_Measurements_At_Alice

This rule ensures the Initiator properly processes the received measurements and stores them for verification.

# Key Exchange

The **Key Exchange** process is an essential part of establishing secure communication between two parties in SPDM (Security Protocol and Data Model). The goal is to enable both parties (the Initiator and the Responder) to establish a shared session key that will be used to secure further communications.

#### Key Exchange Steps

#### Rule: Alice_KE_Request

1. **Alice** generates her Diffie-Hellman public key (`ga`) as `'g'` raised to the power of a randomly chosen private value (`~m`).
2. Alice prepares a message (`Message_Alice`) containing:
    - The type `KEY_EXCHANGE`
    - The Diffie-Hellman public key (`ga`)
    - A nonce (`~Nonce_A`)
3. Alice enters the state `Alice_State_2` while waiting for a response from Bob.
4. The rule tracks the generation of Alice's Diffie-Hellman public key (`ga`), and the preparation of the message.

### Rule: Bob_Key_Exchange_Response

1. **Bob** receives the `KEY_EXCHANGE` message from Alice, which contains:
    - The Diffie-Hellman public key (`ga`)
    - Alice's nonce (`~Nonce_A`)
2. Bob computes the Diffie-Hellman output (`DH_Output`)
3. Bob generates his own Diffie-Hellman public key (`gb`) as `'g'` raised to the power of his private value (`~n`).
4. Bob constructs a transcript of the key exchange, including:
    - The `VCA` message
    - The hash of Bob's certificate (`h(Cert_B)`)
    - The `KEY_EXCHANGE` message from Alice

    - The `KEY_EXCHANGE_RSP` message (excluding HMAC and signature fields)
5. Bob signs the transcript with his private key (`~y`), producing `Signature_Bob`.
6. Bob computes two HMACs over the transcript:
    - `HMAC_Key_Init1` derived using the Diffie-Hellman output and a string `finished init`
    - `HMAC_Key_Resp1` derived using the Diffie-Hellman output and a string `finished resp`
7. Bob generates the final HMAC response (`HMAC_Resp1`) over the transcript.
8. Bob prepares the `KEY_EXCHANGE_RSP` message, which includes:
    - The type `KEY_EXCHANGE_RSP`
    - The Diffie-Hellman public key (`gb`)
    - Bob's nonce (`~Nonce_B`)
    - The signature (`Signature_Bob`)
    - The HMAC (`HMAC_Resp1`)
9. Bob enters the state `Bob_State_2` after sending the response.

## Rule: Alice_Verify_Bob_And_Send_Finish

This rule describes the process where Alice verifies Bob's response to her key exchange request, calculates HMAC values, and sends the final `FINISH` message to complete the key exchange protocol.
**Alice Receives Bob's Response:** Alice first receives a message from Bob (`Message_Bob`), which contains:

- `KEY_EXCHANGE_RSP`: This indicates that the message is Bob's response to Alice's `KEY_EXCHANGE` request.
- `gb`: Bob's Diffie-Hellman public key.
- `~Nonce_B`: Bob's nonce, a random value used to prevent replay attacks.
- `Signature_Bob`: A digital signature from Bob, which proves that the message is indeed from Bob.
- `HMAC_Resp1`: The HMAC (Hashed Message Authentication Code) that Bob has calculated for the message to ensure its integrity and authenticity.

**Computing Diffie-Hellman Output:** Alice computes the Diffie-Hellman output (`DH_Output`) using the formula:

- `DH_Output = gb ^ ~m` Where `gb` is Bob's public key and `~m` is Alice's private value. This `DH_Output` is the shared secret that both Alice and Bob will use to generate keys for further operations.

**Constructing the Key Exchange Transcript (Transcript 2):** Alice then constructs **Key Exchange Transcript 2** (`Key_Exchange_Transcript2`). This transcript includes:

- `VCA`: The Version, Capabilities, and Algorithms (VCA) message exchanged earlier in the process. This message includes details about the protocol version and certificates.
- `h(Cert_B)`: The hash of Bob's certificate, which ensures that Alice has received a valid certificate.
- `KEY_EXCHANGE`: Alice's own `KEY_EXCHANGE` message that she sent earlier.
- `~Nonce_A`: Alice's nonce, which she generated earlier.
- `gb`: Bob's public key.
- `~Nonce_B`: Bob's nonce, included in the response.
- The `FINISH` message type, indicating that the key exchange is complete.

**Calculating HMACs:** Alice now calculates two HMAC values over the **Key Exchange Transcript 2**:

- **HMAC_Key_Resp2**: This is the key used to compute the HMAC for Alice's response. It is derived using the shared `DH_Output`, the transcript, and the string `'finished resp'`.
    - The derivation is: `HMAC_Key_Resp2 = hkdf(DH_Output, Key_Exchange_Transcript2, 'finished resp')`
- **HMAC_Key_Init2**: This is another key derived from the same shared `DH_Output`, but with the string `'finished init'` to differentiate it from `HMAC_Key_Resp2`.
    - The derivation is: `HMAC_Key_Init2 = hkdf(DH_Output, Key_Exchange_Transcript2, 'finished init')`
- **HMAC_Init1**: After deriving `HMAC_Key_Init2`, Alice computes the HMAC (`HMAC_Init1`) using this key and the transcript. This HMAC is used to authenticate the data in the transcript, ensuring that it hasn't been altered.
    - The derivation is: `HMAC_Init1 = hmac(Key_Exchange_Transcript2, HMAC_Key_Init2)`

**Verifying Bob's Signature:** Alice verifies Bob's signature (`Signature_Bob`) to ensure that the message she received was truly sent by Bob and that it hasn't been tampered with. Alice verifies the signature by checking if:

- The signature is valid when verifying with Bob's public key (`pkB`).
- Alice checks if the signature corresponds to the hash of the Key Exchange Transcript 1, which was sent earlier by Alice.

Additionally, Alice checks that the HMAC (`HMAC_Resp1`) that she received from Bob matches the **computed HMAC (`Computed_HMAC_Resp1`)** that Alice calculates using the transcript and the derived `HMAC_Key_Resp1` (from the previous step).

**Generating the FINISH Message:** After verifying Bob's signature and checking the integrity of the response message, Alice prepares the `FINISH` message. This message includes:

- The `FINISH` type, signaling the end of the key exchange.
- The HMAC (`HMAC_Init1`), which ensures the integrity of the key exchange process and prevents any tampering.

**Sending the FINISH Message:** Alice sends the `FINISH` message to Bob, and at the same time, transitions to `Alice_State_4`, which signifies that Alice has completed the key exchange process.

## Rule: Bob_Verifies_Finish_And_Prepares_Session_Key

This rule describes the process where Bob verifies Alice's `FINISH` message, checks the integrity of the communication using HMAC, and then derives the final session key for secure communication.

1. **Bob Receives Alice's FINISH Message:** Bob receives a message from Alice (`< 'FINISH', HMAC_Init1 >`). This message contains:
    - `FINISH`: A signal from Alice indicating that she has completed her part of the key exchange.
    - `HMAC_Init1`: The HMAC Alice computed to authenticate the integrity of the exchange process up to this point.
2. **Transcript Construction:** Bob constructs **Key Exchange Transcript 1** (`Key_Exchange_Transcript1`) based on earlier messages exchanged during the key exchange process.
3. **Updated Transcript Construction:** Bob then constructs Key Exchange Transcript 2 (`Key_Exchange_Transcript2`), which is similar to Transcript 1, but it includes the `FINISH` message from Alice. This transcript represents the key exchange process up to the point where Alice sends the `FINISH` message.
4. **Further Updated Transcript (Transcript 3):** Bob constructs Key Exchange Transcript 3 (`Key_Exchange_Transcript3`), which includes Alice's `FINISH` message and Bob's expected `FINISH_RSP` message. This transcript will be used to compute the final session key and verify the integrity of the key exchange.
5. **Deriving HMAC Keys:** Bob derives two HMAC keys using HKDF (HMAC-based Key Derivation Function) for the verification of the communication and the generation of the session key.
    - **HMAC_Key_Resp3**: This key is derived from the shared secret (`DH_Output`) and the **Key Exchange Transcript 3** using the string `'finished resp'`. This key will be used to generate the HMAC for Bob's response (`HMAC_Resp2`).

    - **HMAC_Key_Init3**: This key is derived from the shared secret (`DH_Output`) and the **Key Exchange Transcript 3**, using the string `'finished init'`.
6. **Verification of Alice's HMAC:** Bob now verifies Alice's `HMAC_Init1`, which Alice computed in the previous step:
    - Bob computes the expected HMAC (`Computed_HMAC_Init1`) for **Key Exchange Transcript 2** using the `finished resp` string, ensuring that Alice's `FINISH` message is valid and that the transcript has not been tampered with.
    - Bob compares Alice's `HMAC_Init1` to `Computed_HMAC_Init1` to verify the integrity of the data. If they match, it confirms that the communication up to this point is secure.
    - **Verification Condition**:
        - `Eq(HMAC_Init1, Computed_HMAC_Init1)`
7. **Deriving the Session Key:** Once Bob verifies the integrity of the key exchange process and ensures that Alice's `FINISH` message is legitimate, Bob derives the **session key** using **HKDF** with **Key Exchange Transcript 3** and the string `'sessionkey'`. This key will be used to secure further communication between Alice and Bob.
    - `Session_Key_Bob = hkdf(DH_Output, Key_Exchange_Transcript3, 'sessionkey')`
8. **Sending Finish Response:** Bob prepares to send a response to Alice. He constructs the `Finish_RSP_Message` containing:
    - `'FINISH_RSP'`: A message indicating that Bob has successfully completed the verification and is ready to finish the key exchange.
    - `HMAC_Resp2`: The HMAC that Bob computes over the **Key Exchange Transcript 3** to ensure the integrity of the final message.
9. Bob sends this `Finish_RSP_Message` to Alice.
10. **State Update:** After sending the `Finish_RSP_Message`, Bob transitions to a new state, marking the completion of the key exchange process and the successful derivation of the session key.

## Rule: Alice_Verifies_Finish_RSP_And_Derives_Session_Key

The rule `Alice_Verifies_Finish_RSP_And_Derives_Session_Key` describes the process by which **Alice** verifies the integrity of Bob's final response message and derives the session key for secure communication.

1. **Finish_RSP_Message**:
    - The final response message from Bob, which includes:
        - `'FINISH_RSP'` (message type).
        - `HMAC_Resp2` (HMAC signature of the message).
2. **VCA** (Protocol Context):

    - Contains the initial request and response messages between Alice and Bob, including their public keys and certificates
3. **Key Exchange Transcripts**:
    - Three versions of the key exchange transcripts, each representing different stages of the exchange:
        - `Key_Exchange_Transcript1`: Initial handshake with public keys and nonces.
        - `Key_Exchange_Transcript2`: Key exchange step including `'KEY_EXCHANGE_RSP'`.
        - `Key_Exchange_Transcript3`: Final step including the `'FINISH'` and `'FINISH_RSP'` messages.
4. **HMAC Verification**:
    - Alice calculates the HMAC on the key exchange transcript using the shared key and the messages exchanged. The calculated HMAC is referred to as `Computed_HMAC_Resp2`.
    - **Verification Step**:
        i. Alice compares the `HMAC_Resp2` received from Bob with the computed `Computed_HMAC_Resp2`.
        ii. If the two HMACs match, Alice can trust that Bob's message has not been altered.
5. **Session Key Derivation**:
    - Once the HMAC verification is successful, Alice proceeds to derive the **session key**.
    - Alice uses **HKDF** (HMAC-based Key Derivation Function) to derive the session key. The function takes the following inputs:
        i. The **shared secret** (`DH_Output`) resulting from the key exchange.
        ii. **Key_Exchange_Transcript3** (the final message exchange transcript).
        iii. The string `'sessionkey'` as a label to specify the key type.
    - The session key derived in this step will be used for encrypting and decrypting future messages between Alice and Bob.
6. **Commit the Session Key**:
    - Alice securely stores the derived **session key** (`Session_Key`) for future use.
    - The session key will provide the necessary cryptographic protection for the ongoing communication.
7. **State Update**:
    - Alice's state is updated to reflect the successful completion of the key exchange, including the derived session key and the verified messages.
8. **Session Key Derived**:
    - The final **session key** (`Session_Key`) is derived and stored securely for future use in encryption and decryption of messages between Alice and Bob.

(b) Show that the following traces exist in your model:

## 1. Lemmas to Ensure Protocol Completion

The following lemmas are written to ensure that both the Initiator and Responder can reach the end of the protocol:

1. **RolesForming_Exists**
2. **Certificate_Generation**
3. **Cert_Issued**
4. **Key_Exchange_Exists0**
5. **Key_Exchange_Exists1**

These lemmas check that the necessary actions are performed, enabling both participants to complete the protocol successfully. All of these lemmas have been formally proved, to verify that both the Initiator and Responder can successfully complete the protocol. These proofs ensure that each critical step—key generation, certificate issuance, and message exchange—occurs as intended, allowing the protocol to reach its conclusion without errors. Please wait for a significant amount of time for these lemmas.

## 2. The Initiator and Responder Can Agree on a Common Session Key

Based on our research and the model we've developed, everything appears to be correct in terms of the protocol's ability for the Initiator and Responder to agree on a common session key. However, we have observed that the process of proving this, while correct, is taking a significant amount of time.

We have conducted exhaustive research and applied various debugging techniques to ensure the protocol's execution is functioning as expected. After thoroughly analyzing the model, we have found no indication of any infinite loops, inappropriate behavior, or errors in the execution of the protocol. All steps leading to the session key agreement are being processed correctly.

Although the verification process is time-consuming, we are confident that the protocol is correct, and both parties will successfully derive a shared session key once the proof completes.

We considered verifying the following lemmas during the VCA phase because this phase involves the exchange of certificates, which play a crucial role in authentication. Since the VCA phase is where the participants exchange their certificates, it seemed like the most appropriate stage to validate authentication properties.

## (c) Prove one-sided authentication of the Responder holds.

For Bob to be authenticated to Alice, Alice must receive a response from Bob that is verifiable.

- Alice sends `Message_A` (`VCA_REQ`, Cert_A, `Protocol_Version1_2`).
- Bob receives `Message_A`, generates `Response_Message` (`VCA_RSP`, Cert_B, `Protocol_Version1_2`), signs it, and encrypts it.
- Alice decrypts `Encrypted_Response`, extracts `Response_Message`, verifies Bob's signature, and accepts Bob's identity.

The verification `Eq(verify(Signature_B, Response_Message, pkB), true)` in `VCA_Response_Processed_By_Alice` confirms that Alice correctly verifies Bob's response.

Since Alice can verify that Bob sent the response and that the message was not altered, Bob is authenticated. Thus, one-sided authentication of the Responder holds. Since Alice successfully verifies Bob's response using **`Eq(verify(Signature_B, Response_Message, pkB), true)`**, Bob's authentication is confirmed. Hence, the lemma comes verified.

## (d) One-Sided Authentication of the Initiator (Alice) Does NOT Hold

For Alice to be authenticated to Bob, Bob must be able to verify that the `VCA_REQ` message indeed came from Alice.

- Bob receives `Message_A`, but no cryptographic verification of Alice's signature occurs in the transition.
- This means Bob does not verify Alice's signature before processing and responding.

One-sided authentication of the Initiator (Alice) does not hold. Alice sends `Message_A` in plaintext, meaning any attacker could spoof this request. Since Bob does not verify Alice's identity before responding, Alice's authentication is completely absent. Hence, the lemma comes falsified.

# Protocol Execution, Challenges Faced and Work Distribution

- To prove the lemmas, we recommend not proving all the lemmas at once. It is advisable to comment out the other lemmas before proving any  individually.
- In all three projects, writing the protocol part was easier, but proving the lemmas was extremely difficult. It was very challenging to pinpoint where the problem was occurring, as Tamarin does not provide clear indications. Most of the time, the lemmas took a long time to prove, and others were getting killed, resulting in a significant waste of time.
- We could have done better if we had more time, as we would have been able to properly apply heuristics and tactics to optimize the process. With more resources, we could have efficiently fine-tuned the approach to resolve the issues and improve the proving time.
- Despite the challenges, we worked hard from the beginning and were able to prove almost all of the lemmas successfully. Our efforts paid off, and we were able to make significant progress, even though some parts were more difficult than others.
- For project distribution, *Mini Mariya Thomas* was responsible for project one, *Areeba T. Shoaib* focused on project two. For project 3, Tamarin modelling - Mini Mariya Thomas, Model Debugging was done by Areeba Tabassum Shoaib and Documentation was done by Mini & Areeba.