Bachelor thesis
Computer Science

Radboud University

Automated state machine learning of IPsec implementations

Author:
Bart Veldhuizen
S4492765

First supervisor/assessor:
Dr. ir. , Joeri de Ruiter
joeri@cs.ru.nl

Second assessor:
Paul Fiterău-Broştean, Msc.
P.Fiterau-Brostean@
science.ru.nl

August 29, 2017

Abstract

As the internet becomes a bigger and bigger part of our daily life we need to be sure that our communication is secure. IPsec ensures this by encrypting messages and authenticating different parts of the communication. Because IPsec operates on the Internet Layer it is able to secure all data packets using the Internet Protocol. The global working of the IPsec protocol is specified in its RFC [7]. However, before IPsec can be used, somebody needs to translate these formal specifications into actual software. These specifications can be very big, complex and sometimes even ambiguous. This makes the manual translation of specification to software a very complex and error prone task.

In this research we will show a way of automatically inferring finite-state machines of IPsec implementations. This serves to give a better insight in how strict these implementations follow their specifications. We will not try to proof that a certain implementation/IPsec itself is safe or give a complete state machine of the IPsec protocol itself.

For our learning process we use the L* learning algorithm, the random-words equivalence algorithm and a self-made mapper to translate between these algorithms and the implementations. Although this set up can be used to model every IPsec implementation supporting IKEv2, we will limit our scope to the Strongswan and Libreswan implementations.

We will show that our state machine models can be used to analyze the working of a specific IPsec implementation and compare different implementations. We We do this by describing examples of expected and interesting behaviour for each model.

Contents

Chapter 1

Introduction

In the last decade we have drastically increased the time we spend on our computers, mobile phones and televisions. To properly use these devices we constantly send data over the internet. This data can reveal a lot about someones personal life and needs to be protected from unwanted access. Therefore we have security protocols, standardized and secure ways for computers to communicate with each other. IPsec is one of these protocols. It provides end-to-end security and operates on the Internet Layer. This means that IPsec can protect all other applications and protocols which run on top of the Internet Layer. IPsec ensures security by encrypting messages and authenticating both the sender and the message itself. The specifications describing IPsec and its underlying mechanics have been updated several times and are currently described in RFC 4301 [7].

You would assume that having a secure protocol means having a secure way of communicating. However, a protocol has to be implemented before it can be used. This has proven to be an error-prone step, as a lot of vulnerabilities are caused by a wrong implementation of a correct protocol. This can have several causes, sometimes the protocol itself is not clear enough, often it is a simple mistake caused by the sheer size and complexity of these specifications. Both can result in unreliable, incompatible or insecure software. Therefore it is important to review how much the implementations follows the specifications. We did this by automatically inferring a state machine from IPsec implementations and comparing this to the corresponding specifications. This does not mean that we can prove an implementation to be safe or correct. But it does give us more insight in the way IPsec is implemented and how strictly these implementations follow the specifications.

We will use the L* algorithm to learn the IPsec state machines. Then we will use the randomwords equivalence algorithm to check the inferred state machine. The translation between these algorithms and the actual IPsec messages will be handled by our self-made mapper. In this thesis we will first describe the necessary parts of IPsec and the protocols it relies

on. Subsequently we will explain state machines and how we automatically infer them. We will explain which part of the IPsec protocol we decided to model and what our final set up looks like. Finally we will discuss the state machines inferred by our Learner.

Chapter 2

IPsec Background

2.1 IPsec

2.1.1 General

IPsec is a security protocol used to create a secure two-way communication between two endpoints, which can either be a single host or a whole network. Depending on the configuration it can offer confidentiality, data integrity, access control, and data source authentication.

It offers many different modes and supports virtually every cryptographic algorithm, which makes it suitable for a lot of different tasks. These can range from connecting whole networks to the tunneling of one program.

Because IPsec operates on the Internet Layer, it also secures data send over the transport and application layer.

The IPsec specification specifies a list of cryptographic algorithms which have to be supported by every implementation. Implementers are free to extend these algorithms to suit their own needs. This makes IPsec a very versatile protocol which can be used in very different scenarios.

2.1.2 AH & ESP

IPsec provides two different security protocols, AH and ESP, to secure an IP datagram. These protocols specify which data is protected and how this data is protected. The choice between these security protocols influences the security services provided by IPsec.

The Authenticated Header format (AH) provides data integrity and data origin authentication with optionally anti-replay protection being optional. This ensures that only the sender can send, re-send or edit a valid message. The messages can however be delayed and read, which can be necessary for the proper functioning of, for example, a firewall. Because AH authenticates an actual IP datagram it can only authenticate fields which are not altered during normal propagation of the datagram. This means that AH does not

support the changing of IP-addresses and/or ports and therefore does not support standard Network Address Translation (NAT), but instead relies on UDP encapsulation of the whole message.

The Encapsulating Security Payload format (ESP) provides confidentiality, data-origin authentication, data integrity, anti-replay protection and limited traffic-flow confidentiality. ESP supports encryption and/or authentication mode. Only using encryption enables an attacker to change the unencrypted part of a message and is therefore strongly discouraged. The key difference between ESP and AH is confidentiality, ESP uses encryption to make the message unreadable for third parties. In transport mode, ESP only provides integrity and authentication for the payload and not for the whole packet. UDP encapsulation needs to be enabled in order to use TCP or UDP combined with NAT.

If necessary AH and ESP can be combined to serve specific security goals. This is done by nesting the connections. For example, we could first encrypt a packet with ESP after which we authenticate it with AH. The receiver does this the other way around.

2.1.3 Tunnel & Transport mode

IPsec also offers two different modes of operation, which determine how the protected data is send.

Tunnel mode is used to create virtual private networks (VPNs). The original packet is taken and used as the payload for a new IP packet. This means that the whole inner (original) packet is authenticated and/or encrypted, depending on whether AH, ESP or both are used. The outer (new) packet is sent over the network with possibly a new destination. This means that the newly created packet will first arrive at the IPsec tunnel endpoint and from there the inner (original) packet will traverse the network to its final destination.

In transport mode only the payload of the original message is encrypted and/or authenticated. For AH this means that it can only authenticate the payload and several header fields which are not altered during transit. For ESP this means that it provides confidentiality when encryption is enabled and integrity when authentication is enabled. ESP only protects the payload of the message.

2.1.4 Security Associations

Security Associations (SAs) are one-way secure tunnels with specified cryptographic algorithms and corresponding keys. This means that we need at least two SAs to set up a two-way communication. In the Security Association Database, which stores all the cryptographic keys, all SAs are uniquely identified by the Security Parameter Index (SPI) and destination address.

This means that multicasting is possible by multiple database entries and multiple SAs can be used for the same parties. This enables different data to be protected by different algorithms and/or different modes like AH/ESP and Tunnel/Transport mode.

2.2 IKEv2

2.2.1 General

IPsec uses the Internet Key Exchange (IKE) protocol to safely set up Security Associations between multiple parties. We will be using the latest version of the protocol, IKEv2 [6]. IKEv2 is a combination of the older IKEv1, ISAKMP and Oakley protocols. IKEv1 and ISAKMP specified the way two parties set up a Security Association. Oakley provides perfect forward secrecy by using the Diffie-Hellman key exchange algorithm. IKEv2 parties can identify themselves with either certificates, the extensible authentication protocol or pre-shared keys. We will be using pre-shared keys in the form of a simple password known by both sides. IKEv2 works in message exchanges, each valid request message will have one corresponding reply message. Each IKE SA has an Initiator and a Responder. The Initiator role is given to the party sending the Initialization request. An IKE session begins by exchanging the Initialization and Authentication messages. After setting up the IKE SA we can create more SAs or start transferring information.

2.2.2 Initializing

The Initialization exchange (called IKE_SA_INIT) is used to negotiate cryptographic algorithms, nonces and Diffie-Hellman values. The IKE_SA_INIT consists of four parts which are the same for the Initiator and the Responder.

- The Header: This contains the SPI (used to identify the session), version numbers and some flags describing the role of the sender (Initiator/Responder) and message (request/reply).
- The SA1 payload lists all the supported cryptographic algorithm of the initiator and the chosen cryptographic algorithm of the responder. These algorithms are used to secure the IKE SA.
- The KE field defines the public Diffie-Hellman value used by the sender of the message in this session.
- The Nonce field defines the nonce used by the sender in this session.

This exchange is sent in plain and will later be authenticated by the Authentication exchange. Now each party can generate SKEYSEED which is

generated from the exchanged nonces and Diffie-Hellman keys. SKEYSEED is used to generate all encryption, authentication and child (see 'Other Messages') keys. Each direction of messages has its own key.

2.2.3 Authenticating

The authenticating exchange authenticates both the current and previous exchange. It also negotiates the SA used by IPsec. This exchange is encrypted and authenticated by the agreed upon security suite of the Initialization step. A basic authentication message consists of at least 5 parts.

- The Header: This contains the SPI (used to identify the session), version numbers and some flags describing the role of the sender (Initiator/Responder) and message (request/reply). *From here on the other fields are encrypted and authenticated.*
- Identification Payload: Used to identify the sender of the message. Enables a single server to have multiple identities.
- Authentication Payload: Authenticates the initialization exchange using a special authentication key derived from SKEYSEED.
- SA2 Payload: This is used to negotiate the security protocol (AH or ESP) and corresponding cryptographic algorithms used by IPsec. The Initiator offers a list of supported algorithms and the Responder selects one of them.
- Traffic Selectors: These determine the IP-addresses and ports which will be forwarded through the IPsec SA. There is a separate Traffic Selector for each side of the IPsec SA. Parties send both Traffic Selectors to be sure the correct information is forwarded.

The receiver of an authentication message must verify whether all Encryption and Authentication is done correctly. If so, the party knows they communicate with the right person. If anything went wrong during the exchange, a Notify message has to be sent describing the problem and the SA of the IKE remains unchanged.

2.2.4 Other Messages

There are two other types of message exchanges, one for the agreement of further SAs and one for the passing of control messages.

The CREATE_CHILD_SA message exchange is needed for rekeying current SAs and creating new SAs. This means that a IKEv2 session can produce multiple SAs for IPsec tunnels and itself. The new SA uses a different nonce and SPI, possibly using different cryptographic algorithms and/or different Diffie-Hellman groups.

Informational messages should be send in the secure IKE SA tunnel so they are authenticated and encrypted. Each message can contain any number of Notification, Delete and Configuration payloads. There may be messages send outside of the IKE SA in case of unknown SPIs or incompatible IKE versions.

Chapter 3

Automated Modelling

3.1 Mealy Machines

We will use Mealy machines to model the implementations we learn in this research.

3.1.1 Characteristics

A Mealy machine is a deterministic finite-state machine where for every state and input combination there is at most one corresponding transition and output [9]. It has some key characteristics which are useful for our state machine modelling:

- Deterministic: This means that the same input sequence will always lead to the same output sequence. This represents the working of IPsec implementations which reply to the same message sequence in the same way.
- Finite set of states: We want to end up with a finite-state machine which models every input sequence. A finite amount of states represents the limited number of different states in which an IPsec implementation can be.
- Transition function: Mealy machines take both the current state and input to generate an output. This is important because the previous message sequence influences the reply of an IPsec implementation.

Mealy machines also have a limited input and output alphabet, just like we are limited in our messages by the IPsec specifications.

3.1.2 Working

Mealy machines always start in the same initial state. When given an input it will follow the corresponding outgoing transition. This transition is deter-

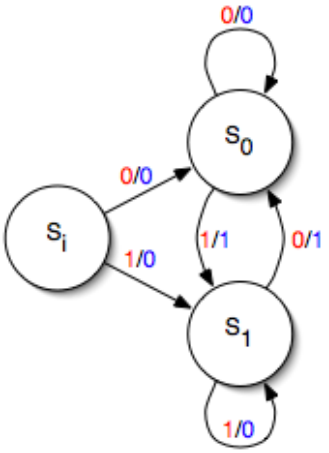

Figure 3.1: Example Mealy Machine

mined by the current state and input symbol. Each transition has an output which will be returned after following this transition. The state where the transition points to becomes the new current state. These steps can be repeated when the input consists of more than one symbol. To illustrate this, we give an example with the input 101 for the state machine given in Figure 3.1. The start state is S_i and both the input and output alphabet consist of the symbols 1 and 0.

- S_i with input 1 brings us to S_1 with output 0 and remaining input 01
- S_1 with input 0 brings us to S_0 with output 1 and remaining input 1
- S_0 with input 1 brings us to S_1 with output 1 and no remaining input

Thus the output of this example is 011.

3.2 State machine learning

3.2.1 Learning Process

Teacher

Our learning process requires a so-called teacher. The teacher is assumed to know the full state machine and can answer simple questions about the state machine. This is used by the algorithms explained below. Given a certain input sequence the teacher is able to return the corresponding output sequence. Because we are inferring the state machine for the first time we need to implement something that can simulate the teacher. We will therefore send these input sequences directly to the IPsec implementation. This is done by our mapper which is explained in section 4.2. Although the

IPsec implementation itself is not explicitly aware of its own state machine, it can obviously answer which output it returns given a certain input.

Learning Algorithm

The L* algorithm [2] is used to learn the finite-state machine corresponding to the System Under Learning (SUL)[1], which has often been used in black-box scenarios. This means that we can see the output given an input, but we cannot see anything that is happening in between. In our case this means that we send an IPsec message sequence to the SUL and learn the state machine based on the returned IPsec message sequence. Once the L* algorithm completes its hypothesis of the state machine it passes this to the equivalence algorithm.

Equivalence Algorithm

The equivalence algorithm checks whether the inferred state machine corresponds to that of the SUL. We will use the randomwords algorithm for this. It checks the model by generating a random input sequence and checking the answer of the SUL against the inferred model. In our research we set the minimum length of these random input sequences to 5 and the maximum length to 11. If the SUL and inferred model return the same output string the algorithm continues with a different input string. After a certain amount of successful tries the algorithm will conclude it can not find a counterexample, in our research we require 5000 successful tries. However, if the output sequences differ, the algorithm will give this counterexample to the learning algorithm to show that its inferred state machine is incorrect. The learning algorithm will then continue to search for the state machine of the SUL.

3.2.2 Implementing

We will use existing implementations of these algorithms rather than implementing them ourselves. Both of the tools have been used before in related research as described in section 6.1.

LearLib is an open java implementation of existing state machine inferring algorithms [2]. We will use their implementation of the L* star learning algorithm and the 'randomwords' equivalence algorithm for our state machine learning.

[1] From now on we will refer to the IPsec implementation we are trying to model as SUL.

[2] Learnlib. https://learnlib.de/

StateLearner will serve as a interface between the mapper and the algorithms implemented in Learnlib [3]. This enables us to easily send and receive queries to/from these algorithms.

[3] Joeri de Ruiter. Statelearner. https://github.com/jderuiter/statelearner

Chapter 4

Implementation

4.1 Scope of modelling

In this section we will discuss which aspects of IPsec and IKEv2 we are going to model. Since IPsec is a very modular protocol it enables the user to configure it in many different ways. The goal of our research is to automatically extract a state machine from various IPsec implementations. We will therefore implement a minimal amount of different IPsec settings while still trying to extract the full state machine of the SUL. For our input alphabet we will only use valid IPsec messages, although these valid messages may be part of invalid sequences of messages. This means that we can, for example, send messages over already closed SAs, repeat messages twice or skip necessary messages. We will discuss IKEv2 separately from IPsec since this part of the protocol is the most interesting for state machine learning.

4.1.1 IPsec

In our set up we will only configure the IPsec SA to use ESP. The difference between setting up a AH or a ESP SA is simply a protocol number in the initialization message and optionally choosing different types of algorithms. Once the IPsec SA is set up there will be no state changes in that particular SA unless messages are sent over the corresponding IKEv2 SA. We choose ESP because it enables both confidentiality and integrity and thus provides the security of AH and more in transport mode.

In IPsec we have two main modes of operation, tunnel and transport mode. Tunnel mode secures the whole IP datagram while the security features of transport mode only affect part of the IP datagram. Just like the previously chosen security architecture, the difference between the two while setting up a SA for IPsec is minimal. In this case it is a matter of adding a notify message in the IPsec SA set up. Because the mode of operation only has influence on the way a packet is encrypted and/or authenticated we will only implement one. We choose tunnel mode because it can protect the

whole IP datagram and is strongly recommended by the RFCs describing it.

The cryptographic algorithms we choose to encrypt and authenticate our IP datagrams with do not have any consequences for the way the IPsec SA is set up. It will only result in the use of a different algorithm while encrypting/authenticating an IP datagram. We will use the following cryptographic algorithms:

- Encryption algorithm: AES-CBC 128-bit
- Integrity algorithm: HMAC_SHA1_96

We will not use extended sequence numbering. Enabling this would result in using 64-bit sequence numbers instead of the original 32-bit.

4.1.2 IKEv2

IPsec relies on IKEv2 for setting up SAs and the associated keys and algorithms. Since initializing, authenticating, re-keying and notifying is handled by IKEv2 the majority of our learned state machine will probably model IKEv2. Because we are interested in learning a state machine, we implement the messages which we expect to change or influence the state of the SUL. This results in the following list of supported IKEv2 messages. The first row is the exchange type as described in Section 2, the second row describes the actual message.

- IKE_SA_INIT: Initialization message
- IKE_AUTH: Authentication message
- CREATE_CHILD_SA: Rekey IKE SA
- CREATE_CHILD_SA: Rekey ESP SA
- INFORMATIONAL: Delete current IKE
- INFORMATIONAL: Delete old IKE
- INFORMATIONAL: Delete current ESP (over old IKE)
- INFORMATIONAL: Delete old ESP (over old IKE)
- INFORMATIONAL: Delete current ESP (current IKE)
- INFORMATIONAL: Delete old ESP (current IKE)
- INFORMATIONAL: Test current IKE
- INFORMATIONAL: Test old IKE

- IPsec: Test current ESP
- IPsec: Test old ESP

Here we will discuss per exchange type which messages we choose to implement, which not and why we made this choice. In general we will not implement optional payload defined outside the IKEv2 RFC [6].

Initialization exchange

Here the cryptographic algorithms, nonces and Diffie-Hellman values will be exchanged. Since this is the first message and we use the same encryption algorithms for each exchange we will just implement one Initialization message. This message will contain the necessary payloads SA, nonce and Diffie-Hellman values. We will not include optional notify payloads like "NAT_DETECTION_SOURCE_IP" or "NAT_DETECTION_DESTINATIO N_IP". These payloads serve to detect network configurations, but once an SA is established should have no effect on message flow of our learned state machine.

Authentication exchange

Just like in the initialization exchange there is just one possible message. Our Authentication message will contain the Identification, Authentication, SA, Traffic Selector$_{initiator}$ and Traffic Selector$_{responder}$ payloads. Since we use pre-shared keys we will not use any extra authentication payload using certificates.

Create Child SA exchange

There are three different types of Create Child SA exchanges. The first one takes care of rekeying the IKE SA. This means that the IKE will get new SPIs, new nonces, new Diffie-Hellman values and possibly different cryptographic algorithms. The IKEv2 RFC [6] describes only one possible implementation of this message, which we will follow.
The second message takes care of rekeying an existing IPsec SA, which is of the type ESP or AH. The old IPsec SA will be deleted if the new SA is correctly set up. Since we only use ESP we will only implement this message for ESP rekeying. We will only include the necessary payload and the optional Diffie-Hellman payload will be omitted. Difie-Hellman values influence the concrete values of the cryptographic keys and therefore do not influence the state machine. Finally we can use the Create Child SA to create a new IPsec SA, which can be either ESP or AH. This new IPsec SA will exist alongside the old IPsec SA. Because we only use ESP we will only implement this message to set up a new ESP SA. Since having either one

or two ESP SAs can influence the learned state machine we will implement both. For the same reasons as described above we will omit the optional Diffie-Hellman exchange.

Informational exchange

An informational exchange can consist of any combination of Delete, Notify and Configuration payloads. An empty informational exchange is used to either test an IKE SA or acknowledge the closing of that IKE SA. We will implement the Delete and the empty payload. The empty message should not change the state of the connection but can determine whether it is open or not. This can be useful for determining the current state. The Delete payload can either close the IKE SA it is sent over or any of its Child ESP SAs. We implement the Delete payload because it closes connections and thus changes the state of the connection. We implement six types of Delete messages, two for the IKE SAs and four for the ESP SAs. We decided to send the Delete ESP messages over both the old and new IKE SA. The specifications state that Delete ESP messages should be send over the new IKE. But because sending a Delete ESP message over the old IKE once was a valid message (before rekeying the IKE SA), we still want to see whether it influences the state of a connection.
The notify payload is used to exchange error and status information. The error messages can give information about a connection but do not effect the connections themselves. We already have multiple ways of determining the state of a connection which makes this exchange redundant. Of the status types only REKEY_SA has direct influence over the connection and we therefore did implement it. It determines whether we create a new Child ESP SA or rekey the current ESP SA.
Configuration payloads are used to exchange information which would normally be provided by the Dynamic Host Configuration Protocol (DHCP). This enables an IKE peer to use the Local Access Network (LAN) of the other peer like their own. While this information can influence the fields of the ESP messages it should not affect the state of the implementation. We will therefore not implement this payload.

4.2 Mapper

4.2.1 General

Our mapper will translate packets back and forth between the learner and the IPsec implementation. It will receive instructions from the learner in the form of packet names, as described above, and will translate these into valid IKEv2 messages sent to the server. Once the server answers, either

by sending a message or not answering at all, it will decode this message and update local variables where necessary. We keep track of identification information (SPIs), nonces, Diffie-Hellman keys and the actual keys used for encryption, authentication and key derivation. These variables are necessary for future exchanges and must be stored. Besides this absolutely necessary information we keep our mapper stateless. This is important as having states in the mapper could result in having the learner infer the mapper states as well as the SUL states. Once all values are extracted we send the type of the reply back to the learner.

4.2.2 Scapy

Scapy is a packet manipulation tool often used in the field of information security. It allows the user to easily craft, send, receive and parse packets while still maintaining the raw packet. We will use Scapy as our main tool for communicating with the IPsec implementations. Scapy enables us to easily create messages and payloads as described in the RFC. We will also use it to decode IP datagrams received from the IPsec implementation. Because Scapy 3.0.0 does not have IPsec/IKE/ISAKMP as fully tested modules and we wanted some extra freedom while crafting packets, we decided to implement the Transform, Authentication, Identification, Traffic Selector, Delete and Notify payloads ourselves. Since a lot of values are different between IKE SAs we also needed to write functions around all existing payloads in order to properly fill the fields. These payloads are still send inside a Scapy message and also decodable by Scapy.

4.2.3 Security Services

We implemented the majority of the cryptographic procedures. This consists of the key derivation algorithm, Diffie-Hellman shared secret calculation, Message encryption/decryption, Message authentication and Authentication payloads. This is done for both IKE and ESP. For the actual implementation of cryptographic algorithms used in these procedures we use the PyCrypto library [1].

4.3 Final set up

Our final set up looks like Figure 4.1, on the left side we have the L* algorithm which is implemented by LearnLib and wrapped by the StateLearner. This is the part which does the actual state machine learning. It sends the names of the messages it wants to send to the mapper. The mapper will take these names and translate them to actual IP datagrams. After receiving the

[1] Dwayne Litzenberger. Pycrypto. https://pypi.python.org/pypi/pycrypto/2.6.1

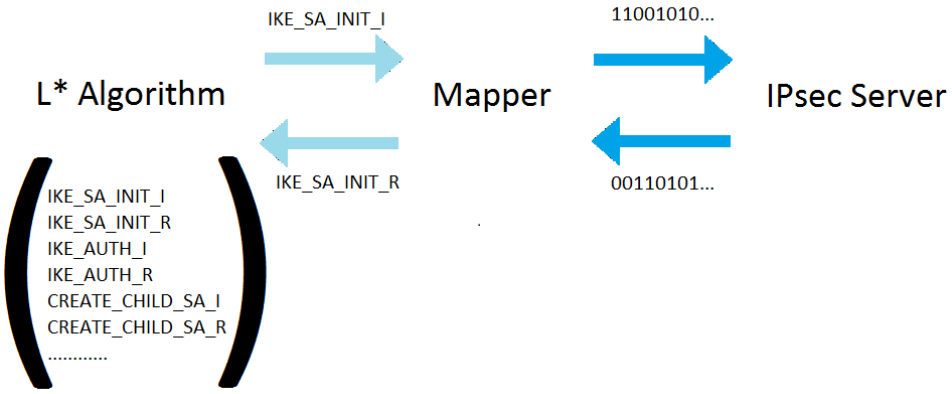

Figure 4.1: Final Set Up

reply it updates local values, translates the datagram back to a message name and sends this to the learning algorithm. The Learner will use these replies to infer the state machine of the SUL. This back and forth translating continues until our Learner finds a acceptable model of the SUL.

Chapter 5

Analysis

We learned models for two implementations, to be precise Strongswan 5.3.5 and Libreswan 3.20 (netkey). We wanted to learn the state machine of implementations which everybody could use in practice, which resulted in the following criteria. First of all, the implementation has to be publicly available and free to use. Secondly, it has to implement the latest version of IPsec. This means that we do not model combinations of protocols like L2TP/IPsec or IPsec with IKEv1. Thirdly, it has to have some source code change in the last 6 months. This ensures that we model an implementation which is still actively being managed. These criteria resulted in the selection of Strongswan and Libreswan. The inferred models can be found in Appendix A, as well as their cleaned versions. In the cleaned versions every transition that does not cause a state change is removed.

5.1 Strongswan

Strongswan was originally based on the FreeS/WAN project but is completely rewritten and does therefore not share any code with its ancestor. It currently supports Linux, Android, FreeBSD, Mac OS X, Windows and iOS. The original inferred model for Strongswan 5.3.5 can be found in Figure A.1 and the cleaned version can be found in Figure A.2.

5.1.1 Expected behaviour

- The transition from state 0 to state 1 is the 'IKE_AUTH' exchange. All other transitions originating in state 0 result in an 'ERROR' and cause no state change. This is what we expect as the 'IKE_INIT' followed with a 'IKE_AUTH' message is the specified way to open a new IKE SA as stated by the specifications [6]. There is one exception which we will discuss later.

- We can see that opening and then closing a ESP SA result in the

same state. The messages included are 'CREATE_NEW_ESP', which creates a new ESP, and 'DEL_NEW_ESP_NEW_IKE', which closes the last opened ESP SA over the last opened IKE SA. This behaviour can be seen in states 2 and 7, 3 and 9 and finally 1 and 4.

- State 5 has no outgoing transitions. All incoming transitions are 'DEL_NEW_IKE', which is expected as this is the designed way of closing a IKE SA. There is one exception which we will discuss later.
- All state changes are caused by transitions which return a non-error value as output. This is expected as an error indicates a time-out and should therefore not result in a state change. There is one exception which we will discuss later.

5.1.2 Interesting behaviour

- If we send a test message over a just initialized IKE SA it is immediately closed. This test message needs to be send authenticated and encrypted and can therefore only be send by the one of the communicating parties. Although the RFC [6] states that the first two messages should be 'IKE_INIT' and 'IKE_AUTH', it does not specify what should happen if it receives another message, the relevant specification is discussed in 5.3.1. Since this message causes the only outgoing transition from state 0, except for 'IKE_AUTH', it might be interesting to find out why.
- We can see that the 'REKEY_IKE' messages result in a loop between two states. The implementation does not respond to any message on the new IKE SA until a valid message is send over the old IKE SA. This might be because the developers interpreted the specification described in 5.3.2 as saying that the old IKE SA needs to be closed immediately after opening a new IKE SA. However, this behaviour does seem to be contradictory to the specification described in 5.3.3, which states that when the initiator sends a valid message over the new IKE SA, the responder is assured that the initiator wants to communicate over that IKE SA. The behaviour can be seen in states 1 and 6, 4 and 10, 2 and 8 and finally 7 and 11.

5.1.3 Conclusion

The Strongswan implementation is very small and consists of building blocks which are repeated multiple times, as described above. It seems interesting that only a test message can interfere with the usual opening exchanges, while all other messages do not. It is not possible to send any messages over the new IKE SA after a 'REKEY_IKE', which is contradictory to 5.3.3, but

this behaviour seems to be caused by prioritizing a strict interpretation of 5.3.2. It might be interesting for the developers of Strongswan to look at both interesting behaviours and see whether it was intended like this or not.

5.2 Libreswan

Just like Strongswan, Libreswan is based on the FreeS/WAN codebase. However, instead of rewriting all the code, it uses the existing codebase and only includes extra features which FreeS/WAN did not have. The original inferred model for Libreswan 3.20 (netkey) can be found in Figure A.3 and the cleaned version can be found in Figure A.4.

5.2.1 Expected behaviour

- The transition from state 0 to state 1 is the 'IKE_AUTH' exchange. Because this is the only transition outward from state 0 we can conclude that the other messages fail on an uninitialized IKE SA. This behaviour follows the specification as described in 5.3.1.
- State 5 has no outgoing transitions. All messages sent result in a 'ERROR' reply. This means that the connection is completely closed in this state.
- We can see that opening and then closing a ESP SA results in the same state. The messages involved are 'CREATE_NEW_ESP', which creates a new ESP, and 'DEL_NEW_ESP_NEW_IKE', which closes the last opened ESP SA over the last opened IKE SA.

5.2.2 Interesting behaviour

- States 5, 26, 38, 41, 43, 49, 61 and 74 all act as final states. These states just differ in the fact that they either reply a 'NOTIFY' or an 'ERROR' to a specific message. To be precise, the Libreswan sends back a 'INVALID_IKE_SPI' notification, telling our mapper that the used SPI no longer corresponds to a open IKE SA. The relevant specification for this behaviour is discussed in 5.3.4.
- States 2, 7, 19, 21, 27 and 67 all act as 'pass-through' states to the fully closed final state 5. This means that the connection does not immediately end in the final state 5 when a 'DEL_OLD_IKE' or 'DEL_NEW_IKE' message is send, but instead first passes through one of these states. These states only differ from state 5 in the fact that they return a 'NOTIFY' in response to some messages instead of a error. This 'NOTIFY' message is of the type 'INVALID_IKE_SPI' which notifies the sender that the used SPI does not correspond to

a open IKE SA. The relevant behaviour is discussed in 5.3.4. When a 'REKEY_ESP', 'CREATE_NEW_ESP' or 'REKEY_IKE' message is send to any of these states, it is answered by a 'ERROR' and the state changes to the final state 5. This behaviour is not discussed anywhere, but seems a unusual way to reach a final state.

- There are several transitions with an 'ERROR' response. This seems strange as you would expect a response for a message which causes a state change. All of these 'ERROR' transitions result from an earlier 'REKEY_IKE' combined with a message send over the old IKE SA. This indicates that when we are re-keying a IKE SA and we send something over the old IKE SA the Libreswan implementation sees this as a sign that the old IKE SA needs to remain open. This behaviour does not seem to be contradictory to the relevant specifications as described in 5.3.2.

- States 32, 34, 37, 40, 42, 46, 47, 48, 53, 60, 65, 66, 70, 71 and 75 only allow 'DEL_OLD_ESP_OLD_IKE', 'DEL_NEW_ESP_OLD_IKE' and/or 'DEL_OLD_IKE' messages to be send. This is the result from sending a 'REKEY_IKE' over the same IKE SA twice. The second time a 'REKEY_IKE' is send over such a connection the Libreswan implementation only responds to delete messages related to the old IKE SA, including ESP SAs opened on that IKE SA, until it is closed. This seems to be contradictory to the specification described in 5.3.3. However, when a 'REKEY_IKE' is send the first time, Libreswan does not force the old IKE SA to be closed. Although both behaviours seem to be in line with 5.3.2, the inconsistency might be a reason to look further into this behaviour.

5.2.3 Conclusion

The Libreswan implementation has a lot of interesting behaviour. Just like with Strongswan, not answering a valid message over the new IKE SA after receiving a 'REKEY_IKE' message seems to be contradictory to 5.3.3. Besides this, Libreswan has inconsistent behaviour, messages with an 'ERROR' response which still cause a state change and multiple final states. It might be interesting for the developers of Libreswan to find out what causes this behaviour.

5.3 Related specifications

In this section we will quote and discuss the specifications most relevant to our observed behaviour.

5.3.1 Opening a IKE SA

The IKEv2 RFC [6] chapter 1.2 states the following:

> Communication using IKE always begins with IKE_SA_INIT and IKE_AUTH exchanges (known in IKEv1 as Phase 1). . . . The first pair of messages (IKE_SA_INIT) negotiate cryptographic algorithms, exchange nonces, and do a Diffie-Hellman exchange [DH]. The second pair of messages (IKE_AUTH) authenticate the previous messages, exchange identities and certificates, and establish the first Child SA.

From this we can conclude that a IKE SA should be opened by a 'IKE_INIT' exchange followed by a 'IKE_AUTH' exchange. However, the specifications do not discuss what should happen if there is a different message involved.

5.3.2 Deleting a IKE SA after a rekey

The IKEv2 RFC [6] chapter 2.8 states the following:

> After the new equivalent IKE SA is created, the initiator deletes the old IKE SA, and the Delete payload to delete itself MUST be the last request sent over the old IKE SA.

This makes it clear that the last request send over a IKE SA must be the Delete payload. It is however unclear whether the old IKE SA needs to be closed immediately after opening a new IKE SA or whether the initiator can wait and close it later on.

5.3.3 Responding on the new IKE SA

The IKEv2 RFC [6] chapter 2.8 states the following:

> The responder can be assured that the initiator is prepared to receive messages on an SA if either (1) it has received a cryptographically valid message on the other half of the SA pair, or (2) the new SA rekeys an existing SA and it receives an IKE request to close the replaced SA. When rekeying an SA, the responder continues to send traffic on the old SA until one of those events occurs.

So the responder, in this case the IPsec implementation, knows that the initiator will answer on the new IKE SA if it received a cryptographically valid message on the new IKE SA or when the old IKE SA is closed. This means that if the old IKE SA is still open but there is a valid message send over the new IKE SA, the responder knows that the initiator wants to communicate over the new IKE SA.

5.3.4 Sending outside of the IKE SA

The IKEv2 RFC [6] chapter 1.5 states the following:

> There are some cases in which a node receives a packet that it cannot process, but it may want to notify the sender about this situation.
>
> o If an ESP or AH packet arrives with an unrecognized SPI. This might be due to the receiving node having recently crashed and lost state, or because of some other system malfunction or attack.
>
> o If an encrypted IKE request packet arrives on port 500 or 4500 with an unrecognized IKE SPI. This might be due to the receiving node having recently crashed and lost state, or because of some other system malfunction or attack.
>
> o If an IKE request packet arrives with a higher major version number than the implementation supports.

Although a closed connection is not explicitly listed as being a reason to send a notify message, not recognizing a SPI in itself can be a reason to send a notify message.

5.4 Comparison

Based on the discussed expected behaviour, the models look quite alike. However, the Libreswan model has a lot more states than the Strongswan model. This is due to the superfluous final states, 'pass-through' states and states which force you to close a specific IKE SA. This makes the Libreswan model a lot harder to understand and too large to easily grasp. The Libreswan model responds with a lot of 'INVALID_IKE_SPI' messages when the connection is already closed. Both Strongswan and Libreswan ignore messages send over the new IKE SA after receiving a 'REKEY_IKE', although Libreswan only does this after receiving this message twice.

Chapter 6

Related Work

This section covers two different fields of research, to be precise automated state model inferring and the analysis of IPsec. Since these two topics rarely overlap they are discussed separately.

6.1 Automated state model inferring

Several papers use the same set up for automatically inferring state machines as described in this thesis. The L* algorithm is used to learn the state machines and some kind of automated translation is done between this algorithm and the protocol that it is trying to model. This set up has been used to model security protocols like SSH [11], but also very basic internet protocols like TCP [8]. A paper describing a similar set up for modelling smart cards used in banking serves to show that this process can also be used on real world objects [4]. Similar research relied on passively examining message exchanges to create probabilistic protocol state machines [13], instead of actively sending messages. Although this set up differs from ours in an important way, it serves to show that state machines can be automatically inferred by merely looking at message exchanges.

6.2 IPsec

A relevant study was done by the Oulu University Secure Programming Group, in which they tested the robustness of IPsec implementations by sending both valid and invalid messages in varying sequences. This research was limited to IKE phase 1, which means that it only involves the IKE SA. The research was conducted in 2005 and is based on now outdated RFCs. It concluded that many implementations failed to perform in a robust manner and multiple vulnerabilities were found [12]. These vulnerabilities were all found after sending invalid messages and were related to the implementations themselves.

Another paper describes a way to extract passwords from people using IKE aggressive mode. This involves sniffing a hash of the pre-shared secret and brute-forcing the corresponding password offline [10]. Although this attack is very costly and IKE aggressive mode is not specified in the latest IKE RFC, it could still be used on outdated systems with poor passwords.

IPsec makes use of default Diffie-Hellman groups to secure the connections. There has been research on the dangers of using a small prime for Diffie-Hellman groups. When enough servers use the same small prime it can become appealing for an attacker to 'break the prime' of the Diffie-Hellman group used. It is argued that a nation state could perform such a attack on a 1024-bit prime [1]. The RFC [6] specifies both a 768-bit and 1024-bit prime. IKEv2 is however extended to use up to 8192-bit primes [5].

Chapter 7

Future Work

This research can be extended in several ways. We will discuss the main possibilities here.
It can be interesting to use the set up of this thesis to model other IPsec implementations. This could provide more insight in the way and strictness of which they were implemented. It could also help in determining differences between implementations and versions of implementations. This information can be useful for developers of IPsec software to get a clearer view of the working of their own and other people's software.
The models generated in this research could also be combined with a model-based tester to validate certain security properties of IPsec. Model-based testing has already been implemented for SSH [3]. A successful check could give more confidence in the security of IPsec while an unsuccessful run can help in improving the software. If the checking of IPsec models could be automated we could combine this with our own research. This would enable completely automated testing of IPsec implementations which first infers the state machine and then determines if the implementation follows certain security requirements.
The current research could also be extended by adding fuzzing to the mapper. This means that we change values in the messages send. This could be anything from sequence numbers to keys used for certain cryptographic algorithms. These messages could potentially uncover more states and even serve to find security bugs.
IPsec has several messages and extensions that we did not implement in this research. This enables further research to create a more complete model of IPsec implementations. It can also be interesting to implement different forms of authentication and or encryption. Comparing the state machines inferred with the use of different cryptographic algorithms and or protocol extensions could also give more insight in the way IPsec is implemented.
The technique used in this thesis can also be used to model other protocols. These could be similar security protocols like L2TP or any other state-full

protocol like HTTP or FTP. Only the mapper would need to be modified in order to support the messages specified by these protocols.

Chapter 8

Conclusions

Our research has shown that we can automatically infer state machines from IPsec implementations. This serves as evidence that the techniques used in this research are useful in modeling security protocols. Analyzing these models proved to be useful in determining whether the implementations we tested adhered to the corresponding specifications. The learned models also showed to be effective in comparing different implementations and it has given us a better insight in the working of these implementations. To be more precise, these models helped us in finding inconsistent behaviour, behaviour which possibly deviated from the specifications, superfluous states, ERROR transitions and unusual closing sequences of the selected implementations. All of this combined shows us that automated state machine inferring can provide us with a much better understanding of the working of security protocols, in our case IPsec.

Bibliography

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*, October 2015.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[3] Erik Boss. *Evaluating implementations of SSH by means of model-based testing.* Bachelor Thesis. Radboud Universiteit, 2012.

[4] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using Lego®. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego, CA, 2014. USENIX Association.

[5] Internet Engineering Task Force. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*, 2003.

[6] Internet Engineering Task Force. *Internet Key Exchange Protocol Version 2*, 2005.

[7] Internet Engineering Task Force. *Security Architecture for IP*, 2005.

[8] Ramon Janssen. *Learning a State Diagram of TCP Using Abstraction.* Bachelor Thesis. Radboud Universiteit, 2014.

[9] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, Sept 1955.

[10] Michael Thumann and Enno Rey. *PSK Cracking using IKE Aggressive Mode.* Enno Rey Netzwerke GmbH, 2003.

[11] Max Tijssen. *Automatic modeling of SSH implementations with state machine learning algorithms.* Bachelor Thesis. Radboud Universiteit, 2013.

[12] Universit of OULU. *PROTOS Test-Suite: c09-isakmp*, 2006.

[13] Yipeng Wang, Zhibin Zhang, Danfeng (Daphne) Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: A probabilistic approach. In *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, 2011.

Appendix A

Appendix

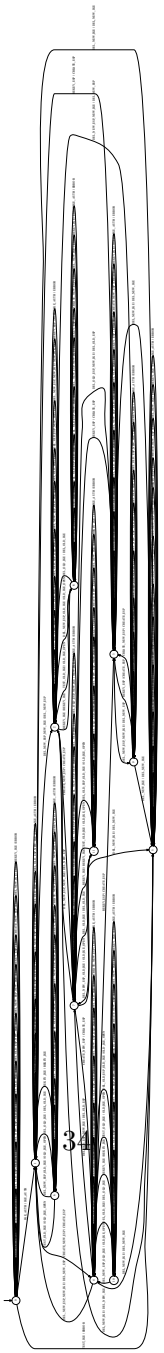

Figure A.1: Original Strongswan Model

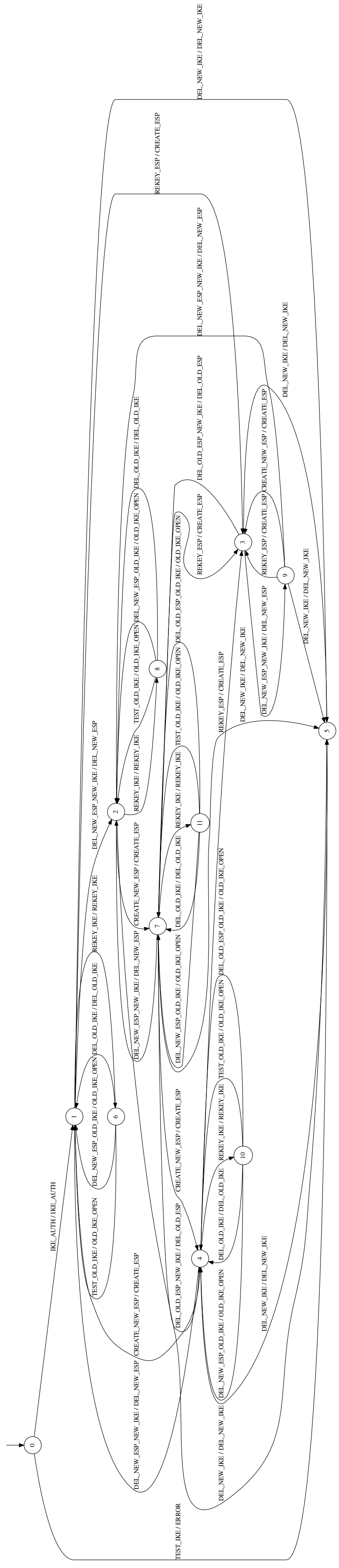

Figure A.2: Cleaned Strongswan Model

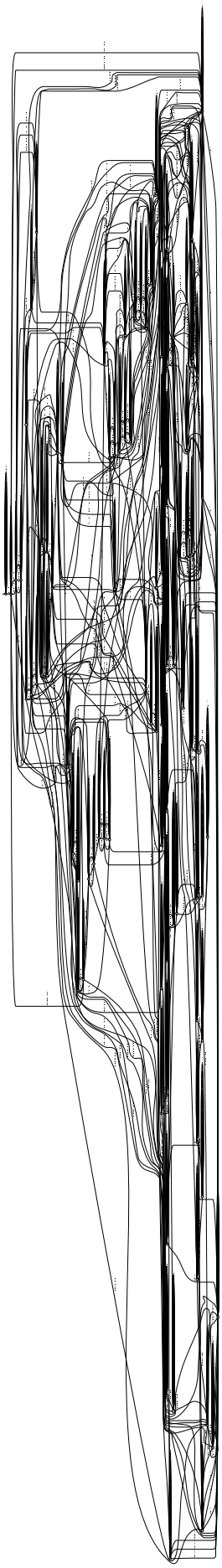

Figure A.3: Original Libreswan Model

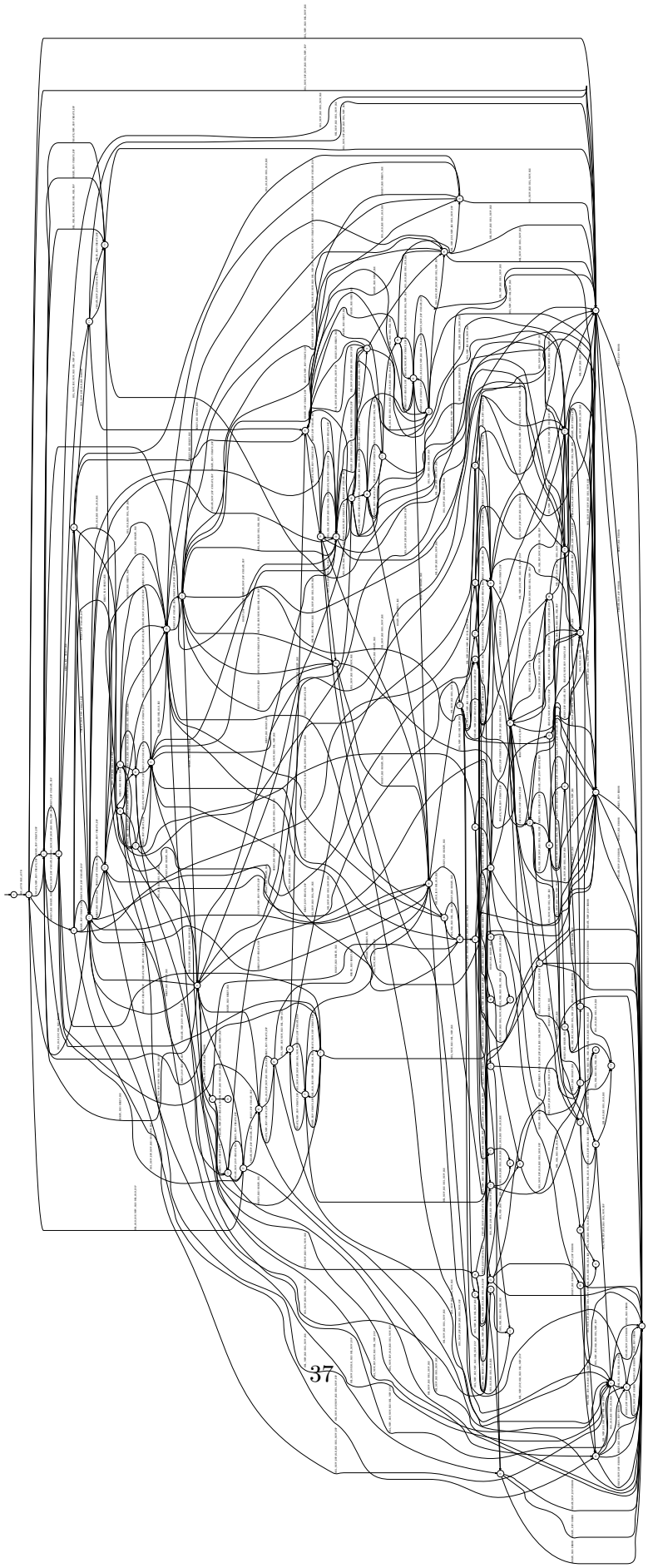

Figure A.4: Cleaned Libreswan Model