Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol

Benjamin Wunderling

Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol

Benjamin Wunderling B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl-Ing. Dr. Bernhard K. Aichernig
Institute of Software Technology (IST)

Graz, 10 Nov 2021

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The document uploaded to TUGRAZonline is identical to the present thesis.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Dokument ist mit der vorliegenden Arbeit identisch.

Date/Datum

Signature/Unterschrift

Abstract

Writing a thesis is a vast, overwhelming endeavour. There are many obstacles and false dawns along the way. This thesis takes a fresh look at the process and addresses new ways of accomplishing this daunting goal.

The abstract should concisely describe what the thesis is about and what its contributions to the field are (what is new). Market your contributions to your readership. Also make sure you mention all relevant keywords in the abstract, since many readers read *only* the abstract and many search engines index *only* title and abstract.

This thesis explores the issues concerning the clear structuring and the academic criteria for a thesis and presents numerous novel insights. Special attention is paid to the use of clear and simple English for an international audience, and advice is given as to the use of technical aids to thesis production. Two appendices provide specific local guidance.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I am indebted to my colleagues at the ISDS and the Know-Center who have provided invaluable help and feedback during the course of my work. I especially wish to thank my advisor, Keith Andrews, for his immediate attention to my questions and endless hours of toil in correcting draft versions of this thesis.

Special mention goes to Christian Gütl, Irene Isser, and Josef Moser for their help in translating the thesis abstract into German and to Bernhard Zwantschko and Didi Freismuth for ample supplies of coffee. Please remember to replace this tongue-in-cheek acknowledgements section with your own version!

Last but not least, without the support and understanding of my wife, pleasant hours with my girlfriend, and the tolerance of my friends, this thesis would not have been possible.

Keith Andrews

Graz, Austria, 10 Nov 2021

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews 2021].

Chapter 1

Introduction

1.1 Motivation

Virtual Private Network (VPN) are used to allow secure communication over an insecure channel. They function by creating a secure encrypted tunnel through which users can send their data. Example use cases include additional privacy from prying eyes such as Internet Server Providers, access to region-locked online content and secure remote access to company networks. The importance of VPN software has increased dramatically since the beginning of the COVID-19 pandemic due to the influx of people working from home [Abhijith and Senthilvadivu 2020]. This makes finding vulnerabilities in VPN software more critical than ever. Internet Protocol Security (IPsec) is a popular VPN protocol and most commonly uses the Internet Key Exchange protocol (IKE) protocol to share authenticated keying material between involved parties. Therefore, IKE and IPsec are sometimes used interchangeably. We will stick to the official nomenclature of using IPsec for the full protocol and IKE for the key exchange only. IKE has two versions, IKEv1 and IKEv2, with IKEv2 being the newer and recommended version [Barker et al. no date]. However, despite IKEv2 supposedly replacing its predecessor, IKEv1, sometimes also called Cisco IPsec, is still in widespread use today. This is reflected by the company AVM to this day only offering IKEv1 support for their popular FRITZ!Box routers [GmbH 2022]. Additionally, IKEv1 is also used for the L2TP/IPsec protocol, one of the most popular VPN protocols according to NordVPN [Ferguson and Schneier 2021]. The widespread usage of IPsec-IKEv1, combined with its relative age and many options makes it an interesting target for security testing.

1.2 Research Problems and Goals

State machines of protocol implementations are useful tools in state-of-the-art software testing. They have, e.g., been used to detect specific software implementations, or to generate test cases automatically [Pferscher and Aichernig 2021; Pferscher and Aichernig 2022]. Mealy machines are a type of state machine that can be used to describe the behavior a system when faced with external input. Often we are interested in testing software without knowing its exact inner workings. We call these systems black-box systems. However, despite lacking information on the inner structure of a black-box system, the state machine of the system can still be extracted. One method of generating the state machine of such a system is to use active automata learning. A notable example of an active automata learning algorithm is the L^* algorithm by Angluin [Angluin 1987a]. In L^*, a learner queries the system under learning (SUL) and constructs an automaton describing the behavior of the SUL through its responses. This automaton is then compared with the SUL, adapting it if they show different behaviors. The resulting automaton then fully describes the behavior of the SUL.

By combining automata learning with fuzzing or similar software testing techniques, network protocols can be extensively and automatically tested without requiring access to their source code. Guo et al. [Guo

et al. 2019] tested IPsec-IKEv2 using automata learning and model checking, however so far, no studies have focused on IKEv1 in the context of automata learning. Therefore our goal was to black-box test the IPsec-IKEv1 protocol using automata learning in combination with automata-based fuzzing. We used the active automata learning framework AALpy [Muškardin et al. 2022] with a custom mapper to learn the state machines of various IPsec-IKEv1 server implementations. We then further utilized the learned models for fuzzing and fingerprinting.

1.3 Structure

This thesis is structured as follows. Chapter 2 gives an overview of related and relevant literature. Chapter 3 introduces necessary background knowledge, covering the IPsec-IKEv1 protocol, Mealy machines, automata learning and fuzzing. Our learning setup, custom mapper and fuzzing methodology are presented in chapter 4. In chapter 6 we present and analyze the learned models and the results of the fuzzing tests. Finally we summarize the thesis in chapter 7 and discuss future work.

Chapter 2

Related

use workshop paper stuff

The aim of this chapter is to give a brief overview of related work, focusing mainly on automata learning and testing of secure communication protocols. 1987 The concept of learning through the means of membership and equivalence queries was introduced in 1987 by Angluin [Angluin 1987a]. Angluin presented an algorithm for learning regular languages from queries and counterexamples, called L^*. In it, a student questions a teacher and constructs a hypothesis based on its responses. The hypothesis is then tested through equivalence queries which check if the hypothesis correctly matches the regular language being learned. While the L* algorithm was originally designed to learn deterministic finite automata (dfa), it can be simply extended to work for Mealy machines by making use of the similarities between dfa and Mealy machines, as shown by Steffen et al. [Steffen et al. 2011]. Over time, many related and improved algorithms were published, such as the one proposed by Rivest and Schapire in 1993 in which homing sequences were used to infer finite automata [Rivest and Schapire 1993]. Another, more recent algorithm came in the form of a redundancy-free active automata learning approach titled TTT by Isberner et al [Isberner et al. 2014]. In this algorithm, essential data is stored in three tree data structures, stripping away unessential information.

Model learning network protocols for the purpose of testing is a more recent development, with models of protocols like SSH [Fiterău-Broştean et al. 2017], or TCP [Fiterău-Broştean et al. 2016] being learned and used for model checking. Both Novickis et al. [Novickis et al. 2016] and Daniel et al. [Daniel et al. 2018] learned models of the related OpenVPN protocol and used the learned models to perform protocol fuzzing. In a work by Pferscher and Aichernig [Pferscher and Aichernig 2021], learned models were used to fingerprint Bluetooth Low Energy devices (BLE), showing yet another possible use case of automate learning. Guo et al. [Guo et al. 2019] used automata learning to learn and test the IPsec-IKEv2 protocol. They used the LearnLib [1] library for automata learning and performed model checking of the protocol, using the learned state machine. In contrast, our work focuses on the IPsec-IKEv1 protocol, the predecessor of IPsec-IKEv2, which, to the best of our knowledge, has not yet been tested with methods utilizing automata learning. The protocols differ greatly on a packet level, with IKEv1 needing more than twice the amount of packets to establish a connection than IKEv2. Additionally we used the AALpy [2] library for automata learning and focused on fuzzing and fingerprinting as opposed to model checking.

[1] https://learnlib.de/

[2] https://github.com/DES-Lab/AALpy

Chapter 3

Preliminaries

3.1 Mealy Machines

taken from workshop paper - is this allowed, or should I rewrite this?

Mealy machines are a modeling formalism for reactive systems such as communication protocols. Mealy machines are finite-state machines in which each transition is labeled with an input and the corresponding output action. More formally, a Mealy machine is defined as a 6-tuple $M = \{S, s_0, I, O, \delta, \lambda\}$, where S is a finite set of states, $s_0 \in S$ is the initial state, I is a finite set called the input alphabet, O is a finite set called the output alphabet, δ is the state-transition function $\delta\colon S \times I \to S$ that maps a state and an element of the input alphabet to another state in S. In other words, the choice of each new state is defined by the current state and an external input. Finally, λ is the output function $\lambda\colon S \times I \to O$ which maps a state-input alphabet pair to an element of the output alphabet O. We use Mealy machines to model the state of learned IPsec implementations.

3.2 Automata Learning

Automata learning refers to methods of learning the state model, or automaton, of a system through an algorithm or process. Automata learning algorithms generate a model that describes the behavior of the SUL. We differentiate between active and passive automata learning. In passive automata learning (PAL), models are learned based on a given data set describing the behavior of the SUL, e.g. log files. In contrast, in active automata learning (AAL) the SUL is queried directly. In this paper, we will focus on AAL and will, moving on, be referring to it as automata learning or AAL interchangeably.

One of the most influential AAL algorithms was introduced in 1987 by Dana Angluin, titled "Learning regular sets from queries and counterexamples" [Angluin 1987b]. In this seminal paper, Angluin introduced the concept of the minimally adequate teacher (MAT) as well as the L^* learning algorithm. Variants of the L^* algorithm are still used for learning deterministic automata to this day, for example by the AAL python library AALpy [Muškardin et al. 2022]. While the original L^* algorithm was designed to learn deterministic finite automata (DFA) formalizing regular languages, the algorithm can be extended to learn Mealy machines [Niese 2003]. While many modern implementations, including AALpy use improved versions of L^*, such as using homing sequences [Rivest and Schapire 1993], fundamentally they still resemble the original algorithm by Angluin.

The MAT concept is also used by other active learning algorithms. Kearns and Vazirani [Kearns and Vazirani 1994] propose an active learning algorithm that uses an underlying tree-based data structure called a classification tree to construct a model. We refer to their learning algorithm as KV. Published later than L^*, it boasts a more compact method of representing learned data, which, on average, leads to

the *KV* algorithm requiring less membership queries than L^* to learn a system. Especially for learning internet protocols and other systems where communication with the SUL can be very time consuming, this can result in a significant performance increase. Both algorithms used in this paper are briefly explained below.

3.2.1 L*

L^* uses a MAT model to learn an unknown regular language L. This means, that we define both a learner and a teacher. The teacher must respond to two types of queries posed by the learner, namely membership and equivalence queries. Queries are built using a fixed input alphabet Σ where $L \subseteq \Sigma^*$ must hold. Membership queries consist of a word $s \in \Sigma^*$ and must be answered with either "yes" if $s \in L$, or "no" if not. In other words, membership queries are used to check if a given word is part of the language being learned. Equivalence queries on the other hand, consist of a regular language L_{prop}, proposed by the learner. The teacher must answer with "yes" if $L_{prop} \equiv L$, or else return a counterexample c proving the two languages are different, so $c \in L(S) \iff c \notin L$. In other words, equivalence queries are used to verify if the learner has successfully learned the target language L or if not, return a counterexample detailing the differences. The results of the membership queries are stored in an observation table $O = (S, E, T)$, where S is a prefix-closed set of strings representing candidates for states of L_{prop}, E a suffix-closed set of strings used to distinguish between candidates and T a transition function $(S \cup S \cdot \Sigma) \cdot E \rightarrow 0, 1$. Essentially, if visualized as a 2D array where the rows are labeled with elements in $(S \cup S \cdot \Sigma)$ and columns with elements in E, the entries in the table are ones, if the word created by appending the row-label to the column-label is accepted by L and zeros if not. The goal of L^* is to learn a DFA acceptor for L using the observation table. S-labeled rows correspond to states in the acceptor under construction. E-labeled columns represent individual membership query results. For the observation table to be transformable into a DFA acceptor, it must first be closed and consistent.

Closed is defined as for all $t \in S \cdot \Sigma$ there exists an $s \in S$ so that $row(t) = row(s)$. In other words, that no new information is gained by expanding the S-set by any word in Σ. If an observation is not closed, it is fixed by adding t to S and updating the table rows through more membership queries. Consistent means, that $\forall s_1, s_2 \,|\, row(s_1) = row(s_2) \implies \forall \sigma \in \Sigma \,|\, row(s_1 \cdot \sigma) = row(s_2 \cdot \sigma)$, or in other words, appending the same word to identical states should not result in different outcomes. If an observation table is inconsistent, it is made consistent again by adding another column to the table with the offending σ as its label and again updating the table rows through more membership queries.

Listing 3.1 shows the workings of the basic L^* algorithm by Angluin. The function $populate(O)$ extends T to $(S \cup S \cdot \Sigma) \cdot E$ by asking membership queries for all table entries still missing membership information. At the start of the algorithm, the observation table is initialized with $S = E = \{\epsilon\}$. Next, until a equivalence query succeeds, the observation table is repeatedly brought to a closed and consistent state by expanding the S and E sets respectively. Once both closed and consistent, L_{prop} is constructed from O and used in an equivalence query. If the equivalence query returns "yes", the algorithm terminates, returning the learned DFA. If not, the returned counterexample is used to update the observation table and the algorithm loops back to line 5.

3.2.2 KV

Another notable AAL algorithm is the KV algorithm published in 1994 by Kearns and Vazirani Kearns and Vazirani 1994. It is designed to work with the same MAT framework as L^*, but aims to minimize the amount of membership queries needed to learn a finite automaton M. The KV algorithm does this by organizing learned information in an ordered binary tree called a classification tree C_T as opposed to the table structure utilized by L^*. Intuitively, L^* must perform membership queries for every entry in the observation table to differentiate between possible states, whereas *KV* requires only a subset to distinguish them.

In the KV algorithm, learned data is stored in two sets called the access strings set S and the distinguishing strings set D. Every string $s \in S$ represents a distinct and unique state of the automaton M. In other words, any s when applied starting in the initial state of M leads to a unique state $M[s]$. The distinguishing strings set is defined as the set of strings $d \in D$ where for each pair $s, s' \in S, s \neq s'$ there exists a $d \in D$ such that either $M[s \cdot d]$ or $M[s' \cdot d]$ is an accepting state. D is used to ensure that their are no ambiguous states. The sets S, D are organized in a binary tree called the classification tree C_T where parent nodes are strings from D and the leaf nodes are strings from S. The root node is set to the empty string λ. For each node of the tree, starting from the root node, each right subtree contains access strings to accepting states while left subtrees contain access strings to rejecting states of M. Given a new string s', we simply start at the root nodes, then sift down the tree by executing a membership query for $s' \cdot \lambda_1$ and depending on if the query returns “yes" or “no" continuing with the left or right subtree until we reach a leaf node labeled with s. If $s' = s$ then the states are equivalent, otherwise the classification tree is updated to include another leaf node representing the newly learned distinct state s'. The main learning loop of the KV algorithm is shown in more detail in Listing 3.2. Following the initialization of the classification table, new states learned from counterexamples are repeatedly added until an equivalence query is successful. The $Update(C_T, c)$ function adds a new leaf to the C_T based on a counterexample c returned from an equivalence query.

```
1  Initialization:
2  Set observation table O = (S, E, T) with S, E = {ϵ}.
3  populate(O).
4
5  repeat:
6    while O is not closed or not consistent do
7      if O is not closed then
8        choose s1 ∈ S, σ ∈ Σ such that
9        row(s1 · σ) ≠ row(s) ∀s ∈ S
10       add s1 · σ to S
11       populate(O)
12     end
13     if O is not consistent then
14       choose s1, s2 ∈ S, σ ∈ Σ and e ∈ E such that
15       row(s1) = row(s2) and T(s1 · σ · e) ≠ T(s2 · σ · e)
16       add σ · e to E
17       populate(O)
18     end
19   end
20   Construct L_prop from O and perform an equivalence query.
21   if query returns a counterexample c then
22     add all prefixes of c to S
23     populate(O)
24   end
25 until teacher replies "yes" to equivalence query L_prop ≡ L
26 return L_prop
```

Listing 3.1: L^* algorithm

```
1  Initialization:
2    Set root node of C_T to ε.
3    Perform membership query on ε to determine if the initial state is
         accepting or not.
4    Construct hypothesis automaton M̂ consisting of only the initial state,
         with self-transitions for all other transitions.
5    Add two access strings ε and the counterexample string c.
6
7  repeat:
8    Construct hypothesis automaton M̂ from C_T.
9    Equivalence query(M̂)
10   if: query returns "yes" then
11     return M̂
12   end
13
14   Update(C_T, c)
15 end
```

Listing 3.2: *KV* algorithm

3.3 Fuzzing

Fuzzing, or fuzz-testing, is a technique in software testing in which lots of random, invalid or unexpected data is used as input for a program. The goal is to elicit crashes or other anomalous behavior from the system under test (SUT) that might serve as an indication of a software bug. To this end, lots of data is generated and sent to the SUT. Fuzzers mainly differ on the method of data generation and can be roughly categorized as generation-based or mutation-based fuzzers. Generation-based fuzzers generate their data from scratch, whereas mutation-based fuzzers modify, or "mutate" existing data. Additionally one can categorize fuzzers based on how much knowledge they have on the SUT, or in other words, how much information on the expected input structure is known by the fuzzer. To ensure that a fuzzer can test all relevant parts of a SUT, additional information on the structure of the SUT may be utilized. For example, one could use a model representation of the SUT to generate more relevant inputs and achieve better coverage. This technique is known as model-based fuzzing and is used in this thesis. While fuzzers come in many different shapes and forms, their core function is usually the same in that data is generated, sent to the SUT and then the SUT is observed for strange behavior.

Two popular examples of fuzzers are american fuzzy lop (AFL) [Zalewski 2020] and boofuzz [Pereyda 2022]. AFL is a software fuzzer, written mainly in C, that uses genetic algorithms in combination with instrumentation to achieve high code coverage of the SUT. The instrumentation has to be added to the the target by compiling the SUT using a custom compiler provided by AFL. AFL has been successfully used to find bugs in many high-profile projects such as OpenSSH [1], glibc [2] and even linux memory management [3].

Boofuzz on the other hand is a Python-based fuzzing library most commonly used for protocol fuzzing. As such, it does not require instrumentation to function. Instead, it supports building blueprints of protocols to be fuzzed via low-level building blocks called primitives. It allows users to specify protocol requests to be sent to the SUT and mark which portions of the request should be fuzzed via settings in the primitives. The data generated by the framework depends on the used blocks and settings, and then

[1] https://lists.mindrot.org/pipermail/openssh-commits/2014-November/004134.html
[2] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=772705
[3] https://bugs.chromium.org/p/project-zero/issues/detail?id=1431

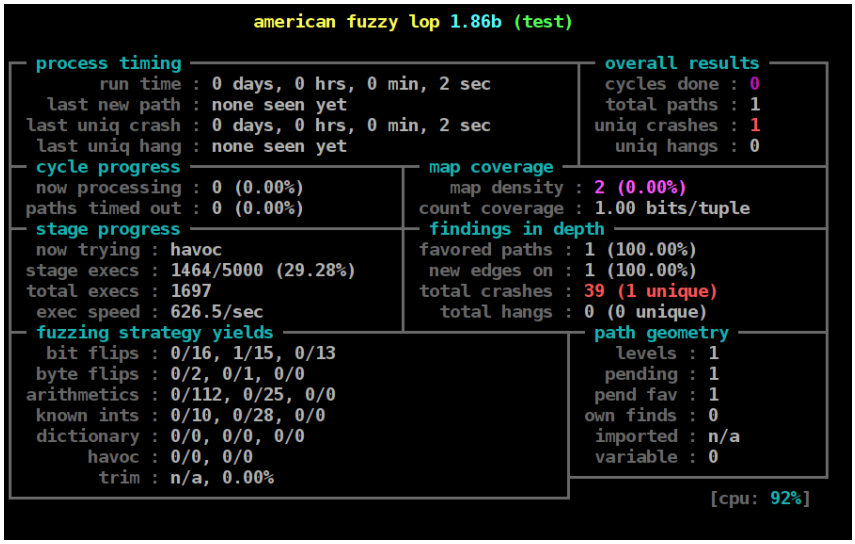

Figure 3.1: AFL fuzzer

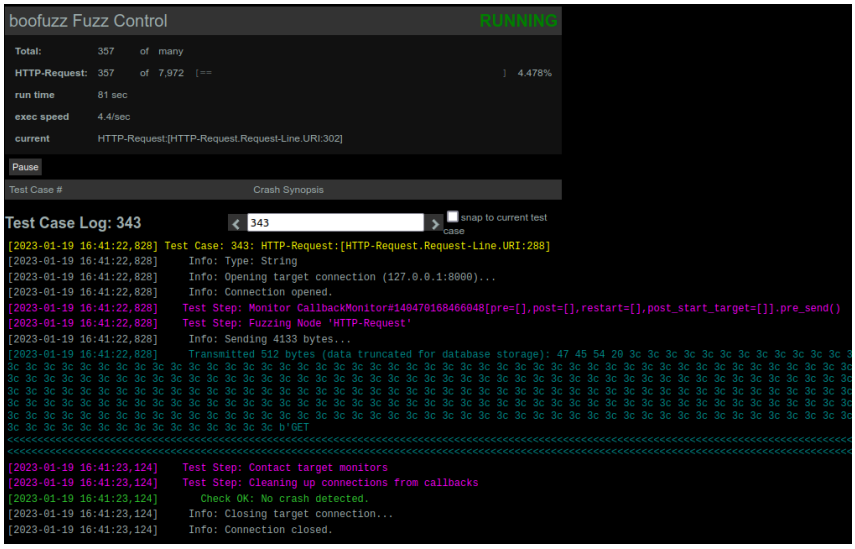

Figure 3.2: Boofuzz fuzzer

created mutations based on the specified protocol structure. Additionally the SUT can be monitored for crashes and other unexpected behavior and the framework can furthermore be instructed to restart or reset the SUT when needed. In this paper, we use boofuzz to generate our values for fuzzing, as detailed in Chapter 5.

3.4 IPsec

VPNs are used to extend and or connect private networks across an insecure channel (usually the public internet). They can be used e.g. to gain additional privacy from prying eyes such as Internet Server Providers, access to region-locked online content or secure remote access to company networks. Many different VPN protocols exit, including PPTP, OpenVPN and Wireguard. IPsec or IP Security, is a VPN layer 3 protocol used to securely communicate over an insecure channel. It is based on three sub-protocols, the IKE protocol, the Authentication Header (AH)) protocol and the Encapsulating Security Payload (ESP) protocol. IKE is mainly used to handle authentication and to securely exchange as well as manage keys. Following a successful IKE round, either AH or ESP is used to send packets securely between parties. The main difference between AH and ESP is that AH only ensures the integrity and authenticity of messages while ESP also ensures their confidentiality through encryption.

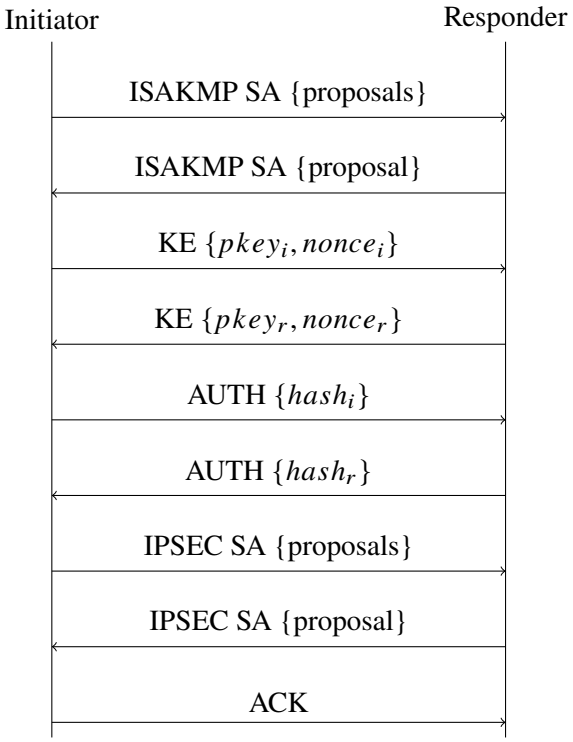

Figure 3.3: IKEv1 between two parties

The IKEv1 protocol works in two main phases, both relying on the Internet Security Association and Key Management Protocol (ISAKMP). Additionally, phase one can be configured to proceed in either Main Mode or Aggressive Mode. A typical exchange between two parties, an initiator and a responder, using Main Mode for phase one, can be seen in Figure 3.3. In phase one (Main Mode), the initiator begins by sending a Security Association (SA) to the responder. A SA essentially details important security attributes required for a connection such as the encryption algorithm and key-size to use, as well as the authentication method and the used hashing algorithm. These options are bundled in containers called proposals, with each proposal describing a possible security configuration. While the initiator can send multiple proposals to give the responder more options to choose from, the responder must answer with only one proposal, provided both parties can agree upon one of the suggested proposals. This initial communication is denoted as *ISAKMP SA* in Figure 3.3. Subsequently, the two parties perform a Diffie-Hellman key exchange, denoted as *KE*, and send each other nonces used to generate a shared secret key *SKEYID* as detailed in Listing 3.3. PSK refers to the pre-shared key, Ni/Nr to the initiator/responder

nonce and CKY-I/CKY-R to the initiator/responder identifier cookie. Note that IKEv1 allows using various different authentication modes aside from PSK, including public key encryption and digital signatures. *SKEYID* is used as a seed key for all further session keys *SKEYID_d*, *SKEYID_a*, *SKEYID_e*, with g^{xy} referring to the previously calculated shared Diffie-Hellman secret and prf to a pseudo-random function (in our case, HMAC). Following a successful key exchange, all further messages of phase one and two are encrypted using a key derived from *SKEYID_e* and *SKEYID_a* for authentication. Finally, in the last section of phase one *AUTH*, both parties exchange and verify hashes to confirm the key generation was successful. Once verification succeeds, a secure channel is created and used for phase two communication. If phase one uses Aggressive Mode, then only three packets are needed to reach phase two. While quicker, the downside of Aggressive Mode is that the communication of the hashed authentication material happens without encryption. This means, that using short pre-shared keys in combination with Aggressive Mode is inherently insecure, as the unencrypted hashes are vulnerable to brute-force attacks provided a short key-size [4]. The shorter phase two (Quick Mode) begins with another SA exchange, labeled with *IPSEC SA* in Figure 3.3. This time, however, the SA describes the security parameters of the ensuing ESP/AH communication and the data is sent authenticated and encrypted using the cryptographic material calculated in phase one. This is followed by a single acknowledge message, *ACK*, from the initiator to confirm the agreed upon proposal. After the acknowledgment, all further communication is done via ESP/AH packets, using *SKEYID_d* as keying material.

```
1  # For pre-shared keys:
2  SKEYID = prf(PSK, Ni_b | Nr_b)
3
4  # to encrypt non-ISAKMP messages (ESP)
5  SKEYID_d = prf(SKEYID, g^xy | CKY-I | CKY-R | 0)
6
7  # to authenticate ISAKMP messages
8  SKEYID_a = prf(SKEYID, SKEYID_d | g^xy | CKY-I | CKY-R | 1)
9
10 # for further encryption of ISAKMP messages in phase two
11 SKEYID_e = prf(SKEYID, SKEYID_a | g^xy | CKY-I | CKY-R | 2)
```

Listing 3.3: IKE Keying

In addition to the packets shown in Figure 3.3, IKEv1 also specifies and uses so called ISAKMP Informational Exchanges. Informational exchanges in IKEv1 are used to send ISAKMP Notify or ISAKMP Delete payloads. Following the key exchange in phase one, all Informational Exchanges are sent encrypted and authenticated. Prior, they are sent in plain. ISAKMP Notify payloads are used to transmit various error and success codes, as well as for keep-alive messages. ISAKMP Delete is used to inform the other communication partner, that a SA has been deleted locally and request that they do the same, effectively closing a connection.

Compared to other protocols, IPsec offers a high degree of customizability, allowing it to be fitted for many use cases. However, in a cryptographic evaluation of the protocol, Ferguson and Schneier Ferguson and Schneier [1999] criticize the complexity arising from the high degree of customizability as the biggest weakness of IPsec. To address its main criticism, IPsec-IKEv2 was introduced in RFC 7296 to replace IKEv1 [Kaufman et al. 2014]. Nevertheless, IPsec-IKEv1 is still in wide-spread use to this day, with the largest router producer in Germany, AVM, still only supporting IKEv1 in their routers [GmbH 2022]. We

[4] https://nvd.nist.gov/vuln/detail/CVE-2018-5389

use IPsec-IKEv1 with Main Mode and ESP in this paper and focus on the IKE protocol as it is the most interesting from an AAL and security standpoint.

Chapter 4

Learning

4.1 Environment Setup

We developed and tested our custom mapper using two VirtualBox 6.1 virtual machine (VM)s running standard Ubuntu 22.04 LTS distributions. Both VMs were allotted 4GB of memory and one CPU core. All communication took place in an isolated virtual network to eliminate possible external influences. During learning, all power saving options and similar potential causes of disruptions were disabled. For larger learning attempts, the IPsec server was restarted before each attempt to ensure identical conditions. We designated one VM as the initiator and one as the responder to create a typical client-server setup. The used Strongswan config files can be found in Appendix TODO. The open source IPsec implementation Strongswan[1] was installed on the responder VM and set to listen for incoming connections from the initiator VM. We used the Strongswan version US.9.5/K5.15.0-25-generic, installed using the default Ubuntu package manager, apt. The Strongswan server was configured to use pre-shared keys for authentication and default recommended security settings. Additionally, it was configured to allow unencrypted notification messages, which we used to reset the connection during the learning process. For learning, we used the Python library AALpy[2] version 1.2.9 in conjunction with the packet manipulation library Scapy[3], version 2.4.5. Significant effort was put into expanding the ISAKMP Scapy module to support all packets required for IPsec as the module lacked many features out-of-the-box. The provided Python script, *IPSEC_IKEv1_SUL* demonstrates how we use the AALpy in conjunction with our custom mapper to communicate with and learn the model of an IPsec server.

[1] https://www.strongswan.org/
[2] https://github.com/DES-Lab/AALpy
[3] https://scapy.net/

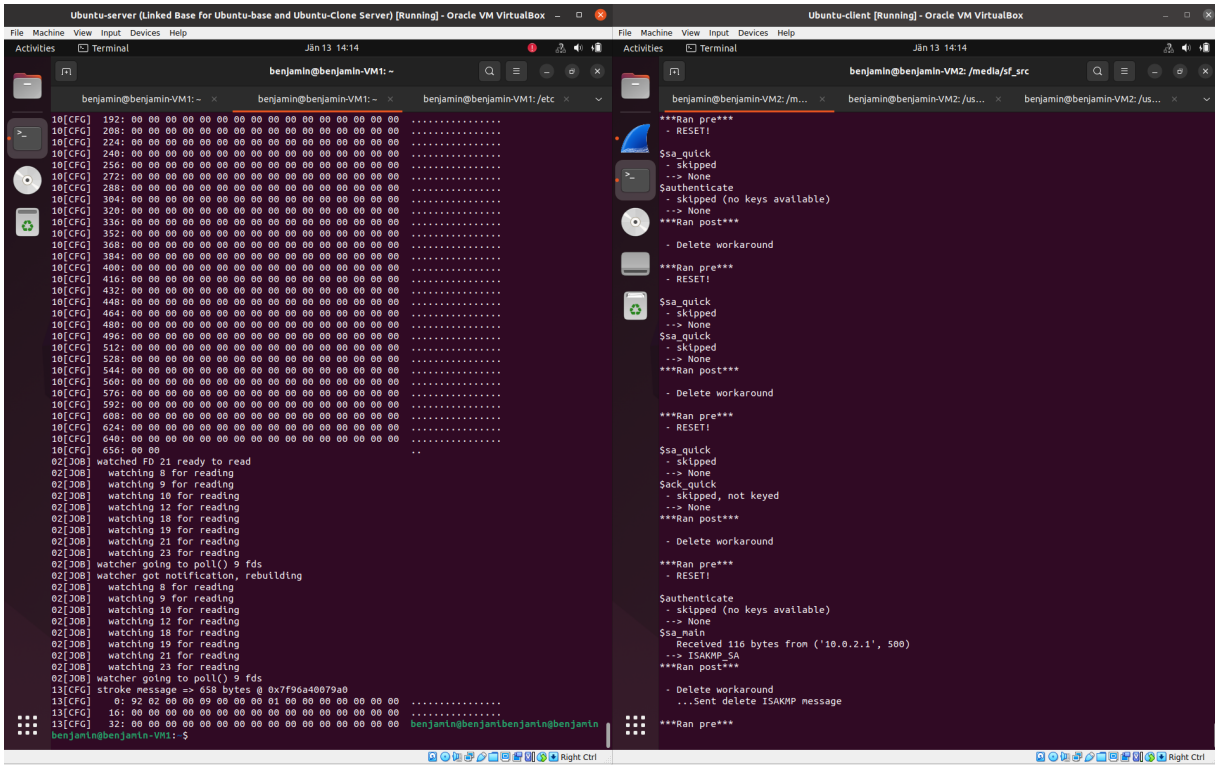

Figure 4.1: Learning setup, server left, learner right.

4.2 Combating Non-determinism

Despite all the precautions detailed in Section 4.1, the IPsec server still exhibited non-deterministic behavior, resulting in variance among the learned automata. While the majority of learned models were identical, the rest were significantly different, as can be seen when comparing Figure 4.2 to the more common ones shown in Chapter 6. To help decrease the remaining non-deterministic behavior, timeouts were added to all requests in order to give the server more time to correctly work through all incoming requests. This measure helped further decrease the amount of outlying automata learned, however it did not fully fix the issue. Examination of the outliers led to the discovery that all outlying behavior was concentrated around so-called retransmissions. Essentially, the IKE specification allows for previous messages to be retransmitted if deemed useful. A possible trigger could be the final message of an IKE exchange being skipped / lost. For example, if instead of an *AUTH* message, the server receives a phase two *IPSEC SA* message, the server would not know if it missed a message or if their was an error on the other parties side. According to the ISAKMP specification in RFC 2408 [Maughan et al. 1998], the handling of this situation is unspecified, however two possible methods are proposed. Firstly, if the *IPSEC SA* message can be verified somehow to be correct, the server may ignore the missing message and continue as is. Secondly, the server could retransmit the message prior to the missing one to force the other party to respond in kind. Strongswan appears to implement these retransmissions and due to internal timeouts of connections, seem to trigger in a not-quite-deterministic fashion. These retransmissions also explain the strange transitions seen in phase two of Figure 4.2.

While interesting for fingerprinting, we wanted a deterministic automaton as a base case for automata-based fuzzing, so we implemented checks in our mapper to allow the ignoring of retransmissions. The retransmission-filtering can be easily enabled or disabled through a simple flag and works by checking the message ID of incoming server responses against a list of previous message IDs (excluding zero, as it is the default message ID for phase one). If a repeated message ID is found, it is flagged as a retransmission and depending on the current filtering rule, ignored. With this addition, the IPsec server

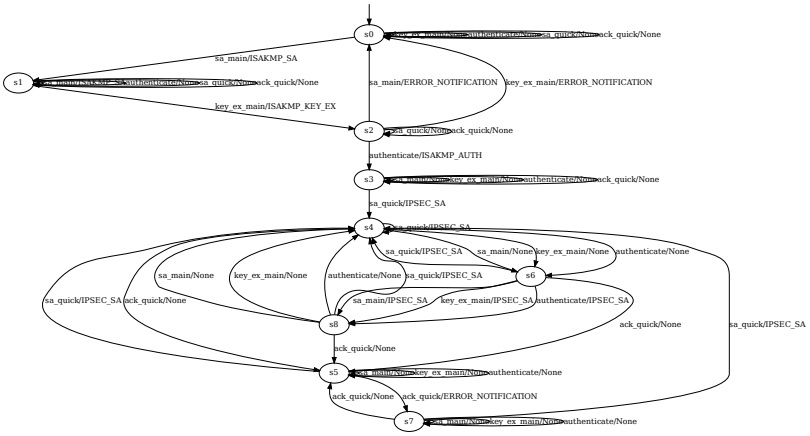

Figure 4.2: An example variant learned model.

became 100% deterministic, allowing the learning of a very clean automata, as shown in Figure 6.3. As an additional method of dealing with non-determinism but still keeping retransmissions, we can also catch non-determinism errors as they occur and repeat the offending query several times. If upon the first rerun the non-determinism does not occur again, we accept the existing value as the correct one and continue. If however, it persists for a set amount of repetitions with the same constant server response, we assume that the original saved response was incorrect and update it to the new one. With this non-determinism correcting added, the automata learning also works with no more non-determinism errors and the learned automata are consistent with one another.

4.3 Learning Setup

We used the Python automata learning library AALpy to learn our automata. It boasts support for deterministic, non-deterministic and stochastic automata, with support for various formalisms for each automata type. We chose deterministic Mealy machines to describe the IPsec server. However, learning automata with AALpy follows the same pattern, regardless of the type of automata.

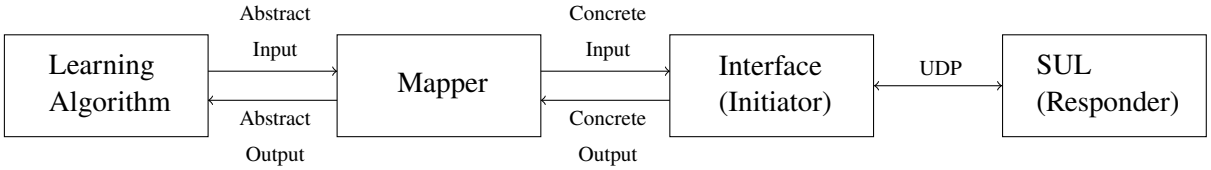

Figure 4.3: Automata Learning Setup

Figure 4.3 gives an overview of the learning process, adapted from Tappler et al. Tappler et al. 2017. To begin, the learning algorithm sends abstract inputs chosen from the input alphabet to the mapper class, which converts it to concrete inputs. The concrete inputs are then sent to the SUL, by means of a communication interface. In our case, the mapper class comprises the major portion of our work and converts the abstract words into actual IPsec packets that can be sent to the SUL Strongswan server via UDP packets. This separation between abstract and concrete in/outputs allows for easy future modifications to the message implementations, including fuzzing support, as well as increasing the readability of our code.

To begin learning an automaton with AALpy, we must first choose a suitable input alphabet encompassing the language known by the server, as well as the learning algorithm to be used. Our chosen input alphabet consists of the initiator-to-responder messages shown in Figure 3.3. We use both the L^* and *KV* algorithms for learning with a state prefix equivalence oracle that provides state-coverage by means of random walks started from each state. The equivalence oracle is used by the chosen learning algorithm to test for conformance between the current hypothesis and the SUL, giving either a counterexample on failure, or confirmation that we have learned the SUL. This corresponds to an equivalence query. We also enabled several optional AALpy features including caching and non-determinism checks to improve the learning process. An overview of relevant code can be seen in Listing 4.1. Line three shows the used input alphabet, line four the used equivalence oracle and line 5 the used learning algorithm. Both the equivalence oracle and learning algorithm take the input alphabet and an object representing the SUL as parameters, were the SUL is defined as shown below in 4.2. The equivalence oracle is then also passed as a parameter to the the learning algorithm which also has a few additional optional parameters specifying the type of automaton to learn and enabling non-determinism checking and caching.

```
1  # Code example detailing AAL with AALpy
2
3  input_al = ['sa_main', 'key_ex_main', 'authenticate', 'sa_quick', '
       ack_quick']
4  eq_oracle = StatePrefixEqOracle(input_al, sul, walks_per_state=10,
       walk_len=10)
5  learned_ipsec = run_Lstar(input_al, sul, eq_oracle=eq_oracle,
       automaton_type='mealy', cache_and_non_det_check=True)
```

Listing 4.1: Equivalence Query code

To learn the target server, we defined a *step* method and *reset* functionality as can be seen in Listing 4.2. We use *step*, seen in line three, to execute one input action from the current query and *reset*, lines eight through 12, to revert the SUL to an initial clean state. Used in combination, they correspond to membership queries, were *step* is used to query the SUL with one letter of the input alphabet and *reset* ensures identical starting conditions for each full query. *pre* is called before each membership query and *post* afterwards. The abstract input chosen from the input alphabet is passed on to the mapper class for further processing. We can see in line four, that a function, corresponding to the input, is called in the mapper class and the return value is passed on to the learning algorithm.

Our mapper class implements methods for each communication step in a typical IPsec-IKEv1 exchange, as described in Section 3. This includes the *ISAKMP SA*, *KE*, *AUTH*, *IPSEC SA*, *ACK* and *DELETE* messages. The *DELETE* message is special in that it actually sends two packets which is required to delete all existing connections to the Strongswan server. Also its behavior is highly dependent on the current state of the protocol. For these two reasons, it was mostly left out of the learning process. Furthermore, the mapper class contains a variety of helper functions used to handle the decryption and encryption of packets as well as parse received informational messages. To illustrate our mapper functions, (simplified) excerpts from *ISAKMP SA* function are shown in Listing 4.3. It shows how a Scapy packet is constructed out of many different individually configurable layers and fields, allowing for a high degree of flexibility and customizability. We can see in line five, how an ISAKMP transform is created, encompassing various security parameters. This is packed into a ISAKMP proposal packet first and then the resulting packet is packet into an ISAKMP SA packet in line six. The SA packet is appended to a generic top level ISAKMP packet in line eight. In line nine we then send the ISAKMP packet to the SUL and receive its response (if

```
# code excerpt from IPSEC_IKEv1_SUL.py

def step(self, letter):
  func = getattr(self.ipsec, letter)
  ret = func()
  return ret

def pre(self):
  self.ipsec.reset()

def post(self):
  self.ipsec.delete()
```

Listing 4.2: Membership Query code

any) back, already converted into a matching Scapy object. The response then undergoes a retransmission check and is then parsed and relevant data is used to update local values in lines 11 through 16.

```
# code excerpt from IPSEC_MAPPER.py

def sa_main(self, ...):
  ...
  tf = [('Encryption', 'AES-CBC'), ('KeyLength', 256), ('Hash', 'SHA'), (
      'GroupDesc', '1024MODPgr'), ('Authentication', 'PSK'), ('
      LifeDuration', 28800)]
  sa_body_init = ISAKMP_payload_SA(prop=ISAKMP_payload_Proposal(trans_nb
      =1, trans=ISAKMP_payload_Transform(num=1, transforms=tf)))

  policy_neg = ISAKMP(init_cookie=cookie_i, next_payload=1, exch_type=2)/
      sa_body_init
  resp = self._conn.send_recv_data(policy_neg)

  if (ret := self.get_retransmission(resp)):
    # retransmission handling
    ...

  # Response handling (checks response code, decrypts if necessary,
      updates relevant local values)
  ...
```

Listing 4.3: Excerpt of SA Main Mapper code

The IPsec packets generated by the mapper class are passed on to our communication class, which acts as an interface for the SUL and handles all incoming and outgoing UDP communication. Additionally, it parses responses from the SUL into valid Scapy packets and passes them on to the mapper class. The mapper class then parses the received Scapy packets and returns an abstract output code representing the received data to the learning algorithm. This code corresponds to the type of received message, or in the case of an error response (informational message), the error type. For fuzzing purposes, several common error types were grouped together into categories and the error category was used as the return value. Finally, the abstract error codes are returned to the learning algorithm which uses it update its internal

data structures and improve its understanding of the SUL.

4.4 Design Decisions and Problems

As messages will be sent in random order during learning, we require a robust framework that correctly handles en/decryption of messages. For key management, we simply store the current base-keys but keep track of initialization vector (IV)s on a per message-ID (M-ID) basis. Additionally, we keep track of the M-IDs of server responses to detect and handle retransmissions of old messages. Each request, we store the response for use in the next message and update affected key material as needed. Most notably, the IVs are updated almost every request and differ between M-IDs. Informational requests also handle their IVs separately. For each request that we send, if available, we try to parse the response, decrypting it if necessary and resetting or adjusting our internal variables as required to match the server. To keep track of all the different M-IDs, we use a Python dictionary to map M-ID to relevant keying and IV information. Usually, IVs are updated to the last encrypted block of the most recently sent or received message, though this behavior varies slightly between phases and for informational messages. Keeping track of IVs is required to continuously be able to parse encrypted server responses and extract meaningful information.

To ensure that we receive all responses, we add a timed wait for each server response, after which a None response is logged. In the case of no response, we can directly return None and need no further handling. Otherwise, we check the response M-ID against our list of previous M-IDs to detect retransmissions and then depending on the configured retransmission-handling rule, either ignore or use the corresponding previous response. To save some time when not ignoring retransmissions, we keep a dictionary mapping M-IDs to their parsed response codes, allowing us to skip the parsing stage fro retransmitted messages. If no retransmission is detected, we check that the message type matches the expected one and if so, parse the message further to update local values and extract a response code. If the message type does not match, it is usually an informational message, indicating some sort of error. In that case we decrypt the message using the corresponding parameters (as they are calculated and saved differently for informational messages), and return a code indicating the error being reported. Finally we catch and log unimplemented message types, but this case should not occur unless fuzzing breaks something.

We use the Python library Scapy to construct ISAKMP packets as required by the IKEv1 protocol. More exactly, we use the ISAKMP package that defines a generic top-level ISAKMP package as well as several more specific payloads that it can contain. Parsing was made more difficult by the fact that Scapy does not support / implement all the packets required by IPsec-IKEv1. To solve this problem, we implemented all missing packets in the Scapy ISAKMP class and used this modified version. Specifically, we added support for ISAKMP Informational packets, including resolving all commonly supported error codes, ISAKMP Delete packets, NAT-D, additional SA attributes for ISAKMP and ESP. Additionally we improved the ISAKMP Transform, Proposal and ID packets.

In addition to all the ISAKMP packets, Automata learning requires a SUL reset method to be able to return to an initial starting point after each query. We implement this using a combination of the ISAKMP *DELETE* request and general ISAKMP informational error messages. While *DELETE* works for established connections in phase two of IKE, we require informational error messages to trigger a reset in phase one, as delete does not work here sufficiently. Implementation was hindered at times by unclear RFC-specifications, but this was overcome by manually comparing packet dumps and Strongswan logs to fix encryption errors.

Each concrete mapping function in our mapper class can be run normally, with an injected error or with arbitrary values for the respective fields of the resulting packet. This allows us to learn different variations of the IPsec server. For example, our mapper class allows us to very easily switch between learning the

server model when presented with valid inputs, and the model of the server when introduced to invalid, malformed messages in combination with valid ones. Additionally, this design of the mapper functions will make fuzz testing specific protocol messages quite simple. The model of the server presented with malformed inputs will serve as the basis for future model based fuzz testing and can be seen in Chapter 6.

Since bugfixes and new features required a lot of testing and automata learning can be a very time-intensive process, we implemented several performance improvements to speed up the learning process. First we reduced the timeouts down to a minimum amount needed to still get deterministic results. Next we categorized the server informational responses according to their severity and impact and then grouped the most common ones together under the same abstract response code. This decreased the amount of states that had to be learned while at the cost of some information, but since any deviations or non-deterministic behavior would have been caught by the framework, we are confident that no important information was lost. Finally we switched out the L^* learning algorithm for KV, as KV can be more performative for learning environments where membership queries are expensive operations. As IKEv1 is a networks protocol with quite a bit of communication in each phase and we additionally have to implement small timeouts to wait for the server, each individual membership query can take several seconds. With hundreds of membership queries required to learn the IPsec server, this results in a lot of time spent running the algorithm. Consequently, any decrease to the amount of membership queries should, in theory, lead to an overall decrease in runtime. Since AALpy supports the KV, switching between the two learning algorithms is as easy as setting a simple flag as shown in line three of Listing 4.4 below. The KV algorithm, as expected, required less membership queries to learn the SUL and consequently significantly improved the speed at which models could be learned. The detailed comparison of runtime statistics between L^* and KV can be found in Chapter 6.

```
1  # code excerpt from IPSEC_IKEv1_SUL.py
2
3  if kv:
4    learned_ipsec, info = run_KV(input_al, sul, eq_oracle, automaton_type='
         mealy', cex_processing='rs')
5  else:
6    learned_ipsec, info = run_Lstar(input_al, sul, eq_oracle=eq_oracle,
         automaton_type='mealy', cache_and_non_det_check=True)
```

Listing 4.4: Switching Learning Algorithms

Chapter 5

Fuzzing

5.1 Environment Setup

We ran all our fuzzing tests in the same virtual network setup we used for our automata learning, on the same Ubuntu 22.04 LTS distributions. We again designated one VM as the initiator which would send the fuzzed messages and the other one as the responder to create a typical client-server setup. All settings on the used VMs remained the same as while learning to ensure that no discrepancies were introduced by different environment settings. The SUT was also the same Strongswan server used for learning. The major difference to learning is that for fuzzing, we no longer require AALpy. Our only real dependency, apart from our mapper class, is the Python fuzzing framework boofuzz, version 0.4.1, which we use for input generation.

5.1.1 Adapting the Model

While we had already successfully learned a (deterministic) model of the SUL when exposed to expected inputs, this proved to be not particularly useful for model-based fuzzing, as each fuzz case would be treated as new behavior. Instead, prior to fuzzing, we learn a new model, again with retransmission-filtering enabled, but this time also with an expanded input alphabet. In addition to the previous input alphabet letters, we add an erroneous version of each input letter that maps to an IKE packet with some sort of error or malformation, as shown in Listing 5.1. This doubles the size of our input alphabet. The goal is to learn the behavior of the SUL when exposed to typical errors or malformed packets that could arise during fuzzing, to be able to filter away as much uninteresting information as possible and focus on more unusual behavior. An example of such a malformed packet could be an incorrect length field, a wrong hash value or simply an unsupported SA option. Since we have designed our mapper class in such a way as to allow for easy manipulation of packets, this was an easy change to implement. Some additional server responses had to be parsed correctly, but all in all, not much had to be changed. The resulting model can be seen in Figure 6.4 and was used as our reference model while testing.

```
input_al = ['sa_main', 'key_ex_main', 'authenticate', 'sa_quick', '
    ack_quick', 'sa_main_err', 'key_ex_main_err', 'authenticate_err', '
    sa_quick_err', 'ack_quick_err']
```

Listing 5.1: The Updated Input Alphabet

5.1.2 Fuzzing Setup

As we had already designed our mapper class in such a way as to allow for easy fuzzing, the only thing missing was a source of values to use for fuzzing. For this purpose, we used the open source fuzzing library boofuzz[1], which is a successor of the popular Sulley[2] fuzzer. Boofuzz is usually used by first defining a protocol in terms of blocks and primitives that define the contents of each message, and then using those definitions to generate large amounts mutated values for testing. However, as we already had a very flexible way of sending manipulated IPsec packets, we decided to only use the data generation features of boofuzz, forgoing the protocol definitions. To get relevant fuzz values for each field, we mapped each fuzzable field to a specific boofuzz primitive data type and then used that to generate our data. We now have fuzz data generation, but are still lacking a way of choosing which runs to use for our tests, what exactly to fuzz and how to make use of our reference model.

The general procedure for model-based fuzzing is to fuzz the target system, while at the same time keeping track of the expected outputs on a reference model, to be able to identify new states and interesting behavior. To this end, we implemented a very simple parser for the .dot files of our learned model, as well as a converter between .dot files and state machines. Using this state machine, we can then simply see if a fuzzed input sent to the SUT results in the same next state and response as it does on our reference model, by simply following the edge correspondng to the input being fuzzed. If we notice a mismatch, we will have found a new state and hitherto unexplored behavior of the SUT. While our fuzzer does follow this general pattern, due to the IPsec-IKEv1 protocol being rather complicated, fuzzing every possible field for every possible combination and concatenation of inputs would be an immense task that goes far beyond the scope of this masters thesis. Instead, we implemented several techniques to limit the amount of fuzzing to be done in a way that aims to still maximize the chances of discovering new states and potential bugs.

Firstly, instead of fuzzing every possible field of the protocol, we instead chose 5-10 key fields from each packet that looked to be the most impactful and representative for that type of message. We focused on length fields, SA proposals and hashes / keys, but also added general fields, such as the responder / initiator cookies. All the chosen fields were added as parameters to their respective mapper class methods and default to their usual values. Packets can then have all or just some of their fields fuzzed, chosen randomly. Input data for chosen fields are then obtained from a boofuzz generator of a type matching the data of the field..

The next step was the run-generation phase in which we look to generate a set of runs consisting of input alphabet words, where one of the letters will be randomly chosen to be fuzzed. Our first idea was to go on random walks through the state machine and mirror the messages sent to the SUT as well, but the problem here at least for truely random walks was a lot of wasted queries in phase one and not enough state coverage in phase two. Therefore, since we had already generated a number of runs during model learning that guarantees state-coverage (at least for the learned automaton), we decided to repurpose those runs for fuzzing. The problem with this approach however, was that the resulting set of runs was rather large. So, in an effort to reduce the fuzzing space, we employed an additional filtering phase before the actual in-depth fuzzing. In this phase, we go through each run one by one and randomly designate one of its input alphabet letters as the fuzzing target. Then we test each fuzzable field of that packet in the context of the run with a greatly reduced set of fuzz values and compare the results to the expected outcomes using our state machine. If new behavior is found, the run and fuzz target passes the filtering. Runs in which no new behavior is discovered are discarded. This allows us to focus our resources on testing those configurations in which it is more likely to discover new behavior and therefore also bugs. And since many of the runs are very similar, this decreases the chance of discarding interesting runs, as chances are good that a run that contains at least a relevant subset of the discarded run will pass.

[1] https://github.com/jtpereyda/boofuzz
[2] https://github.com/OpenRCE/sulley

Following the automatic filtering, we go over the results and manually check and remove / reduce the number of identical or not relevant cases. For example, we noticed that every run in which cookies were fuzzed led to new states, due to new cookies indicating a completely new connection and since we learned the model with static initiator cookies, this will always lead to a new state. Finally, following the automatic and manual filtering, we arrived at a list of 175 runs, compared to roughly ten times the number before filtering. The filtered list of runs can be found in Appendix and contains all the discovered runs APPENDI that exhibited new behavior. Instead of our approach using existing runs and filtering, we also developed a more promising method of run generation using scored mutation that is described in Subsection 5.1.3.

Diagram for fuzzing process

5.1.3 Mutation-based Testcase Generation

While the run generation method described above does work, in practice it is too slow without the added filtering stages. However, even with the added filtering stages, fuzzing the remaining 175 runs still took well over several days. As an alternative, we developed the following run generation method, which results in only a single run to be tested. The goal is to generate a run which has the highest possible chance of reaching the largest amount of interesting states. To this end, we designed a method for scored, mutation-based testcase generation.

In its simplest form, mutation-based testcase generation refers to the generation of testcases, in our case runs, by repeatedly mutating a base case. In other words, we generate testcases by repeatedly applying small changes to a base testcase. These small changes, or mutations, could for example be swapping a bit, or changing a letter. In our fuzzer, we implemented two mutation operations. Adding a new letter of the input alphabet and swapping an existing letter in the run with its opposite version (so an erroneous version becomes a valid one, and vice versa). For our mutation base we can either use a random input alphabet letter, or pass a run to be used. We added this second option since IPsec phase one packets have to be sent in a specific order to successfully authenticate. Remembering back to our attempt to use fully random runs for fuzzing, we know that phase two is explored far less than phase one. Therefore, we can start the mutations with phase one already completed if we so desire.

At this point, we had implemented very basic mutation-based testcase generation, but the runs generated were often of low interest. As our goal is to generate as meaningful a run as possible, we decided to score the generated mutations and only keep a mutation, if its score is higher than its predecessor. Intuitively, the score given should serve as a representation of how easy it is to find new states while fuzzing a given run. To calculate the score of a mutated run, we test each fuzzable field of every letter in the run using the minimal list of fuzzing values used previously for our initial filtering phase and record the amount of new states discovered. To calculate the final score, we divide the total sum of new states by the length of the run. While the scoring does take some time, as we have to test at least five fields per letter, and we test every letter in the run compared to a randomly chosen one before, it is still much faster than the initial filtering phase provided the length of the mutated run does not come close to the number of runs need to learn the model. However, we do need to run this scoring step after every mutation, so for long mutations (>30 mutations) the total runtime can exceed that of the previous filtering step. The key difference is, that after the mutations are completed, the result will be a single run that is more interesting than any previous generated run, as opposed to the 175 runs after filtering. Finally, we weighted the mutation operations to make adding a new letter at the end of the word more likely than at a random index, as well as increasing the likelihood of the last letter being flipped over random letters. These changes try to decrease the chances of wasting time changing existing interesting configurations instead of adding to them, but still leaves the possibility given enough mutations. An example mutation in which prependeding the letter sa_main to the existing run leads to a significant score increase can be seen in Listing 5.2. The most interesting run found within 20 mutations is shown in line seven.

Comparing the two methods, the filtering one takes far longer, but does explore a much higher number

of states. In contrast, the mutation-based approach is much faster, and focuses on a small sample of states, but generates that sample to be as interesting as possible in regards to the amount of new states that can be discovered through it.

These are not final results and still have the mentioned error, will update once I have the correct results

```
1 Mutation: 11, score: 82.0
2 ['sa_main', 'key_ex_main_err', 'key_ex_main', 'sa_main', 'authenticate', '
      sa_main', 'sa_quick', 'sa_main_err', 'authenticate']
3
4 Mutation: 12, score: 115.5
5 ['sa_main', 'sa_main', 'key_ex_main_err', 'key_ex_main', 'sa_main', '
      authenticate', 'sa_main', 'sa_quick', 'sa_main_err', 'authenticate']
6 ...
7 Mutation: 16, score: 130.66666666666666
8 ['sa_main', 'sa_main', 'key_ex_main_err', 'key_ex_main', 'sa_main', '
      authenticate', 'sa_main', 'sa_quick', 'sa_main_err', 'ack_quick_err', '
      sa_main_err', 'authenticate']
```

Listing 5.2: Mutations

Chapter 6

Evaluation

In this chapter, we present the results of our model learning and model-based fuzzing. We begin with a discussion and comparison of the various learned models in Section 6.1 and then move on to presenting the results of our model-based fuzzing in Section 6.2.

6.1 Learning Results

rewrite / expand on this portion, detail exact configurations / should I detail it again if I already did so in methods?

Over the course of our work, we learned a wide variety of different models. In the following, we present the four most relevant ones, all learned from a Linux Strongswan U5.9.5 server, using both the KV and L^* learning algorithms. Error codes have been simplified for better readability. As our SUL had some issues with non-determinism while retransmissions were enabled, one major differentiating factor in our models is whether retransmission-filtering was enabled for the learning process. This had a significant impact on the resulting learned model, with the version without filtering boasting more than twice the number of states than the one with. Additionally, even when using the methods to combat non-determinism described in Chapter 4, Section 4.2 the resulting models still occasionally differed when not filtering out retransmissions. Therefore, while still suitable for fingerprinting, the non-filtered models were not used for fuzzing, as we desired a completely deterministic model to serve as our baseline.

6.1.1 Learned Models

Figures 6.1 and 6.2 show the two most commonly learned models when not filtering retransmissions. Roughly than 80% of all learning attempts without filtering resulted in one of these two models, which we will refer to as the common models, with the other 20% being a non-uniform assortment of outliers, an example of which can be seen in Figure 6.2.

The first common model, presented in Figure 6.1 took approximately 52 minutes (3092 seconds) to learn with the KV algorithm, spread over seven learning rounds, and consists of 10 states. Of those 52 minutes, roughly half were used for state exploration / membership queries and the other half for conformance checking, with conformance checking taking slightly longer (1501 vs 1591 seconds). 171 membership queries were performed by the learning algorithm in 2047 steps, whereas 100 equivalence queries were performed for conformance checking in 1826 steps. In contrast, when learned with the L^* algorithm, model learning took almost 85 minutes (5094 seconds) over five learning rounds. Here, the split between state exploration and conformance checking was more distinct, with state exploration taking up approximately 68% of the total runtime and conformance checking only requiring the remaining 32% (3489 vs 1605 seconds). Notably, the time needed for conformance checking remained largely the same

between the two algorithms, however the difference in state exploration / membership queries is quite large. We discuss this behavior in more detail including a statistical comparison of the two algorithms in Subsection 6.1.2.

Moving on to an examination of the model itself, we can clearly see a separation between the two phases. Phase one completes in state *S3*, and phase two begins right thereafter. While phase one looks very clean and is in fact identical to the model learned with retransmission-filtering enabled, phase two has many strange transitions caused by retransmissions. For example, the transition from state *S5* to *S7* via *sa_main* returns a valid *IPSEC SA* response. This should be impossible, as phase one messages are ignored while in phase two. However, due to specific timings of retransmissions, this phase one input results happens to be listening for a response when the SUL sends a retransmission for previous *sa_quick* message. Next we can see a transition throughout states *S6* to *S11*. In these states, we can see that phase one messages, which are usually ignored in phase two, result in *IPSEC SA* responses. This behavior is caused by retransmissions being treated as responses to phase one messages sent in phase two, which are in fact ignored by the server. Another noticeable property of the learned automata, is that past state *S2*, no paths lead back to the initial state. This is due to the fact that we did not include the delete command in the alphabet for this learned model. Adding delete adds transitions from every state back to the initial one, but also dramatically increases the runtime and non-deterministic behavior of the SUL, as even more retransmissions are triggered. While not part of our input alphabet, it could be included in future work.

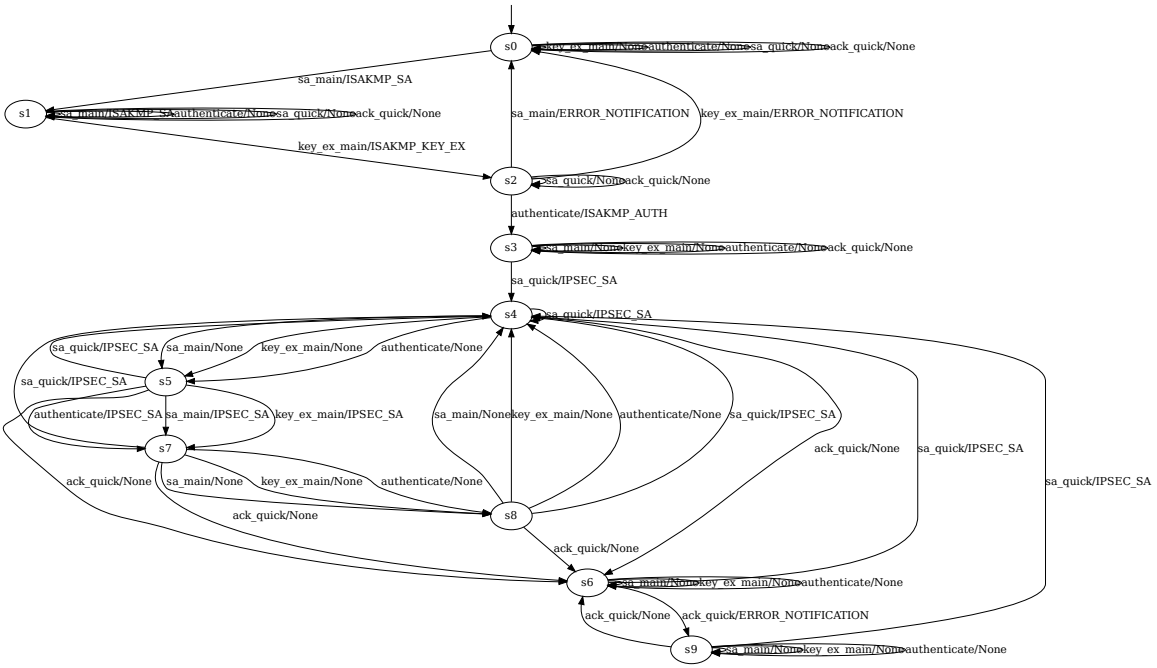

Figure 6.1: First commonly learned model with retransmissions.

The second common model, seen in Figure 6.2, took approximately 75 minutes (4507 seconds) to learn using the *KV* algorithm. The model took nine rounds to learn, and consists of 12 states. Of those 75 minutes, roughly 53$ were used for state exploration / membership queries and the other 47% (2382 vs 2126 seconds). 215 membership queries were performed by the learning algorithm in 2219 steps, whereas 120 equivalence queries were performed for conformance checking in 1964 steps. In contrast, when learned with the L^* algorithm, model learning took significantly longer, running for 125 minutes

(7520 seconds) over five learning rounds. Here, the split between state exploration and conformance checking was again very distinct, with state exploration taking up approximately 71% of the total runtime and conformance checking only requiring the remaining 29% (5393 vs 2126 seconds). Again, the time needed for conformance checking remained largely the same between the two algorithms, however the difference in state exploration / membership queries is even larger.

Examining the model, we can again see a clear separation between the two phases. Phase one for this model is identical to the previous one. Phase two shows a larger number of retransmission-induced strange behavior over the states *S5*, *S7*, *S8*, *S10*, *S11* and *S9*. Again we have phase one inputs, such as *sa_main*, resulting in the valid phase two output, *IPSEC SA*. This behavior is again triggered by specific timings of retransmissions. Same as in Figure 6.1, no paths past state *S2* lead back to the initial state.

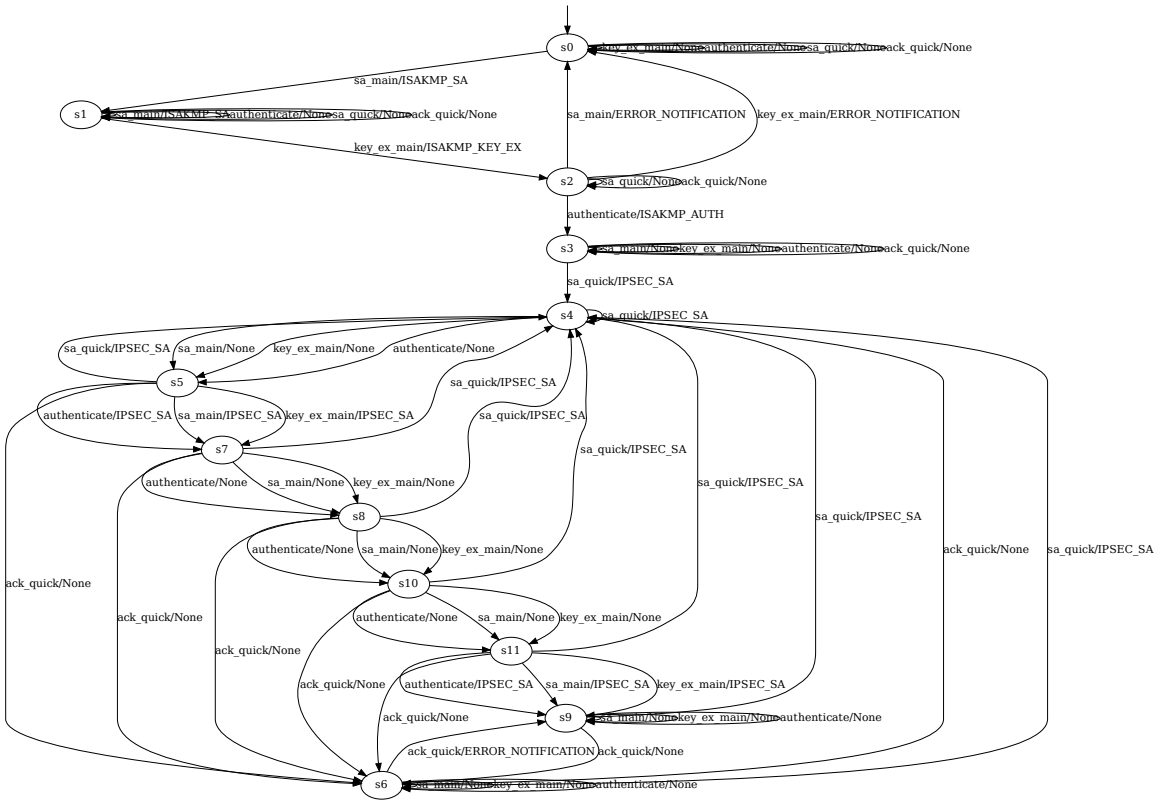

Figure 6.2: Second commonly learned model with retransmissions.

In comparison, when learning the same server using retransmission-filtering, all non-deterministic behavior vanishes and we get the model shown in Figure 6.3 every learning attempt. The model has only 6 states and therefore was learned much more quickly than the previous ones, with learning requiring only approximately 21 minutes (1266 seconds) using the *KV* algorithm. Learning happened over four rounds, where the time was distributed between state exploration and conformance checking in a 60-40 split (519 vs 747 seconds). In comparison, when learned with the L^* algorithm, learning took roughly 36 minutes (2157 seconds), spread over two learning rounds. Of that time, state exploration required roughly 55% compared to the 45% needed for conformance checking (1188 vs 969 seconds). Compared to *KV*, state exploration / membership queries took more than twice the amount of time to complete.

Looking at the resulting model more closely, we can see that the first four states are again identical to the previous model. This is due to the fact that the retransmissions only triggered for phase two messages

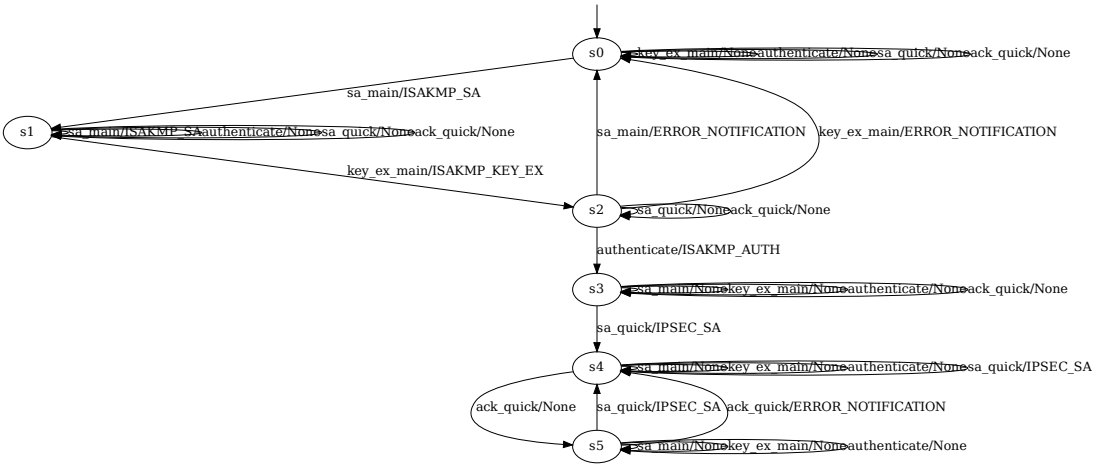

Figure 6.3: Clean model learned using retransmission filtering

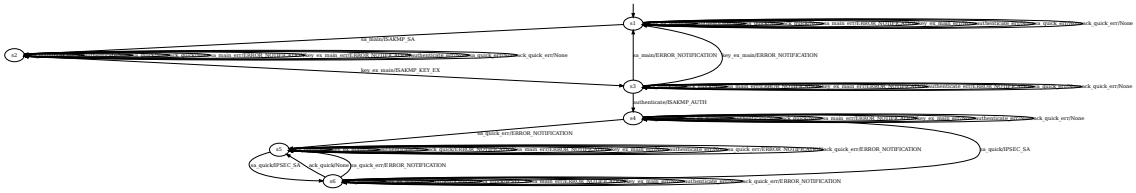

Figure 6.4: Model with malformed messages

and since they are our only source of non-determinism, we see no differences here. However, the phase two states look wildly different, showing a streamlined behavior that fits our reference IKE exchange (see Figure 3.3) almost perfectly. The only small difference lies in the additional state *S5* which loops back to state *S4* with an *IPSEC SA* or *ACK* message. This behavior demonstrates how one can create multiple IPsec SAs on a single IKE SA channel. And once a new IPsec SA has been established, we can again send another *ACK* message. In other words, the extra state is there to show that we cannot acknowledge a single IPsec SA twice, but need to first create a new one.

TODO: analyze error model Figure 6.4 shows a model learned with retransmission-filtering enabled. Additionally, the input alphabet was expanded to include an additional erroneous version of each letter that maps to a malformed IKE message. This model served as the basis for our model-based fuzzing and was, thanks to the retransmission-filtering, 100% deterministic. It took a total of

TODO: rerun learning with Lstar alg, as runtime seems sus

The model looks largely identical to the previous model, apart from some additional self-transitions and one additional error transition from state *S4* to *S5*. Here, state *S4* corresponds to the previous *S6*. The error transition simply means that we need to create a valid IPsec SA before we can acknowledge it.

Learning Algorithm Performance (Averages)		
Metric	**L***	**KV**
Learning Rounds	2	4
Total Time (s)	3036	2296
Time Learning Algorithm (s)	**1624**	**879**
Time Equivalence Checks (s)	1412	1417
Learning Membership Queries	**177**	**79**
Learning Steps	867	753
Equivalence Oracle Queries	60	60
Equivalence Oracle Steps	748	991
Membership Queries Saved by Caching	14	27

Table 6.1: Comparison L^* and KV

6.1.2 Comparing KV and L^*

Table 6.1 shows average performance statistics over five learning runs each, with retransmission-filtering enabled. The same hardware and software configurations were used as described in Chapter 4.1 with the learning program set up on a VirtualBox 6.1 VM allotted 4GB of memory and one CPU core. We used all the basic packets for our input alphabet, so *sa_main*, *key_ex_main*, *authenticate*, *sa_quick* and *ack_quick*. The model learned is the clean model seen in Figure 6.3. Table 6.1 shows the metric on the left and the respective averages for the L^* and KV learning algorithms respectively on the right. Interesting results are highlighted in bold. From top to bottom, the metrics measured are as follows. Learning rounds refers to the number of rounds the learning algorithms had to run for, or in other words, how many attempts they needed to correctly learn the SUL. Total time is the total time needed by the algorithm from start to the finished model. The total time can be split into time spent on the learning algorithm and time spent on equivalence queries. Learning membership queries refers to the number of membership queries sent to the SUL while learning steps to the steps in the learning algorithm itself. Analogously, equivalence oracle queries refers to the equivalence queries sent to the SUL and equivalence oracle steps to the steps needed by the equivalence oracle implementation. Finally, membership queries saved by caching details the performance boost gained by caching membership queries, with the value indicating the number of queries saved.

As the only difference between the two configurations tested was the choice of learning algorithm, intuitively we expect relevant fields to vary the most with equivalence oracle field to be largely unchanged. This intuition is confirmed by our experiments, wherein while the time spent on equivalence queries was very similar, the time spent on membership queries differed greatly. The L^* algorithm required more than double the number of membership queries than its KV counterpart. As membership queries are the main performance bottleneck in our setup, this change of course led to a significantly better runtime for KV, with total time spent on the learning algorithm being close to half that of the L^* algorithm. This difference in time spent on the learning algorithm meant, that for this experiment, the KV algorithm learned a model in roughly 75% of the time needed by the L^* algorithm. Looking only at the learning algorithm, KV performed roughly twice as well as its counterpart.

Little variance was observed throughout previous learning attempts so this small sample size is believed to be representative. However, for even more accurate results the experiment should be carried out again for more runs.

Redo with
20 runs. A
standard d
such statis

6.1.3 Library Error

Another notable finding from the model learning phase, was the discovery of a bug in a used Python Diffie-Hellman key exchange library. The bug was only found thanks to the exhaustive number of packets sent with our mapper class and due to the non-determinism checks implemented in AALpy. Despite our best efforts in removing the non-deterministic behavior from our learning process, we would still get occasional non-determinism errors at random points while learning. This problem persisted over several weeks due to the fact that the errors occurred randomly and only sporadically during some learning attempts. Initially we believed this to be also caused by retransmissions, but since the problems persisted even after introducing retransmission-filtering, that possibility was ruled out. The other option was of course problems in our implementation of the IPsec protocol. Therefore, a lot of time was invested into painstakingly comparing logs and packet captures between our implementation and the SUL to ensure that everything lined up, since AALpy was still reporting non-determinism errors. Finally we discovered a discrepancy between the two and through it, that the problems were not in fact caused by our implementation, but by a used Python library. It turns out there was a very niche bug in a used Diffie-Hellman Python library where, if the most significant byte was a zero, it would be omitted from the response, causing the local result to be one byte shorter than the value calculated by the SUL. As this would only occur in the rare case where the MSB of the DH exchange was zero, this explains the random and difficult to reproduce nature of the bug. This behavior was undocumented and happened in a function call that allowed specifying the length of the returned key. As the library is not a very widespread one, the impact of this bug is presumably not very high. Regardless, it could compromise the security of affected systems and therefore the maintainer of the library has been notified of the problem. Due to the elusive nature of this bug, it would very likely not have been noticed without the exhaustive communication done by the model learning process and without seeing the slight differences in the resulting models that did not crash during the learning process.

6.2 Fuzzing Results

Fuzzing results

Compare mutation based fuzzing and filtering results / runtimes

Chapter 7

Conclusion

test

Bibliography

Abhijith, M and K Senthilvadivu [2020]. *Impact Of VPN Technology On It Industry During Covid-19 Pandemic*. IJEAST. 2020 (cited on page 1).

Andrews, Keith [2021]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. 10 Nov 2021. https://ftp.isds.tugraz.at/pub/keith/thesis/ (cited on page xi).

Angluin, Dana [1987a]. *Learning regular sets from queries and counterexamples*. Information and computation 75.2 (1987), pages 87–106 (cited on pages 1, 3).

Angluin, Dana [1987b]. *Learning regular sets from queries and counterexamples*. Information and Computation 75.2 (1987), pages 87–106. ISSN 0890-5401. doi:https://doi.org/10.1016/0890-5401(87)90052-6. https://www.sciencedirect.com/science/article/pii/0890540187900526 (cited on page 5).

Barker, Elaine, Quynh Dang, Sheila Frankel, Karen Scarfone, and Paul Wouters [no date]. *Guide to IPsec VPNs*. en. doi:https://doi.org/10.6028/NIST.SP.800-77r1 (cited on page 1).

Daniel, Lesly-Ann, Erik Poll, and Joeri de Ruiter [2018]. *Inferring OpenVPN state machines using protocol state fuzzing*. 2018 IEEE European Symposium On Security And Privacy Workshops (Euros&PW). IEEE. 2018, pages 11–19 (cited on page 3).

Ferguson, Niels and Bruce Schneier [1999]. *A cryptographic evaluation of IPsec* (1999) (cited on page 11).

Ferguson, Niels and Bruce Schneier [2021]. *The best VPN protocols* (2021). https://nordvpn.com/de/blog/protocols/ (cited on page 1).

Fiterău-Broştean, Paul, Ramon Janssen, and Frits Vaandrager [2016]. *Combining model learning and model checking to analyze TCP implementations*. International Conference on Computer Aided Verification. Springer. 2016, pages 454–471 (cited on page 3).

Fiterău-Broştean, Paul, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg [2017]. *Model learning and model checking of SSH implementations*. Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. 2017, pages 142–151 (cited on page 3).

GmbH, AVM Computersysteme Vertriebs [2022]. *Connecting the FRITZ!Box with a company's VPN*. https://en.avm.de/service/vpn/tips-tricks/connecting-the-fritzbox-with-a-companys-vpn/. 2022 (cited on pages 1, 11).

Guo, Jiaxing, Chunxiang Gu, Xi Chen, and Fushan Wei [2019]. *Model learning and model checking of IPsec implementations for Internet of Things*. IEEE Access 7 (2019), pages 171322–171332 (cited on pages 1, 3).

Isberner, Malte, Falk Howar, and Bernhard Steffen [2014]. *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning*. International Conference on Runtime Verification. 2014, pages 307–322 (cited on page 3).

Kaufman, Charlie, Paul Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen [2014]. *Internet key exchange protocol version 2 (IKEv2)*. Technical report. 2014 (cited on page 11).

Kearns, Michael J and Umesh Vazirani [1994]. *An introduction to computational learning theory*. MIT press, 1994 (cited on pages 5–6).

Maughan, Schertler, Schneider, and Turner [1998]. *Internet Security Association and Key Management Protocol*. RFC 2408. 1998 (cited on page 14).

Muškardin, Edi, Bernhard K Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler [2022]. *AALpy: an active automata learning library*. Innovations in Systems and Software Engineering (2022), pages 1–10 (cited on pages 2, 5).

Niese, Oliver [2003]. *An integrated approach to testing complex systems*. 2003 (cited on page 5).

Novickis, Tomas, Erik Poll, and Kadir Altan [2016]. *Protocol state fuzzing of an OpenVPN*. PhD Thesis. PhD thesis. MS thesis, Fac. Sci. Master Kerckhoffs Comput. Secur., Radboud Univ, 2016 (cited on page 3).

Pereyda, Joshua [2022]. *boofuzz Documentation*. `https://boofuzz.readthedocs.io/`. 2022 (cited on page 8).

Pferscher, Andrea and Bernhard K Aichernig [2021]. *Fingerprinting Bluetooth Low Energy devices via active automata learning*. International Symposium on Formal Methods. Springer. 2021, pages 524–542 (cited on pages 1, 3).

Pferscher, Andrea and Bernhard K Aichernig [2022]. *Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning*. NASA Formal Methods Symposium. Springer. 2022, pages 373–392 (cited on page 1).

Rivest, Ronald L. and Robert E. Schapire [1993]. *Inference of Finite Automata Using Homing Sequences*. Information & Computation 103.103 (1993), pages 51–73 (cited on pages 3, 5).

Steffen, Bernhard, Falk Howar, and Maik Merten [2011]. *Introduction to Active Automata Learning from a Practical Perspective*. Jun 2011, pages 256–296. ISBN 978-3-642-21454-7. doi:10.1007/978-3-642-21455-4_8 (cited on page 3).

Tappler, Martin, Bernhard K. Aichernig, and Roderick Bloem [2017]. *Model-Based Testing IoT Communication via Active Automata Learning*. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). 2017, pages 276–287. doi:10.1109/ICST.2017.32 (cited on page 15).

Zalewski [2020]. *american fuzzy lop*. `https://github.com/google/AFL`. 2020 (cited on page 8).