Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol

Benjamin Wunderling

Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol

Benjamin Wunderling BSc

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 endeavor. 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 [2].

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 [10]. This makes finding vulnerabilities in VPN software more critical than ever. IPsec is a popular VPN protocol suite 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 [7] and IKEv2 [9], with IKEv2 being the newer and recommended version, according to a report by the National Institute of Standards and Technology [5]. 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 [16]. Additionally, IKEv1 is also used for the L2TP/IPsec protocol, one of the most popular VPN protocols according to NordVPN [12]. 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

Behavioral models are a useful tool in testing and verifying the correctness of complex systems. A model simulating the behavior of a system gives an abstract high-level overview of its inner workings, allowing for much easier understanding than code review. Additionally, the model can be used e.g. to automatically generate test cases, or to fingerprint / detect specific software implementations [29, 30]. Despite their usefulness, the availability of accurate models may be limited for the several reasons. Firstly, the manual creation of an accurate behavioral model of a complex system can be a tedious and error-prone process. Secondly, the model must be updated every time the system is changed, i.e. functionality is added or changed in any way. Automata learning has proven itself as a useful technique for automatically generating behavioral models of various communication protocols, e.g. Bluetooth Low Energy, TLS, SSL or MQTT [TODO: CITATIONS!] .

Active automata learning (AAL) is an automata learning technique in which a behavioral model is generated by actively querying a system to gain information about it. Notable examples of AAL algorithms include the L^* algorithm by Angluin [3] and the KV algorithm by Kearns and Vazirani [21].

AAL is of particular interest for security testers, as it can also be applied to systems where knowledge about its inner workings is lacking, e.g. due to its implementation code being closed-source. We call such systems black-box systems. It is not unusual for VPNs and other security-critical software to be closed-source and therefore a black-box system for testers. Fortunately, AAL allows for the creation of a behavioral model of a black-box system, which can then be used for model-based testing approaches.

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. [17] 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 AAL framework AALpy [25] with a custom mapper to learn the state machines of various IPsec-IKEv1 server implementations. We then further utilized the learned models for model-based fuzzing, creating a custom fuzzing framework supporting multiple types of input-sequence generation.

1.3 Structure

This thesis is structured as follows. Chapter 2 gives an overview of the related 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 5. In Chapter 7, learned models and the results of the fuzzing tests are showcased and analyzed. Finally, Chapter 8 summarizes the thesis and discusses future work.

Chapter 2

Related Work

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.

Model learning is a popular tool for creating behavioral models of network and communication protocols. The learned models showcase the behavior of the system under learning (SUL) and can be analyzed to find differences between implementation and specification. Furthermore, the learned models can be used for additional security testing techniques, e.g., for model checking or model-based testing.

Model learning has been applied to a variety of different protocols, including many security-critical ones. De Ruiter and Poll [32] automatically and systematically analyzed TLS implementations by using the random inputs sent during the model learning process to test the SUL for unexpected and dangerous behavior. The unexpected behavior then had to be manually examined for impact and exploitability. Tappler et al. [39] similarly analyzed various MQTT broker/server implementations, finding several specification violations and faulty behavior. Furthermore, the 802.11 4-Way Handshake of Wi-Fi was analyzed by Stone et al. [37] using automata learning to test implementations on various routers, finding servery vulnerabilities. Fiterau and Brostean combined model learning with the related field of model checking, in which an abstract model is checked for specified properties to ensure correctness. In their work they learned and analyzed both TCP [14] and SSL [15] implementations, showcasing several implementation deviations from their respective RFC specifications. The Bluetooth Low Energy (BLE) protocol was investigated by Pferscher and Aichernig [29]. In addition to finding several behavioral differences between BLE devices, they were able to distinguish the individual devices based on the learned models, essentially allowing the identification of hardware, based on the learned model (i.e. fingerprinting).

Specifically within the domain of VPNs, Novickis et al. [27] and Daniel et al. [8] learned models of the OpenVPN protocol and showcased how to use the learned models to perform protocol fuzzing. In contrast to our approach, they chose to learn a more abstract model of the entire OpenVPN session, where details about the key exchange were abstracted in the learned model.

Even more closely related to our work, Guo et al. [17] used automata learning to learn and test the IPsec-IKEv2 protocol setup to use certificate-based authentication. They used the LearnLib [19] library for automata learning and performed model checking of the protocol, using the learned state machine. In contrast, the predecessor to IPsec-IKEv2, IPsec-IKEv1, differs greatly on a packet level, with IKEv1 needing more than twice the amount of packets to establish a connection than IKEv2 and also being far more complex to set up. Guo et al. highlight the complexity of IKEv1 repeatedly in their work, which emphasizes the need to also test the older version of the protocol as well, especially seeing as it is still in widespread use today [16].

IPsec-IKEv1 is frequently fuzzed, however until now, without employing learning-based testing methods. For example, Yang et al. built a custom mutation-based fuzzer for the IKEv1 protocol, focusing

on known vulnerabilities of the protocol [42]. Tsankov et al. discovered an unknown IKE vulnerability through the use of a semi-valid input coverage fuzzer, focusing on inputs where all but one constraint are fulfilled. Additionally, many popular IPsec libraries utilize OSS-Fuzz [34], ensuring that they are regularly fuzzed using the LibFuzzer, AFL++ and Hongfuzz fuzzing libraries [33, 13, 1].

In contrast, while our work also focuses on the IPsec-IKEv1 protocol, we approach it as a black-box system, using model learning to extract a behavioral model of the SUL and using that model for model-based fuzzing. We use the learned models for model-based fuzzing, employing search-based and genetic fuzzing techniques to further optimize the fuzzing process. Zeller et al. describe these fuzzing techniques and more in their comprehensive book on fuzzing [44]. In doing so, together with the fuzzer by Yang et al., our work completes the coverage of learning-based testing approaches for both IKE versions.

Chapter 3

Preliminaries

3.1 Mealy Machines

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 a corresponding output action. More formally, a Mealy machine is defined as a 6-tuple $M = \{S, s_0, \Sigma, O, \delta, \lambda\}$, where S is a finite set of states, $s_0 \in S$ is the initial state, Σ is a finite set called the input alphabet, O is a finite set called the output alphabet, δ is the state-transition function $\delta\colon S \times \Sigma \to S$ 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 $\lambda\colon S \times \Sigma \to O$ 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, models are learned based on a given data set describing the behavior of the SUL, e.g. log files. In contrast, in 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" [4]. In this seminal paper, Angluin introduced the concept of the minimally adequate teacher (MAT) as well as an algorithm for learning regular languages from queries and counterexamples, called L^*. Note, that while the L^* algorithm was originally designed to learn deterministic finite automata (DFA) of regular languages, it can be simply extended to work for Mealy machines by making use of the similarities between DFA and Mealy machines [18, 35]. As Mealy machines are our chosen formalism for modeling learned reactive systems, we will assume throughout this paper that we employ the Mealy machine variants of all mentioned learning algorithms.

3.2.1 MAT Framework

L^* uses the MAT framework to learn a Mealy machine representation of the behavior of an unknown system. To this end, the MAT framework defines both a learner and a teacher. The teacher must respond to two types of queries posed by the learner, namely membership and equivalence queries. On a conceptual level, the learner poses membership queries to the teacher in order to build a model of the SUL and equivalence queries, to verify if the model accurately describes the behavior of the system. Membership

queries consist of an input $s \in \Sigma^*$, where Σ is the input alphabet. The teacher receives the membership query and executes it on the SUL, returning the response of the SUL to the learner. In other words, using the Mealy machine definition from before, membership queries are used to learn the mapping between inputs $s \in \Sigma^*$ to an outputs $o \in O$. Once enough information has been gathered using membership queries, the learner constructs a Mealy machine representation of the SUL Mprop and proposes it to the teacher as an equivalence query. The teacher must confirm if Mprop is equivalent to the SUL, answering with "yes" if the equivalence holds true, or else returning a Mealy machine counterexample c, proving their non-equivalence. In other words, equivalence queries are used to verify if the learner has successfully learned the Mealy machine representation of the SUL or if not, return a counterexample detailing the differences. If a counterexample is returned by the teacher, the learner uses this to update its model to include the new information and starts the cycle anew. This cycle of repeatedly gathering information through membership queries and proposing a model through an equivalence query until it is confirmed to be correct is showcased in Listing 3.1 below. We can see that the algorithm does not end until the teacher confirms the equivalence in line 10.

```
1  Initialization
2
3  repeat:
4    Membership query
5
6    if: Sufficient information has been gathered through membership queries
7      Construct Mealy Machine proposal Lprop.
8      Equivalence query(Lprop)
9
10     if: Query returns "yes" then
11       return Lprop
12     else:
13       Parse counterexample
14     fi
15   fi
16 end
```

Listing 3.1: MAT algorithm

3.2.2 L*

L^* uses the MAT framework and stores the results of the membership queries in a data structure called the observation table. Once the observation table has been filled with enough information on the SUL, an equivalence query is performed. If successful, the algorithm terminates, otherwise, the observation table is expanded to include the information learned from the received counterexample. The observation table has two notable properties that must hold before an equivalence query can be constructed, namely closedness and consistency.

Closed implies that the table includes all possible combinations of inputs relevant to the current candidate states (for the next proposed Mealy machine) in the observation table. Or in other words, that for each candidate state and each $s \in \Sigma^*$ there is a clearly defined next state in the transition function δ of the learned Mealy machine.

Consistent means, that appending the same input to identical states in the observation table should not result in different outcomes. In other words, if two states in the same equivalence class produce different outputs when appended with the same input, the table is inconsistent, and needs to be fixed.

If either of these is properties violated, the table must be updated and filled through further membership queries in order to bring the table back into a closed and consistent state.

Variants of the L^* algorithm are still used for learning deterministic automata to this day, e.g., by the AAL python library AALpy [26]. While many modern implementations, including AALpy use improved versions of L^*, such as with advanced counter example processing by Rivest and Shapire [31], fundamentally they still resemble the original algorithm by Angluin.

3.2.3 KV

Another notable AAL algorithm is the KV algorithm published in 1994 by Kearns and Vazirani [21]. Like L^*, it also uses the MAT framework, but aims to minimize the amount of membership queries needed to learn a system. The *KV* algorithm does this by organizing learned information in an ordered tree data structure called a classification tree as opposed to the table structure utilized by L^*. As a trade-off, the *KV* algorithm requires on average more equivalence queries than 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 due to the nature of a the tree data structure. This property makes the *KV* algorithm particularly attractive for scenarios in which membership queries are the more expensive operation (e.g. due to network overhead).

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. First used to test the reliability of Unix utilities [24], it can be applied to test the reliability and security of any system that takes input. While originally just a simple random text-generation program, modern fuzzers are often more complex and boast a variety of features. Modern fuzzers mainly differ on the method of data generation and knowledge about the SUT. On the data generation side, fuzzers 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 regarding the SUT is available to the fuzzer while fuzzing. More specifically, based on how much information regarding the expected input structure and the internal workings of the SUT is known to the fuzzer, one commonly distinguishes between *black box*, *gray box* and *white box* fuzzers. Here, the term black box refers to a system that the fuzzer has no additional knowledge about. The fuzzer can interact with the black box system and receive responses, but its internal workings are hidden from the fuzzer. In the domain software, this translates to closed-source software, the inner workings of which are unknown. White box on the other hand, refers to the opposite case, in which the fuzzer is provided with as much additional information as it requires. This translates to open-source software, where fuzzers can make use of the original source code to further improve the fuzzing process. Gray box fuzzers lie somewhere in between the two extremes, often having access to some additional information regarding the structure or source code of the SUT, but to a lesser degree than white box fuzzers.

How much relevant behavior of the SUT is actually reached and tested by a fuzzer is commonly referred to as the coverage of the fuzzer. To ensure that a fuzzer can test all relevant parts of a SUT, i.e. achieve good coverage, additional information on the structure of the SUT may be utilized. While the source code coverage of the SUT can be used as a suitable metric if it is available, e.g. for white box fuzzers, in black box scenarios, other methods of guaranteeing meaningful coverage are required. One possible solution could be to use a model representation of the SUT to generate more relevant inputs and therefore better coverage. The model can be learned from an entirely black box system, allowing for sensible coverage metrics even without source code access. This technique is known as model-based fuzzing and is the

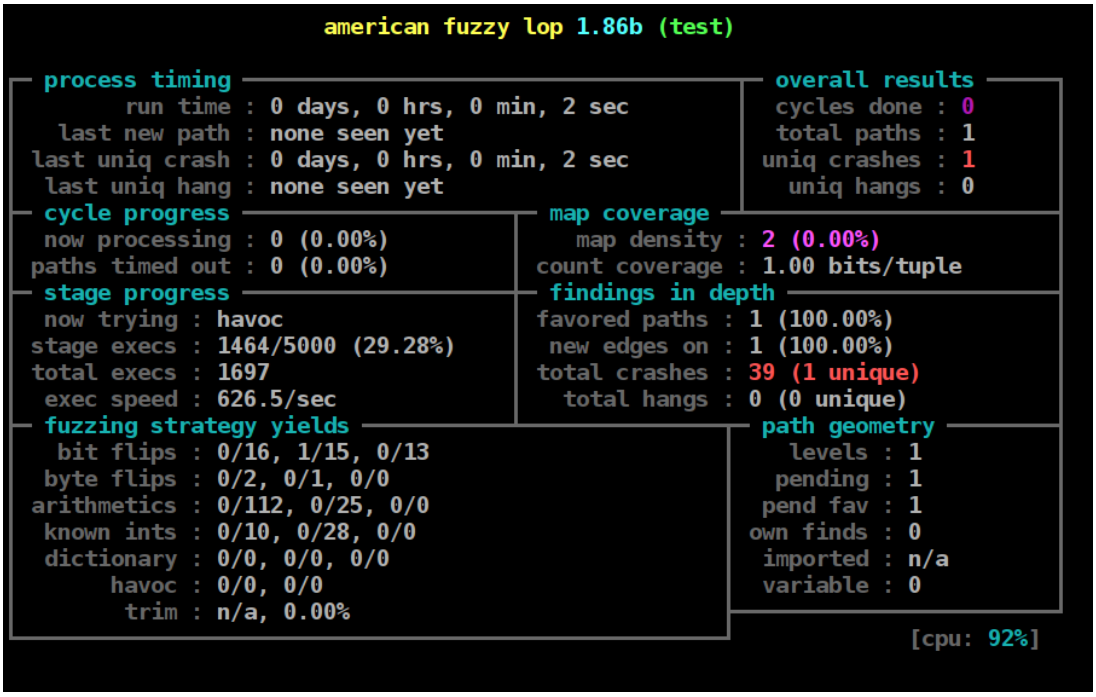

Figure 3.1: AFL fuzzer

fuzzing technique used in this thesis. While fuzzers come in many different shapes and forms, their core function is usually the same in that test data is generated, executed on the SUT and then the SUT is observed for strange behavior.

Two popular examples of fuzzers are american fuzzy lop (AFL) [43] and boofuzz [28]. 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].

On the other hand, boofuzz is a Python-based fuzzing library most commonly used for protocol fuzzing. As such, it does not require code instrumentation to function. Instead, it supports building blueprints of protocols to be fuzzed using primitives and blocks. These can be thought of as representations of various common components of protocols, such as data types including strings, integers and bytes, but also other common features, such as length fields, delimiters and checksums. These allow users to specify protocol requests to be sent to the SUT and explicitly mark which portions of the request should be fuzzed via settings in the primitives. Possible settings on a per primitive/block level include the maximum length of data to be generated while fuzzing and also if the element in question should be fuzzed at all, or just be left at a default values. The option to define which parts of a protocol will be fuzzed at a field-by-field level gives boofuzz a high degree of flexibility. The exact fuzz data generated by the framework depends on the used blocks and settings, and then creates mutations based on the specified protocol structure. For example, a string primitive uses a list of many “bad” strings as a basis for mutation, which it then concatenates, repeats and otherwise mutates. 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 primitives to generate our values for fuzzing, as detailed in

[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

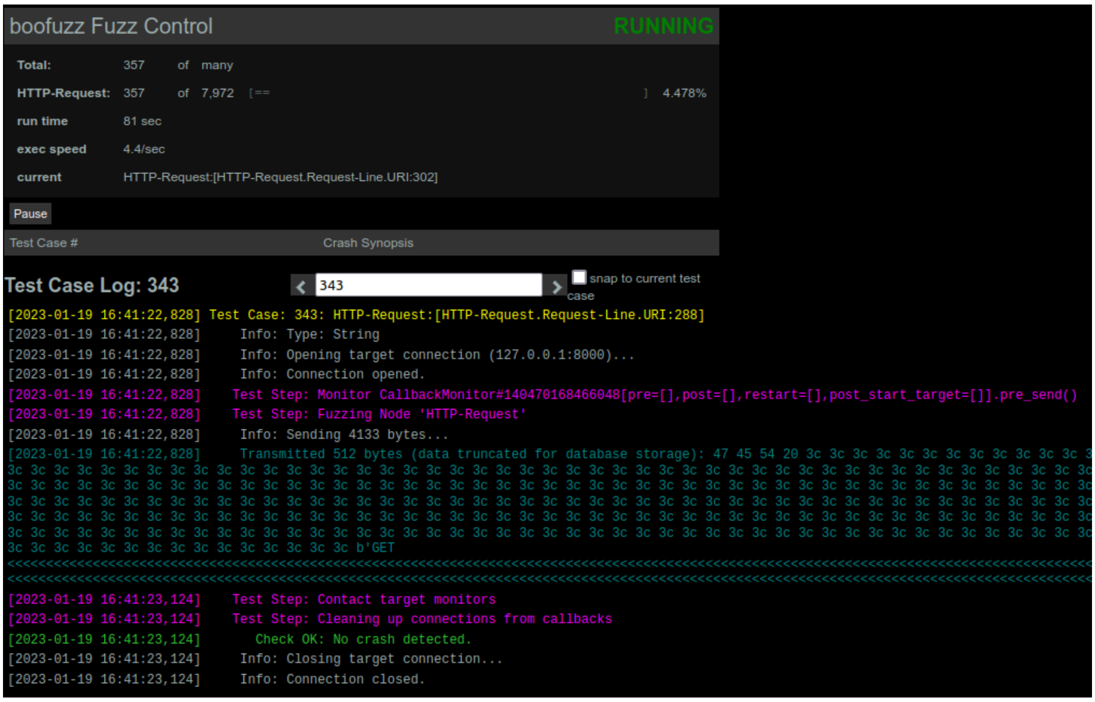

Figure 3.2: Boofuzz fuzzer

Chapter 6.

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 exist, including PPTP, OpenVPN and Wireguard. Internet Protocol Security (IPsec), is a VPN layer 3 protocol suite 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.

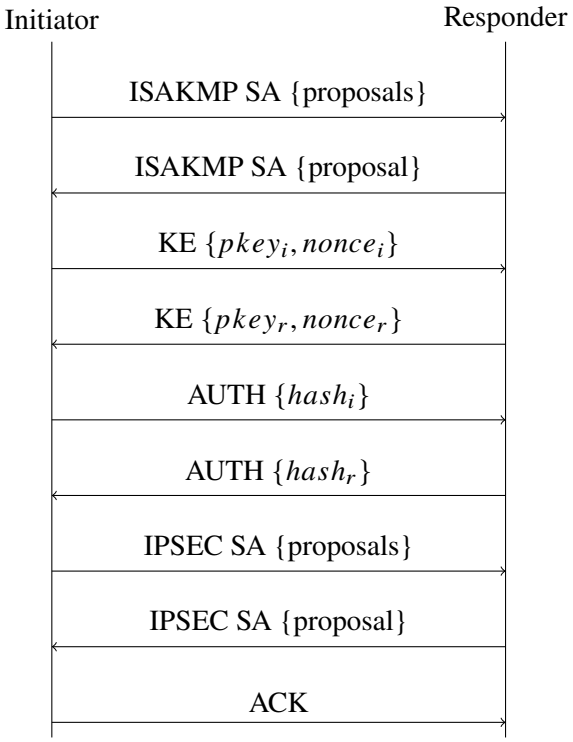

Figure 3.3: IKEv1 between two parties

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 (e.g. an employee connecting to a company network) and a responder (e.g. a company VPN server), using Main Mode for phase one and pre-shared key (PSK) authentication, can be seen in Figure 3.3. In contrast, aggressive mode reduces the number of packets in phase one to only three. While faster, this method is considered to be less secure, as the authentication hashes are sent in clear text. 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 and also exchanges initiator/responder cookies, tokens used as identifiers for the remainder of the connection. Subsequently, the two parties perform a Diffie-Hellman key exchange, denoted as *KE*, with the public key shorted to $pkey$, and send each other nonces used to generate a shared secret key `SKEYID` as detailed in Listing 3.2. Here, `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. The symbol "|" is used to signify concatenation of bytes, not a logical or. 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). 0, 1 and 2 are constants used to ensure that the resulting key material is different and unique for each type of key. 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. Note that the length of the actual encryption key depends on the used encryption algorithm, and is generated by concatenating hashes of `SKEYID_e` and trimming the result until the desired length has been reached. This allows for the `SKEYID_e` key to be used to generate arbitrary-length encryption keys. 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 PSKs 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 3.2: 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

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

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 [11] 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 [20]. 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 [16]. We 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

Environment Setup

This chapter provides an in-depth discussion of the environment used for both learning and fuzzing, focusing on its setup and configuration. The chapter begins with a detailed examination of the virtual machine (VM) setup, providing enough information to allow for the creation of a functionally identical VM environment. Relevant networking optimizations and design choices are highlighted. Following the discussion of the VM configuration, the installation and configuration of the two utilized IPsec VPN servers are examined in detail. Special focus is given to providing a comprehensive overview of the relevant IPsec configuration file options, including which options map to which keywords for the two servers respectively. Additionally, the most notable differences between the two servers are showcased and discussed.

4.1 VM Setup

All model learning and testing took place in a virtual environment using two VirtualBox 6.1 VMs running standard Ubuntu 22.04 LTS distributions (x64). Each VM was allotted 4 GB of memory and one CPU core. To set up the base Ubuntu VM using VirtualBox, we downloaded the Ubuntu image from the official source [1] and created a new generic Ubuntu (x64) VM in VirtualBox, specifying the downloaded Ubuntu image as the target ISO image. Next, we configured the hardware settings as detailed above and set the network to use NAT mode to be able to use the host computers internet connection to install updates and the VPN software. Furthermore, power-saving options and similar potential causes of disruptions were disabled within the VMs as well as on the host computer during testing. Additionally, a shared folder was created on each VM, linked to the local development folder on the host computer. This allowed for very easy testing, as there was no need to copy over Python code after every change. Instead it could simply be run from the mounted folder directly. Design-wise, for each pair of VMs, one was designated as the VPN initiator and one as the responder, to create a typical client-server setup. Following the installation and configuration of the VPN software (explained in detail in Section 4.2), all that remained was the final network configuration.

VirtualBox supports many networking modes for its VMs, including several that allow for inter-VM communication. These include host-only, internal, bridged and NAT-network networking modes. As we wished to minimize external traffic, we configured the VMs to use the internal networking mode, as it is the only one that solely supports VM-VM communication. The internal networking mode works by creating named internal networks that one can assign the network adapters of individual VMs to. All VMs within the same internal network can communicate freely with one-another, but not with the host computer, or any other network for that matter. We created a separate internal test network for each client-server pair of

[1] https://ubuntu.com/download/desktop

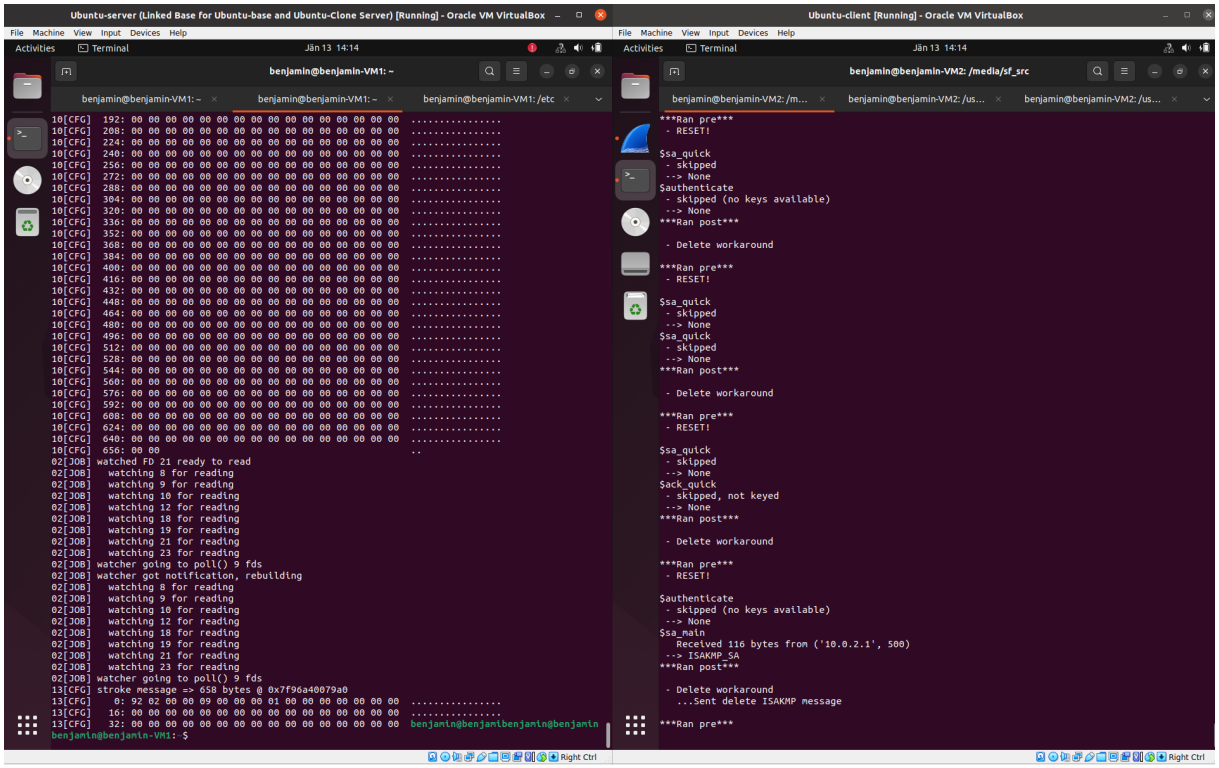

Figure 4.1: Pair of VMs running side by side, showing log output. Responder/server on the left, initiator/client on the right.

VMs, ensuring that all communication is isolated to the two involved parties. One important VirtualBox setting, is to change the network adapter from the default Intel, to the paravirtualized network adapter. Paravirtualized means, that instead of virtualizing networking hardware, VirtualBox simply ensures that packets arrive at their designated destination, through a special software interface in the guest operating system. This leads to a noticeable network performance increase. Within the guest Ubuntu installations, we configured the server to use the 2.0.2.1 and the client to use the 2.0.2.2 IP addresses respectively. We use the 2.0.2.0 network (255.255.255.0 subnet mask) with 2.0.2.0 as our default gateway. VirtualBox handles all of the internal routing, provided the two VMs are in the same internal network. Libreswan requires an additional second internal network with a different IP range to be configured. It is required for SSH-based resetting of the server from the client. This is discussed in Chapter 5 in more detail.

As we use a separate internal network for each pair of VMs, we can leave the IP configurations identical between pairs and only have to change the used internal network name. This makes cloning pairs of VMs very practical, allowing for numerous identical test setups to be created and run at the same time, limited solely by the computing power of the host machine. Figure 5.1 shows such a pair of VMs being run side by side.

4.2 VPN Configuration

The two IPsec implementations learned and tested were strongSwan and Libreswan. Both are popular open source IPsec implementations, with strongSwan featuring support for Linux, Android, FreeBSD, Apple OSX and Windows [38] and being the more widespread choice of the two. The libreswan IPsec implementation supports Linux, FreeBSD and Apple OSX [22]. Both projects can trace their roots back to the now discontinued FreeS/WAN IPsec project, an early IPsec implementation for Linux. Both support IKEv1 and IKE-v2, as well as an extensive list of additional features and authentication methods.

For this project, we installed both implementations and configured them to use IKEv1 with PSKs for authentication. The libreswan implementation uses a so-called *ipsec.conf* configuration file to specify connection details including IKE version, mode and authentication type. The IPsec background service is started/restarted via the commands `ipsec start` or `ipsec restart`. During learning and fuzzing, the IPsec server was restarted before each execution of code, to ensure identical starting conditions.

The configuration file includes a setting to automatically ready the server connection and to wait for incoming connections on startup. On the other hand, strongSwan actually supports two types of configuration files. One more modern one using the Versatile IKE Control Interface, and another legacy option, also using an *ipsec.conf* configuration file. The IPsec service is started using the same commands. To make the configuration file translation between IPsec implementations as straightforward as possible, we chose to use the *ipsec.conf* configuration file for both implementations. What follows is an overview of the used *ipsec.conf* settings for both implementations, as well as the full configuration files, shown in Listings 4.1,4.2.

The *ipsec.conf* settings control most facets of an IPsec connection. The configuration consists of two major sections, *config* and *conn*. The *config* section contains settings related to the general behavior of the IPsec background service that is not limited to an individual connection. In our case, we specify the debugging behavior to log everything in Line 2 and set the *uniqueids* setting to false in Line 3 of both configuration files. The *uniqueids* setting is used to instruct the IPsec implementation to treat individual IDs as globally unique or not. If set to “no”, new exchanges with the same ID are applied to the existing connection instance instead of replacing it. We use this option, to ensure that we do not invalidate existing sessions with each subsequent *ISAKMP SA* exchange. These settings apply to all connections configured in the configuration file.

```
1  config setup
2    charondebug="all"
3    uniqueids=no
4
5  conn vm1tovm2
6    auto=add
7    keyexchange=ikev1
8    authby=secret
9    left=10.0.2.1
10   leftsubnet=10.0.2.0/24
11   right=10.0.2.2
12   rightsubnet=10.0.2.0/24
13   ike=aes256-sha1-modp2048!
14   esp=aes256-sha1!
15   ikelifetime=28800s
16   dpdaction=none
17   keyingtries=%forever
```

Listing 4.1: strongSwan configuration

Connection specific settings are configured in a *conn* block and are given a name as an identifier, as seen in Line 5. The *auto* setting in Line 6 sets the default behavior when starting the IPsec service. Setting it to “add” instructs the connection to be readied and to wait for incoming messages. This option allows us to bring the IPsec server into a clean starting state by using either the `ipsec start` or `ipsec restart` console commands. Using the *keyexchange* or *ikev2* directives respectively in Line 7, we specify that the communication will use the IKEv1 protocol. The *authby* setting in Line 8 specifies the use of PSKs for authentication. The next four following lines specify the involved IP ranges and subnets. The

server or local IP is always referred to as the left side and the external connection partner is referred to as the right side of the IPsec connection. Line 13 specifies the encryption/authentication algorithm to be used for the phase one (ISAKMP) connection with the *ike* keyword. The configured options are read as "cipher-hashing algorithm-modgroup". So in our case, the connections are configured to use 256 bit AES-CBC mode for encryption, SHA-1 as a hashing algorithm and the modp2048 Diffie Hellman Group for key exchanges. Next follows the identical configuration for phase two (IPsec) in Line 14, with the *esp* keyword. We use the same parameters here, as for the prior configuration, however, as keying information for phase two is generated in phase one, we no longer need the modgroup value. The *ikelifetime* keyword in Line 15 determines how long the phase one connection will stay valid before the keys have to be replaced as a brute-force protection. "dpdaction" for strongSwan and "dpddelay" in combination with "dpdtimeout" for libreswan both serve to disable the Dead Peer Detection (DPD) feature of the IPsec implementations. DPD is a feature that allows IPsec servers to probe the status of their connection peers in order to ensure their availability. As this creates additional traffic, we disable this feature for the testing environment. Finally, in Line 17 we allow unlimited keying attempts in order to allow for the efficient fuzzing of phase one messages.

```
1  config setup
2    plutodebug=all
3    uniqueids=no
4
5  conn vm1tovm2
6    auto=add
7    ikev2=no
8    authby=secret
9    left=10.0.2.1
10   leftsubnet=10.0.2.0/24
11   right=10.0.2.2
12   rightsubnet=10.0.2.0/24
13   ike=aes256-sha1-modp2048
14   esp=aes256-sha1
15   ikelifetime=28800s
16   dpddelay=0
17   dpdtimeout=0
18   keyingtries=%forever
```

Listing 4.2: libreswan configuration

As can be seen in Listings 4.1, 4.2, the two configuration files only differ in a select few lines. The main differences are in the debugging setting, as the two implementations use different management backends, and in how DPD is disabled. Here, strongSwan has an explicit keyword to disable the function, while libreswan implicitly disables it if related settings are set to 0. Options not specified in the configuration file are set to their default values. Important default values relevant to the thesis include the "aggressive", "tunnel" and "pfs". The "aggressive" keyword determines if IKE should run in Aggressive mode or Main mode, defaulting to Main mode. As Main mode is the more commonly used, as well as more complicated option (including encryption), we leave this setting at its default value. The other two settings determine the type of VPN connection to establish (defaults to tunnel) and the use of Perfect Forward Secrecy (PFS) (defaults to PFS enabled). Both are left at their default values. Starting the IPsec server with the presented configuration options results in an easily-testable, basic VPN setup.

```
[IKE] shared Diffie Hellman secret => 256 bytes @ 0x7fcdc8012410
[IKE]    0: B4 90 E1 03 B5 2C D5 B2 4C 18 80 A9 68 C5 AA 3B  .....,..L...h..;
[IKE]   16: D5 24 27 EB C5 1C 7C 41 94 40 81 D0 B9 25 52 CB  .$'...|A.@...%R.
[IKE]   32: 66 A8 21 B5 3F 6F 7B 39 E7 A6 5A 68 C8 88 0F B2  f.!.?o{9..Zh....
[IKE]   48: B7 7A CB 51 31 4A A1 D9 A7 60 32 0E BE 65 30 42  .z.Q1J...`2..e0B
[IKE]   64: 3F 5B 58 79 13 8D DE 79 C8 57 51 A3 F8 D7 3E 91  ?[Xy...y.WQ...>.
[IKE]   80: 56 9B 67 09 20 BB 3F 3A 9F 87 45 DA CF 25 99 E2  V.g. .?:..E..%..
[IKE]   96: E7 71 70 82 F4 B4 A3 D5 76 91 0C 5C 08 4A 66 17  .qp.....v..\.Jf.
[IKE]  112: 76 C0 24 44 47 68 8B 86 FF 47 74 6B 4A B6 63 61  v.$DGh...GtkJ.ca
[IKE]  128: A7 C6 45 35 1B 1B FF A2 C5 47 43 E2 B1 A4 D7 C8  ..E5.....GC.....
[IKE]  144: E6 52 F4 9C 10 DE 76 11 C2 62 6F 75 3F 87 A7 0D  .R....v..bou?...
[IKE]  160: B2 DB 8B 18 1C C8 FA 26 D7 DD A2 B4 02 12 AB 81  .......&........
[IKE]  176: 9D F9 A3 4D AF AE 5D 41 4E 52 00 3A 11 F2 0C 32  ...M..]ANR.:...2
[IKE]  192: 63 BC 8C 3A 13 C1 CE 9E D6 16 7F 0E 94 48 B9 73  c..:.........H.s
[IKE]  208: DB 17 E1 A5 3D 75 53 3F F6 1E AA 3F B1 12 C4 E7  ....=uS?...?....
[IKE]  224: C9 A5 0E 32 84 E3 AC 59 46 4B 92 66 E5 DD D4 76  ...2...YFK.f...v
[IKE]  240: 63 C8 00 EA CA DE 14 4A DF 8A 59 F1 9F 91 89 C1  c......J..Y.....
[IKE] SKEYID => 20 bytes @ 0x7fcdc80122d0
[IKE]    0: 09 85 C6 22 57 90 2B CF 1C E2 6C 33 4D 83 14 76  ..."W.+...l3M..v
[IKE]   16: 94 1A F8 07                                      ....
[IKE] SKEYID_d => 20 bytes @ 0x7fcdc8012520
[IKE]    0: 90 56 B1 1C 56 97 8D 48 A9 FF 83 9F 86 09 31 BD  .V..V..H......1.
[IKE]   16: 85 EF C4 D2                                      ....
[IKE] SKEYID_a => 20 bytes @ 0x7fcdc8012670
[IKE]    0: 28 33 5A E0 B5 23 D3 7B 30 66 7C 98 71 E0 46 A6  (3Z..#.{0f|.q.F.
[IKE]   16: 74 2E F6 ED                                      t...
[IKE] SKEYID_e => 20 bytes @ 0x7fcdc8012690
[IKE]    0: 94 50 C3 62 89 C4 CD D3 D4 4B 44 C1 F5 3D B0 11  .P.b.....KD..=..
[IKE]   16: 26 19 81 41                                      &..A
[IKE] encryption key Ka => 32 bytes @ 0x7fcdc80037a0
[IKE]    0: 13 EB 78 54 A5 F1 1C 41 82 41 27 E8 54 7E 19 98  ..xT...A.A'.T~..
[IKE]   16: E1 BE C3 AF F0 21 7A C2 F8 3D AF B3 36 DB 31 85  .....!z..=..6.1.
[IKE] initial IV => 16 bytes @ 0x7fcdc8012690
[IKE]    0: 62 93 69 45 6A 7A BA 02 B6 2E 0C 07 59 82 61 16  b.iEjz......Y.a.
```

Figure 4.2: IPsec log excerpt showing keying information.

4.3 Debugging

During any software development process, roughly 35-50 percent of development time is on the testing and validation of the software [6]. The development of our custom mapper and fuzzer required a very large amount of debugging, leaning more towards the 50 percent side of the aforementioned statistic. A common scenario was, that the mapper class would do some cryptographic calculations, but the server would return an error. Setting the debugging options in the respective configuration files to “all”, has the IPsec implementations log all internal procedures in great detail. This often gave more insight as to why specific packets were being rejected. However, in certain cases, a manual comparison of strongSwan-generated and our custom-generated IPsec packets had to be performed on a byte-by-byte level. The open-source packet sniffing tool Wireshark [36] was used for this purpose. It allowed us to first connect the strongSwan client and then our own mapper class, all the while recording the traffic. This aided greatly in finding packet-level differences between the two implementations, allowing us to find and fix several bugs in ours. Unfortunately for debugging, IPsec, or IKE to be more specific, encrypts large portions of its communication (everything past the first key exchange). This of course made analyzing the packets impossible, as the relevant information would be encrypted. Luckily, strongSwan allows for the logging of encryption keys and connection cookies. However, this setting is disabled by default. To enable it, one has to edit a different configuration file, namely the *strongswan.conf* file, found usually in the same folder as the *ipsec.conf* file. This other configuration file controls IPsec management backend specific options, including a logging level. Setting this level to the highest (four), causes cryptographic keys to also be logged. An excerpt of the relevant log file can be seen in Figure **??**, where one can see the shared Diffie-Hellman secret, as well as all relevant keying material. The actual key used for the encryption of messages is denoted as the *encryption key Ka*. The encryption key is generated by concatenating two hashes of the *SKEYID_e* and trimming to 32 B (for AES-CBC-256 encryption).

Using the encryption key as well as the cookie of an IKE connection, the packets can be decrypted in

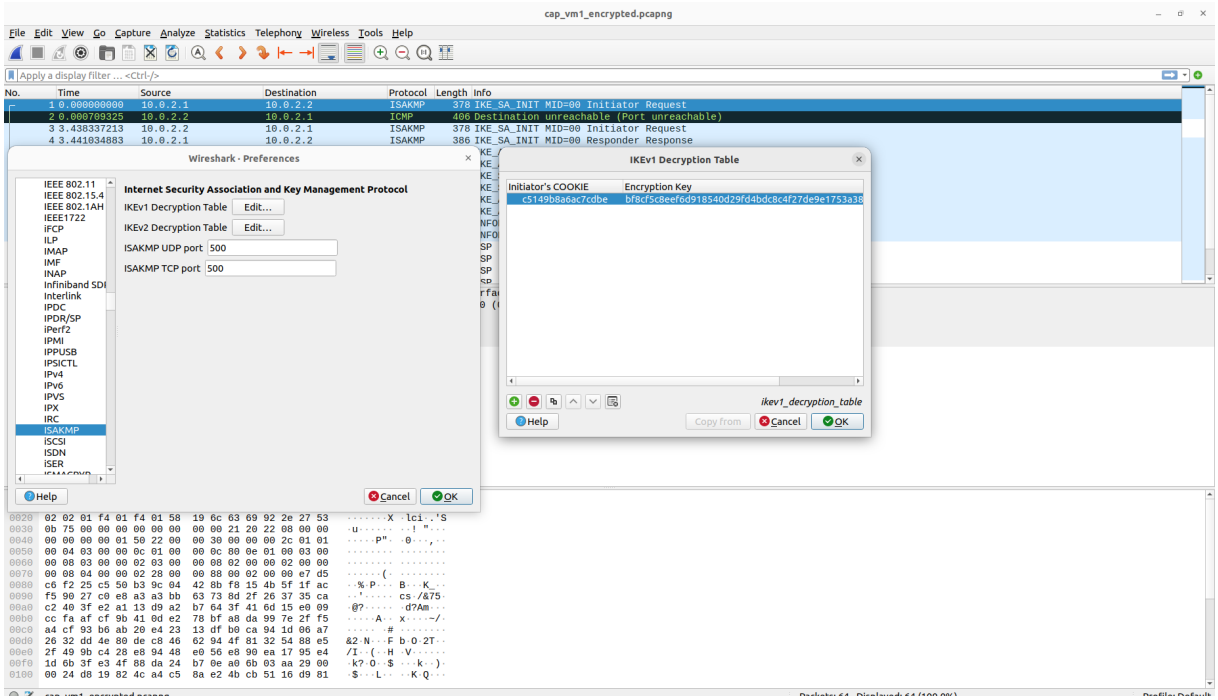

Figure 4.3: Decrypting IPsec packets in Wireshark.

Wireshark, by inputting the values in the corresponding protocol options field, as shown in Figure 4.3. Similar cryptographic information can be found in libreswan using the `ip xfrm state` command. The use of Wireshark for packet-level debugging greatly aided in the development of the custom mapper especially, as oftentimes, the relevant RFC specifications were rather unclear / left open to interpretation. A Wireshark packet dump of a sample IPsec connection establishment between two strongSwan participants, as well as the corresponding decryption key and cookies is provided as supplementary material.

Chapter 5

Model Learning

This chapter covers the learning and experiment setup. It showcases the many steps needed to learn the model of an IPsec server, highlighting various design decisions. Additionally, implementation problems and our proposed solutions are discussed and presented. As until now we have been discussing learning algorithms from a theoretical standpoint, we will begin with a brief definition of automata learning terminology to use going forward that better suits the task of learning a reactive system.

The goal of our learning setup is to learn a Mealy machine that models the SUL. We refer to it as the *learned model*, or simply *model* and *automaton* interchangeably. To this end, we employ a learning algorithm, which requires an input alphabet Σ of packets that are understood by the SUL. We refer to individual elements of the input alphabet as *inputs*, whereas we refer to a chain of (multiple) inputs as an *input sequence*. Each input of an input sequence will be executed on the SUL subsequently. We refer to one execution of our learning program as one learning attempt. A successful learning attempt is one that results in a correct behavioral model of the SUL. As we are working with Mealy machines, the term *output query* is used instead of membership query.

5.1 Learning Setup

The models of two separate IPsec implementations were learned, namely behavioral models of strongSwan and libreswan IPsec server implementations. The IPsec servers were installed and setup on the responder VM, as detailed in Chapter 4. They were configured to listen for incoming connections from the initiator VM, which are generated by our learning setup. On a high level, our learning framework consists of four main parts, as shown in Figure 5.1. These are the learning algorithm, the custom mapper, the communication interface and the SUL. The learning algorithm handles the actual learning via the L^* or *KV* algorithm. This is in large part done using the Python automata learning library AALpy [26] version 1.2.9. AALpy supports deterministic, non-deterministic and stochastic automata, including support for various formalisms for each automata type. We chose deterministic Mealy machines to describe the IPsec server, as they are commonly used to model reactive systems. However, learning automata with AALpy follows the same basic process, regardless of the type of automata used.

The custom mapper makes up a large portion of our work and is used to convert between abstract inputs and actual ISAKMP packets, using the packet manipulation library Scapy[1], version 2.4.5. Scapy was used to parse and create ISAKMP packets, which are used by the communication interface to communicate with the IPsec server. 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

[1] https://scapy.net/

script, *IPSEC_IKEv1_SUL*[2] demonstrates how AALpy can be used in conjunction with our custom mapper and communication interface to communicate with, and learn the model of an IPsec server. What follows, is a more detailed look at how the individual parts of the learning framework work together to learn the model of an IPsec server.

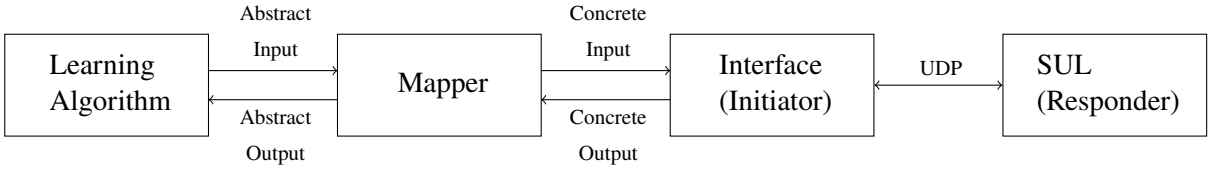

Figure 5.1: Automata Learning Setup

Figure 5.1, adapted from Tappler et al. [40], gives an overview of the model learning process. To begin, the learning algorithm sends abstract inputs chosen from the input alphabet to the mapper class, which converts it to concrete inputs. In other words, the mapper class converts between sequences of abstract and concrete inputs. The concrete inputs are then sent to the SUL, by means of the communication interface. In our case, the mapper class comprises the major portion of our work in the establishment of the learning framework and converts the abstract words into actual IPsec packets that can be sent to the SUL Strongswan server via UDP packets. This alphabet abstraction step simplifies the learning process, as learning the model for all possible inputs of an IPsec server would be tedious at best. Additionally, the separation between abstract and concrete inputs/outputs allows for easy future modifications to the message implementations, including fuzzing support, as well as increasing the readability of our code.

Protocol	Input Alphabet
ISAKMP SA	sa_main
KE	key_ex_main
AUTH	authenticate
IPSEC SA	sa_quick
ACK	ack_quick

Table 5.1: Mapping protocol to input alphabet names

To begin learning an automaton with AALpy, one 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 corresponds to the IKEv1 protocol messages shown in Figure 3.3. The protocol messages map to abstract inputs of the input alphabet as shown in Table 5.1. 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 the SUL has been learned successfully. This corresponds to an equivalence query. Additionally, several optional AALpy features were enabled, including caching and non-determinism checks to improve the learning process. An overview of the relevant learning algorithm initialization code can be seen in Listing 5.1. Line 3 shows the used input alphabet, Line 4 the used equivalence oracle and Line 5 the used learning algorithm. Both the equivalence oracle and learning algorithm take the input alphabet

[2] [TODO: GITHUB LINK]

and an object representing an interface to the SUL as parameters, were the SUL interface is defined as shown below in 5.2 and can execute inputs on, as well as reset the the actual SUL. The equivalence oracle is also passed as a parameter to the learning algorithm with 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',
4  'ack_quick']
5  eq_oracle = StatePrefixEqOracle(input_al, sul, walks_per_state=10,
       walk_len=10)
6  learned_ipsec = run_Lstar(input_al, sul, eq_oracle=eq_oracle,
       automaton_type='mealy', cache_and_non_det_check=True)
```

Listing 5.1: Equivalence Query code

The SUL interface defines the *step* and *reset* methods, as can be seen in Listing 5.2. *step*, seen in Line 3, is used to execute input actions. *reset*, show in lines 8-12, reverts the SUL to an initial state. An output query is a sequence of inputs executed on the SUL. Every input sequence is executed starting from an initial state, hence the need for a *reset* method. Used in combination, *step* and *reset* allow asking output queries to the SUL. *pre* is called before each output query and *post* afterwards. The abstract input chosen from the input alphabet is passed on to the mapper class for further processing. Line 4 shows how a function, corresponding to the abstract input, is called in the mapper class and the return value (abstract output) is passed on to the learning algorithm.

```
1  # code excerpt from IPSEC_IKEv1_SUL.py
2
3  def step(self, input):
4    func = getattr(self.ipsec, input)
5    ret = func()
6    return ret
7
8  def pre(self):
9    self.ipsec.reset()
10
11 def post(self):
12   self.ipsec.delete()
```

Listing 5.2: SUL interface

The mapper class implements methods for each communication step in a typical IPsec-IKEv1 exchange, as described in Section 3, but referred to by their input alphabet name according to Table 5.1. This includes methods for *sa_main*, *key_ex_main*, *authenticate*, *sa_quick*, *ack_quick* packets. Additionally, the mapper class supports *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. It is critical for the correct functioning of *reset* that this input is executed correctly, hence it requires the SUL state be checked after execution, which is not feasible during learning. 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. Informational messages are mainly used in IPsec to return error codes when something goes wrong. To illustrate our mapper class, (simplified) excerpts from the *sa_main* method are shown in Listing 5.3. 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. Line 5 shows how an ISAKMP transform is created, encompassing various security parameters. This transform is packed into a ISAKMP proposal packet first and then the resulting packet is packet into an ISAKMP SA packet in Line 6. The SA packet is appended to a generic top level ISAKMP packet in Line 8. In Line 9 the ISAKMP packet is sent to the SUL and its response (if any) is received. The connection manager, initialized as `self._conn`, handles the actual sending and receiving logic and returns the server response already converted into a matching Scapy object. The Scapy response object then undergoes a retransmission check and is then parsed with relevant data being used to update local values, as indicated in lines 11-16.

```
1  # code excerpt from IPSEC_MAPPER.py
2
3  def sa_main(self, ...):
4  ...
5  tf = [('Encryption', 'AES-CBC'), ('KeyLength', 256), ('Hash', 'SHA'), ('
       GroupDesc', '1024MODPgr'), ('Authentication', 'PSK'), ('LifeDuration',
        28800)]
6  sa_body_init = ISAKMP_payload_SA(prop=ISAKMP_payload_Proposal(trans_nb=1,
        trans=ISAKMP_payload_Transform(num=1, transforms=tf)))
7
8  policy_neg = ISAKMP(init_cookie=cookie_i, next_payload=1, exch_type=2)/
       sa_body_init
9  resp = self._conn.send_recv_data(policy_neg)
10
11 if (ret := self.get_retransmission(resp)):
12 # retransmission handling
13 ...
14
15 # Response handling (checks response code, decrypts if necessary, updates
        relevant local values)
16 ...
```

Listing 5.3: Excerpt of sa_main method 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 by updating the model.

5.2 Design Decisions and Problems

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, our chosen automata learning algorithms require a SUL reset method to be able to return to an initial starting point after each query. Due to design differences between strongSwan and libreswan, we were forced to implement this reset method differently for the two IPsec servers. For strongSwan, we implement reset using a combination of the ISAKMP *DELETE* request and general ISAKMP informational error messages. While *DELETE* alone 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. On the other hand, libreswan does not allow remote resetting of phase one connections at all, so here we were forced to implement the reset method by sending `ipsec restart` commands via a SSH connection directly to the IPsec manager backend. This adds an additional separate medium of communication, outside the existing protocol stack, potentially skewing runtime measurements. Therefore, the strongSwan implementation was used for most benchmarks, as there, all messages sent were part of the IPsec protocol stack and the reset method does not depend on having root access to the SUT (as is unlikely in a black-box scenario).

Each concrete mapping function in our mapper class can be run with a standard configuration, or with arbitrary values for the respective fields of the resulting packet. This allows us to learn different variations of the IPsec servers. For example, our mapper class allows us to very easily switch between learning a server model when presented with valid inputs, and the model of a 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 a server presented with malformed inputs will serve as the basis for future model-based fuzz testing and can be seen in Chapter 7.

As inputs will be sent in many different, potentially unusual, combinations during learning, we require a robust framework that correctly handles the encryption and decryption of IPsec messages. For key management, we simply store the current base-keys but keep track of initialization vectors (IVs) 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. As libreswan also sends retransmissions of phase one messages, we additionally have to store identifying information (nonces, hashes, etc.) to be able to match retransmissions to past messages, as phase one messages all have the same M-ID of zero. 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 from other message types. 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-IDs 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. Implementation of the mapper class, in particular encryption and decryption functionality, was hindered at times by unclear RFC-specifications, but this was overcome by manually comparing packet dumps and IPsec server logs to fix errors, as detailed in Chapter 4.

To ensure that we receive all responses, we add a timed wait for each server response. In the case of no response arriving during the wait, we directly return an empty *None* response and need no further handling.

Otherwise, we check the response M-ID against our list of previous M-IDs to detect retransmissions. A retransmission is when the server returns a previously returned message in response to a new request. In this case, the retransmission M-ID (or other identifier in the case of libreswan phase one messages) is the same for both responses. Our retransmission handling is covered in more detail in Section 5.3. If a retransmission is detected, depending on the configured retransmission-handling rule, it is either ignored or the corresponding previous response is returned. 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 for retransmitted messages and return the saved previously parsed response directly. 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 this 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 during learning and is implemented mainly for later fuzzing.

Since 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 minimal 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 at the cost of some informational loss. However, since any deviations or non-deterministic behavior would have been caught by the learning 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 output 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 output query can take several seconds. With hundreds of output 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 output queries should, in theory, lead to an overall decrease in runtime. Since AALpy supports the *KV* algorithm, switching between the two learning algorithms is as easy as setting a simple flag as shown in line three of Listing 5.4 below. The *KV* algorithm required less output 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 7.

```
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 5.4: Switching Learning Algorithms

5.3 Combating Non-determinism

Despite many precautions taken to create a disturbance-free learning environment, as detailed in Chapter 4, the IPsec servers still exhibited non-deterministic behavior, resulting in variance among the learned models. While the majority of learned models were identical, the outliers were significantly different,

having differing amounts of states and transitions between them. To help decrease the remaining non-deterministic behavior, additional 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 [23], the handling of this situation is unspecified, leaving the exact handling up to the implementations, 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, they seem to trigger in a not-quite-deterministic fashion in phase two of IPsec IKEv1 protocol. libreswan also implements retransmissions, but here they also occur in phase one, forcing us to identify the messages in question in other ways (nonces, hashes, etc.).

While interesting for fingerprinting, as certain models were learned with a much higher frequency than the outliers and they contain a lot of information, a deterministic model was required to serve as a base case for model-based fuzzing. Therefore, checks were added in our mapper to allow for 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) and other identifying information for libreswan phase one retransmissions. If a repeated message is found, it is flagged as a retransmission and depending on the current filtering rule, ignored. With this addition, non-deterministic behavior no longer occurred, allowing the learning of very clean models, as shown in Chapter 7, Figure 7.3 and Figure [TODO: ADD REF TO LIBRESWAN REF MODEL!] . As an additional method of dealing with non-determinism problems, but still keeping retransmissions, non-determinism errors can be caught as they occur and the offending queries repeated several times. If upon the first rerun the non-determinism does not occur again, the existing value can be accepted as the correct one. If however, the non-determinism errors persist for a set amount of repetitions with the same constant server response, it is likely, that the original saved response was incorrect and it can be updated to the new response. However, this method heavily impacts runtime performance, as many queries have to be repeatedly sent. As retransmissions are inherently non-deterministic and we need deterministic models for fuzzing, we decided to use mainly the filtering approach for our models. With the non-determinism correcting added, the automata learning works without non-determinism errors and the learned models are consistent with one another.

Chapter 6

Fuzzing

This chapter presents our model-based fuzzing setup used to test a Strongswan IPsec server. It first gives a high-level overview of our fuzzing process and then goes into more detail on the individual involved components and methodologies. We focus on testing entire input sequences (sequences of inputs of the input alphabet), and present two different methods of generating input sequences for fuzzing, with an emphasis on reducing the total runtime of fuzzing to a reasonable amount.

6.1 Fuzzing Setup

A basic fuzzing loop consists of test data generation, execution on the SUT and observing the SUT for strange behavior. Applied to fuzzing an IPsec server in a potential black-box scenario, some additional considerations and steps are required, as seen in Figure 6.1. Firstly, it must be taken into consideration, that the SUT reacts differently to inputs depending on which state it is in and a lot of states are locked behind specific sequences of valid inputs (e.g. phase two requiring a prior successful phase one). Therefore, it stands to reason that in order to achieve good coverage of SUT functionality, the SUT must be put into specific states prior to receiving each fuzzed test case. To this end, specific input sequences are passed to the fuzzer, which, at random, chooses a single input of each input sequence to fuzz, while still sending the other unchanged inputs surrounding the fuzzed input to the SUT as well. Note that technically, the mapped concrete ISAKMP packet is fuzzed and not the abstract input, as is explained in more detail in Subsection 6.1.4, however, we will continue to refer to it as the "fuzzed input". Additionally, the SUT is reset after each execution of a fuzzed input sequence, ensuring that each fuzzed input is executed in exactly the same starting state of the SUT. Figure 6.1 shows an example of such a fuzzed input sequence, with the fuzzed input being the third input of the sequence. In order to be able to detect strange behavior, a Mealy machine implementation of a learned reference model of the SUT is created, depicted in yellow in Figure 6.1. Each input that is sent to the SUT, depicted in red in Figure 6.1, is also executed on the Mealy machine in parallel, comparing the response codes of the SUT and Mealy machine. A mismatch between the response of the SUT and the expected one returned by the Mealy machine indicates, that a new state or behavior has been discovered. As the Mealy machine was created from the model of the SUT, any new behavior can be treated as interesting behavior worth further investigation. Input sequence generation/selection, new state detection and fuzz input generation is all handled separately from the SUL interface, in a new script called *fuzzing.py* [1].

While this basic fuzzing procedure is rather straightforward, the questions remaining is how to choose or generate the input sequences that are to be fuzzed, how the reference model is created/learned and how

[1] TODO: github or supplementary material link

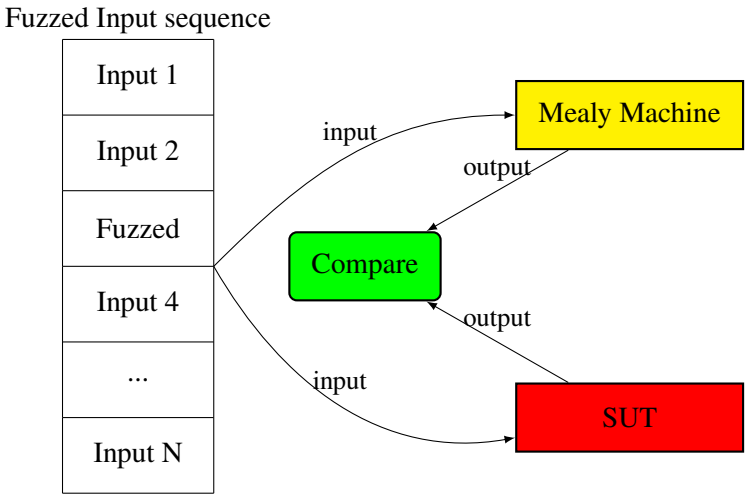

Figure 6.1: Overview of the fuzzing process.

the test data for fuzzed inputs in generated. The following subsections cover the creation of the reference model, as well as two methods of input sequence selection/generation.

6.1.1 Learning the Reference Model

The black-box determination of interesting inputs during fuzzing requires a model of the SUT to extract expected responses from. While a (deterministic) model of the SUL when exposed to expected inputs, had already successfully been learned, this proved to be not particularly useful for model-based fuzzing, as the majority of fuzzed inputs would result in an error message response from the SUT and therefore be treated as a new state. Instead of using the model of only expected behavior, a new model was learned, again using retransmission-filtering, but this time also with an expanded input alphabet. In addition to the previous input alphabet, an erroneous version of each input was added, that maps to an IKE packet with some sort of error or malformation. An example of such a malformed packet could be an incorrect length field, a wrong hash value or simply an unsupported SA option. Adding these new inputs to our input alphabet doubles its size and results in the following input alphabet: *sa_main*, *key_ex_main*, *authenticate*, *sa_quick*, *ack_quick*, *sa_main_err*, *key_ex_main_err*, *authenticate_err*, *sa_quick_err* and *ack_quick_err*. The model learned using the new input alphabet represents the behavior of the SUL when exposed to expected inputs, as well as unexpected inputs, that could arise during fuzzing. Since our mapper class was designed 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 in our mapper class to allow for the sending of erroneous packets.

6.1.2 Detecting New States

As inputs should be executable simultaneously on the SUT as well as on the reference model and have their respective responses compared in order to discover new states, i.e. new behavior. In order to be able to compare the outputs automatically, a Python representation of the learned reference model is required. Conveniently, AALpy is able to parse generated dot files into a corresponding Mealy machine object. This Mealy machine has a current state, as well as a `step` method, taking an abstract input as a parameter and returning the corresponding expected response based on the learned response when applying that input to the current state. Using this Mealy machine, a fuzzed input sent to the SUT can then easily be checked to see if it results in the same next state and response as it did on our reference model. By passing the same input to both the actual SUT and the Mealy machine representation of the reference model, this

comparison becomes very simple and can be easily automated. If the two responses do not match, a new state and therefore hitherto unexplored behavior of the SUT will have been discovered.

Using the Mealy machine representation of our new reference model, we were able to filter out the expected standard error correcting behavior and instead focus on more unusual behavior than previously.

6.1.3 Test Data Generation

As our mapper class was designed to allow for the easy fuzzing of IPsec fields, the only thing missing for test case generation is a source of values to use for fuzzing the individual fields. Our choice was the open source fuzzing library boofuzz[2], which is a successor of the popular Sulley[3] fuzzing framework. 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 our mapper class already had a very flexible way of sending manipulated IPsec packets and the protocol structure is quite rigid, we decided to only use the data generation features of boofuzz, which are mutation-based, forgoing the protocol definitions. To get relevant fuzz values for each field, every fuzzable field was mapped to a matching boofuzz primitive data type and then used that to generate our data. This ensures that each field is fuzzed with relevant data allowed by the protocol, e.g. string inputs are not suddenly used for length fields. While testing completely incompatible types of inputs would also be an interesting experiment in and of itself, we decided to focus on allowed but unexpected inputs, as these seemed the most likely to lead to undocumented/unexpected behavior. Additionally, supporting completely invalid types would have required a complete redesign of our mapper class. To summarize, we now have a rough fuzzing framework, consisting of a reference model to detect new states reached during fuzzing, as well as fuzz data generation, on a field type-by-field type basis. However, we are still missing a way of choosing which, or generating input sequences to insert the fuzzed test data into.

6.1.4 Input Sequence Generation

The final piece missing for our fuzzer is the input sequence-generation phase, in which a set of input sequences is generated/selected. During fuzzing, one (or more) of the inputs in the input sequence will be chosen to be fuzzed. An input is fuzzed, by replacing certain interesting fields of the concrete ISAKMP packet it maps to, with fuzzed values. Fuzzed inputs are embedded in a full input sequence to ensure that the fuzzed input is always executed in the same state, provided the SUT is reset after each sent/received input sequence. The difficulty lies in selecting or generating the input sequences to use for fuzzing in such a way, that the amount of new states discoverable through fuzzing is maximized, i.e. as much interesting new behavior is tested as possible. Effectively, this means that the goal is to achieve good coverage of the interesting behavior of the SUT. As the SUT is a reactive system, this task boils down to finding specific input sequences that lead to said new states. Naturally, one could achieve this by exhaustively fuzzing each field of every input of every possible input sequence, guaranteeing full coverage of the SUT. Unfortunately, due to the IPsec-IKEv1 protocol being rather complicated with IKE exchange messages containing tons of configurable fields, fuzzing even half the possible fields of a basic IKEv1 exchange would be an immense task that goes far beyond the scope and resources of this thesis. In fact, even the most simple packet of the exchange, the final *ACK* message, has at least nine distinct fields, while others have upwards of 50. Instead of trying all possible possibilities, several techniques to limit the amount of fuzzing to be done were implemented in a way that aims to still maximize the chances of discovering new states and potential bugs, while dramatically reducing the required runtime.

Firstly, instead of fuzzing every possible field of the protocol, the selection of fields to fuzz was narrowed down to 5-10 key fields from each packet. As abstract inputs are mapped to concrete packets

[2] https://github.com/jtpereyda/boofuzz

[3] https://github.com/OpenRCE/sulley

in the mapper class, the terminology used will be, that for each input, 5-10 fields are fuzzed. However, it is important to keep in mind, that the actual fuzzing happens on a protocol level, post the abstract-to-concrete mapping. In other words, actual fields of the protocol are fuzzed and not the abstract input words themselves. The fields to be fuzzed were chosen based on our estimation of their impact and chance of leading to errors. Additionally, fields unique to specific messages were prioritized. Length fields, SA proposals and hashes/keys were deemed especially interesting, but also general fields, such as the responder/initiator cookies were added. 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, as needed. When fuzzed, the fields are fed with input from a matching boofuzz data generator as explained in Subsection 6.1.3.

Additionally, three separate approaches for the generation/selection of relevant input sequences to use during fuzzing were implemented. The first method focuses on reusing existing input sequences from the model learning stage to achieve a more widespread coverage, while the second method generates a new input sequence using a search-based approach, that is designed to lead to as many new states as possible when fuzzed. Finally, a genetic algorithm for input sequence generation is showcased. The three methods are presented in more detail below.

6.1.4.1 Filtering-based Input Sequence Selection

Our initial idea when thinking about how to generate our input sequences for fuzzing, was to go on random walks through the Mealy machine and mirror the messages sent to the SUT as well, i.e. random testing. However, the problem here, at least for truly random walks, was that it resulted in a lot of wasted queries in phase one and not enough state coverage in phase two. Therefore, since a large number of input sequences had already been generated during model learning, guaranteeing state-coverage (at least for the learned reference automaton), these input sequences were repurposed for fuzzing. The problem with this approach however, was that the resulting set of input sequences was rather large. So, in an effort to reduce the fuzzing space, an additional filtering phase was added, to not include input sequences deemed uninteresting for in-depth fuzzing. In this initial filtering phase, each input sequence is processed one by one, randomly designating one (or more) of its inputs as the fuzzing target. Next, each fuzzable field of that input (i.e., the interesting fields of the concrete ISAKMP packet that the abstract input maps to) is tested in the context of the input sequence, with a greatly reduced set of fuzz values (3-5 values per field). The results/returned values are then compared to the expected outcomes using our Mealy machine, as explained previously in Subsection 6.1.2. If new behavior is found, the input sequence and choice of fuzzed input(s) within the sequence passes the filtering and are saved. Input sequences 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 not already tested during model learning and therefore also bugs. Note, that by filtering out input sequences and by choosing the inputs to fuzz randomly and not exhaustively, the potential of missing some new states is increased. However, since many of the input sequences from learning are very similar, differing only in a specific suffix, this again decreases the chance of discarding interesting input sequences, as chances are good that an input sequence containing at least a relevant subset of the discarded input sequence will pass.

Following the automatic filtering, the results are examined and identical or not relevant cases are removed manually. For example, we noticed that every input sequence in which cookies were fuzzed led to new states, due to new cookies indicating a completely new connection. Since our implementation learned the model with static initiator cookies, this will always lead to a new state.

Finally, following the automatic and manual filtering, 175 input sequences of various lengths remained, compared to roughly ten times the number before filtering. The filtered list of input sequences can be found in Appendix and contains all the discovered input sequences that exhibited new behavior. While not exhaustive, this approach of input sequence selection through filtering learning input sequences gives good coverage over a variety of different input scenarios. However, fuzzing 175 distinct input sequences takes

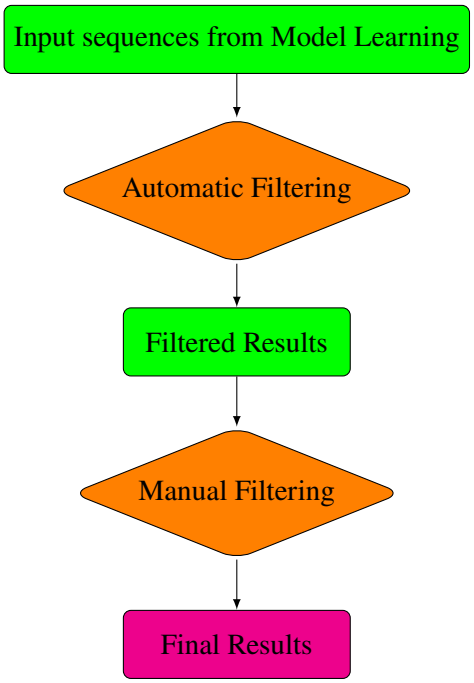

Figure 6.2: Overview of the filtering-based input sequence selection method.

a considerable amount of time (upwards of a full week), due to the long length of some input sequences, as well as the high amount of fuzzed values for each field of each tested input. While the selection of input sequences could certainly be optimized further, for example by adding additional filtering steps, the runtime is likely to remain rather high and would not be suitable for quick scans during security testing. It is however suitable for more in-depth testing. Instead of our approach using existing input sequences and filtering, we also implemented two promising methods of input sequence generation using a search-based approach and a genetic-based approach respectively, that are described in Subsection 6.1.4.2.

6.1.4.2 Search-based Input Sequence Generation

While the input sequence 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 input sequences still took well over several days. As a significantly faster alternative, the following input sequence generation method was developed, which results in only a single input sequence to be tested. The goal is to generate the input sequence which has the highest possible chance of reaching the largest amount of interesting states. To this end, we propose a search-based input sequence generation approach.

In its simplest form, search-based input sequence generation refers to the "searching" for a specific input sequence, that fulfills certain criteria. In our case, these criteria are, that the input sequence found has the highest possible chance of reaching the largest amount of interesting states, while at the same time proving as much state coverage as possible. We do this, by repeatedly applying small changes to a, possibly empty, base case and keeping only those changes deemed beneficial. These small changes, or mutations, could for example be swapping a bit, or changing a letter. Our fuzzer implements two mutation operations, however, more could be added in future work. The first mutation operation consists of adding a new input of the input alphabet to the input sequence. The second mutation operation swaps an existing input in the input sequence with its opposite version (so an erroneous version becomes a valid one, and vice versa). Our base case consists of either a random input, or a specified input sequence to be used. This second option was added since IPsec phase one packets have to be sent in a specific order to successfully authenticate. Remembering back to our attempts to use fully random input sequences for

fuzzing, we know that phase two is explored far less than phase one. Therefore, to speed up the search, the option of starting with phase one already completed was added.

Now that we have the means of generating small changes in input sequences, a method of evaluating the suitability of said changes is still required. As the goal is to generate an input sequence with the highest possible chance of reaching the largest amount of interesting states, while at the same time proving as much state coverage as possible, a fitness function was built to evaluate input sequences based on these goals. Through this fitness function, changes to the current input sequence are given a fitness score and only changes with a higher or equal fitness compared to the previous maximum fitness are accepted. Intuitively, the fitness score given should serve as a representation of how easy it is to find new states while fuzzing a given input sequence, as well as being weighted slightly, to encourage the exploration of new states. To calculate the fitness of an input sequence, each fuzzable field of every input in the input sequence is tested using the minimal list of fuzzing values, used previously in the initial filtering phase of the filtering-based approach. The amount of new states and/or transitions (not in the reference model) discovered, while minimally fuzzing each input of the input sequence is recorded and referred to as s_{new}. The fitness of an input sequence s_{sequence} is calculated as the sum of the number of new states found for each input s_{new}, divided by the number of inputs in the input sequence n, multiplied by the percentage of total states visited of the reference model.

$$s_{\text{sequence}} = \sum_{0}^{n-1} \frac{s_{\text{new}}}{n} \frac{s_{\text{visited}}}{s_{\text{total}}} \tag{6.1}$$

Calculating the fitness of an input sequence does take some time, as at least five fields are tested per input, and every input in the input sequence is minimally fuzzed. In comparison, in the filtering-based approach, only a random choice of input was fuzzed in each input sequence. However, it is still much faster than the initial filtering phase, provided the length of the input sequence being scored does not approach to the number of input sequences tested during model learning, divided by five (the average number of fields tested per input). Note, that while usually significantly faster than the filtering phase, fitness calculation occurs after every change to the input sequence being generated during the search-based approach, so for long input sequences (>30 inputs) the total runtime can exceed that of the previous filtering step. The key difference is, that after the search is completed, the result will be a single input sequence that is more interesting than any previously generated input sequence generated through the search, as opposed to the 175 input sequences after filtering. To help further increase the likelihood of generating interesting input sequences, the change/mutation operations were weighted to make adding a new letter at the end of the word more likely than at a random index, as well as the likelihood of the last letter being flipped over random letters being increased. 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 iterations. The rational being, that by adding more inputs, potentially more unique states can be visited, increasing coverage and potentially increasing the fitness of the input sequence.

An excerpt of a search-based input sequence generation over 50 iterations can be seen in Listing 6.1. The initial sequence, seen in Line 2, shows the starting point for future changes. Next, in Line 5, `authenticate` is mutated into `authenticate_err`. As the calculated fitness of the new input sequence is higher than the previous, the change is accepted. Mutation B in Line 8 inserts a new `sa_main_err` between the `key_ex_main` and `authenticate_err` inputs. This change is also accepted, as the fitness score increases again. Mutation C shows an even more significant fitness increase by almost 0.5 points, by changing `authenticate_err` to `authenticate`. Finally, a `key_ex_main_err` is inserted between `sa_main_err` and `authenticate`. As this change leads to a decrease in fitness, the change is rejected and the search continues using the previous input sequence.

Comparing the filtering and the search-based approaches, the filtering method takes far longer, but explores a higher number of states. In contrast, the search-based approach is much faster (for reasonably

```
1  Initial Sequence:
2  (['sa_main', 'key_ex_main', 'authenticate'])
3
4  Mutation: A, fitness: 1.1666666666666667
5  ['sa_main', 'key_ex_main', 'authenticate_err']
6
7  Mutation: B, fitness: 1.375
8  ['sa_main', 'key_ex_main', 'sa_main_err', 'authenticate_err']
9
10 Mutation: C, fitness: 1.8333333333333333
11 ['sa_main', 'key_ex_main', 'sa_main_err', 'authenticate']
12
13 Mutation D, fitness: 1.7333333333333334
14 (['sa_main', 'key_ex_main', 'sa_main_err', 'key_ex_main_err', '
       authenticate'])
15 Discarded
16 ...
```

Listing 6.1: Search-based input sequence generation.

sized input sequences), but in turn only tests a single input sequence. However, it generates that input sequence to be as interesting as possible in regards to the amount of new states that can be discovered through it.

Both methods found identical findings for the tested Strongswan IPsec server.

6.1.4.3 Genetic Input Sequence Generation

[TODO: :] stuff on genetic

[TODO: :] baseline comparison

Chapter 7

Evaluation

This chapter presents the results of our model learning and model-based fuzzing. Model learning results are presented in Section 7.1, beginning with the models learned using various input alphabets. Models from both examined IPsec implementations are presented and discussed. The presentation of learned models is followed by a comparison of the two used learning algorithms, L^* and KV, as well as the discussion of a library error found during learning. Finally, the fuzzing results are presented and discussed in Section 7.2, comparing the various methods of input sequence generation introduced in Chapter 6.

7.1 Learning Results

Over the course of our work, we learned a variety of different models due to different retransmission-handling settings and choice of inputs alphabets. The following sections showcase 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 5, Section 5.3 the resulting models still occasionally differed when not filtering out retransmissions. Therefore, the non-filtered models were not used for fuzzing, as a completely deterministic model was desired to serve as our baseline when fuzzing the SUT.

7.1.1 Learning Metrics

The comparison of learned models and model learning algorithm performance in subsections 7.1.2 and 7.1.3 is based largely on the the following metrics, saved during the model learning process.

Steps

Steps refers to the number of algorithm steps required by the learning algorithm or the equivalence checking.

Queries

Queries refers to the amount of queries sent during state exploration (membership queries) or during conformance checking (equivalence queries). AALpy supports speeding up model learning by using caching to reduce the number of required membership queries.

Runtime

Runtime refers to the time it took to learn the model. It is further split into state exploration and conformance checking runtimes. Runtime directly correlates to the number of steps and queries. When given in seconds, the runtime is rounded to the nearest second. Note that AALpy refers to output queries as membership queries.

7.1.2 Learned Models

Figures 7.1 and 7.2 show the two most commonly learned models when not filtering retransmissions. Roughly 80% of all models learned without retransmission filtering enabled resulted in one of these two models, which we will refer to as the common models. The other 20% of models were a non-uniform assortment of outliers. Figure 7.3 shows the clean base model learned from the SUL with retransmission filtering enabled. The reference model used for fuzzing is shown in Figure 7.4, also learned with retransmission-filtering enabled, as well as an expanded input alphabet. DOT files of all models are provided in Appendix

APPENDIX/additional resources

. The runtimes of both learning algorithms for all four models are summarized in tables 7.2 and 7.1. All values are averages over multiple learning attempts. Tables use the following abbreviations:

1. States: The number of states in the learned model
2. TT (s): Total time needed to learn the model (in seconds)
3. TL (s): Time spent on state exploration (in seconds)
4. TC (s): Time spent on conformance checking (in seconds)
5. MQ: Number of membership queries sent during the model learning
6. EQ: Number of equivalence queries sent during model learning

First Common Model

The first common model, presented in Figure 7.1, took approximately 52 minutes (3092 seconds) to learn with the *KV* algorithm, spread over seven learning rounds. The model consists of 10 states. Of the 52 minutes total, 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). 462 membership queries were required compared to the 171 of the *KV* algorithm. 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. This behavior is discussed in more detail in Subsection 7.1.3, which includes a statistical comparison of the two algorithms.

Moving on to an examination of the first common 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 complicated transitions caused by retransmissions. For example, all three transitions from

state *s5* to *s7* via *authenticate*, *sa_main* and *key_ex_main*, highlighted in yellow, return 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, our communication interface can occasionally happen to be listening for a server response of a regular phase two communication, when the SUL sends a retransmission for previous *sa_quick* message. This causes our framework to treat the received retransmission as the response for the phase two message, when in fact, it is not. We can see multiple incoming and outgoing transitions of state *s4*, highlighted in red, that further exhibit this same same behavior. 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 our input alphabet for this learned model does not include the delete command. 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.

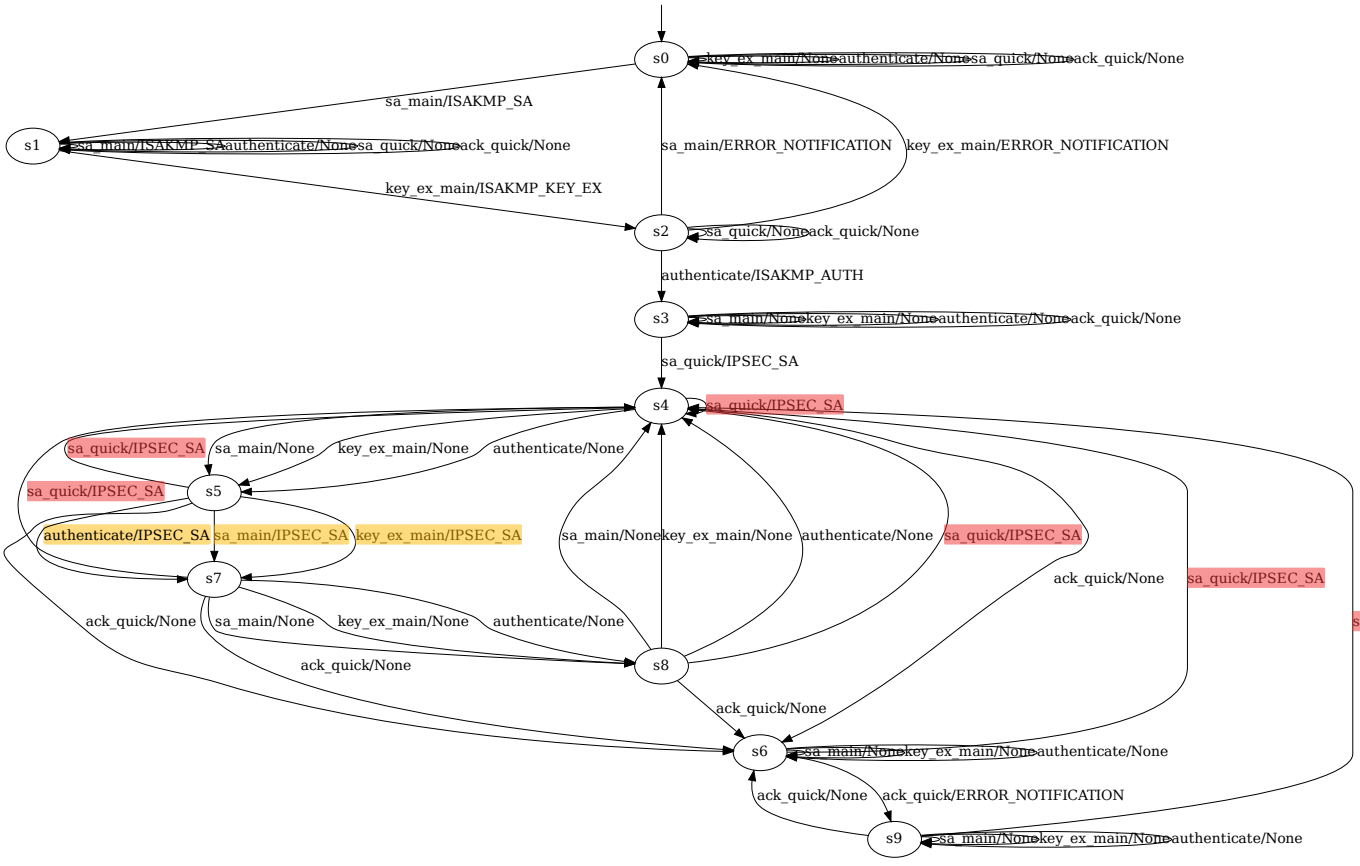

Figure 7.1: First commonly learned model with retransmissions.

Second Common Model

The second common model, seen in Figure 7.2, 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, with L^* taking 522 membership queries.

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, as no retransmission occur there. Same as in Figure 7.1, no paths past state *s2* lead back to the initial state. Phase two shows retransmission-induced strange behavior in the transitions *s5* to *s7*, as well as *s11* to *s9*. The strange behavior is again linked to retransmissions, causing phase one inputs, such as *sa_main*, to result in the valid phase two outputs, such as *IPSEC SA*. The states *s7* and *s11* are separated by two states that do not exhibit any strange behavior, apart from having identical in and outputs. The main difference to the first common model is, that strange behavior occurs in two pairs of states, highlighted in yellow, and that these pairs are separated by two states that do not appear to receive any retransmissions, highlighted in red. This is likely caused by the SUT sending repeated retransmissions, in the same frequency, allowing for two states in between.

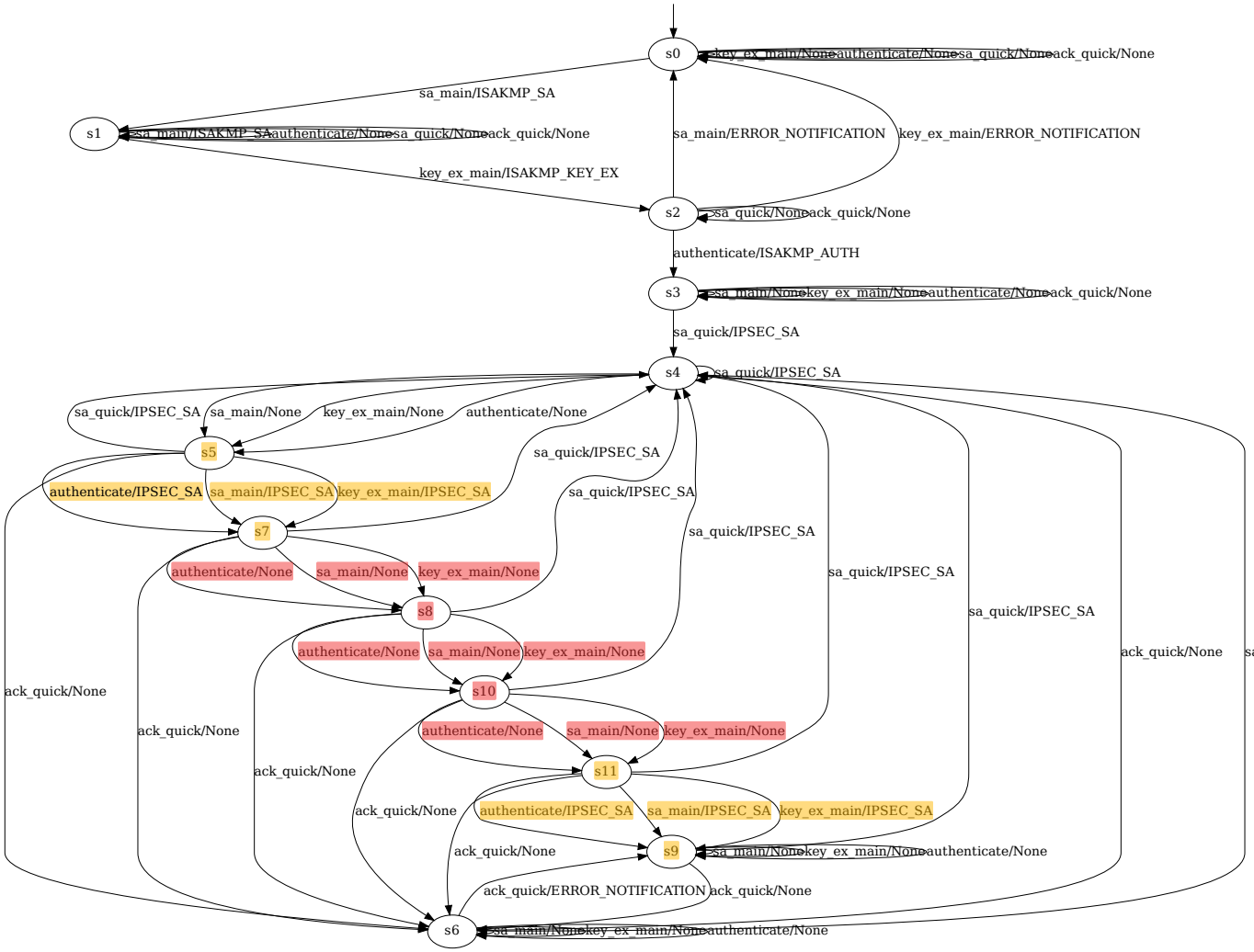

Figure 7.2: Second commonly learned model with retransmissions.

Clean Base Model

In comparison, when learning the same server using retransmission-filtering, all non-deterministic behavior vanishes and we get the model shown in Figure 7.3 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 40-60 split (519 vs 747 seconds). This was the only configuration where the conformance checking took longer than state exploration, as highlighted in Table 7.2. It does however still have the lowest altogether runtime. 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, 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 and since they are our only source of non-determinism, there are 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 shows how multiple IPsec SAs, each created from a single IKE SA channel, can be used interchangeably for different traffic flows but not simultaneously. As soon as a new IPsec SA has been established, another *ACK* message can be sent, to finalize the creation of the new IPsec SA. In other words, the extra state is there to show that a single IPsec SA cannot be acknowledged twice, and instead a new SA must be created first.

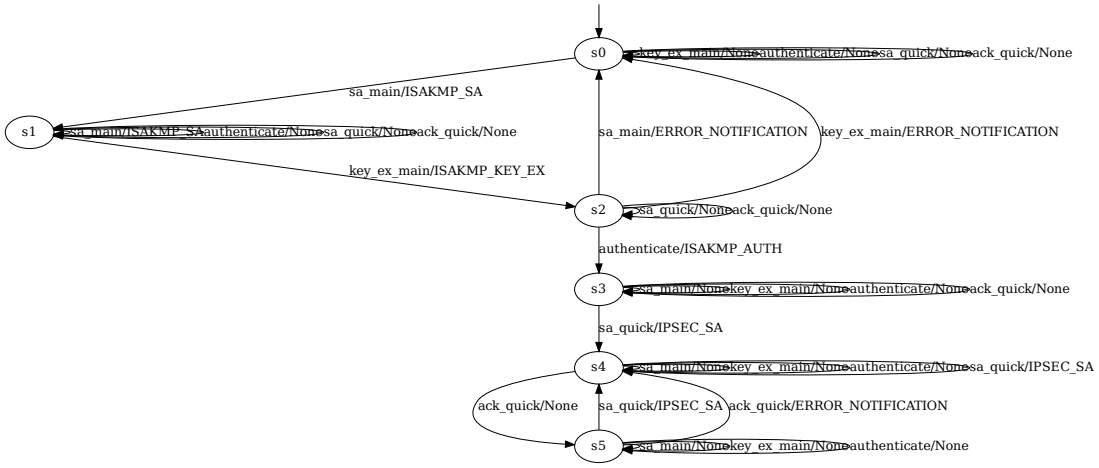

Figure 7.3: Clean model learned using retransmission filtering

Model	States	TT (s)	TL (s)	TC (s)	MQ	EQ
Common A	10	5094	3489	1605	462	100
Common B	12	7520	5393	2126	522	120
Base	6	1652	899	753	177	60
Reference	6	3078	**2500**	578	600	60

Table 7.1: L^* Runtimes of all the learned models

Fuzzing Reference Model

Figure 7.4 shows the reference model used for fuzzing, learned with retransmission-filtering enabled. Additionally, the input alphabet was expanded to include an additional erroneous version of each letter that maps to an erroneous input. Thanks to the retransmission-filtering, this model was, like the base model, 100% deterministic. Using the *KV* algorithm, it took roughly 24 minutes (1447 seconds) to learn. The model took between five and four learning rounds to learn and consists of six states. Roughly 60% of the total learning time was spent on state exploration/membership queries and the remaining 40% on conformance checking (879 vs 568 seconds). Alternatively, when learned using the L^* algorithm, the model took a total of 51 minutes (3078 seconds) to learn over a single learning round. The 51 minutes were split between state exploration and conformance checking in a 81-19 split (2500 vs 578 seconds), with state exploration requiring 600 membership queries. Interesting to note is the large difference in conformance checking runtime between the two algorithms. For learning the reference model using L^*, state exploration takes up more than 80% of the total runtime, which is the highest percentage for all the learned models, as can been seen highlighted in the runtime summary in Table 7.1.

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 a valid IPsec SA must be created before it can be acknowledged.

[TODO: move model to appendix / mention DOT file in appendix]

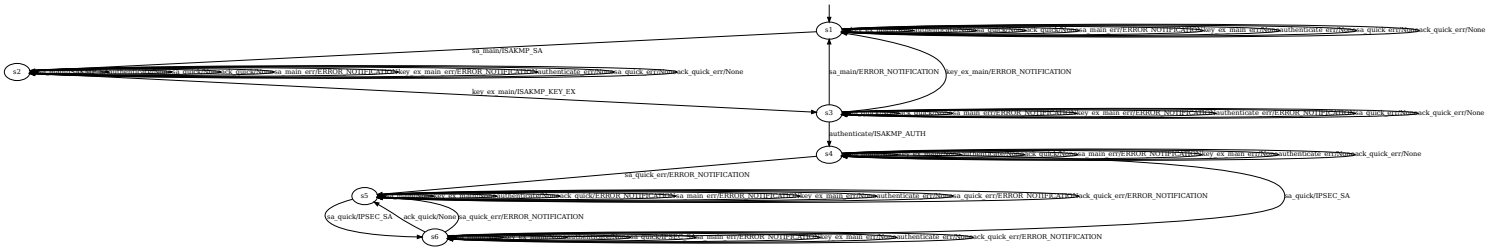

Figure 7.4: Model with malformed messages

Model	States	TT (s)	TL (s)	TC (s)	MQ	EQ
Common A	10	3092	1501	1591	171	100
Common B	12	4507	2382	2126	215	120
Base	6	1214	**480**	734	78	60
Reference	6	1447	879	568	174	60

Table 7.2: KV Runtimes of all the learned models

7.1.3 Comparing KV and L^*

Table 7.3 shows average performance statistics over 20 learning attempts for both learning algorithms. The models were learned with retransmission-filtering enabled. The same hardware and software configurations were used as described in Section **??** 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 7.3. Table 7.3 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 one expects 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, with both requiring the same number of equivalence oracle queries for conformance checking. In contrast, the time spent on membership queries differs greatly between the two model learning algorithms. The L^* algorithm required almost double the number of membership queries than its KV counterpart. As communication with the SUT is the main performance bottleneck and membership queries make up a large portion of this communication, this change naturally 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. As the same equivalence checking algorithm was used for both attempts, the identical number of equivalence oracle queries makes sense. Another noticeable difference can be observed in the number of membership queries saved by caching. Here, KV saves more than double the amount L^* does, indicating a better caching implementation. In summation, we found the KV algorithm to be better suited for our learning setup and solely used it for fuzzing.

l devi-

Little variance was observed throughout all learning attempts so the sample size of 20 learning attempts each is believed to be representative. However, for even more accurate results the experiment should be carried out again for even more runs. Additionally, it might be interesting to compare the performance of various equivalence oracles for this learning setup.

Learning Algorithm Performance (Averages)		
Metric	**L***	**KV**
Learning Rounds	2	4
Total Time (s)	1652	1214
Time Learning Algorithm (s)	**899**	**480**
Time Equivalence Checks (s)	753	734
Learning Membership Queries	**177**	**78**
Learning Steps	856	676
Equivalence Oracle Queries	60	60
Equivalence Oracle Steps	747	934
Membership Queries Saved by Caching	**13**	**30**

Table 7.3: Comparison L^* and KV

7.1.4 Library Error

Another notable finding from the model learning phase, which demonstrates the usefulness of AAL from a testing standpoint, 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, a small discrepancy between the two logs was discovered 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 [41] 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.

7.2 Fuzzing Results

We used model-based fuzzing to test the IPsec IKEv1 SUT. Our fuzzer supports testing inputs in the context of runs, to ensure an identical state on the SUT for each fuzzed input. To that end, we developed a custom fuzzer supporting two methods of generating the input sequences to be tested, filtering-based and mutation-based input sequence generation, as described in Chapter 6. All fuzzing took place in the same isolated network described in Section **??**. The used reference model can be seen above in Figure 7.3. This section presents an evaluation of the results of using our custom fuzzer to fuzz a Strongswan IPsec server. Two different input sequence generation methods were used and contrasted, comparing performance and findings. Both methods found the same issues, however the mutation-based input sequence generation

method proved to be significantly faster.

7.2.1 Findings

Our fuzzer was very successful in finding new states, discovering new cases for almost every input sequence tested with both run-generation methods. As a new state found does not necessary indicate, that the new behavior is harmful, the discovered new states still had to be looked over for particularly interesting behavior. By analyzing the run-fuzzed input combinations leading to new states, we discovered two undocumented instances of the SUT not following RFC specifications. Unfortunately, a lot of "non-findings" were discovered as well, non-findings referring to new states not included in the reference model, but also not exhibiting undocumented or otherwise interesting behavior. Some of these non-findings could be removed by simply specifying some fields which should not be fuzzed. Unfortunately, this approach requires first manually going through and verifying that none of the discovered behavior is interesting. As an alternative approach, the fuzzer could be further improved in future work by reducing the amount of noise by either relearning the reference model with a more advanced mapper class, or by adding newly learned states as soon as they are discovered to avoid the many duplicate new states found.

Want to still try this

For completeness sake, we first present some of the more interesting or common non-findings, followed by the two discovered deviations from the RFC specifications.

An example of a non-finding that is typically categorized as noise is the discovery of new states by changing the initiator/responder cookies during the run. These cookies are used in part to identify the members of a VPN connection. The new states were found, as during the learning of the reference model, the cookies were not changed mid-run, as this causes the SUT to think it is communicating with a different user and if it doesn't know that user, to simply discard the message. As fuzzing the cookie fields did not lead to any errors on the server side and greatly complicated the mapper class, this field was removed from the fuzzing scope.

Another non-finding was discovered while fuzzing the field indicating the number of proposals in the *ISAKMP SA* packet. While testing various randomly chosen lengths, every time the length was set to one, the fuzzer indicated, that a new state had been found. This behavior was caused due to our implementation of both the fuzzer and base reference model. Our *ISAKMP SA* packet always contains exactly one proposal and the packet being fuzzed is assumed to be incorrect. However, in this case, the field is, by fuzzing through random numbers, every so often set to a valid value (namely one), resulting in a mismatch between the expected (error) and actual (normal) response. Yet another cause of many non-findings during fuzzing were the various packets containing hashes, e.g. *ACK* and *AUTH*. Fuzzing these hashes proved to be problematic, as random changes to the hashes cause the server responses to not correspond correctly to the mapper class, making decryption impossible. Therefore, errors had to be manually examined in the Strongswan logs. Seeing as no crashes or other unexpected errors were observed, fuzzing hashes was reduced in the overall fuzzer.

Overall, while looking through and understanding the many new states helped improve our overall understanding of the IPsec protocol, most of them failed to exhibit any undefined/unexpected behavior. However, in two separate cases, deviations from the relevant RFC specifications were observed with the SUT. The first of these was discovered while fuzzing the ISAKMP length field. When manually going through the results of the fuzzer, a significant amount of newly discovered states were found to have been caused by packets with a fuzzed ISAKMP length field. The response to the fuzzed packet was compared to the expected return value according to the reference model. The two values did not conform, causing the fuzzer to indicate that a new state had been found. The mismatch was caused by the reference model expecting an error response to the incorrect ISAKMP length field, while in practice, the SUT ignored the content of the field entirely. This behavior is showcased in Listing 7.1, Line 6, where an error was

expected, but the SUT returned a valid response, despite the length field being the obviously incorrect value of hex $FF000000_{16}$ (4278190080_{10}). The finding was similarly observed with all other tested values, in all ranges from zero to $FFFFFFFF_{16}$, leading us to the conclusion, that the field is in fact completely ignored.

The impact of this finding is presumably rather small, however it might lead to inconsistencies between different IPsec implementations, should they handle this behavior differently. Additionally, RFC 2048 (describing ISAKMP) clearly specifies, that the ISAKMP payload length field must match the length of the entire payload. Seeing as IPsec simply builds on ISAKMP, these requirements should still hold true. In fact, RFC 2409, Section 5 [7] even states very plainly, that IKEv1 exchanges are to conform to the ISAKMP standard. This standard, in Section 5.1 of RFC 2048 [23] states the following.

> ...If the ISAKMP message length and the value in the Payload Length field of the ISAKMP Header are not the same, then the ISAKMP message MUST be rejected.

In other words, according to the RFC, the length field-manipulated ISAKMP packets are invalid and should be rejected. Since the tested Strongswan server does not, this finding falls into the deviations from specifications category.

```
1  Fuzzing isa_len with: b'\xff\x00\x00\x00'
2  Run: ['sa_main_fuzz', 'key_ex_main', 'authenticate', ...]
3  $sa_main_fuzz
4  %%%%%%%%%%%%%%%%%%%%
5  **********
6  Expected: ERROR_NOTIFICATION | Received: ISAKMP_SA
7  **********
8
9  $key_ex_main
10 **********
11 Expected: None | Received: ISAKMP_KEY_EX
12 **********
13
14 $authenticate
15 **********
16 Expected: None | Received: ISAKMP_AUTH
17 **********
18 ...
```

Listing 7.1: Finding showing the ISAKMP length field being ignored

Another interesting finding can be observed when fuzzing the transform `Authentication` field sent at the start of an IKE exchange as part of an *ISAKMP SA* packet. The expected behavior for the fuzzed field was, that the server would return an error response right away, indicating it does not support the proposed unknown `Authentication` method. However, in practice, the error response was only received after the following key exchange packet was sent. Strongswan logs also do not show any errors when parsing in an *ISAKMP SA* packet with a fuzzed `Authentication` transform field. This is due to Strongswan apparently only verifying the content of the `Authentication` transform field during the key-exchange step. RFC 2049, section 5 [7] explicitly states that

> Exchanges conform to standard ISAKMP payload syntax, attribute encoding, timeouts and retransmits of messages, and informational messages– e.g a notify response is sent

	Mutation	Filtering
Runs	1	175
Run Length (Av.)	11	16
Seconds / Run	11	16
Values Tested	2522	44000
Runtime (h)	~8	~194

Table 7.4: Comparison Mutation-based and Filtering-based Fuzzing

when, for example, a proposal is unacceptable, or a signature verification or decryption was unsuccessful, etc.

In particular the second part abort a notification being sent for unacceptable proposals leads us to believe, that the behavior exhibited by the Strongswan `Authentication` transform field is unintended behavior, that deviates from the RFC specification. It is important to note, that only the `Authentication` field exhibited this behavior, all the other tested transform fields (`Encryption`, `KeyLength`, `Hash`, `Group Description`, `Life Type` and `Life Duration`) appear to be checked right away and return errors if invalid/unknown. Another interesting point to note, is that Wireshark logs of the sent packets show, that the response transform has the `Authentication` field set to PSK, despite the fuzzed value sent to it not being supported. This indicates, that for this field Strongswan reverts to a default value if it encounters an invalid input, without logging any errors.

While not necessarily severe findings, the two RFC specification deviations clearly do not conform to ISAKMP and IPsec standards. As deviations from specifications can lead to vulnerabilities and compatibility issues, they should be carefully reviewed to ensure, that they are on purpose and not due to an oversight. In particular the `Authentication` field finding should be checked, as all other fields of that packet exhibited the correct behavior, indicating a potential developer oversight. It is important to thoroughly examine any deviations from established standards to ensure that the system is as secure as possible.

7.2.2 Mutation vs. Filtering-based Input Sequence Generation

This subsection compares the performance of the two input sequence generation methods used for fuzzing. Both methods discovered the same new states and therefore also findings. The main differences between the two methods lies in the amount of coverage achieved and runtime. While the filtering-based method achieves greater coverage over more states due to testing a large amount of different runs, the runtime is also proportionally longer. Table 7.4 shows a comparison of the two used input sequence generation methods, with the mutation-based method being run for a total of 60 mutations. The “Runs” column refers to the number of input sequences generated, while the “Run Length” column indicates the average amount of inputs per run. The ´´Seconds / Run” column displays the average execution time of a single input sequence and the ´´Values Tested” column shows the total number of fuzzed values tested in runs. Finally, the ´´Runtime” column indicates the total runtime of the methods. These metrics allow for a comparison of the efficiency and effectiveness of the two methods in generating and testing input data.

Examining Table 7.4, the most striking difference between the two methods is the difference in number of input sequences generated and the total runtime. While the mutation-based method always only generates a single run, the filtering-based approach generates far more, in our case 175 runs. For the input sequence generation with 60 mutations, the resulting input sequence consisted of 11 inputs. In contrast, the average input sequence length for the filtered input sequences was 16 inputs. The “Run Length” translates directly to the ´´Seconds / Run” metric. Each further input leads to, on average, an additional second of runtime, due to timed waits and timeouts during network communication. As every field

of each input is fuzzed when utilizing mutation-based input sequence generation, the amount of values tested per input sequence is significantly higher for this method. In comparison, only roughly 250 fuzzed values are tested per input sequence when using the filtering-based input sequence generation method. Still, as 175 input sequences have to be tested, despite the significantly less fuzzed values per run, the total amount of values tested is still far larger with the filtering-based method. In fact, a comparison of the two entries in the ´ ´Runtime" column shows that the filtering-based method takes approximately 24 times longer to complete than the mutation-based method. The runtime increases proportionally with the amount of tested inputs and average input sequence length.

Chapter 8

Conclusion

[TODO: improve on this]

8.0.1 Conclusion

In this Section, we showcased the four most relevant learned models and compared them with respect to various metrics including runtime and number of sent queries. Our results shows that our learning setup succeeds in its goal of reliably learning models of the target IPsec IKEv1 server. We contrasted two popular model learning algorithms KV and L^* and explained why we consider KV to be better suited for our learning setup. Additionally, we demonstrated the usefulness of AAL from a testing standpoint by showcasing a crypto library bug found during model learning. Future work could include support for more authentication methods, additional extensions and variable user-cookies. Additionally, it would be interesting to test the model-learning framework with multiple IPsec implementations.

8.0.2 Conclusion

In this section, we presented the results of our fuzzer, highlighting the most interesting findings, most notably, two deviations from the RFC specification. We recommend, that the findings be thoroughly examined, to ensure that they were not created accidentally and do not pose compatibility or security risks. Furthermore, we compared the two different utilized input sequence generation methods - filtering-bas ed and mutation-based. We found that both methods discovered the same findings, indicating their comparable effectiveness in generating and testing data. Therefore, we argue that the methods should be evaluated mainly based on their runtime performance. Comparing the two methods, we found that the mutation-based input sequence generation method performed significantly better, completing the fuzzing roughly 24 times faster than its counterpart (for a 60 mutation deep input sequence generation). This suggests, that the mutation-based method is better suited for larger-scale tests, where an exhaustive test might be too time costly. Overall, our findings highlight the importance of thorough testing and validation of network protocols and their implementations and show how new tools and techniques can be used to help accomplish that.

Bibliography

[1] Robert Swiecki et al. *Honggfuzz*. http://code.google.com/honggfuzz. [Online; accessed 19-April-2023]. 2016.

[2] Keith Andrews. *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/.

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

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

[5] Elaine Barker et al. *Guide to IPsec VPNs*. en. doi:https://doi.org/10.6028/NIST.SP.800-77r1.

[6] Tom Britton et al. *Reversible debugging software*. Cambridge Judge Business School. 2013. %5Curl%7Bhttp://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094%7D.

[7] Harkins Carrel. *Internet Key Exchange (IKE)*. RFC 2409. Jan 1998. https://www.rfc-editor.org/rfc/rfc2409.txt.

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

[9] Kaufman Hoffman Nir Eronen. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 5996. Sep 2010. https://www.rfc-editor.org/rfc/rfc5996.txt.

[10] Anja Feldmann et al. *A year in lockdown: how the waves of COVID-19 impact internet traffic*. Commun. ACM 64.7 (2021), pp. 101–108. doi:10.1145/3465212. https://doi.org/10.1145/3465212.

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

[12] Niels Ferguson and Bruce Schneier. *The best VPN protocols* (2021). https://nordvpn.com/de/blog/protocols/.

[13] Andrea Fioraldi et al. *AFL++: Combining Incremental Steps of Fuzzing Research*. 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Aug 2020.

[14] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. *Combining Model Learning and Model Checking to Analyze TCP Implementations*. Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 454–471. doi:10.1007/978-3-319-41540-6_25. https://doi.org/10.1007/978-3-319-41540-6%5C_25.

[15] Paul Fiterau-Brostean et al. *Model learning and model checking of SSH implementations*. Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017. Ed. by Hakan Erdogmus and Klaus Havelund. ACM, 2017, pp. 142–151. doi:10.1145/3092282.3092289. https://doi.org/10.1145/3092282.3092289.

[16] AVM Computersysteme Vertriebs GmbH. *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.

[17] Jiaxing Guo et al. *Model learning and model checking of IPsec implementations for Internet of Things*. IEEE Access 7 (2019), pp. 171322–171332.

[18] Hardi Hungar, Oliver Niese, and Bernhard Steffen. *Domain-specific optimization in automata learning*. Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15. Springer. 2003, pp. 315–327.

[19] Malte Isberner, Falk Howar, and Bernhard Steffen. *The Open-Source LearnLib*. Computer Aided Verification. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 487–495. ISBN 978-3-319-21690-4.

[20] Charlie Kaufman et al. *Internet key exchange protocol version 2 (IKEv2)*. Tech. rep. 2014.

[21] Michael J Kearns and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[22] libreswan Project. *libreswan documentation*. Website. 2021. https://libreswan.org/wiki/.

[23] Maughan et al. *Internet Security Association and Key Management Protocol*. RFC 2408. 1998.

[24] Barton P Miller, Lars Fredriksen, and Bryan So. *An empirical study of the reliability of UNIX utilities*. Communications of the ACM 33.12 (1990), pp. 32–44.

[25] Edi Muškardin et al. *AALpy: an active automata learning library*. Innovations in Systems and Software Engineering (2022), pp. 1–10.

[26] Edi Muškardin et al. *AALpy: an active automata learning library*. Innovations in Systems and Software Engineering 18 (Mar 2022), pp. 1–10. doi:10.1007/s11334-022-00449-3.

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

[28] Joshua Pereyda. *boofuzz Documentation*. https://boofuzz.readthedocs.io/. 2022.

[29] Andrea Pferscher and Bernhard K Aichernig. *Fingerprinting Bluetooth Low Energy devices via active automata learning*. International Symposium on Formal Methods. Springer. 2021, pp. 524–542.

[30] Andrea Pferscher and Bernhard K Aichernig. *Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning*. NASA Formal Methods Symposium. Springer. 2022, pp. 373–392.

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

[32] Joeri de Ruiter and Erik Poll. *Protocol State Fuzzing of TLS Implementations*. 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, 2015, pp. 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[33] K. Serebryany. *libFuzzer: a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html. LLVM Project [Online; accessed 19-April-2023]. 2015.

[34] Kostya Serebryany. *OSS-Fuzz-Google's continuous fuzzing service for open source software*. USENIX Security symposium. USENIX Association. 2017.

[35] Muzammil Shahbaz and Roland Groz. *Inferring mealy machines*. FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2. Springer. 2009, pp. 207–222.

[36] Richard Sharpe, Ed Warnicke, and Ulf Lamping. *Wireshark User's Guide*. Accessed April 28, 2023. n.d. https://www.wireshark.org/docs/wsug_html_chunked/.

[37] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. *Extending Automated Protocol State Learning for the 802.11 4-Way Handshake*. Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I. Ed. by Javier López, Jianying Zhou, and Miguel Soriano. Vol. 11098. Lecture Notes in Computer Science. Springer, 2018, pp. 325–345. doi:10.1007/978-3-319-99073-6_16. https://doi.org/10.1007/978-3-319-99073-6%5C_16.

[38] strongSwan Project. *strongSwan documentation*. Website. 2021. https://docs.strongswan.org/docs/5.9/.

[39] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. *Model-Based Testing IoT Communication via Active Automata Learning*. 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017. IEEE Computer Society, 2017, pp. 276–287. doi:10.1109/ICST.2017.32. https://doi.org/10.1109/ICST.2017.32.

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

[41] TOPDapp. *py-diffie-hellman*. https://github.com/TOPDapp/py-diffie-hellman. 2021.

[42] Huan Yang et al. *IKE vulnerability discovery based on fuzzing*. Security and Communication Networks 6.7 (2013), pp. 889–901. doi:https://doi.org/10.1002/sec.628. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.628. https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.628.

[43] Zalewski. *american fuzzy lop*. https://github.com/google/AFL. 2020.

[44] Andreas Zeller et al. *Search-Based Fuzzing*. In: *The Fuzzing Book*. Retrieved 2023-01-07 15:11:42+01:00. CISPA Helmholtz Center for Information Security, 2023. https://www.fuzzingbook.org/html/SearchBasedFuzzer.html.