Benjamin Wunderling[1], BSc

Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

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

Co-Supervisor
Dipl.-Ing. Andrea Pferscher BSc
Institute of Software Technology (IST)

Graz, 05. September 2023

[1] E-mail: benjamin.wunderling@gmail.com
© Copyright 2023 by the author

AFFIDAVIT

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 text document uploaded to TUGRAZonline is identical to the present master's thesis.

. . . . .	. . . . .
Date	Signature

Abstract

In light of the growing number of individuals working from home, the importance of secure Virtual Private Network communication protocols has increased significantly. Consequently, the need for rigorous security testing of Virtual Private Network software has become more critical than ever before. In this thesis, we employ active automata learning to demonstrate how behavioral models of two black-box Internet Protocol Security Virtual Private Network implementations can be automatically learned. We subsequently utilize the learned models to facilitate security testing the Internet Protocol Security implementations through model-based fuzzing. More specifically, we first learn behavioral models of strongSwan and libreswan Internet Protocol Security servers using both the L^* and KV model learning algorithms. Learned models of both Internet Protocol Security implementations are presented and the utilized learning algorithms are compared and evaluated. The learned models are then used to perform model-based fuzzing of the respective implementations. While fuzzing, new behavior is detected by utilizing the previously-learned models as a reference. Additionally, we employ search-based methods, including a genetic algorithm-based approach, aided by the learned models, in order to enhance the effectiveness of fuzzing. Our fuzzing analysis uncovered several new security issues, including several standard violations and a potential deadlock state. Furthermore, our model learning approach led to the discovery of a bug in a cryptographic Python library.

Keywords: IPsec · VPN · Active automata learning · Model-based fuzzing

Kurzfassung

Aufgrund der stetig steigenden Anzahl an Personen, die von zu Hause aus arbeiten, hat die Wichtigkeit sicherer Virtual Private Network-Kommunikationsprotokolle erheblich zugenommen. Infolgedessen ist die Notwendigkeit strenger Sicherheitskontrollen für Virtual Private Network-Software wichtiger denn je zuvor. In dieser Masterarbeit verwenden wir active automata learning, um Verhaltensmodelle von zwei Internet Protocol Security Virtual Private Network-Implementationen, strongSwan und libreswan, automatisiert zu lernen. Dazu verwenden wir die L^* und KV Automatenlern-Algorithmen. Die Verhaltensmodelle und die fürs Lernen verwendeten Automatenlern-Algorithmen werden präsentiert und verglichen. Anschließend werden die erlernten Modelle verwendet um modell-basiertes Fuzzing der jeweiligen Implementierungen zu betreiben. Während dem Fuzzing wird neues Verhalten mithilfe der zuvor erlernten Modelle entdeckt. Des Weiteren setzen wir suchbasierte Methoden ein, inklusive eines genetischen Algorithmus, um die Effektivität des Fuzzings zu verbessern. Unsere Tests haben einige potenzielle Sicherheitslücken enthüllt, darunter mehrere Verstöße gegen Standards und ein potenzieller Deadlock-Zustand. Darüber hinaus hat unser Automatenlernen zu der Entdeckung eines Fehlers in einer kryptografischen Python-Library geführt.

Schlagworte: IPsec · VPN · Aktives Automatenlernen · Modell-basiertes Fuzzing

Acknowledgements

First and foremost, I would like to thank my coadvisor at TU Graz, Dipl.-Ing. Andrea Pferscher, whose patience, advice and draft corrections made this thesis possible.

I would also like to thank Ao.Univ.-Prof. Dipl-Ing. Dr. Bernhard K. Aichernig for his support of my choice in topic and his invaluable feedback and draft corrections.

Last but not least, a special thanks goes to the most important person in my life, Alex, my better half, whose constant motivation and food helped keep me sane during long days of debugging.

Benjamin Wunderling
Graz, Austria, 03. July 2023

Danksagung

Zuallererst möchte ich meiner Zweitbetreuerin an der TU Graz, Dipl.-Ing. Andrea Pferscher, danken, dessen Geduld, Ratschläge und Korrekturen diese Arbeit erst ermöglicht haben.

Des Weiteren möchte ich Ao.Univ.-Prof. Dipl-Ing. Dr. Bernhard K. Aichernig für seine Unterstützung meiner Themenwahl, seine Korrekturarbeit und für sein wertvolles Feedback danken.

Zu guter Letzt möchte ich einen besonderen Dank an die wichtigste Person in meinem Leben richten - Alex, meine bessere Hälfte. Ihre ständige Motivation und Kochkünste haben mir geholfen, selbst bei langen Tagen des Debuggens den Verstand nicht zu verlieren.

Benjamin Wunderling
Graz, Österreich, 03. Juli 2023

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research Problem and Goals . . . 1
1.3 Structure . . . 2

2 Related Work **3**

3 Preliminaries **5**
3.1 Mealy Machines . . . 5
3.2 Automata Learning . . . 6
3.2.1 MAT Framework . . . 6
3.2.2 L^* . . . 7
3.2.3 KV . . . 8
3.3 Fuzzing . . . 8
3.4 IPsec . . . 11

4 Environment Setup **15**
4.1 VM Setup . . . 15
4.2 VPN Configuration . . . 16
4.3 Debugging . . . 19

5 Model Learning **21**
5.1 Learning Setup . . . 21
5.2 Design Decisions and Problems . . . 24
5.3 Combating Non-determinism . . . 26

6 Fuzzing **29**
6.1 Fuzzing Setup . . . 29
6.1.1 Learning the Reference Model . . . 29
6.1.2 Detecting New Behavior . . . 30
6.1.3 Test Data Generation . . . 31
6.1.4 Input-Sequence Generation . . . 31

7 Evaluation **39**
7.1 Learning Results . . . 39
7.1.1 Learning Metrics . . . 40
7.1.2 Learned strongSwan Models . . . 40
7.1.3 Learned libreswan Models . . . 50
7.1.4 Comparing KV and L^* . . . 52
7.1.5 Library Error . . . 53
7.2 Fuzzing Results . . . 54
7.2.1 Findings . . . 54
7.2.2 Comparison of Input-Sequence Generation Methods . . . 58

8 Conclusion **63**
8.1 Summary 63
8.1.1 Discussion 63
8.2 Future Work 64

Appendices

A DOT Files **66**

Bibliography **77**

List of Figures

3.1 Comparison of Mealy (left) and Moore (right) machines. 5
3.2 An example Mealy machine, including its definition as a 6-tuple. 5
3.3 Control flow graph depicting the typical model-learning loop using the minimally adequate teacher framework. 7
3.4 Runtime view of the AFL fuzzer. 9
3.5 Runtime view of the boofuzz fuzzer. 10
3.6 An IKEv1 protocol exchange between two parties. 11

4.1 Pair of VMs running side by side, showing log output. Responder/server on the left, initiator/client on the right. 16
4.2 Internet Protocol Security (IPsec) log excerpt showing keying information. 19
4.3 Decrypting IPsec packets in Wireshark. 20

5.1 Overview of our automata learning setup 21

6.1 Overview of the fuzzing process for a single input sequence. 30
6.2 Overview of the filtering-based input sequence selection method. 33
6.3 Overview of the genetic algorithm-based input-sequence generation method. 37

7.1 First commonly learned model of strongSwan server with retransmissions enabled. . . . 42
7.2 Second commonly learned strongSwan server model with retransmissions enabled. . . . 43
7.3 A rare outlier model, learned with retransmissions enabled. 45
7.4 Clean strongSwan model learned using retransmission-filtering. 47
7.5 (Simplified) strongSwan model with malformed messages in the input alphabet. 48
7.6 Clean libreswan model learned using retransmission-filtering. 50
7.7 (Simplified) libreswan model with malformed messages in the input alphabet. 51

List of Tables

5.1 Mapping protocol to input alphabet names. 22

7.1 KV Runtimes of all the learned strongSwan models. 46
7.2 L^* Runtimes of all the learned strongSwan models. 49
7.3 Runtimes averages of both learning algorithms of all the learned strongSwan models. . . 49
7.4 Comparison of the L^* and KV learning algorithms. 52
7.5 Comparison of input sequence generation runtimes using the strongSwan server. 58
7.6 Comparison of the fitness scores of input-sequence generation methods. 59
7.7 Comparison of generated input-sequence fuzzing runtimes on the strongSwan server. . . 60
7.8 Comparison of input sequence generation and subsequent fuzzing total runtimes using the strongSwan server. 61

Listings

3.1 IKE keying material construction. 12
4.1 Configuration options of the strongSwan server. 17
4.2 Configuration options of the libreswan server. 18
5.1 Model-learning framework initialisation code excerpt. 22
5.2 SUL interface methods code excerpt. 23
5.3 Excerpt of mapper class sa_main method code. 24
5.4 Switching learning algorithms in code. 26
6.1 Search-based input-sequence generation example mutations. 35
7.1 Discovered finding showing the ISAKMP length field being ignored. 55
7.2 Discovered finding showing the Authentication field not being validated. 56

List of Acronyms

AAL active automata learning

AFL american fuzzy lop

AH Authentication Header

DFA deterministic finite automaton

DPD Dead Peer Detection

ESP Encapsulating Security Payload

IKE Internet Key Exchange protocol

IPsec Internet Protocol Security

ISAKMP Internet Security Association and Key Management Protocol

IV initialization vector

MAT minimally adequate teacher

M-ID message-ID

MSB most significant byte

OSS open source software

PFS Perfect Forward Secrecy

PSK pre-shared key

SA Security Association

SUL system under learning

SUT system under test

VM virtual machine

VPN Virtual Private Network

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 [12]. 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 [5] and IKEv2 [10], with IKEv2 being the newer and recommended version, according to a report by the National Institute of Standards and Technology [3]. 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 mainly offering IKEv1 support for their popular FRITZ!Box routers [2]. Additionally, IKEv1 is also used for the L2TP/IPsec protocol, one of the most popular VPN protocols according to NordVPN [25]. 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 Problem 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 [28, 29]. 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 to be a useful technique for automatically generating behavioral models of various communication protocols, e.g., Bluetooth Low Energy [28], TCP [15], SSL [16] or MQTT [38].

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 [1] and the KV algorithm by Kearns and Vazirani [20]. 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.

When testing complex systems, such as network protocol implementations, a popular testing technique is fuzz testing or fuzzing. During fuzzing, a system is bombarded by a large amount of random, invalid or otherwise unusual data, with the goal of triggering unexpected behavior. If such behavior is discovered, it can then be further examined by professionals or other software. Two important aspects while fuzzing are on the one hand, choosing what exactly to fuzz, and, furthermore, how unexpected behavior is defined/discovered by the fuzzer. Especially when fuzzing black-box systems, intelligent fuzzing can be difficult, as usually no information about the underlying source code is available. This makes choos-

ing the fuzzing targets, as well as deciding which responses constitute interesting, unexpected behavior more challenging. The goal of this thesis is to showcase, how automata learning can be combined with fuzzing, in order to intelligently test a target black-box system. To this end, we first learned a model of the target system and subsequently used the learned model for model-based fuzzing. In model-based fuzzing, the learned model of a target black-box system can be used as a frame of reference to determine if unexpected behavior has been successfully discovered.

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, we chose the IPsec-IKEv1 protocol as the focus of this thesis, using automata learning in combination with automata-based fuzzing to automatically learn and test two IPsec server implementation. We used the learning framework AALPY [24] with a custom mapper to actively learn the state machines of two popular 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 (sequences of inputs, where individual inputs are to be fuzzed). Drawing inspiration from search-based testing techniques [44], we implemented search-based and genetic input-sequence generation methods.

In doing so, several deviations from the RFC specification, a potential deadlock state and a cryptographic Python library bug were discovered.

1.3 Structure

This thesis is structured as follows. Chapter 2 gives an overview of the related literature. It focuses on automata learning of security-critical communication protocols, as well as model-based testing. Additionally, an examination of automated testing for VPN protocols, particularly IPsec-IKEv1 and IPsec-IKEv2, is presented. Chapter 3 introduces necessary background knowledge, covering the IPsec-IKEv1 protocol, Mealy machines, automata learning and fuzzing. First Mealy machines are introduced, as they are well suited to model reactive systems such as an IPsec server implementation. Subsequently, automata learning and especially the minimally adequate teacher (MAT) framework is explained, also highlighting the key differences between the two learning algorithms used in this thesis, L^* and KV. Following that, an overview of fuzzing is given, explaining different types of fuzzers and introducing the fuzzer used in this thesis. Finally, the IPsec-IKEv1 protocol is explained in detail, going over a typical exchange between two parties. Chapter 4 covers our VPN setup, IPsec configurations and test environment. Our learning setup, custom mapper and solutions to non-determinism problems are presented in Chapter 5. Code excerpts serve to illustrate key features of our custom mapper class. Following model learning, our fuzzing methodology and input-sequence generation are explained in Chapter 6. In Chapter 7, learned models of both IPsec implementations and the results of the corresponding fuzzing tests are showcased and analyzed. Additionally, a detailed performance comparison between the two used learning algorithms, L^* and KV, is presented. Furthermore, the various methods of input-sequence generation used during fuzzing are evaluated. Finally, Chapter 8 summarizes the thesis and discusses future work.

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 security-critical communication protocols.

Automata or 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 to help guide additional security testing measures, such as fuzzing, or pentesting.

Model learning has been applied to a variety of different protocols, including many security-critical ones. When applied to security-critical communication protocols, the model learning is often called protocol-state fuzzing. De Ruiter and Poll [7] 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. [38] 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. [36] using automata learning to test implementations on various routers, finding servery vulnerabilities. Fiterau and Brostean combined model learning with model checking, in which an abstract model is checked for specified properties to ensure correctness and security. In their work, they learned and analyzed both TCP [15] and SSL [16] implementations, showcasing several implementation deviations from the respective RFC specifications. The Bluetooth Low Energy (BLE) protocol was investigated by Pferscher and Aichernig [28]. 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. [26] and Daniel et al. [6] performed protocol-state fuzzing of the OpenVPN protocol. 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. Novickis et al. discovered several non security-critical contradictions between the official documentation and actual implementation.

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. Through their work, they discovered several deviations from the RFC specification. 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 [2].

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 [41]. Additionally, several popular IPsec libraries, including strongSwan, utilize the open source software (OSS) fuzzing framework, OSS-Fuzz [33], ensuring that they are regularly fuzzed using the LibFuzzer, AFL++ and Hongfuzz fuzzing libraries [32, 14, 11].

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

based on metaheuristics 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 model-based testing approaches for both IKE versions.

The model-learning section of this thesis builds upon our prior work, published and presented as part of the Workshop on Applications of Formal Methods and Digital Twins [23]. This thesis substantially expands on the previous work by learning additional models and leveraging the learned models for the purpose of model-based fuzzing.

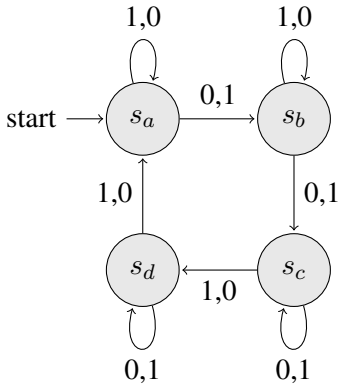
3 Preliminaries

3.1 Mealy Machines

Mealy machines are a type of deterministic finite-state machine commonly used to model reactive systems such as communication protocols. In Mealy machines, each state transition is labeled with and input and a corresponding output. In other words, transitions between states are triggered by inputs and also generate an output. The outputs depend on both the current state and the received input. Due to their ability to model both inputs and outputs, Mealy machines are an especially useful tool for modeling reactive systems, systems that react to inputs with outputs, such as communication protocols. In contrast, Moore finite-state machines define their outputs only through the current state. Figure 3.1 showcases this difference, with the Mealy machine on the left having input/output pairs for each transition, while the Moore machine on the right only has outputs.

Figure 3.1: Comparison of Mealy (left) and Moore (right) machines.

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 next state is defined by the current state and an 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. Figure 3.2 shows an example Mealy machine. The state-machine and its corresponding 6-tuple definition are shown side-by-side.



$$
\begin{aligned}
S &= \{s_a, s_b, s_c, s_d\} \\
s_0 &= s_a \\
\Sigma &= \{0, 1\} \\
O &= \{0, 1\} \\
\delta &= \{ \\
&\quad s_a \times 0 \to s_b,\ s_a \times 1 \to s_a, \\
&\quad s_b \times 0 \to s_a,\ s_b \times 1 \to s_b, \\
&\quad s_c \times 1 \to s_d,\ s_c \times 0 \to s_c, \\
&\quad s_d \times 1 \to s_a,\ s_d \times 0 \to s_d, \\
&\quad \} \\
\lambda &= \{ \\
&\quad s_a \times 0 \to s_b,\ s_a \times 1 \to s_a, \\
&\quad s_b \times 0 \to s_a,\ s_b \times 1 \to s_b, \\
&\quad s_c \times 1 \to s_d,\ s_c \times 0 \to s_c, \\
&\quad s_d \times 1 \to s_a,\ s_d \times 0 \to s_d, \\
&\quad \}
\end{aligned}
$$

Figure 3.2: An example Mealy machine, including its definition as a 6-tuple.

State transitions are shown as a mapping between state x input tuples and a corresponding next state. For example, when in state s_a, the input 0 leads to the state s_b. Similarly, the output function δ maps the same state x input tuple to the corresponding output, namely 1. In this thesis, Mealy machines, due to their ability to express dependencies between both inputs and outputs, are used to model the state of learned IPsec implementations.

3.2 Automata Learning

Automata learning refers to methods of learning a behavioral model of a system through an algorithm or process. The learned model 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 active automata learning (AAL) the SUL is queried directly. In this paper, we will focus on AAL and will, moving on, be referring to it as automata learning or AAL interchangeably.

One of the most influential AAL algorithms was introduced in 1987 by Dana Angluin, titled "Learning regular sets from queries and counterexamples" [1]. In this seminal paper, Angluin introduced the concept of the 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, 34]. 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 output and equivalence queries. Note that in the original L^* paper, the term membership query is used instead of output queries. The term output queries is used when learning Mealy machines. On a conceptual level, the learner poses output queries to the teacher in order to build a behavioral model of the SUL and equivalence queries, to verify if the model accurately describes the behavior of the system. Output queries consist of an input $s \in \Sigma^*$, where Σ is the input alphabet. The teacher receives the output 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, output 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 output queries, the learner constructs a Mealy machine representation of the SUL and proposes it to the teacher as an equivalence query. The teacher must confirm if the proposal is equivalent to the SUL, answering with "yes" if the equivalence holds true, or else returning a counterexample that represents an input sequence that reveals the behavioral difference between the provided model and the SUL, 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 output queries and proposing a model in an equivalence query until it is confirmed to be correct is showcased in Figure 3.3. We can see that the algorithm does not end until the teacher confirms that the proposed model is equivalent.

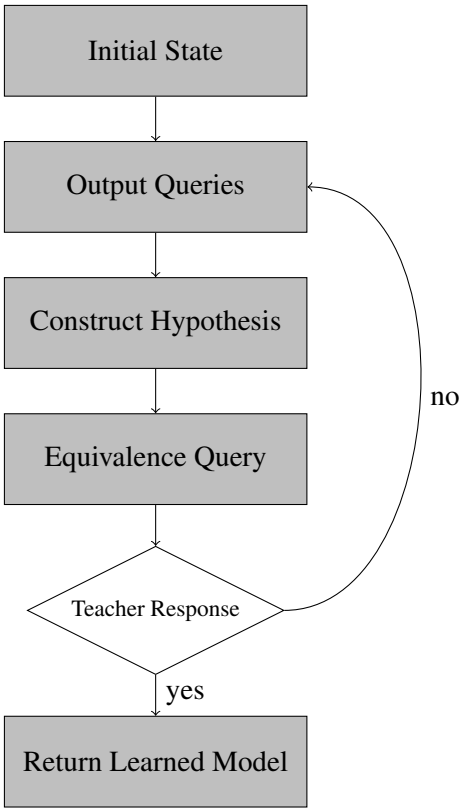


Figure 3.3: Control flow graph depicting the typical model-learning loop using the minimally adequate teacher framework.

3.2.2 L*

L^* uses the MAT framework and stores the results of the output queries in a data structure called the observation table. Once the observation table has been filled with enough information on the SUL, it is used to construct a model. The model is then proposed to the teacher by performing an equivalence query. 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 one of these properties is violated, the table must be updated and filled through further output 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 Python learning library AALPY [24]. While many modern implementations, including AALPY, use improved versions of L^*, such as with advanced counterexample processing by Rivest and Schapire [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 [20]. Like L^*, it also uses the MAT framework, but aims to minimize the size of the underlying data structure 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 output 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 the tree data structure. This property makes the KV algorithm particularly attractive for scenarios in which output 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 [22], it can be applied to test the reliability and security of any system that takes input. While originally just simple random text-generation programs, 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.

- 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 of 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 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.

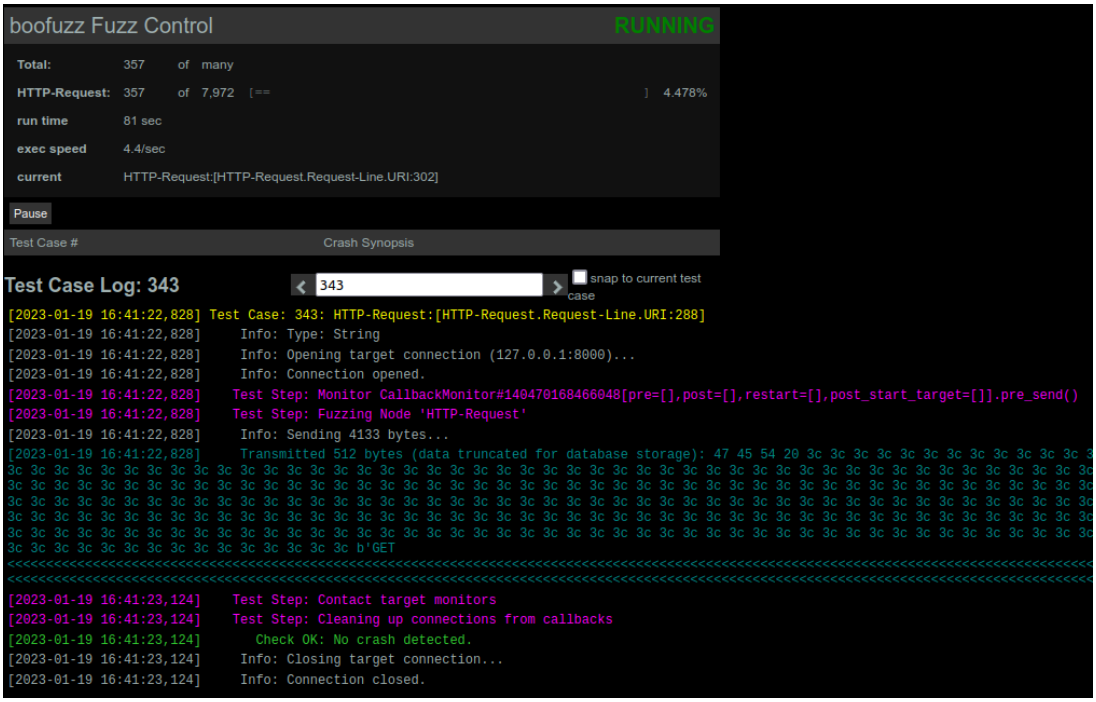
```
                    american fuzzy lop 1.86b (test)

┌─ process timing ──────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec         │  cycles done : 0      │
│   last new path : none seen yet                       │  total paths : 1      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 2 sec         │ uniq crashes : 1      │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 2 (0.00%)             │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 1 (100.00%)            │
│ stage execs : 1464/5000 (29.28%)    │  new edges on : 1 (100.00%)            │
│ total execs : 1697                  │ total crashes : 39 (1 unique)          │
│  exec speed : 626.5/sec             │   total hangs : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/16, 1/15, 0/13                      │    levels : 1          │
│  byte flips : 0/2, 0/1, 0/0                         │   pending : 1          │
│ arithmetics : 0/112, 0/25, 0/0                      │  pend fav : 1          │
│  known ints : 0/10, 0/28, 0/0                       │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 0/0, 0/0                              │  variable : 0          │
│        trim : n/a, 0.00%                            ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu: 92%]
```

Figure 3.4: Runtime view of the AFL fuzzer.

Two popular examples of fuzzers are american fuzzy lop (AFL) [43] and boofuzz [27]. 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. A runtime screenshot of AFL fuzzing can be seen in Figure 3.4 [8]. It shows various metrics, including measures for coverage and various fuzzing strategies. 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 value. 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. An example screenshot of boofuzz fuzzing a network protocol can be seen in Figure 3.5. It shows the long fuzzed message being sent to the SUT. In this thesis we use boofuzz primitives to generate our values for fuzzing, as detailed in Chapter 6.

[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


boofuzz Fuzz Control
RUNNING
Total: 357 of many
HTTP-Request: 357 of 7,972 [==] 4.478%
run time 81 sec
exec speed 4.4/sec
current HTTP-Request:[HTTP-Request.Request-Line.URI:302]
Pause
Test Case #
Crash Synopsis
Test Case Log: 343
343
snap to current test case
[2023-01-19 16:41:22,828] Test Case: 343: HTTP-Request:[HTTP-Request.Request-Line.URI:288]
[2023-01-19 16:41:22,828] Info: Type: String
[2023-01-19 16:41:22,828] Info: Opening target connection (127.0.0.1:8000)...
[2023-01-19 16:41:22,828] Info: Connection opened.
[2023-01-19 16:41:22,828] Test Step: Monitor CallbackMonitor#140470168466048[pre=[],post=[],restart=[],post_start_target=[]].pre_send()
[2023-01-19 16:41:22,828] Test Step: Fuzzing Node 'HTTP-Request'
[2023-01-19 16:41:22,828] Info: Sending 4133 bytes...
[2023-01-19 16:41:22,828] Transmitted 512 bytes (data truncated for database storage): 47 45 54 20 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c
3c 3c 3c 3c 3c 3c 3c 3c 3c 3c

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.

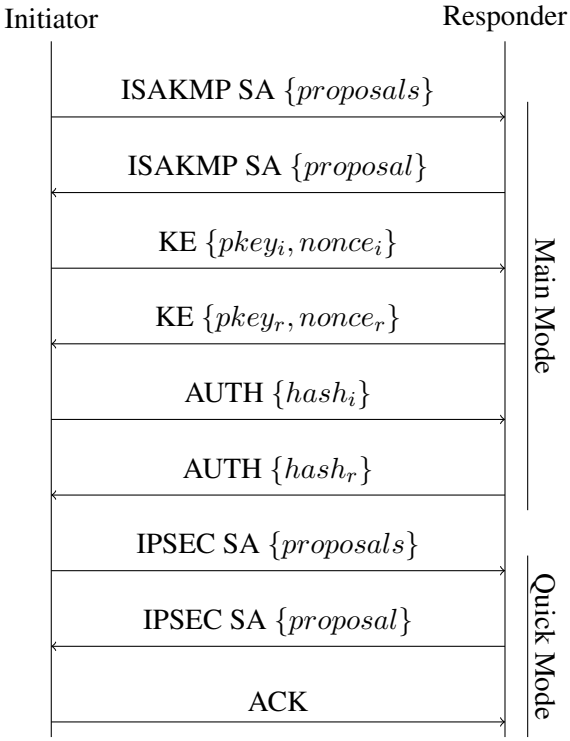


Figure 3.6: An IKEv1 protocol exchange between two parties.

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 3.6. 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. Each proposal describes 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 3.6 and also exchanges initiator/responder cookies.

These are 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 3.1, Line 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`, seen on lines 5, 8 and 11 respectively. Here, g^{xy} refers 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. As previously mentioned, if phase one uses Aggressive Mode, only three packets are needed to reach phase two and 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 3.6. 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 3.1: IKE keying material construction.

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

In addition to the packets shown in Figure 3.6, IKEv1 also specifies and uses so called ISAKMP Informational Exchanges. Informational exchanges in IKEv1 are used to send ISAKMP *Notify* or ISAKMP *Delete* payloads. Following the key exchange in phase one, all Informational Exchanges are sent encrypted and authenticated. Prior, they are sent in plain. ISAKMP *Notify* payloads are used to transmit various error and success codes, as well as for keep-alive messages. ISAKMP *Delete* is used to inform the other communication partner that a SA has been deleted locally and request that they do the same, effectively closing a connection.

Compared to other protocols, IPsec offers a high degree of customizability, allowing it to be fitted for many use cases. However, in a cryptographic evaluation of the protocol, Ferguson and Schneier [13] 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 [10]. IPsec-IKEv2 was presented as a simpler replacement for IPsec-IKEv1, with an IKEv2 exchange requiring only four messages, compared to the nine of its predecessor. Nevertheless, IPsec-IKEv1 is still in wide-spread use to this day, with the largest router producer in Germany, AVM, until recently only supporting IKEv1 in their routers [2]. 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 due to it containing the majority of cryptographic operations.

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, strongSwan and libreswan, 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 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 10.0.2.1 and the client to use the 10.0.2.2 IP addresses respectively. We use the 10.0.2.0 network (255.255.255.0 subnet mask) with 10.0.2.0 as our default gateway. VirtualBox handles all of the internal routing, provided the two VMs are in the same internal network. The libreswan setup 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 more detail in Chapter 5.

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

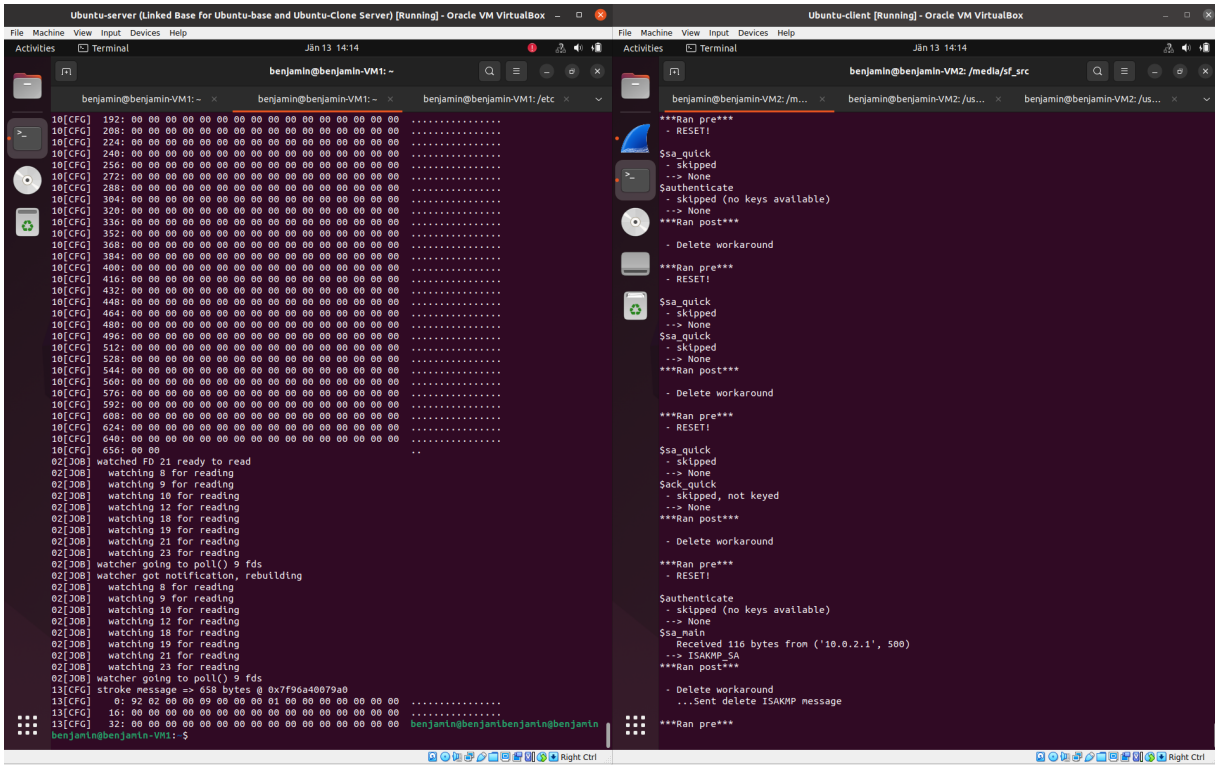


Figure 4.1: Pair of VMs running side by side, showing log output. Responder/server on the left, initiator/client on the right.

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 4.1 shows such a pair of VMs being run side by side. The left VM is running the IPsec server and the right VM is running the model-learning code.

4.2 VPN Configuration

The two IPsec implementations learned and tested were strongSwan U5.9.5/K5.15.0-25-generic and libreswan U3.32/K5.15.0-41-generic (U referring to the userspace and K to the kernel portions of the ipsec implementation). Both are popular open source IPsec implementations, with strongSwan featuring support for Linux, Android, FreeBSD, Apple OSX and Windows [37] and being the more widespread choice of the two. The libreswan IPsec implementation supports Linux, FreeBSD and Apple OSX [39]. 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 IKEv2, 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. In contrast, strongSwan actually supports two types of configuration files. One more modern one using the Versatile IKE Control Interface (VICI), 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 and 4.2.

```
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: Configuration options of the strongSwan server.

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 implementations whether 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 and to not have the server ignore identical IDs. These settings apply to all connections configured in the configuration file.

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* (Listing 4.1) or *ikev2* (Listing 4.2) 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 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". In our case, the connections are configured to use $256\,\mathrm{bit}$ 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 config-

uration, 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 serve both 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: Configuration options of the libreswan server.

As can be seen in listings 4.1 and 4.2, the two configuration files only differ in a 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 “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 the 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 mode) 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.

4.3 Debugging

During any software development process, roughly 35-50% of development time is on the testing and validation of the software [4]. The development of our custom mapper and fuzzer required a very large amount of debugging, leaning more towards the 50% 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 or libreswan-generated and our custom-generated IPsec packets had to be performed on a byte-by-byte level. The open-source packet sniffing tool Wireshark [35] 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 made analyzing the packets impossible, as the relevant information would be encrypted. Fortunately, 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 4.2, 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).

```
[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.

Using the encryption key as well as the cookie of an IKE connection, the packets can be decrypted in 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. To solve these issues, one of our most successful strategies was to manually compare the actual IPsec client to IPsec server connection packets, byte by byte, with those generated by our custom mapper class. While very time-consuming and tedious, it did give a very low-level understanding of the protocol, as well as a greater appreciation for the strongSwan/libreswan projects and their vast array of features. 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[2]. Fortunately, all the effort invested into debugging our implementation resulted in a very detailed logging and testing toolkit for all parts of the implemented protocol. Additionally, once completed, the mapper class greatly reduced the time needed for further testing and fuzzing of the IPsec servers due to its flexible implementation.

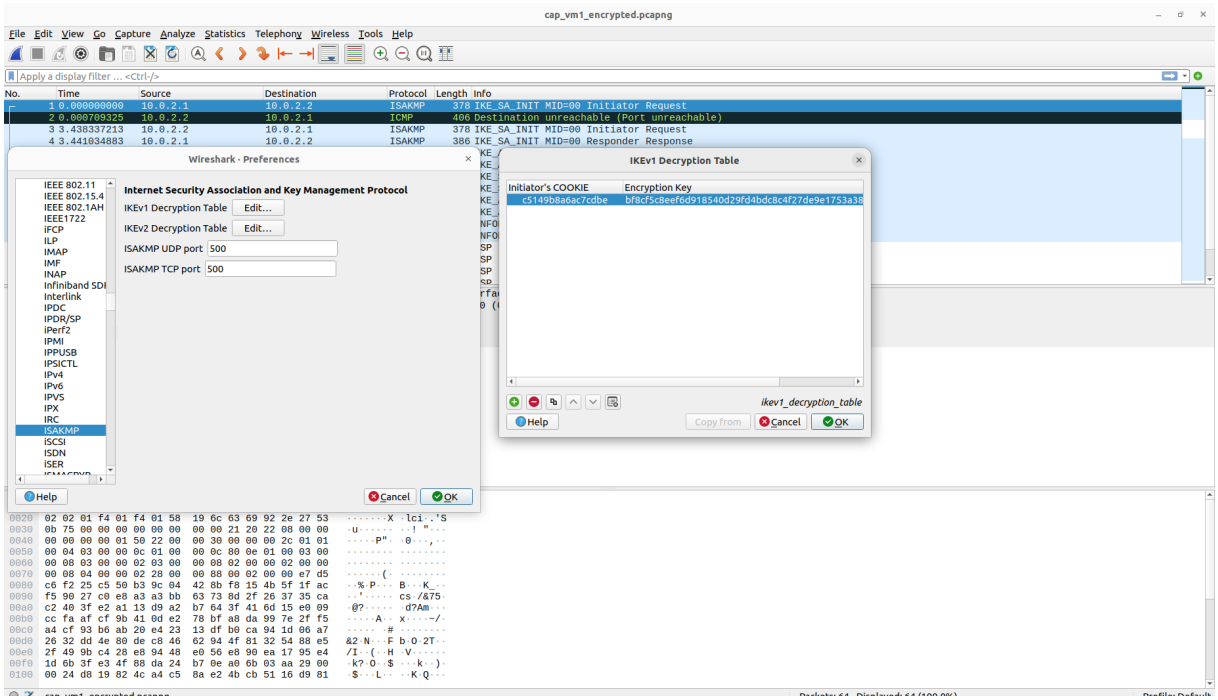


Figure 4.3: Decrypting IPsec packets in Wireshark.

[2] `https://github.com/benjowun/VPN-AL`

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 in order to, going forward, use terminology better suited for 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* or *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 [24] 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 script *IPSEC_IKEv1_SUL* (see supplementary material[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.

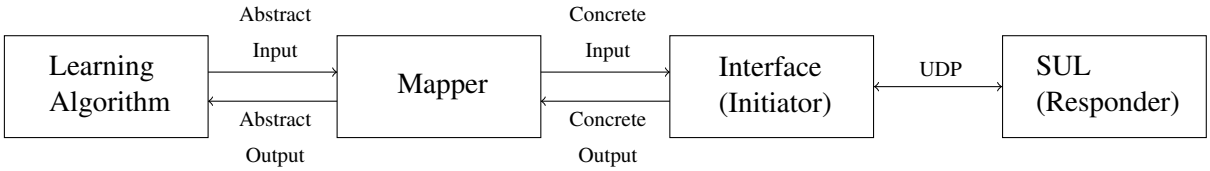


Figure 5.1: Overview of our automata learning setup

[1] https://scapy.net/
[2] https://github.com/benjowun/VPN-AL

Figure 5.1, adapted from Tappler et al. [38], 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 3.6. 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 which 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, denoted as *in*, Line 4, the used equivalence oracle and Line 6, the used learning algorithm. Both the equivalence oracle and learning algorithm take the input alphabet and an object representing an interface to the SUL as parameters, were the SUL interface is defined as shown below in Listing 5.2 and can execute inputs on, as well as reset 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  in = ['sa_main', 'key_ex_main', 'authenticate', 'sa_quick', 'ack_quick']
4  eq_oracle = StatePrefixEqOracle(in, sul, walks_per_state=10, walk_len=10)

5
6  learned_ipsec = run_Lstar(in, sul, eq_oracle=eq_oracle, automaton_type='
       mealy', cache_and_non_det_check=True)
```

Listing 5.1: Model-learning framework initialisation code excerpt.

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. An output query is a sequence of inputs, chosen from the input alphabet, that is executed on the SUL. Each abstract input of this sequence 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. Every sequence of inputs is executed starting from an initial state. Consequently, a *reset* method is required, in order to revert the SUL back to the initial state. Used in combination, *step* and *reset* allow asking output queries to the SUL. In our SUL interface, the *reset* function is split into both a *pre* and a *post* half. *pre*, seen on Line 8, is called before each output query and ensures that our local state is reset. *post*, seen on on Line 11, is called afterwards and ensures that the SUL is also returned to an initial state.

```
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 methods code excerpt.

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 and only used for *reset* purposes. 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 wrapped in 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 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 and onward.

```
# code excerpt from IPSEC_MAPPER.py

def sa_main(self, ...):
  ...
  tf = [('Encryption','AES-CBC'), ('KeyLength',256), ('Hash','SHA'), ...]

  sa_body_init = ISAKMP_payload_SA(...,ISAKMP_payload_Proposal(...,trans=
      ISAKMP_payload_Transform(..., transforms=tf)))

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

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

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

Listing 5.3: Excerpt of mapper class 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), indicates 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 to 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* packets do 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 it 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 effortlessly 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. For 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. The 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 3 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, ...)
5
6  else:
7    learned_ipsec, info = run_Lstar(input_al, sul, eq_oracle, ...)
```

Listing 5.4: Switching learning algorithms in code.

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 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 [21], 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 fuzzing reference models, as shown in Chapter 7, Figure 7.4 and Figure 7.7. 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 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.

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 new 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, new behavior 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 the test data for fuzzed inputs is generated. The following subsections cover the creation of the reference model, as well as two methods of input-sequence 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 new behavior. Instead of using the model describing only expected behavior, a new model was learned using retransmission-filtering. Moreover, an expanded input alphabet was utilized, constructed as follows. In addition to the previous input alphabet, an erroneous version of each input was added, mapping 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 an unsupported SA option.

[1] `https://github.com/benjowun/VPN-AL`

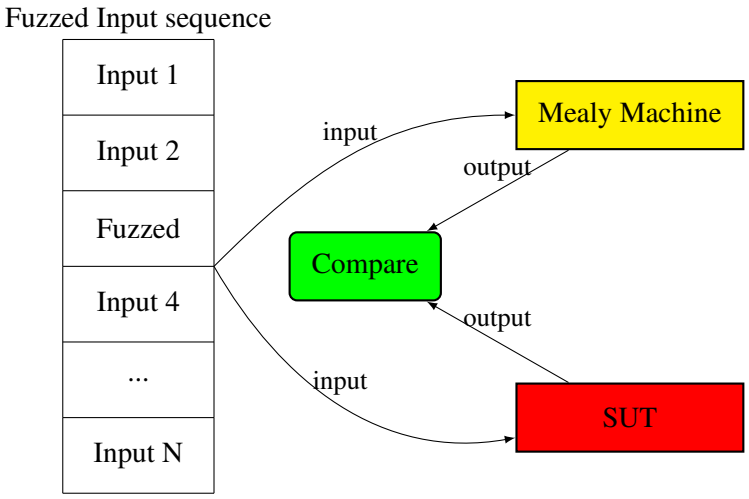


Figure 6.1: Overview of the fuzzing process for a single input sequence.

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 generally, not much had to be changed in our mapper class to allow for the sending of erroneous packets.

6.1.2 Detecting New Behavior

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 behavior, i.e., new responses and transitions not in the reference model. 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, 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 behavior triggered 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. 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 that 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 behavior 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 the discovery of said new behavior. 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 behavior 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 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

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

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. The second method generates a new input sequence using a search-based approach, that is designed to lead to the discovery of as much new behavior as possible when fuzzed. Finally, a genetic algorithm for input-sequence generation is showcased. The three methods are presented in more detail below.

Filtering-based Input-Sequence Generation

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 are generated during model learning, guaranteeing state-coverage (at least for the learned automata), these input sequences can be repurposed for fuzzing. To this end both output and equivalence queries from learning the reference model were recorded, giving us an extensive suite of input sequences to fuzz. Note that since caching was enabled, cache hits meant that particular input sequence was not recorded. However, as cache hits indicate that no new information would be learned from the input in question, inputs skipped due to caching can be safely ignored. Be that as it may, there still was a significant problem with this method of fuzzing input-sequence generation, namely that the resulting set of input sequences was very large. 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, depending on how thoroughly the SUT is to be tested) 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 phase 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 behavior is increased. In other words, we can no longer guarantee full state-coverage of the SUT. 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 behavior, due to new cookies indicating a completely new connection. Since our implementation learned the model with static initiator cookies, this will always lead to the discovery of new behavior.

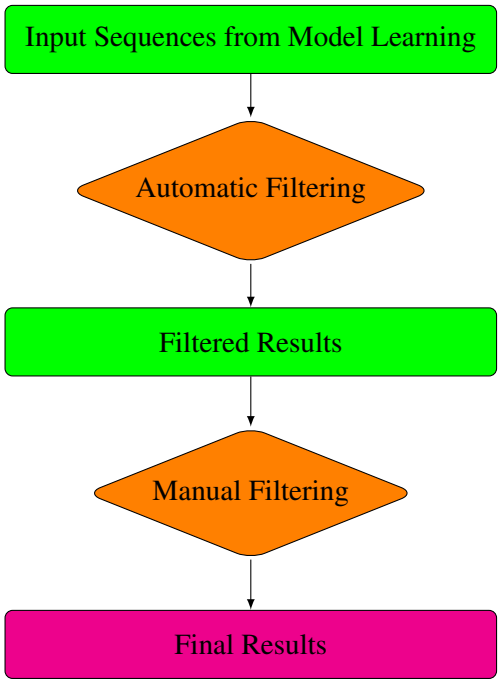


Figure 6.2: Overview of the filtering-based input sequence selection method.

Finally, following the automatic and manual filtering, 55 input sequences of various lengths remained for the strongSwan server, compared to the 220 from before filtering. The filtered list of input sequences can be found in our supplementary material[4] 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. Additionally, the resulting list of input sequences can serve as a good test-suite, giving a large degree of coverage over the behavior of the SUT. However, fuzzing 55 distinct input sequences takes a considerable amount of time (several days), 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. Due to the long runtime, the filtering-based input sequence selection method was only performed for the strongSwan SUT. While the selection of input sequences could certainly be optimized further, e.g., 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.

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 55 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, new information. To this end, we propose a search-based input-sequence generation approach.

[4] `https://github.com/benjowun/VPN-AL`

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 new behavior, 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 (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 new behavior, 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 score are accepted. The condition *higher or equal* was chosen in order to allow the algorithm to overcome plateaus (ares with equal fitness scores) in the search landscape. Intuitively, the fitness score given should serve as a representation of how easy it is to find new behavior while fuzzing a given input sequence, as well as being weighted slightly, to encourage the exploration of new behavior. To calculate the fitness of an input sequence, each fuzzable field of every input in the input sequence is tested using a minimal list of fuzzing values (also used in the initial filtering phase of the filtering-based approach). The amount of new behavior (states/transitions/return-values not in the reference model) discovered, while minimally fuzzing each field of each input of the input sequence, is recorded and referred to as b_{new}. In other words, an input sequence is fuzzed with a reduced pool of fuzz-values, and the amount fuzzed values leading to new behavior is referred to as b_{new}. The fitness of an input sequence f_{seq} is calculated as the sum of the number of new behavior found for each input b_{new}, divided by the number of inputs in the input sequence n, multiplied by the percentage of total states of the reference model visited during score calculation, as seen in Equation 6.1.4. The last fraction, $\frac{s_{\text{visited}}}{s_{\text{total}}}$, is effectively the state coverage of the reference model, that fuzzing the input sequence will achieve. This helps encourage the algorithm to visit new states and not just stay in the initial one should it finds a large number of new behavior there.

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

If the fitness score of a changed input sequence is better or equal than its predecessor, the new input sequence is accepted and the algorithm continues with it as its base sequence. We use the criteria "greater than or equal" when comparing fitness scores, to increase the chance of finding high-scoring input sequences hidden behind equal fitness areas. As a stopping criterion, we simply specify a number of mutations the input sequence is to undergo, after which the current best performing sequence is returned. Future work could look to improve this criterion, e.g., by specifying a minimum fitness level that is to be achieved. The fitness score is calculated once after each mutation.

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 fuzzed using a reduced set of fuzz-values. In comparison, in the filtering-based approach, only a randomly chosen input in each input sequence was fuzzed. 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. 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, as opposed to the 55 input sequences after filtering. However, this single input sequence will, on average, find significantly more interesting new behavior. 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 rationale 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. Line 2, shows a sequence corresponding to phase one of the IKE protocol. 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` in Line 11. Finally, a `key_ex_main_err` is inserted between `sa_main_err` and `authenticate` in Line 14. As this change leads to a decrease in fitness, the change is rejected and the search continues using the previous input sequence. The algorithm terminates after the specified number of mutations, so in this case, after 50.

```
1   ...
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 example mutations.

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 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. The results are discussed in more detail in Chapter 7.

One significant shortcoming of the search-based method is that it is prone to getting stuck in local maxima while attempting to optimize an input sequence w.r.t. its fitness score. While getting stuck in local maxima is somewhat combated by weighting the mutation operations in such a way as to promote visiting new states, a scenario where the algorithm repeatedly cycles between two mutations is certainly possible. Still, one can argue that a local maxima still is worth fuzzing, as it will feature more interesting behavior than other, similar input sequences, making it a good fuzzing target. Nevertheless, getting stuck in local maxima can lead to the resulting inputs sequence not covering the full behavior of the SUT. Therefore, to minimize the chances of getting stuck in such local maxima, we implemented a genetic algorithm-based approach.

Genetic Input-Sequence Generation

As an evolution of our search-based approach, we implemented a genetic algorithm-based approach of input-sequence generation. It improves on search-based method with regards to test coverage, fitness score and resistance to getting stuck in local maxima. As explained in Chapter 3, genetic algorithms work by simultaneously cultivating many different populations, subjecting them to small mutations and generating new populations by crossing the best-performing existing ones. Only the populations that benefited the most from the mutations, decided based on the respective fitness scores before and after the mutation, are kept after each round of mutations. The other populations are discarded. The surviving populations are then crossed with each other, generating new populations that are essentially children of two well-performing parent populations. Finally, randomly-generated populations are added to the group in order to introduce more randomness and to promote more extreme mutations.

Our implemented genetic algorithm-based approach works analogously, with each population being an input sequence. At the start, an initial set of input sequences is randomly generated or specified. This population is depicted in orange in Figure 6.3. Subsequently, until the maximum number of iterations is reached, the populations are repeatedly mutated and scored, using the same fitness function as in the search-based approach. Figure 6.3 shows mutation in yellow and scoring in red. The main difference to the search-based approach is that, instead of only working with one input sequence, here many different ones are processed in parallel (technically, our search-based approach is a randomized hillclimber algorithm, with our genetic algorithm-based approach being very similar, but with multiple populations instead of only one). After each scoring round, a specifiable percentage of the most fit new input sequences is chosen to survive the round. The rest are discarded. The surviving input sequences are used to generate crossover input sequences, shown in green in Figure 6.3, with the crossover containing half of the input sequence of each parent. Any remaining empty population slots are then filled with new randomly generated populations, of a random length between one and and slightly longer than the currently longest sequence. Finally, the algorithm starts over, using the current set of populations as the new starting sequences. The algorithm terminates after a specified number of repetitions of these steps, returning the remaining populations/input sequences.

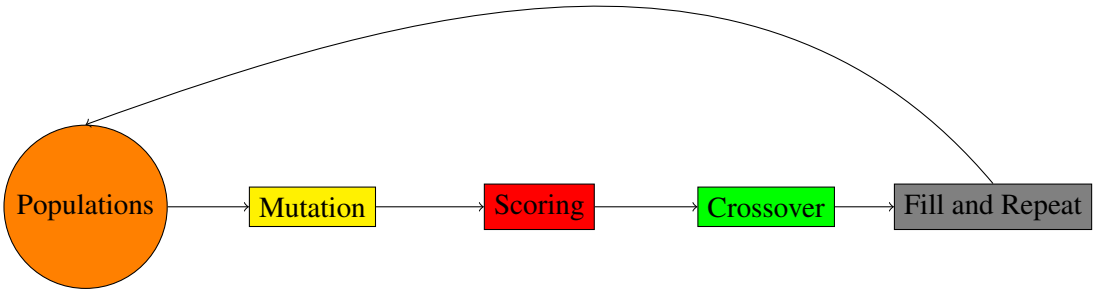


Figure 6.3: Overview of the genetic algorithm-based input-sequence generation method.

Using this approach, well-performing input sequences can be crossed to, hopefully, generate even better performing child input sequences. Additionally, through the continuous addition of randomly generated input sequences, as well as the large amount of populations being mutated at a time, the genetic approach has a much more rapid rate of change than its single-population search-based counterpart. Furthermore, mutations that do occur tend to being more intense (meaning large changes) mutations than the single-sequence search-based approach, presumably also due to the addition of random input sequences and the crossing of populations. Additionally, due to the noise added by the randomly generated populations, the genetic method is more resistant to getting stuck in local maxima (again going back to the greater rate of change, giving a higher chance of escaping local maxima). As a consequence, for similar runtimes, the genetic algorithm generates considerably better-scoring input sequences, when compared to its single-population search-based counterpart. While the genetic algorithm-generated input sequence did not discover any additional findings when fuzzed, the approach seems promising, as it outperforms the search-based approach given a similar runtime. Both the search-based, as well as the genetic algorithm-based approaches heavily outperformed randomly generated baseline input sequences, both with regards to calculated fitness, as well as with regard to the discovered findings. Chapter 7 features a full comparison of all used input-sequence generation methods, contrasting the different input-sequence generation methods with regard to their runtime, as well as the runtime when fuzzing their generated input sequences.

7 Evaluation

This chapter presents the results of our model learning and model-based fuzzing for IPsec-IKEv1. Model learning results are presented in Section 7.1, beginning with the models learned without retransmission-filtering. All model learning and testing took place in a virtual environment using two VirtualBox 6.1 VMs running standard Ubuntu 22.04 LTS distributions, as described in 4. Next, models of both examined IPsec implementations, with retransmission-filtering enabled, are presented and discussed, highlighting differences between the models of the two IPsec implementations. 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 for both IPsec servers 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 of both IPsec servers, due to varying retransmission-handling settings and choices of the input alphabet. 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 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.

The following sections first introduce the most relevant metrics used to evaluate the learned models. Next, the two most commonly learned models without retransmission-filtering, both learned from a Linux strongSwan U5.9.5 server, are presented. Finally, models of both servers using the basic and extended input alphabets with retransmission-filtering enabled are presented. All models were learned using both the KV and L^* learning algorithms. Error codes have been simplified for better readability and DOT files of all models are provided in Appendix A, as well as in the supplementary material[1]. Note that AALPY refers to output queries as membership queries.

[1] `https://github.com/benjowun/VPN-AL`

7.1.1 Learning Metrics

The comparison of learned models and model learning algorithm performance in sections 7.1.2 and 7.1.4 is based largely on the following metrics, saved during the model learning process.

Steps

Steps refers to the number of inputs executed on the SUL. Referred to as steps, input steps or inputs interchangeably.

Queries

Queries refers to the amount of queries sent during state exploration (output queries) or during conformance checking (equivalence queries). AALPY supports speeding up model learning by using caching to reduce the number of required output 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 input steps and queries. When given in seconds, the runtime is rounded to the nearest second.

7.1.2 Learned strongSwan Models

Figures 7.1 and 7.2 show the two most commonly learned strongSwan 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, an example of which is given in Figure 7.3. Figure 7.4 shows the clean base model learned from the strongSwan SUL with retransmission-filtering enabled. The strongSwan reference model used for fuzzing is shown in Figure 7.5, also learned with retransmission-filtering enabled, as well as an expanded input alphabet. Figure 7.6 shows the clean base model learned from the libreswan server. Figure 7.7 shows the corresponding libreswan fuzzing reference model.

The strongSwan runtimes of both learning algorithms for all four models are summarized in tables 7.1 and 7.2. All values are averages over multiple learning attempts. Note that, while all listed values are averages, the strongSwan clean base model was learned significantly more often than the other models, making the corresponding statistics the most accurate. This is due to the fact that these models were used as the basis of the comparison between the KV and L^* learning algorithms, as presented in Section 7.1.4. Consequently, as the learning of strongSwan clean base models served a dual purpose, their learning was prioritized. Other models were learned as often as time and resources allowed, usually around ten times per model. The libreswan runtimes were omitted from the statistics, as a workaround had to be used to reset the IPsec host VM between output queries, potentially skewing the results. Relevant statistics, such as the number of required input steps (i.e. the number of inputs executed on the SUL) and queries are still presented. The 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. OQ: Number of output queries sent during the model learning
6. CT: Size of the conformance testing suite, i.e. the number of performed conformance tests

First Common Model

The first common model, learned from the strongSwan server, is presented in Figure 7.1 and took approximately 52 minutes (3092 seconds) to learn with the KV algorithm, spread over seven learning rounds. The model consists of ten states. Of the 52 minutes total, roughly half were used for state exploration/output queries and the other half for conformance checking, with conformance checking taking slightly longer (1501 vs 1591 seconds). 171 output queries were performed by the learning algorithm in 2047 steps, whereas 100 conformance tests were performed for equivalence 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 output queries (2922 steps) were required compared to the 171 queries 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/output queries is quite large. However, conformance checking required only 1379 inputs steps than with the KV algorithm. This behavior is discussed in more detail in Section 7.1.4, 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 with the transition from state *s3* to *s4*. While phase one looks very clean and is in fact identical to the model learned with retransmission-filtering enabled, phase two has many unexpected 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 to be 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 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 ISAKMP delete to the input alphabet creates 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.

Second Common Model

The second common model of the strongSwan server, 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 twelve states. Of those 75 minutes, roughly 53% were used for state exploration/output queries and the other 47% (2382 vs 2126 seconds). 215 output queries were performed by the learning algorithm, requiring 2219 steps, whereas 120 conformance tests were performed, requiring 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/output queries is even larger, with L^* taking 522 output queries. State exploration required 4225 inputs to complete, whereas conformance checking required 1745.

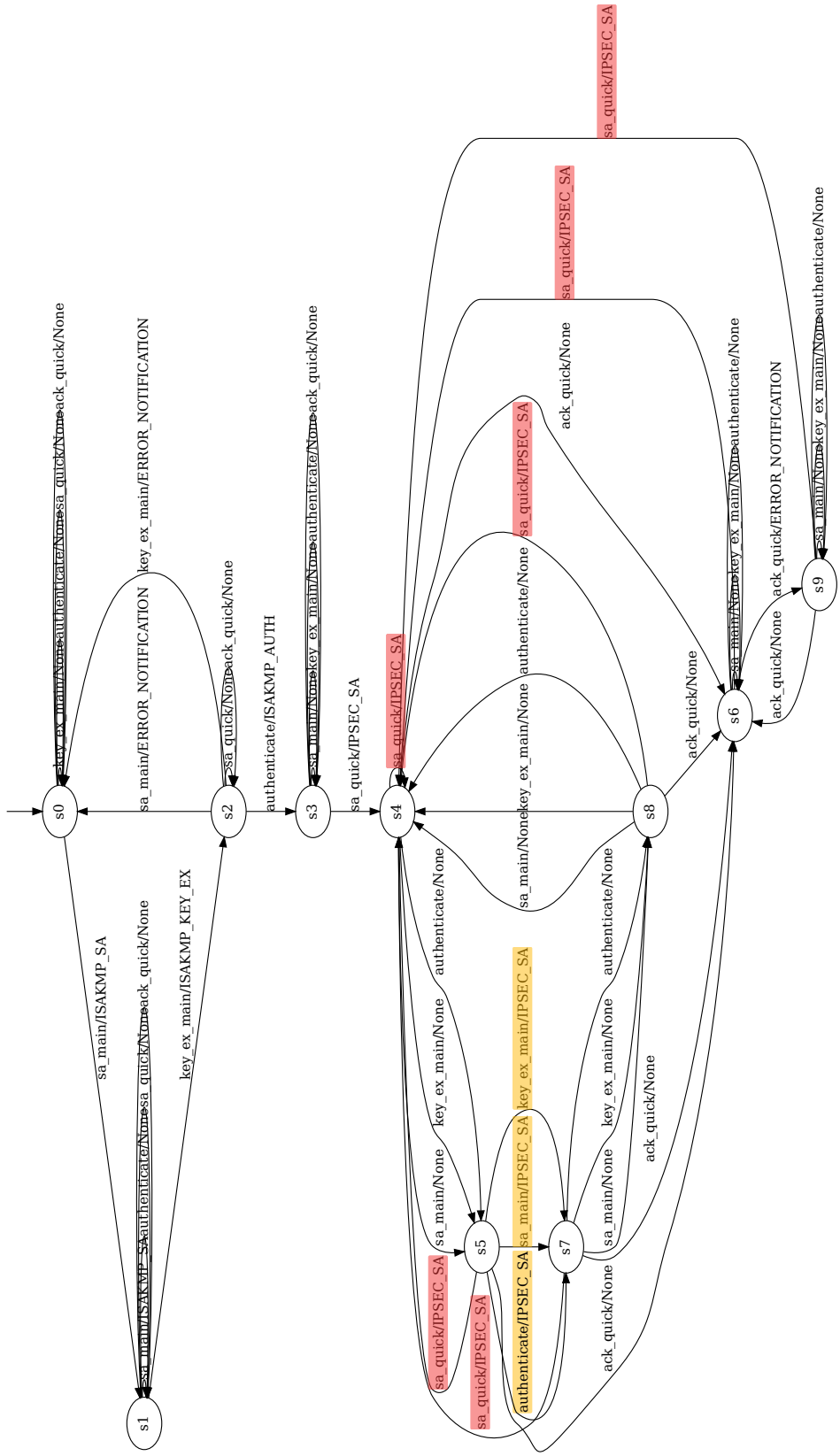


Figure 7.1: First commonly learned model of strongSwan server with retransmissions enabled.

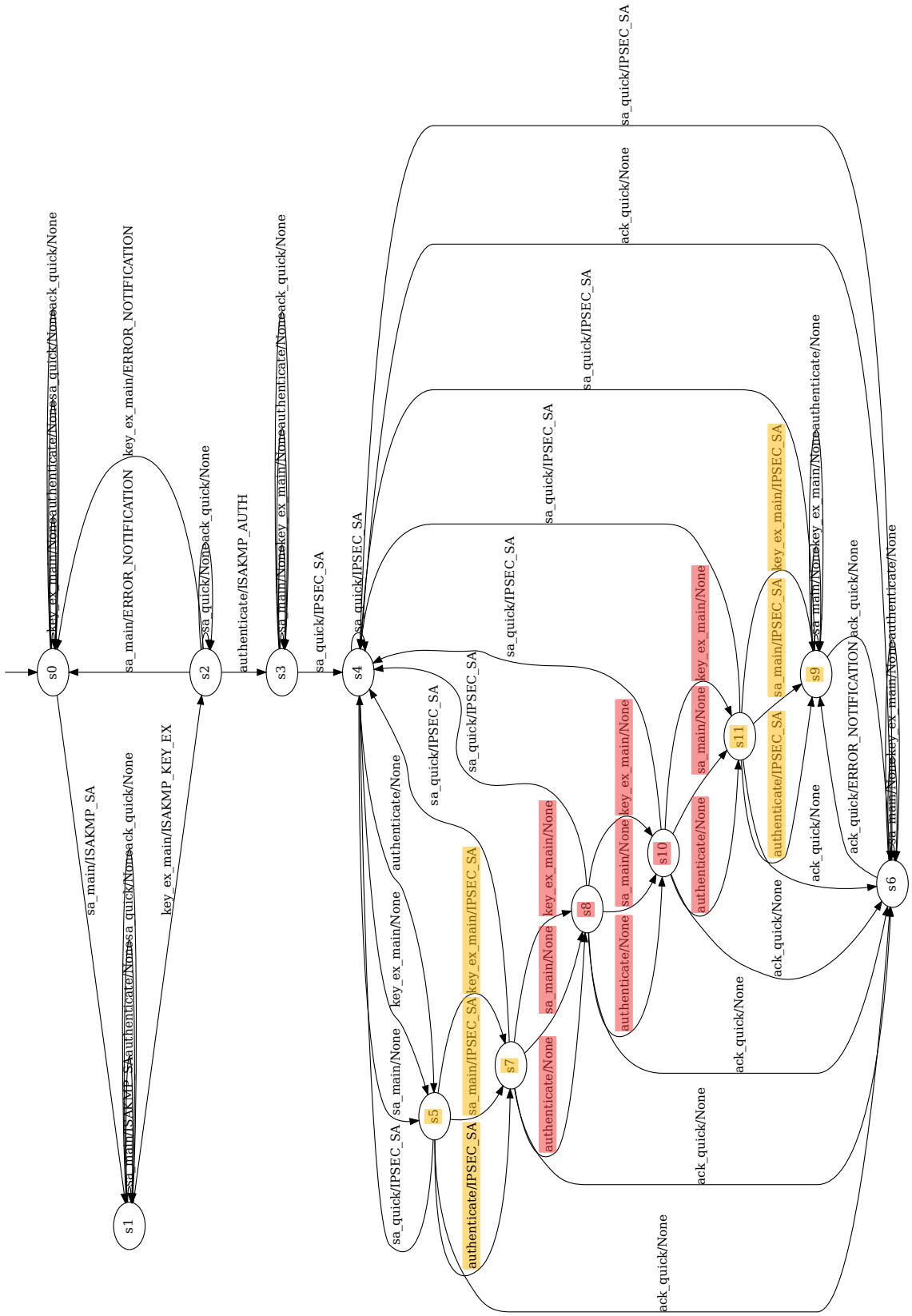


Figure 7.2: Second commonly learned strongSwan server model with retransmissions enabled.

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 inputs 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.

Other outliers

Figure 7.3 shows a sample outlier model of the strongSwan IPsec server, learned using the L^* algorithm in approximately 200 minutes (12187 seconds). It consists of 13 states, which were learned over five learning rounds. The runtime was shared between state exploration and conformance checking in a 60/40 split. State exploration took 5188 input steps to complete, whereas conformance checking only took 1847 steps. Unfortunately, due to the random nature of the outliers, this exact model could not be learned using the KV algorithm within the allotted time frame. When model learning the IPsec server without enabling retransmission-filtering, roughly 20% of the learned models differed from the two already presented common models. Figure 7.3 shows one such model. They all showcased slight deviations, such as additional states and/or transitions cause by server retransmissions.

The presented outlier model showcases characteristics of both common models, with states *s5* and *s7* (highlighted in yellow) resembling states *s6* and *s9* of the first common model. Additionally, the central states of the outlier model (highlighted in red) resemble those of the second common model in that it features a long chain of retransmissions allowing otherwise invalid phase one messages in phase two.

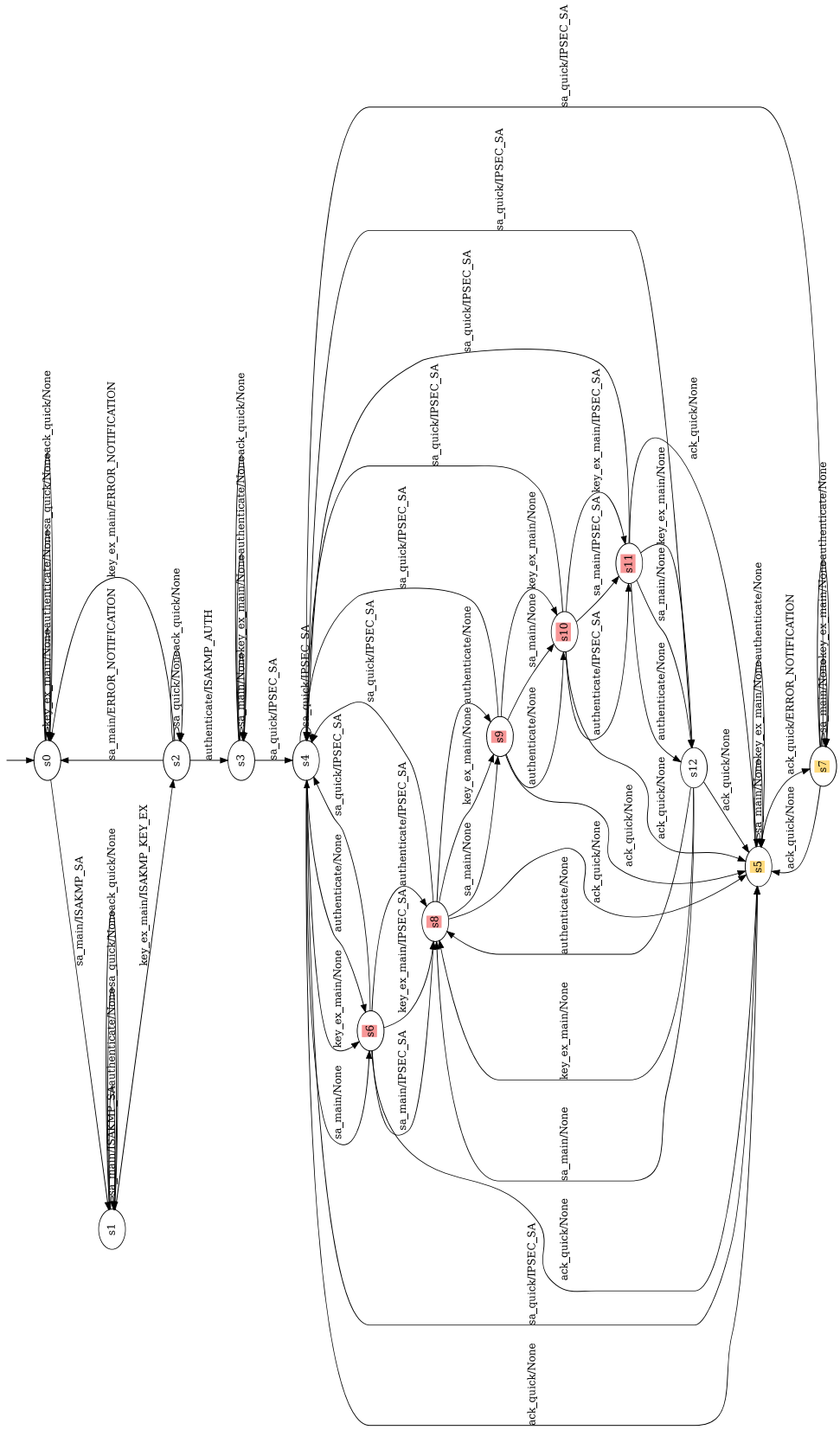


Figure 7.3: A rare outlier model, learned with retransmissions enabled.

Clean Base Models

In comparison, when learning the same server implementation with retransmission-filtering enabled, all non-deterministic behavior vanishes and we get the model shown in Figure 7.4 for every learning attempt. The model has only six 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, taking 676 input steps, where the time was distributed between state exploration and conformance checking in a 40/60 split (519 vs 747 seconds). Conformance checking took an average of 934 input steps to complete. This was the only configuration where the conformance checking took longer than state exploration, as highlighted in Table 7.1. 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), with state exploration requiring 856 and conformance checking requiring only 747 input steps. Compared to KV, state exploration/output queries took more than twice the amount of time to complete.

Model	States	TT (s)	TL (s)	TC (s)	OQ	CT
First Common	10	3092	1501	1591	171	100
Second Common	12	4507	2382	2126	215	120
Base	6	1214	480	734	78	60
Reference	6	1447	879	568	174	60

Table 7.1: KV Runtimes of all the learned strongSwan models.

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 are 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 3.6) 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.

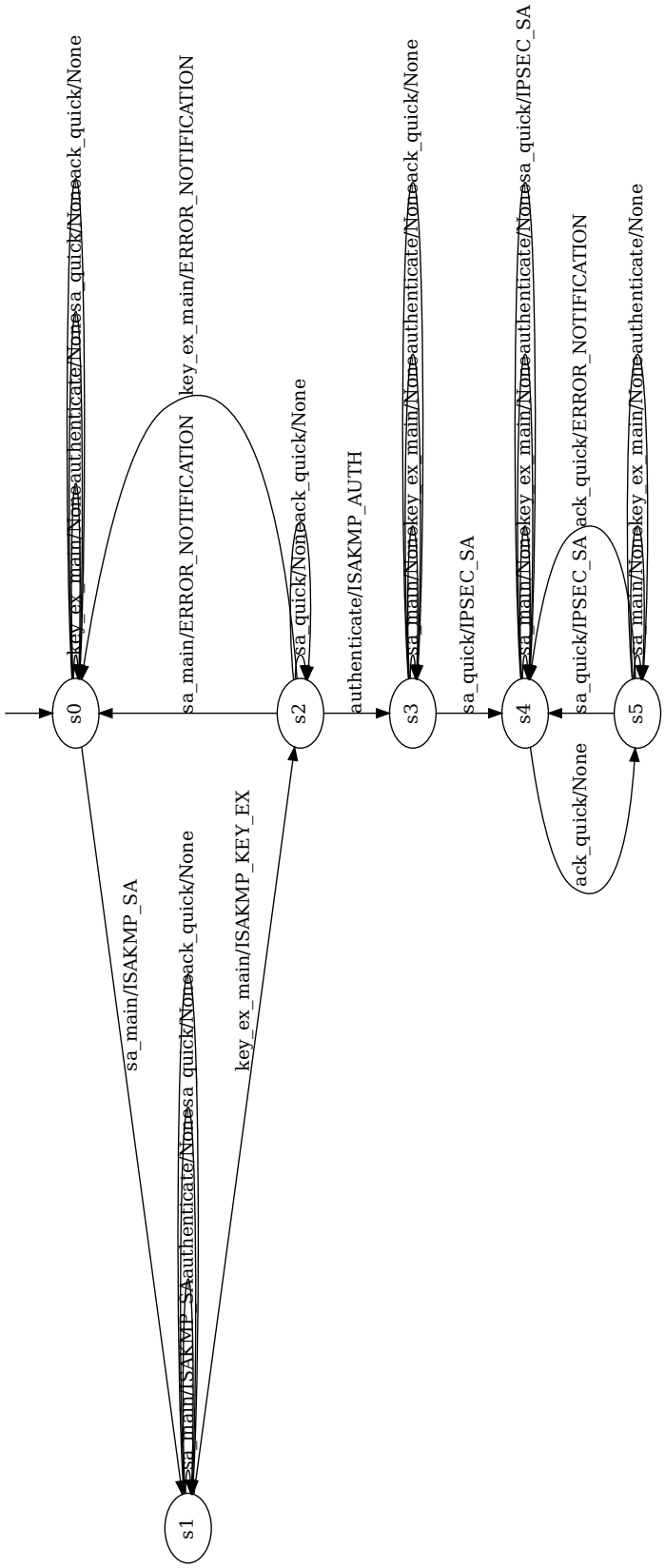


Figure 7.4: Clean strongSwan model learned using retransmission-filtering.

Fuzzing Reference Models

Figure 7.5 shows the (simplified, with self-transitions removed for readability) reference model used for fuzzing the strongSwan server, learned with retransmission-filtering enabled. Additionally, the input alphabet was expanded to include an additional erroneous version of each input that maps to an erroneous input. Thanks to the retransmission-filtering, the same model could be learned every time. The DOT file of the full model is provided in Appendix A. Using the KV algorithm, it took roughly 24 minutes (1447 seconds) to learn. The model took between five and four learning rounds to complete and consists of six states. Roughly 60% of the total learning time was spent on state exploration/output queries and the remaining 40% on conformance checking (879 vs 568 seconds). State exploration took an average of 1556 input steps to complete. In comparison, conformance checking required only 944 inputs. 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 80/20 split (2500 vs 578 seconds), with state exploration requiring 600 output queries. For this algorithm, learning took 2600 inputs and the equivalence checking required 740 input steps. 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 7.2.

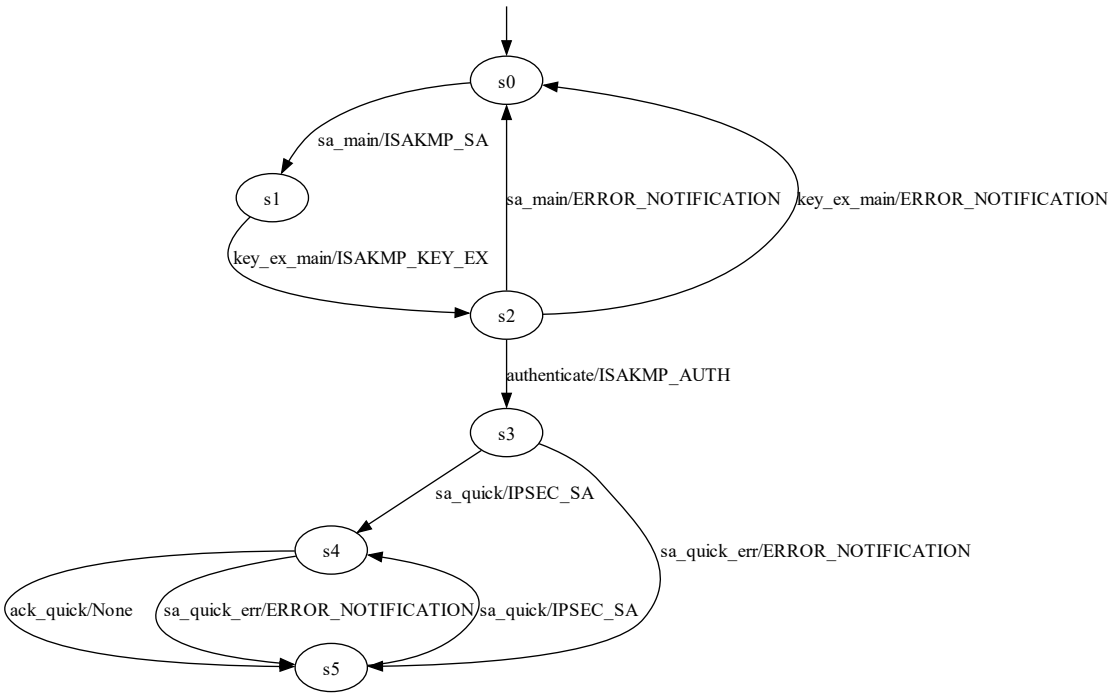


Figure 7.5: (Simplified) strongSwan model with malformed messages in the input alphabet.

The model looks largely identical to the previous model, apart from some additional self and error transitions (self transitions not shown here in the simplified version). Notably, there is an error transition from state *s4* to *s5*, via *sa_quick_err*. This error transition implies that a valid IPsec SA is required for *ack_quick* to work. In other words, the erroneous SA is insufficient for the server to accept an ensuing acknowledgment. If attempted, the strongSwan server returns an error message. For the same reason there is another error transition from state *s3* to *s5*, highlighting that one cannot complete the IPsec handshake with invalid IPsec SA packets. In summation, the mentioned error transitions show that a valid IPsec SA must be created before it can be acknowledged. Note that the DOT files of all models can be found in Appendix A and larger PDF files of all models are available as supplementary material[2].

Model	States	TT (s)	TL (s)	TC (s)	OQ	CT
First Common	10	5094	3489	1605	462	100
Second Common	12	7520	5393	2126	522	120
Base	6	1652	899	753	177	60
Fuzzing Ref.	6	3078	2500	578	600	60

Table 7.2: L^* Runtimes of all the learned strongSwan models.

Table 7.3 shows the average statistics of both learning algorithms over all four learned strongSwan models (averages of L^* and KV combined in one cell). As expected, models consisting of more states took longer to learn than those with less. For the two models with an identical amount of states, namely the Base and Fuzzing Reference models, the combined number of queries sent is a more suited indicator for the expected runtime. In general, the combined number of queries serves as a good indicator for how long each model will take to learn.

Model	States	TT (s)	TL (s)	TC (s)	OQ	CT
Common A	10	4048	2495	1553	317	100
Common B	12	6014	3888	2126	369	120
Base	6	1433	690	744	128	60
Fuzzing Ref.	6	2263	1690	573	387	60

Table 7.3: Runtimes averages of both learning algorithms of all the learned strongSwan models.

[2] `https://github.com/benjowun/VPN-AL`

7.1.3 Learned libreswan Models

Figure 7.6 shows the libreswan clean base model, learned using the same input alphabet and retransmission-filtering rules. The corresponding strongSwan model is the strongSwan Base model, shown in Figure 7.4. The model is very simple, consisting of only four states. Learning occurred over two rounds using the KV, and only one round, using the L^* algorithm. Despite the aforementioned reset workaround potentially affecting runtimes and, therefore, them being excluded from statistics, some runtime-agnostic metrics, such as the number of required queries and steps, are still provided. Using the KV algorithm, 36 output queries and 321 steps were required by the learning algorithm. Conversely, L^* required 100 output queries and 350 input steps. Both algorithms required 40 conformance tests for equivalence checking, with the KV algorithm however requiring roughly 660 steps, compared to the 460 of the L^* algorithm. With respect to runtime, KV did outperform L^*, but it is important to again note that any runtime measurements for libreswan are unreliable and cannot be compared with the strongSwan runtimes, due to the reset workaround.

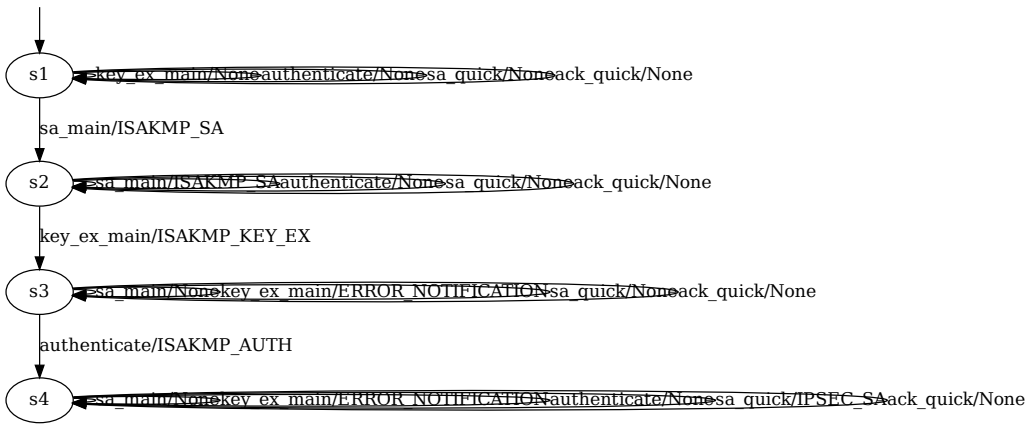


Figure 7.6: Clean libreswan model learned using retransmission-filtering.

The learned libreswan base model itself is noticeably smaller than its strongSwan equivalent, having two less states. In addition, the libreswan model is very linear, with each state being forced to occur subsequently. This is due to how libreswan handles errors and invalid messages, completely ignoring most error types that would lead to connection resets with strongSwan. This behavior is also what forced us to implement a workaround for the reset functionality, as error messages do not cause the libreswan phase one ISAKMP connection to be closed. Another key difference is that, following the initial key exchange in phase one, the libreswan server does return error messages when presented with another unencrypted key exchange request, but does not close the affected connection. In contrast, strongSwan ignores them in phase two resulting in a None response in the model. However, if an additional key exchange request is sent to a strongSwan server before phase one is completed, the server returns an error and aborts the connection. The key difference here being the closing of the connection on error, which again explains the more complicated strongSwan model, compared to the linear libreswan one. This difference can, e.g., be seen when comparing the transition of Figure 7.4, state *s2* to *s0*, with state *s3* of Figure 7.6, noting the lack of a transition back to the initial state for the libreswan model. Finally, there is a distinct lack of error messages for not-yet established IPsec SA acknowledgments on the libreswan server, resulting in both *sa_quick* and *ack_quick* remaining in the same state.

The libreswan version of the fuzzing reference model is again simpler than its strongSwan counterpart, consisting of one less state (five vs six) than the strongSwan fuzzing reference model. The DOT file of the full automata can be found in Appendix A. Figure 7.7 shows a strongly simplified model, without self-transitions, highlighting the new state *s2*. It looks largely identical to the libreswan base model, with the notable addition of a left branch in the model (state *s1* to *s2*), where an erroneous *sa_main_err* packet is sent at the start of the connection. This causes the server to become impossible to establish a connection with (with our setup), as libreswan ignores any further attempts with the same initiator cookie, following an invalid one, regardless of their correctness. The server does not leave this error state any more, as, due to the *uniqids* setting mentioned in Chapter 4, all following *ISAKMP_SA* packets will use the same ID. While admittedly a rather uncommon server setup (primarily used here due to simplify our mapper class), it could be considered a design flaw, to block valid *ISAKMP_SA* packets from replacing a broken connection when using the *uniqids* setting, as this renders the connection unusable. A valid use-case for this setting could be wanting to connect multiple users using a single IKE connection.

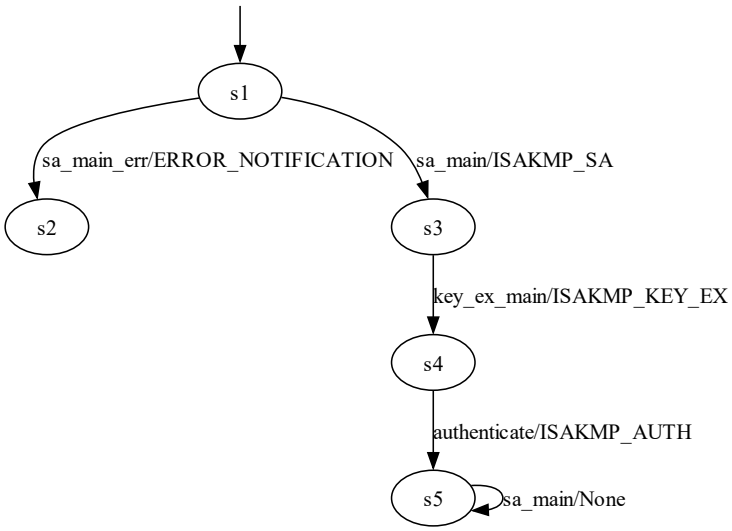


Figure 7.7: (Simplified) libreswan model with malformed messages in the input alphabet.

As switching to a randomly generated initiator cookies system would have negatively affected the strongSwan setup, the decision was made against a random-initiator cookie redesign. Especially since the libreswan setup already required a workaround for its reset method, potentially skewing results, the additional reworks were considered not worth the effort, but should be considered for future work.

In summation, one can say that the strongSwan server is stricter with their error handling, closing connections, if errors are encountered, whereas libreswan is more error-tolerant, preferring to keep connections regardless. These implementation details are clearly visible in the presented models, with the libreswan model having considerably less states, due to less error-handling being performed.

7.1.4 Comparing KV and L^*

Table 7.4 shows average performance statistics over 20 learning attempts for both learning algorithms on the strongSwan server. The models were learned with retransmission-filtering enabled. The same hardware and software configurations were used as described in Chapter 4 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 7.4. Table 7.4 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 hypotheses they needed to propose to the teacher in order 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 state exploration and time spent on equivalence queries. Learning output queries refers to the number of output queries sent to the SUL while learning steps to the number of individual inputs executed on the SUT. Analogously, equivalence oracle queries refers to the equivalence queries sent to the SUL and equivalence oracle steps to the inputs executed by the equivalence oracle. Finally, output queries saved by caching details the performance boost gained by caching output queries, with the value indicating the number of queries saved.

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 Output Queries	**177**	**78**
Learning Steps	856	676
Equivalence Oracle Queries	60	60
Equivalence Oracle Steps	747	934
Output Queries Saved by Caching	**13**	**30**

Table 7.4: Comparison of the L^* and KV learning algorithms.

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 output queries differs greatly between the two model-learning algorithms. The L^* algorithm required almost double the number of output queries than its KV counterpart. As communication with the SUT is the main performance bottleneck and output 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 output 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.

Little variance was observed throughout all learning attempts. Therefore 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 input sequences. Additionally, it might be interesting to compare the performance of various equivalence oracles for this learning setup.

7.1.5 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. This led us to the conclusion that the problems were not caused by our implementation, but in fact by a used Python library. It turns out there was a very niche bug in a used Diffie-Hellman Python library [40], where if the most significant byte (MSB) 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 two IPsec IKEv1 SUTs. Our fuzzer supports testing inputs in the context of input sequences, to ensure an identical state on the SUT for each fuzzed input. To that end, we developed a custom fuzzer supporting multiple methods of generating the input sequences to be tested, filtering, search and genetic algorithm-based input-sequence generation, as described in Chapter 6. All fuzzing took place in the same isolated network described in Chapter 4. The used fuzzing reference models can be seen above in figures 7.4 and 7.7, for strongSwan and libreswan respectively. This section presents the results of using our custom fuzzer to fuzz both a strongSwan, as well as a libreswan IPsec server. Different input-sequence generation methods were used and contrasted, comparing performance and discovered findings. Most methods found the same issues, however the search-based input-sequence generation method proved to be the fastest and the genetic method resulted in the best fitness score. Randomly generated input sequences serving as a baseline did not discover all the findings.

7.2.1 Findings

Our fuzzer was very successful in discovering new behavior not covered by the reference model, finding new cases for almost every input sequence tested with both input-sequence generation methods. As discovered new behavior does not necessarily indicate that the new behavior is harmful, the discovered new transitions still had to be manually parsed for particularly interesting behavior. By analyzing the specific concrete inputs leading to new transitions, we discovered two undocumented instances of the SUT not following RFC specifications. Unfortunately, a lot of "non-findings" were discovered as well, with non-findings referring to new behavior not covered by the reference model, but also not exhibiting undocumented or otherwise interesting behavior. Some of these non-findings could be removed by specifying some fields which should not be fuzzed. Unfortunately, this approach requires manually going through and verifying that none of the discovered behavior is interesting. Alternatively, it might be possible to use the discovered new behavior to generate a suite of tests checking for common exploitable vulnerabilities, such as buffer overflows and memory leaks. However, this goes beyond the scope of this thesis.

To further reduce the amount of information that had to be manually reviewed, we improved the fuzzer by temporarily adding newly learned states and transitions to the reference model as soon as they are discovered. This stops the fuzzer from repeatedly discovering the same new behavior with different variations of the same fuzzed input, leading to far less manual reviewing being required. 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. Unless otherwise mentioned, all findings were found on both tested IPsec servers.

An example of a non-finding that is typically categorized as noise is the discovery of new behavior by changing the initiator/responder cookies as part of fuzzing. These cookies are used in part to identify the members of a VPN connection. The new behavior was detected, as during the learning of the reference model, the cookies were not changed mid-input sequence execution, as this causes the SUT to think it is communicating with a different user and if it doesn't know that user, to 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 causes the server response encryption to no longer match 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 examining and testing the discovered new behavior helped improve our overall understanding of the IPsec protocol, it rarely resulted in the discovery of any actual undefined/undocumented behavior. However, in two separate cases, actual deviations from the relevant RFC specifications were observed with the SUTs.

The first of these deviations was discovered while fuzzing the ISAKMP header length field. This field is specified by RFC 2409 [21] (ISAKMP protocol definition), as the length of the entire ISAKMP packet. When manually going through the results of the fuzzer, a significant amount of the newly discovered behavior was found to have been caused by packets with a fuzzed ISAKMP header length field. Comparing the responses to the fuzzed packets revealed that they did not match the expected return values according to the reference model. As expected and actual return values did not conform the fuzzer correctly concluded that a new state had been found. The mismatch between actual and expected values was caused by the reference model expecting an error response to the fuzzed ISAKMP length field, while in practice, both of the tested SUTs 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 having 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 by both IPsec servers.

```
1  Fuzzing isa_len with: b'\xff\x00\x00\x00'
2  Input sequence: ['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: Discovered finding showing the ISAKMP length field being 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 heavily relies on ISAKMP, these requirements should still hold true. This is especially relevant, as strongSwan even lists RFC 2408 as one of the "Core Standards" they adhere to on their own website [30]. In fact, RFC 2409 (IKEv1), Section 5 [5] states very plainly, that IKEv1 exchanges are to conform to the ISAKMP standard. This standard, in Section 5.1 of RFC 2048 [21] 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 and libreswan servers do not, this finding falls into the deviations from specifications category.

Another interesting finding can be observed on strongSwan servers 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 from the tested strongSwan server after the key exchange packet was sent as well. 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. Listing 7.2 shows the results of fuzzing the strongSwan SUT using a modified unsupported ISAKMP_SA `Authentication` transform field.

```
Fuzzing tf with: [('Encryption', 'AES-CBC'), ('KeyLength', 256), ('Hash',
    'SHA'), ('GroupDesc', '1024MODPgr'), ('Authentication', '65535'), ('
   LifeType', 'Seconds'), ('LifeDuration', 28800)]

Run: ['sa_quick_err', 'ack_quick', 'sa_main_fuzz', 'sa_quick_err', '
   authenticate_err', 'sa_quick_err', 'key_ex_main', 'authenticate', '
   sa_main', 'key_ex_main']

$sa_main_fuzz

**********
Expected: ERROR_NOTIFICATION | Received: ISAKMP_SA
**********
```

Listing 7.2: Discovered finding showing the Authentication field not being validated.

Listing 7.2, Line 8 shows that the SUT does not return an error and instead returns a valid *ISAKMP SA* response packet. This is problematic, as RFC 2049, section 5 [5] 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 when, for example, a proposal is unacceptable, or a signature verification or decryption was unsuccessful, etc."

In particular the second part regarding 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. This further supports the theory that validity checks for this particular field were forgotten. 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. This is problematic in its own right, as errors being belatedly reported makes debugging more difficult. The main impact of this finding is, therefore, most likely its potential in making debugging more difficult. It is important to highlight that this finding was only observed on the strongSwan server, the libreswan server seems to implement it correctly.

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 intentional 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.

Finally, reviewing the learned libreswan model in Figure 7.7 revealed a potential deadlock state, state *s2*. The learned model suggests that the libreswan IPsec server does not recover from an invalid initial *ISAKMP SA* packet. Further testing showed this to be accurate, with all packets being ignored following the initial erroneous one. While likely limited to our particular IPsec configuration, notably the *uniqueids* setting allowing us to reuse the existing connection, this could prove problematic in certain edge cases, where IDs are not unique and due to network errors, the first packet is invalid. In such a scenario, the connection would remain unusable until the server is restarted. This problem does not occur with the strongSwan implementation, as strongSwan destroys the ISAKMP connection when an erroneous packet is received prior to keying.

7.2.2 Comparison of Input-Sequence Generation Methods

This subsection compares the performance of the utilized input-sequence generation methods used for fuzzing. The methods are filtering, search and genetic algorithm-based input-sequence generation approaches and are explained in detail in Chapter 6. Additionally, random input sequences were used to serve as a baseline when comparing the other input-sequence generation methods. All methods other than the random baseline led to input sequences that discovered all above-mentioned non-trivial findings during fuzzing. However, the methods differ greatly with regards to the amount of coverage achieved and their runtime.

As the filtering-based input-sequence generation method results in many different input sequences, that are then all fuzzed, naturally its coverage is inherently higher than the other methods, which result in only a single (or a select few) input sequence(s). This is due to the filtering-based approach reusing the input sequences already generated during model learning of the fuzzing reference model, bringing with it a guarantee of state-coverage (for at least the reference model). While this guarantee does not necessarily still hold after removing some input sequences due to filtering, considering the large number of remaining ones, the resulting coverage after fuzzing all of them will almost certainly be significantly higher than when limited to one, or very few, input sequences. However, as a trade-off, the runtime of fuzzing the input sequences generated through the filtering-based method is exceedingly long, as can be seen in Table 7.7. The runtime of the filtering step itself is less than the other methods, as seen in Table 7.5, and also more consistent, as the duration of the filtering algorithm is less random than that of the search or genetic-based methods. However, the fuzzing runtime is far longer than that of the other methods, as all the remaining input sequences are fuzzed one after the other. This leads to the fuzzing runtime of the filtering-based approach being more easily measurable in days instead of hours. Unfortunately, due to the extremely long runtime and our limited resources, this method could only be used sparingly, making the resulting statistics less tested than the others.

Method	Runtime (h)
Filtering	7
Search	19.5
Genetic	15.6

Table 7.5: Comparison of input sequence generation runtimes using the strongSwan server.

In comparison, the search-based input-sequence generation method results in only a single input sequence to be fuzzed, albeit one that is then fuzzed in more detail (all inputs of the sequence instead of only one are fuzzed). Using the search-based approach described in Chapter 6, a single input sequence is continuously refined with regards to a fitness score. The fitness score describes the suitability of the input sequence in finding new behavior, as well as achieving a high degree of state-coverage when fuzzed. Table 7.6 shows a comparison of our generated and random baseline input sequences, examining their average fitness scores with regards to the strongSwan server. Two baseline sequence lengths were used as a comparison, one consisting of eight inputs and the other out of 18. These lengths mirror the lengths of the best-scoring input sequences generated through search/genetic-based methods. The fitness score for the filtering method was calculated by calculating the average fitness score of all the input sequences produced by the filtering method. While considerably more performative than both baseline sequences, the filtered input sequence fitness score is barely half as good as that of the search-based method. Comparing search with the two baseline sequences shows that it significantly outperforms both. The poor results of the baseline sequences can be explained by the baseline fuzzing often getting stuck in the initial state/phase and therefore lacking coverage of large parts of the SUT. This naturally leads to a poor score, as hardly any interesting behavior is found and the coverage is poor. As both of the discovered findings were discoverable in the initial state however, no concrete differences in the

actual discovered findings were observed for that state. However, fuzzing the baseline sequences resulted in far fewer findings, especially the ISAKMP length finding, as far fewer states were covered. The filtering method does discover all findings, however the fuzzing runtime is disproportionately longer, as shown in Table 7.7. Additionally, while the average runtime of the search-based method is inferior to the filtering-based method, see Table 7.5, the actual fuzzing is considerably quicker, making the total runtime (sequence generation + fuzzing) also quicker. Consequently we argue that the search-based method significantly outperforms both the baseline and filtering-based methods, making it more likely to discover interesting behavior while fuzzing, in less time, even if no additional findings were found for the specific SUTs in question.

Method	Fitness Score
Baseline (8)	0.0542
Baseline (18)	0.392
Filtering (avg)	1.4712
Search	2.8125
Genetic	4.7

Table 7.6: Comparison of the fitness scores of input-sequence generation methods.

While a good trade-off in terms of time and fitness score, the effectiveness of the search-based method can be further enhanced by incorporating genetic algorithm components. As described in Chapter 6, our genetic input-sequence generation method works similarly to the search-based approach, however, instead of mutating only one input sequence, multiple sequences called populations are mutated simultaneously. The top scoring populations with regard to the fitness function are then crossed to create new sequences with characteristics of both parent sequences. Combined with randomly generated sequences to introduce more variability, this method outperforms all other implemented input-sequence generation methods significantly, as can be seen in Table 7.6. The genetic algorithm-based input-sequence generation method was configured to resemble the search-based approach with regard to the number of mutations performed on the populations. Regardless, the genetic method outperformed both the search-based and filtering-based approaches, generating an input sequence with almost double the fitness score than that of the search-based sequence, and more than thrice the score than that of the average filtering-based sequence, with similar average runtimes to the search-based approach. Note that the runtimes for search and especially genetic algorithm-based methods have a rather high variance, as the runtime is strongly tied to the length of input sequence, which is to a certain degree, random. Furthermore, the genetic algorithm introduces populations of a random length, often significantly shorter than the current average, whereas the search-based method input sequence tends to only grow longer as it progresses. As many of the scored sequences of the genetic algorithm-based approach were short/mostly invalid due to their random generation, less inputs were actually executed on the SUT. These two attributes led to the genetic algorithm runtime being, on average, slightly better than the search-based method, however the standard deviation was significantly higher. Additionally, fuzzing the search-based and the genetic input sequences both discovered all known findings, however, the genetic sequences provided better state coverage of the reference automaton, further cementing it as an improvement over the single-population search-based method.

Table 7.7 shows the fuzzing runtime comparison of fuzzing the previously generated input sequences on the strongSwan SUT. The search-based method was set to run for 50 iterations and the genetic algorithm-based method was configured to use ten populations and to run for five iterations (totaling the same 50 iterations for both methods). The filtering-based approach used the input sequences from learning the reference model of the strongSwan SUT and all runtime data was measured using the strongSwan setup. The "Fuzzed In. Seq." row refers to the number of input sequences generated by the method in question that are then fuzzed. Note that, while the genetic algorithm-based input-sequence generation method technically generates several input sequences in a population, only the best-scoring sequence was fuzzed. The "In. Seq. Length" row indicates the average amount of inputs per run. The "Seconds/In. Seq." row displays the average execution time of a single input sequence and the "Values Tested" row shows the total number of fuzzed values tested during fuzzing rounded to the nearest hundred. Finally, the "Runtime" row indicates the total runtime of fuzzing the input sequence(s) in question. These metrics allow for a comparison of the efficiency and effectiveness of the two methods in generating and testing input data.

	Filtering	**Search**	**Genetic**
Fuzzed In. Seq.	55	1	1
In. Seq. Length	14	8	18
Seconds/In. Seq.	14	8	18
Values Tested	192500	2800	4500
Runtime (h) Fuzzing	53	7	24

Table 7.7: Comparison of generated input-sequence fuzzing runtimes on the strongSwan server.

Examining the strongSwan fuzzing runtime statistics shown in Table 7.7, the most striking difference is the difference in number of input sequences fuzzed. While the filtering input-sequence generation method resulted in 55 input sequences, the other two methods only result in one sequence. Naturally, as all 55 sequences are then fuzzed, the fuzzing runtime is by far the longest for this method. This is despite the fact that for the filtering input sequences, only one input is fuzzed, while for the other two methods, all inputs are fuzzed.

Each additional input added to an input sequence leads to, on average, an additional second of runtime when executing the sequence, due to timed waits and timeouts during network communication. Therefore, "In.Seq Length" translates directly to the "Seconds/In.Seq." metric. When fuzzing, we execute the entire sequence for each value inserted into the field being fuzzed. Consequently, the total runtime when fuzzing an input sequence can be approximated as the length of the sequence (the number of inputs), multiplied by the number of values tested while fuzzing the sequence. The average number of tested values on a single input is approximately 250 values. This lets us approximate the runtime of fuzzing an input sequence as

$$r = l \cdot n \cdot v \tag{7.1}$$

where l is the length or average length of the tested input sequence(s), n the number of input sequences to be fuzzed and v the average number of values tested when fuzzing one input, so 250. Consequently, we can say the runtime of our fuzzer is directly proportionate to the number of fuzzed input sequences and the length of the tested input sequence(s). There is some variance with v depending on the type of tested fields, but in our case it is mostly constant.

Finally, Table 7.8 shows the full runtimes for all used input-sequence generation methods. We can again clearly see that despite having the shortest generation runtime, the exceedingly long fuzzing runtime makes the filtering-based approach by far the slowest. The search-based method finishes in less than half the time needed by the filtering-based approach, and discovered the same findings. However, its state coverage was less. In contrast, the genetic algorithm-based approach ran slightly longer than the search-based one, but still much faster than the filtering-based approach and had better coverage than the search-based approach.

Method	**Runtime (h)**
Filtering	60
Search	26.5
Genetic	31.6

Table 7.8: Comparison of input sequence generation and subsequent fuzzing total runtimes using the strongSwan server.

8 Conclusion

8.1 Summary

In this thesis, we learned behavioral models of two IPsec server implementations, strongSwan and libreswan. We presented the learned models of both implementations, comparing them with respect to various metrics including runtime and the number of required queries. Our results show that our learning setup succeeds in its goal of reliably learning models of the target IPsec IKEv1 server implementations. We contrasted two popular model-learning algorithms KV and L^* and explained why we consider KV to be better suited for our learning setup. Furthermore, we highlighted the difficulties that arose during model learning, such as issues with non-determinism, and presented our proposed solutions. Additionally, we demonstrated the usefulness of AAL from a testing standpoint by showcasing a Python cryptography library bug found during model learning.

We used the learned models to perform model-based fuzzing of the IPsec implementations under test. We presented the results of our fuzzer, highlighting the most interesting findings, most notably, two deviations from the RFC specification, as well as a potential deadlock. We recommend that the findings be thoroughly examined to ensure that they do not pose compatibility or security risks. Furthermore, we compared the various utilized input sequence-generation methods to each other and a random baseline. For our SUTs, all non-baseline input sequence-generation methods discovered all findings. However, significant differences in the calculated fitness scores of the respective methods were observed, with the genetic algorithm-based method performing the best. As the used fitness score takes the state coverage into account, the genetic method will likely have the greatest chance at finding interesting behavior. However, from a pure runtime perspective, the search-based method was the most efficient for the SUT at hand. The filtering-based method was by far the slowest, with fuzzing of the generated input sequences taking several days longer than when fuzzing the genetic-algorithm generated input sequences. While perhaps slightly more thorough, the comparable fitness score of the genetic-algorithm generated input sequences, combined with its significantly faster runtime fuzzing them, leads us to the conclusion that, of the tested methods, the genetic algorithm-based input sequence-generation method is the one best suited for further fuzzing work.

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.

8.1.1 Discussion

Model learning and fuzzing was made more difficult due to implementation differences between libreswan and strongSwan. Perhaps the most significant such difference is the error handling. While strongSwan tends to send all expected informational error notification packets, libreswan, especially in phase one, does not. Additionally, while strongSwan destroys unfinished connections upon receiving an error, libreswan ignores most errors. This behavior forces us to implement a workaround for our *reset* method for libreswan, using an additional SSH connection to manually reset the server. While functional in theory, in practice, such a connection is usually unrealistic in a real world black-box testing scenario, as communication to the server is usually limited to specific ports and root access is even more unlikely. Still, the learned libreswan models serve to showcase the implementation differences between the two not unrelated projects (both implementations share a common ancestor in FreeS/WAN). Furthermore, as the two implementations behave so differently, it is possible to identify them based on their learned models, and even simpler, their responses (or non-responses) to a set of informational requests. Additionally, analyzing and comparing the learned models helped increase our understanding of the intricacies of the IPsec protocol, leading to the discovery of a potential libreswan deadlock. This demonstrates that model learning has considerable value, even beyond its security applications. It can serve as a useful tool

for aiding our comprehension of complex protocols and software by offering an easily understandable, abstracted overview of the SUL.

Regarding the fuzzing of the two SUTs, little to no differences in the number of discovered findings were observed between the utilized input sequence-generation methods. However, the fitness scores show that the genetic algorithm-based method outperforms the other methods with respect to coverage and runtime. Therefore, it is reasonable to assume that the genetic method would discover more findings than the other methods, if applied to a more vulnerable SUT. Of the discovered findings, the strongSwan exclusive `Authentication` transform field bug is likely to be the most disruptive, potentially making debugging connection establishments more difficult. While we believe a severe security impact to be unlikely, we still recommend to examine the presented findings, as any undocumented behavior can be a source of bugs and compatibility issues.

8.2 Future Work

Regarding our IPsec mapper class and 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 further IPsec implementations.

The developed fuzzing code can be applied to other protocols and implementations, simplifying potential future work testing additional IPsec implementations and/or similar protocols. Especially a study on the effectiveness of the genetic algorithm-based input sequence-generation method compared to the other methods of input generation on various SUTs could prove interesting. Additionally, more methods of input-sequence generation could be explored.

Appendices

A DOT Files

Collection of relevant DOT[9] files.

strongSwan

strongSwan first common model

```
digraph common1 {
  s0 [label=s0];
  s1 [label=s1];
  s2 [label=s2];
  s3 [label=s3];
  s4 [label=s4];
  s5 [label=s5];
  s6 [label=s6];
  s7 [label=s7];
  s8 [label=s8];
  s9 [label=s9];
  s0 -> s1  [label="sa_main/ISAKMP_SA"];
  s0 -> s0  [label="key_ex_main/None"];
  s0 -> s0  [label="authenticate/None"];
  s0 -> s0  [label="sa_quick/None"];
  s0 -> s0  [label="ack_quick/None"];
  s1 -> s1  [label="sa_main/ISAKMP_SA"];
  s1 -> s2  [label="key_ex_main/ISAKMP_KEY_EX"];
  s1 -> s1  [label="authenticate/None"];
  s1 -> s1  [label="sa_quick/None"];
  s1 -> s1  [label="ack_quick/None"];
  s2 -> s0  [label="sa_main/ERROR_NOTIFICATION"];
  s2 -> s0  [label="key_ex_main/ERROR_NOTIFICATION"];
  s2 -> s3  [label="authenticate/ISAKMP_AUTH"];
  s2 -> s2  [label="sa_quick/None"];
  s2 -> s2  [label="ack_quick/None"];
  s3 -> s3  [label="sa_main/None"];
  s3 -> s3  [label="key_ex_main/None"];
  s3 -> s3  [label="authenticate/None"];
  s3 -> s4  [label="sa_quick/IPSEC_SA"];
  s3 -> s3  [label="ack_quick/None"];
  s4 -> s5  [label="sa_main/None"];
  s4 -> s5  [label="key_ex_main/None"];
  s4 -> s5  [label="authenticate/None"];
  s4 -> s4  [label="sa_quick/IPSEC_SA"];
  s4 -> s6  [label="ack_quick/None"];
  s5 -> s7  [label="sa_main/IPSEC_SA"];
  s5 -> s7  [label="key_ex_main/IPSEC_SA"];
  s5 -> s7  [label="authenticate/IPSEC_SA"];
  s5 -> s4  [label="sa_quick/IPSEC_SA"];
  s5 -> s6  [label="ack_quick/None"];
  s6 -> s6  [label="sa_main/None"];
  s6 -> s6  [label="key_ex_main/None"];
  s6 -> s6  [label="authenticate/None"];
  s6 -> s4  [label="sa_quick/IPSEC_SA"];
  s6 -> s9  [label="ack_quick/ERROR_NOTIFICATION"];
  s7 -> s8  [label="sa_main/None"];
```

```
  s7 -> s8  [label="key_ex_main/None"];
  s7 -> s8  [label="authenticate/None"];
  s7 -> s4  [label="sa_quick/IPSEC_SA"];
  s7 -> s6  [label="ack_quick/None"];
  s8 -> s4  [label="sa_main/None"];
  s8 -> s4  [label="key_ex_main/None"];
  s8 -> s4  [label="authenticate/None"];
  s8 -> s4  [label="sa_quick/IPSEC_SA"];
  s8 -> s6  [label="ack_quick/None"];
  s9 -> s9  [label="sa_main/None"];
  s9 -> s9  [label="key_ex_main/None"];
  s9 -> s9  [label="authenticate/None"];
  s9 -> s4  [label="sa_quick/IPSEC_SA"];
  s9 -> s6  [label="ack_quick/None"];
  __start0 [label="", shape=none];
  __start0 -> s0  [label=""];
}
```

strongSwan second common model

```
digraph common2 {
  s0 [label=s0];
  s1 [label=s1];
  s2 [label=s2];
  s3 [label=s3];
  s4 [label=s4];
  s5 [label=s5];
  s6 [label=s6];
  s7 [label=s7];
  s8 [label=s8];
  s9 [label=s9];
  s10 [label=s10];
  s11 [label=s11];
  s0 -> s1  [label="sa_main/ISAKMP_SA"];
  s0 -> s0  [label="key_ex_main/None"];
  s0 -> s0  [label="authenticate/None"];
  s0 -> s0  [label="sa_quick/None"];
  s0 -> s0  [label="ack_quick/None"];
  s1 -> s1  [label="sa_main/ISAKMP_SA"];
  s1 -> s2  [label="key_ex_main/ISAKMP_KEY_EX"];
  s1 -> s1  [label="authenticate/None"];
  s1 -> s1  [label="sa_quick/None"];
  s1 -> s1  [label="ack_quick/None"];
  s2 -> s0  [label="sa_main/ERROR_NOTIFICATION"];
  s2 -> s0  [label="key_ex_main/ERROR_NOTIFICATION"];
  s2 -> s3  [label="authenticate/ISAKMP_AUTH"];
  s2 -> s2  [label="sa_quick/None"];
  s2 -> s2  [label="ack_quick/None"];
  s3 -> s3  [label="sa_main/None"];
  s3 -> s3  [label="key_ex_main/None"];
  s3 -> s3  [label="authenticate/None"];
  s3 -> s4  [label="sa_quick/IPSEC_SA"];
  s3 -> s3  [label="ack_quick/None"];
  s4 -> s5  [label="sa_main/None"];
  s4 -> s5  [label="key_ex_main/None"];
  s4 -> s5  [label="authenticate/None"];
  s4 -> s4  [label="sa_quick/IPSEC_SA"];
  s4 -> s6  [label="ack_quick/None"];
  s5 -> s7  [label="sa_main/IPSEC_SA"];
  s5 -> s7  [label="key_ex_main/IPSEC_SA"];
  s5 -> s7  [label="authenticate/IPSEC_SA"];
  s5 -> s4  [label="sa_quick/IPSEC_SA"];
  s5 -> s6  [label="ack_quick/None"];
  s6 -> s6  [label="sa_main/None"];
  s6 -> s6  [label="key_ex_main/None"];
  s6 -> s6  [label="authenticate/None"];
  s6 -> s4  [label="sa_quick/IPSEC_SA"];
  s6 -> s9  [label="ack_quick/ERROR_NOTIFICATION"];
  s7 -> s8  [label="sa_main/None"];
  s7 -> s8  [label="key_ex_main/None"];
  s7 -> s8  [label="authenticate/None"];
  s7 -> s4  [label="sa_quick/IPSEC_SA"];
  s7 -> s6  [label="ack_quick/None"];
  s8 -> s10  [label="sa_main/None"];
  s8 -> s10  [label="key_ex_main/None"];
  s8 -> s10  [label="authenticate/None"];
```

```
  s8 -> s4  [label="sa_quick/IPSEC_SA"];
  s8 -> s6  [label="ack_quick/None"];
  s9 -> s9  [label="sa_main/None"];
  s9 -> s9  [label="key_ex_main/None"];
  s9 -> s9  [label="authenticate/None"];
  s9 -> s4  [label="sa_quick/IPSEC_SA"];
  s9 -> s6  [label="ack_quick/None"];
  s10 -> s11  [label="sa_main/None"];
  s10 -> s11  [label="key_ex_main/None"];
  s10 -> s11  [label="authenticate/None"];
  s10 -> s4  [label="sa_quick/IPSEC_SA"];
  s10 -> s6  [label="ack_quick/None"];
  s11 -> s9  [label="sa_main/IPSEC_SA"];
  s11 -> s9  [label="key_ex_main/IPSEC_SA"];
  s11 -> s9  [label="authenticate/IPSEC_SA"];
  s11 -> s4  [label="sa_quick/IPSEC_SA"];
  s11 -> s6  [label="ack_quick/None"];
  __start0 [label="", shape=none];
  __start0 -> s0  [label=""];
}
```

strongSwan base model

```
digraph strongBase {
  s0 [label=s0];
  s1 [label=s1];
  s2 [label=s2];
  s3 [label=s3];
  s4 [label=s4];
  s5 [label=s5];
  s0 -> s1  [label="sa_main/ISAKMP_SA"];
  s0 -> s0  [label="key_ex_main/None"];
  s0 -> s0  [label="authenticate/None"];
  s0 -> s0  [label="sa_quick/None"];
  s0 -> s0  [label="ack_quick/None"];
  s1 -> s1  [label="sa_main/ISAKMP_SA"];
  s1 -> s2  [label="key_ex_main/ISAKMP_KEY_EX"];
  s1 -> s1  [label="authenticate/None"];
  s1 -> s1  [label="sa_quick/None"];
  s1 -> s1  [label="ack_quick/None"];
  s2 -> s0  [label="sa_main/ERROR_NOTIFICATION"];
  s2 -> s0  [label="key_ex_main/ERROR_NOTIFICATION"];
  s2 -> s3  [label="authenticate/ISAKMP_AUTH"];
  s2 -> s2  [label="sa_quick/None"];
  s2 -> s2  [label="ack_quick/None"];
  s3 -> s3  [label="sa_main/None"];
  s3 -> s3  [label="key_ex_main/None"];
  s3 -> s3  [label="authenticate/None"];
  s3 -> s4  [label="sa_quick/IPSEC_SA"];
  s3 -> s3  [label="ack_quick/None"];
  s4 -> s4  [label="sa_main/None"];
  s4 -> s4  [label="key_ex_main/None"];
  s4 -> s4  [label="authenticate/None"];
  s4 -> s4  [label="sa_quick/IPSEC_SA"];
  s4 -> s5  [label="ack_quick/None"];
  s5 -> s5  [label="sa_main/None"];
  s5 -> s5  [label="key_ex_main/None"];
  s5 -> s5  [label="authenticate/None"];
  s5 -> s4  [label="sa_quick/IPSEC_SA"];
  s5 -> s4  [label="ack_quick/ERROR_NOTIFICATION"];
  __start0 [label="", shape=none];
  __start0 -> s0  [label=""];
}
```

strongSwan fuzzing reference model

```
digraph strongRef {
  s0 [label=s0];
  s1 [label=s1];
  s2 [label=s2];
  s3 [label=s3];
  s4 [label=s4];
  s5 [label=s5];
  s0 -> s1  [label="sa_main/ISAKMP_SA"];
  s0 -> s0  [label="key_ex_main/None"];
  s0 -> s0  [label="authenticate/None"];
  s0 -> s0  [label="sa_quick/None"];
  s0 -> s0  [label="ack_quick/None"];
  s0 -> s0  [label="sa_main_err/ERROR_NOTIFICATION"];
  s0 -> s0  [label="key_ex_main_err/None"];
  s0 -> s0  [label="authenticate_err/None"];
  s0 -> s0  [label="sa_quick_err/None"];
  s0 -> s0  [label="ack_quick_err/None"];
  s1 -> s1  [label="sa_main/ISAKMP_SA"];
  s1 -> s2  [label="key_ex_main/ISAKMP_KEY_EX"];
  s1 -> s1  [label="authenticate/None"];
  s1 -> s1  [label="sa_quick/None"];
  s1 -> s1  [label="ack_quick/None"];
  s1 -> s1  [label="sa_main_err/ERROR_NOTIFICATION"];
  s1 -> s1  [label="key_ex_main_err/ERROR_NOTIFICATION"];
  s1 -> s1  [label="authenticate_err/None"];
  s1 -> s1  [label="sa_quick_err/None"];
  s1 -> s1  [label="ack_quick_err/None"];
  s2 -> s0  [label="sa_main/ERROR_NOTIFICATION"];
  s2 -> s0  [label="key_ex_main/ERROR_NOTIFICATION"];
  s2 -> s3  [label="authenticate/ISAKMP_AUTH"];
  s2 -> s2  [label="sa_quick/None"];
  s2 -> s2  [label="ack_quick/None"];
  s2 -> s2  [label="sa_main_err/ERROR_NOTIFICATION"];
  s2 -> s2  [label="key_ex_main_err/ERROR_NOTIFICATION"];
  s2 -> s2  [label="authenticate_err/ERROR_NOTIFICATION"];
  s2 -> s2  [label="sa_quick_err/None"];
  s2 -> s2  [label="ack_quick_err/None"];
  s3 -> s3  [label="sa_main/None"];
  s3 -> s3  [label="key_ex_main/None"];
  s3 -> s3  [label="authenticate/None"];
  s3 -> s4  [label="sa_quick/IPSEC_SA"];
  s3 -> s3  [label="ack_quick/None"];
  s3 -> s3  [label="sa_main_err/ERROR_NOTIFICATION"];
  s3 -> s3  [label="key_ex_main_err/None"];
  s3 -> s3  [label="authenticate_err/None"];
  s3 -> s5  [label="sa_quick_err/ERROR_NOTIFICATION"];
  s3 -> s3  [label="ack_quick_err/None"];
  s4 -> s4  [label="sa_main/None"];
  s4 -> s4  [label="key_ex_main/None"];
  s4 -> s4  [label="authenticate/None"];
  s4 -> s4  [label="sa_quick/IPSEC_SA"];
  s4 -> s5  [label="ack_quick/None"];
  s4 -> s4  [label="sa_main_err/ERROR_NOTIFICATION"];
  s4 -> s4  [label="key_ex_main_err/None"];
  s4 -> s4  [label="authenticate_err/None"];
  s4 -> s5  [label="sa_quick_err/ERROR_NOTIFICATION"];
```

```
  s4 -> s4  [label="ack_quick_err/ERROR_NOTIFICATION"];
  s5 -> s5  [label="sa_main/None"];
  s5 -> s5  [label="key_ex_main/None"];
  s5 -> s5  [label="authenticate/None"];
  s5 -> s4  [label="sa_quick/IPSEC_SA"];
  s5 -> s5  [label="ack_quick/ERROR_NOTIFICATION"];
  s5 -> s5  [label="sa_main_err/ERROR_NOTIFICATION"];
  s5 -> s5  [label="key_ex_main_err/None"];
  s5 -> s5  [label="authenticate_err/None"];
  s5 -> s5  [label="sa_quick_err/ERROR_NOTIFICATION"];
  s5 -> s5  [label="ack_quick_err/ERROR_NOTIFICATION"];
  __start0 [label="", shape=none];
  __start0 -> s0  [label=""];
}
```

libreswan

librewan base model

```
digraph libreBase {
  s1 [label=s1];
  s2 [label=s2];
  s3 [label=s3];
  s4 [label=s4];
  s1 -> s2  [label="sa_main/ISAKMP_SA"];
  s1 -> s1  [label="key_ex_main/None"];
  s1 -> s1  [label="authenticate/None"];
  s1 -> s1  [label="sa_quick/None"];
  s1 -> s1  [label="ack_quick/None"];
  s2 -> s2  [label="sa_main/ISAKMP_SA"];
  s2 -> s3  [label="key_ex_main/ISAKMP_KEY_EX"];
  s2 -> s2  [label="authenticate/None"];
  s2 -> s2  [label="sa_quick/None"];
  s2 -> s2  [label="ack_quick/None"];
  s3 -> s3  [label="sa_main/None"];
  s3 -> s3  [label="key_ex_main/ERROR_NOTIFICATION"];
  s3 -> s4  [label="authenticate/ISAKMP_AUTH"];
  s3 -> s3  [label="sa_quick/None"];
  s3 -> s3  [label="ack_quick/None"];
  s4 -> s4  [label="sa_main/None"];
  s4 -> s4  [label="key_ex_main/ERROR_NOTIFICATION"];
  s4 -> s4  [label="authenticate/None"];
  s4 -> s4  [label="sa_quick/IPSEC_SA"];
  s4 -> s4  [label="ack_quick/None"];
  __start0 [label="", shape=none];
  __start0 -> s1  [label=""];
}
```

libreswan fuzzing reference model

```
digraph libreRef {
  s1 [label=s1];
  s2 [label=s2];
  s3 [label=s3];
  s4 [label=s4];
  s5 [label=s5];
  s1 -> s3  [label="sa_main/ISAKMP_SA"];
  s1 -> s1  [label="key_ex_main/None"];
  s1 -> s1  [label="authenticate/None"];
  s1 -> s1  [label="sa_quick/None"];
  s1 -> s1  [label="ack_quick/None"];
  s1 -> s2  [label="sa_main_err/ERROR_NOTIFICATION"];
  s1 -> s1  [label="key_ex_main_err/None"];
  s1 -> s1  [label="authenticate_err/None"];
  s1 -> s1  [label="sa_quick_err/None"];
  s1 -> s1  [label="ack_quick_err/None"];
  s2 -> s2  [label="sa_main/None"];
  s2 -> s2  [label="key_ex_main/None"];
  s2 -> s2  [label="authenticate/None"];
  s2 -> s2  [label="sa_quick/None"];
  s2 -> s2  [label="ack_quick/None"];
  s2 -> s2  [label="sa_main_err/None"];
  s2 -> s2  [label="key_ex_main_err/None"];
  s2 -> s2  [label="authenticate_err/None"];
  s2 -> s2  [label="sa_quick_err/None"];
  s2 -> s2  [label="ack_quick_err/None"];
  s3 -> s3  [label="sa_main/ISAKMP_SA"];
  s3 -> s4  [label="key_ex_main/ISAKMP_KEY_EX"];
  s3 -> s3  [label="authenticate/None"];
  s3 -> s3  [label="sa_quick/None"];
  s3 -> s3  [label="ack_quick/None"];
  s3 -> s3  [label="sa_main_err/None"];
  s3 -> s3  [label="key_ex_main_err/None"];
  s3 -> s3  [label="authenticate_err/None"];
  s3 -> s3  [label="sa_quick_err/None"];
  s3 -> s3  [label="ack_quick_err/None"];
  s4 -> s4  [label="sa_main/None"];
  s4 -> s4  [label="key_ex_main/ERROR_NOTIFICATION"];
  s4 -> s5  [label="authenticate/ISAKMP_AUTH"];
  s4 -> s4  [label="sa_quick/None"];
  s4 -> s4  [label="ack_quick/None"];
  s4 -> s4  [label="sa_main_err/None"];
  s4 -> s4  [label="key_ex_main_err/ERROR_NOTIFICATION"];
  s4 -> s4  [label="authenticate_err/None"];
  s4 -> s4  [label="sa_quick_err/None"];
  s4 -> s4  [label="ack_quick_err/None"];
  s5 -> s5  [label="sa_main/None"];
  s5 -> s5  [label="key_ex_main/ERROR_NOTIFICATION"];
  s5 -> s5  [label="authenticate/None"];
  s5 -> s5  [label="sa_quick/IPSEC_SA"];
  s5 -> s5  [label="ack_quick/None"];
  s5 -> s5  [label="sa_main_err/None"];
  s5 -> s5  [label="key_ex_main_err/ERROR_NOTIFICATION"];
  s5 -> s5  [label="authenticate_err/None"];
  s5 -> s5  [label="sa_quick_err/None"];
  s5 -> s5  [label="ack_quick_err/None"];
```

```
  __start0 [label="", shape=none];
  __start0 -> s1  [label=""];
}
```

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987. (Cited on pages 1 and 6.)

[2] AVM Computersysteme Vertriebs GmbH. Connecting the FRITZ!Box with a company's VPN. `https://en.avm.de/service/vpn/connecting-the-fritzbox-with-a-companys-vpn-ipsec/`, 2022. Online; accessed 13-January-2023. (Cited on pages 1, 3 and 13.)

[3] Elaine Barker, Quynh Dang, Sheila Frankel, Karen Scarfone, and Paul Wouters. Guide to IPsec VPNs. Technical Report 800-77, Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, June 2020. (Cited on page 1.)

[4] Tom Britton, Li-Jung Jeng, Gary Carver, Pei-Yuan Cheak, and Toby Katzenellenbogen. Reversible debugging software. Cambridge Judge Business School, 2013. (Cited on page 19.)

[5] Harkins Carrel. The Internet Key Exchange (IKE). RFC 2409, The Internet Society, January 1998. https://www.rfc-editor.org/rfc/rfc2409.txt. (Cited on pages 1, 56 and 57.)

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

[7] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 193–206. USENIX Association, 2015. (Cited on page 3.)

[8] Deetah. American Fuzzy Lop Screenshot. `https://upload.wikimedia.org/wikipedia/commons/0/05/American_fuzzy_lop's_afl-fuzz_running_on_a_test_program.png`, 2015. [Online; accessed 30-May-2023]. (Cited on page 9.)

[9] John Ellson, Emden Gansner, Stephen North, and Eleftherios Koutsofios. *Drawing graphs with dot*. AT&T Research., `https://www.graphviz.org/doc/info/lang.html`, October 2022. [Online; accessed 31-August-2023]. (Cited on page 66.)

[10] Kaufman Hoffman Nir Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, The Internet Society, September 2010. https://www.rfc-editor.org/rfc/rfc5996.txt. (Cited on pages 1 and 13.)

[11] Robert Swiecki et al. Honggfuzz. `https://github.com/google/honggfuzz`, 2016. [Online; accessed 01-September-2023]. (Cited on page 3.)

[12] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. A year in lockdown: how the waves of COVID-19 impact internet traffic. *Communications of the ACM*, 64(7):101–108, 2021. (Cited on page 1.)

[13] Niels Ferguson and Bruce Schneier. A cryptographic evaluation of IPsec. Technical report, Counterpane Internet Security, Inc., 1999. https://api.semanticscholar.org/CorpusID:16387997. (Cited on page 13.)

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. (Cited on page 3.)

[15] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016. (Cited on pages 1 and 3.)

[16] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits W. Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, pages 142–151. ACM, 2017. (Cited on pages 1 and 3.)

[17] Jiaxing Guo, Chunxiang Gu, Xi Chen, and Fushan Wei. Model learning and model checking of IPsec implementations for Internet of Things. *IEEE Access*, 7:171322–171332, 2019. (Cited on pages 2 and 3.)

[18] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, pages 315–327. Springer, 2003. (Cited on page 6.)

[19] Malte Isberner, Falk Howar, and Bernhard Steffen. The Open-Source LearnLib. In *Computer Aided Verification*, pages 487–495, Cham, 2015. Springer International Publishing. (Cited on page 3.)

[20] Michael J Kearns and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994. (Cited on pages 1 and 8.)

[21] Maughan, Schertler, Schneider, and Turner. Internet Security Association and Key Management Protocol. RFC 2408, The Internet Society, 1998. https://www.rfc-editor.org/rfc/rfc2408.html. (Cited on pages 26, 55 and 56.)

[22] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990. (Cited on page 8.)

[23] Edi Muškardin, Bernhard K Aichernig, Andrea Pferscher, and Benjamin Wunderling. Mining Digital Twins of a VPN Server. *Preproceedings of the Workshop on Applications of Formal Methods and Digital Twins*, 2023. (Cited on page 4.)

[24] Edi Muškardin, Bernhard K Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering*, 18:1–10, 03 2022. (Cited on pages 2, 7 and 21.)

[25] Jomilė Nakutavičiūtė. The best VPN protocols. `https://nordvpn.com/de/blog/protocols/`, 2021. NordVPN [Online; accessed 31-August-2023]. (Cited on page 1.)

[26] Tomas Novickis, Erik Poll, and Kadir Altan. *Protocol state fuzzing of an OpenVPN*. PhD thesis, Radboud University, 2016. (Cited on page 3.)

[27] Joshua Pereyda. boofuzz Documentation. `https://boofuzz.readthedocs.io/`, 2022. [Online; accessed 19-April-2023]. (Cited on page 9.)

[28] Andrea Pferscher and Bernhard K Aichernig. Fingerprinting Bluetooth Low Energy devices via active automata learning. In *Formal Methods: 24th International Symposium, FM 2021s Proceedings 24*, pages 524–542. Springer, 2021. (Cited on pages 1 and 3.)

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

[30] strongSwan Project. IPsec and Related Standards. https://docs.strongswan.org/docs/5.9/features/ietf.html, 2021. [Online; accessed 13-May-2023]. (Cited on page 56.)

[31] Ronald L. Rivest and Robert E. Schapire. Inference of Finite Automata Using Homing Sequences. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 411–420, 1989. (Cited on page 7.)

[32] K. Serebryany. libFuzzer: a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2015. LLVM Project [Online; accessed 19-April-2023]. (Cited on page 3.)

[33] Kostya Serebryany. OSS-Fuzz-Google's continuous fuzzing service for open source software. In *USENIX Security symposium*. USENIX Association, 2017. (Cited on page 3.)

[34] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2*, pages 207–222. Springer, 2009. (Cited on page 6.)

[35] Richard Sharpe, Ed Warnicke, and Ulf Lamping. Wireshark User's Guide. https://www.wireshark.org/docs/wsug_html_chunked/, n.d. [Online; accessed 28-April-2023]. (Cited on page 19.)

[36] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 325–345. Springer, 2018. (Cited on page 3.)

[37] strongSwan Documentation. strongSwan documentation. https://docs.strongswan.org/docs/5.9/, 2021. (Cited on page 16.)

[38] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. Model-Based Testing IoT Communication via Active Automata Learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 276–287. IEEE, 2017. (Cited on pages 1, 3 and 22.)

[39] The Libreswan Project. libreswan documentation. https://libreswan.org/wiki/, 2021. [Online; accessed 26-April-2023]. (Cited on page 16.)

[40] TOPDapp. py-diffie-hellman. https://github.com/TOPDapp/py-diffie-hellman, 2021. [Online; accessed 22-November-2022]. (Cited on page 53.)

[41] Petar Tsankov, Mohammad Dashti, and David Basin. Semi-valid input coverage for fuzz testing. In *2013 International Symposium on Software Testing and Analysis, ISSTA 2013 - Proceedings*, pages 56–66. Association for Computing Machinery, 07 2013. (Cited on page 3.)

[42] Huan Yang, Yuqing Zhang, Yu-pu Hu, and Qi-xu Liu. IKE vulnerability discovery based on fuzzing. *Security and Communication Networks*, 6(7):889–901, 2013. (Cited on page 3.)

[43] Michal Zalewski. american fuzzy lop. https://github.com/google/AFL, 2020. (Cited on page 9.)

[44] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Search-based fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. [Online; accessed 07-January-2023]. (Cited on pages 2 and 4.)