Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol

Benjamin Wunderling

Model Learning and Fuzzing of the IPsec-IKEv1 VPN Protocol

Benjamin Wunderling BSc

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

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

Graz, 10 Nov 2021

Abstract

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

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

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

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

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

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

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

Keith Andrews

Graz, Austria, 10 Nov 2021

Credits

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

- The thesis was written using Keith Andrews' skeleton thesis [1].

Chapter 1

Introduction

1.1 Motivation

Virtual Private Network (VPN) are used to allow secure communication over an insecure channel. They function by creating a secure encrypted tunnel through which users can send their data. Example use cases include additional privacy from prying eyes such as Internet Server Providers, access to region-locked online content and secure remote access to company networks. The importance of VPN software has increased dramatically since the beginning of the COVID-19 pandemic due to the influx of people working from home [8]. This makes finding vulnerabilities in VPN software more critical than ever. Internet Protocol Security (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 [7], with IKEv2 being the newer and recommended version, according to a report by the National Institute of Standards and Technology [4]. However, despite IKEv2 supposedly replacing its predecessor, IKEv1, sometimes also called Cisco IPsec, is still in widespread use today. This is reflected by the company AVM to this day only offering IKEv1 support for their popular FRITZ!Box routers [13]. Additionally, IKEv1 is also used for the L2TP/IPsec protocol, one of the most popular VPN protocols according to NordVPN [10]. The widespread usage of IPsec-IKEv1, combined with its relative age and many options makes it an interesting target for security testing.

1.2 Research Problems and Goals

State machines of protocol implementations are useful tools in state-of-the-art software testing. They have, e.g., been used to detect specific software implementations, or to generate test cases automatically [26, 27]. Mealy machines are a type of state machine that can be used to describe the output of a system when given the current state and an external input. Often we are interested in testing software without knowing its exact inner workings, for example due to the software being closed-source. We call these systems black-box systems. It is not unusual for VPN software to be closed-source and therefore a black-box system for testers. However, despite lacking information on the inner structure of a black-box system, the state machine of the system can still be extracted. One method of generating the state machine of such a system is to use active automata learning. A notable example of an active automata learning algorithm is the L^* algorithm by Angluin [2]. In L^*, a learner queries the system under learning (SUL) and constructs a behavioral model of the SUL through its responses.

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. [14] tested IPsec-IKEv2 using automata learning and model checking, however so far, no studies have focused on IKEv1 in the context of automata learning. Therefore our goal was to black-box test the IPsec-IKEv1 protocol using automata learning in combination with automata-based fuzzing. We used the active automata learning framework AALpy [21] with a custom mapper to learn the state machines of various IPsec-IKEv1 server implementations. We then further utilized the learned models for fuzzing and fingerprinting.

1.3 Structure

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

Chapter 2

Related Work

The aim of this chapter is to give a brief overview of related work, focusing mainly on automata learning and testing of secure communication protocols.

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

Model learning has been applied to a variety of different protocols, including many security-critical ones. Joeri de Ruiter and Erik Poll [29] 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. [32] 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. [31] using automata learning to test implementations on various routers, finding servery vulnerabilities. Fiterau and Brostean combined model learning with the related field of model checking, in which an abstract model is checked for specified properties to ensure correctness. In their work they learned and analyzed both TCP [11] and SSL [12] implementations, showcasing several implementation deviations from their respective RFC specifications. The Bluetooth Low Energy (BLE) protocol was investigated by Pferscher and Aichernig [26]. 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. [24] and Daniel et al. [6] learned models of the OpenVPN protocol and showcased how to use the learned models to perform protocol fuzzing. In contrast to our approach, they chose to learn a more abstract model of the entire OpenVPN session, where details about the key exchange were abstracted in the learned model. Even more closely related to our work, Guo et al. [14] used automata learning to learn and test the IPsec-IKEv2 protocol setup to use certificate-based authentication. They used the LearnLib [16] library for automata learning and performed model checking of the protocol, using the learned state machine. In contrast, our work focuses on the IPsec-IKEv1 protocol, the predecessor of IPsec-IKEv2, which has not yet been tested with methods utilizing automata learning. Pre-shared keys (PSKs) are used for authentication and the learned model serves as a basis for fuzz-testing. The two protocol versions differ 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 [13]. Our work completes the coverage of learning-based testing approaches for both IKE versions.

Chapter 3

Preliminaries

3.1 Deterministic Finite Automata

A deterministic finite automaton (DFA) is a finite-state machine that is defined as a quintuple $D = \{S, s_0, \Sigma, \delta, F\}$ 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, δ is the state-transition function $\delta\colon S \times \Sigma \longrightarrow S$ that maps a state and an element of the input alphabet to another state in S and F is the set of accepting states. DFA do not have any output.

3.2 Mealy Machines

Mealy machines are a modeling formalism for reactive systems such as communication protocols. Mealy machines are finite-state machines in which each transition is labeled with an input and a corresponding output action. More formally, a Mealy machine is defined as a 6-tuple $M = \{S, s_0, \Sigma, O, \delta, \lambda\}$, where S is a finite set of states, $s_0 \in S$ is the initial state, Σ is a finite set called the input alphabet, O is a finite set called the output alphabet, δ is the state-transition function $\delta\colon S \times \Sigma \longrightarrow S$ that maps a state and an element of the input alphabet to another state in S. In other words, the choice of each new state is defined by the current state and an external input. Finally, λ is the output function $\lambda\colon S \times \Sigma \longrightarrow O$ which maps a state-input alphabet pair to an element of the output alphabet O. The main difference to a DFA is, that Mealy machines have an output and DFA do not. We use Mealy machines to model the state of learned IPsec implementations.

3.3 Automata Learning

Automata learning refers to methods of learning the state model, or automaton, of a system through an algorithm or process. Automata learning algorithms generate a model that describes the behavior of the SUL. We differentiate between active and passive automata learning. In passive automata learning, models are learned based on a given data set describing the behavior of the SUL, e.g. log files. In contrast, in 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” [3]. In this seminal paper, Angluin introduced the concept of the minimally adequate teacher (MAT) as well as an algorithm for learning regular languages from queries and counterexamples, called L^*. In it, a student questions a teacher and constructs a hypothesis based on its responses. The hypothesis is then tested through equivalence queries which check if the hypothesis correctly matches the regular language being learned. While the L* algorithm was originally designed to learn DFA, it can be simply extended to work for Mealy machines by making use of the similarities between DFA and Mealy machines [15, 30]. Variants of the L^* algorithm are still used for learning deterministic automata to this day, e.g. by the AAL python library AALpy [22]. While the original L^* algorithm was designed to learn DFA formalizing regular languages, the algorithm can be extended to learn Mealy machines [23]. While many modern implementations, including AALpy use improved versions of L^*, such as with advanced counter example processing by Rivest and Shapire [28], fundamentally they still resemble the original algorithm by Angluin.

The MAT concept is also used by other active learning algorithms. Kearns and Vazirani [18] proposed an active learning algorithm that uses an underlying tree-based data structure called a classification tree to construct a model. We refer to their learning algorithm as KV. Published later than L^*, it boasts a more compact method of representing learned data, which, on average, leads to the KV algorithm requiring less membership queries than L^* to learn a model. Especially for learning internet protocols and other systems where communication with the SUL can be very time consuming, this can result in a significant performance increase. However, the KV algorithm does on average require more equivalence queries to learn a system. Both algorithms used in this paper are briefly explained below.

3.3.1 L*

L^* uses the MAT framework to learn an unknown regular language L. This means, that we define both a learner and a teacher. The teacher must respond to two types of queries posed by the learner, namely membership and equivalence queries. Queries are built using a fixed input alphabet Σ where $L \subseteq \Sigma^*$ must hold. Membership queries consist of a word $s \in \Sigma^*$ and must be answered with either "yes" if $s \in L$, or "no" if not. In other words, membership queries are used to check if a given word is part of the language being learned. The learner proposes a regular language Lprop and sends it to the teacher as an equivalence query. The teacher must confirm if Lprop is equivalent to L, answering with "yes" if the equivalence holds true, or else returning a counterexample c proving the two languages are different. So $c \in L\text{prop} \iff c \notin L$. In other words, equivalence queries are used to verify if the learner has successfully learned the target language L or if not, return a counterexample detailing the differences. The results of the membership queries are stored in an observation table $O = (S, E, T)$, where S is a prefix-closed set of words representing candidates for states of L_{prop}, E a suffix-closed set of words used to distinguish between candidates and T a transition function $(S \cup S \cdot \Sigma) \cdot E \rightarrow 0, 1$, with $S \cdot \Sigma$ being the concatenation of words in S with words in Σ and $\cup$ referring to the union of two sets. Essentially, if visualized as a 2D array where the rows are labeled with elements in $(S \cup S \cdot \Sigma)$ and columns with elements in E, the entries in the table are ones, if the word created by appending the row-label to the column-label is accepted by L and zeros if not.

The goal of L^* is to learn a DFA acceptor for L using the observation table. S-labeled rows correspond to states in the acceptor under construction. E-labeled columns represent individual membership query results. For the observation table to be transformable into a DFA acceptor, it must first be closed and consistent.

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

Listing 3.1 shows the workings of the basic L^* algorithm by Angluin. At the start of the algorithm, the observation table is initialized with $S = E = \{\epsilon\}$ in Line 2. The function $populate(O)$ in Line 3 extends T to $(S \cup S \cdot \Sigma) \cdot E$ by asking membership queries for all table entries still missing membership information. Next, until a equivalence query succeeds, the observation table is repeatedly brought to a closed and consistent state by expanding the S and E sets respectively as detailed previously. This occurs in the while loop in Line 6 until the observation table is brought to a closed and consistent state. Following this, L_{prop} is constructed from O and used in an equivalence query in Line 20. If the equivalence query returns "yes", the algorithm terminates, returning the learned DFA. If not, the returned counterexample is used to update the observation table and the algorithm loops back to Line 5.

3.3.2 KV

Another notable AAL algorithm is the KV algorithm published in 1994 by Kearns and Vazirani [18]. It is designed to work with the same MAT framework as L^*, but aims to minimize the amount of membership queries needed to learn a finite automaton M. The *KV* algorithm does this by organizing learned information in an ordered binary tree called a classification tree C_T as opposed to the table structure utilized by L^*. As a trade-off, the *KV* algorithm requires on average more equivalence queries than L^*. Intuitively, L^* must perform membership queries for every entry in the observation table to differentiate between possible states, whereas *KV* requires only a subset to distinguish them.

In the *KV* algorithm, learned data is stored in two sets called the access strings set S and the distinguishing strings set D. The information contained is stored in a binary tree called the classification tree C_T. Every word $s \in S$ represents a distinct and unique state of the automaton M. In other words, any s when applied starting in the initial state of M leads to a unique state $M[s]$, where $M[s]$ signifies the state reached by applying s to the finite automaton M. The distinguishing strings set is defined as the set of words $d \in D$ where for each pair $s, s' \in S, s \neq s'$ there exists a $d \in D$ such that either $M[s \cdot d]$ or $M[s' \cdot d]$ is an accepting state. D is used to ensure that their are no ambiguous states. The binary tree C_T is formed from S, D by setting parent nodes to be words from D and leaf nodes as words from S. The root node is

```
1   Initialization:
2   Set observation table $O = (S,E,T)$ with $S,E = \{\epsilon\}$.
3   $populate(O)$.
4
5   repeat:
6     while $O$ is not closed or not consistent do
7       if $O$ is not closed then
8         choose $s_1 \in S, \sigma \in \Sigma$ such that
9         $row(s_1 \cdot \sigma) \neq row(s) \; \forall s \in S$
10        add $s_1 \cdot \sigma$ to $S$
11        $populate(O)$
12      end
13      if $O$ is not consistent then
14        choose $s_1, s_2 \in S, \sigma \in \Sigma$ and $e \in E$ such that
15        $row(s_1) = row(s_2)$ and $T(s_1 \cdot \sigma \cdot e) \neq T(s_2 \cdot \sigma \cdot e)$
16        add $\sigma \cdot e$ to $E$
17        $populate(O)$
18      end
19    end
20    Construct $L_{prop}$ from $O$ and perform an equivalence query.
21    if query returns a counterexample $c$ then
22      add all prefixes of $c$ to $S$
23      $populate(O)$
24    end
25  until teacher replies "yes" to equivalence query $L_{prop} \equiv L$
26  return $L_{prop}$
```

Listing 3.1: L^* algorithm

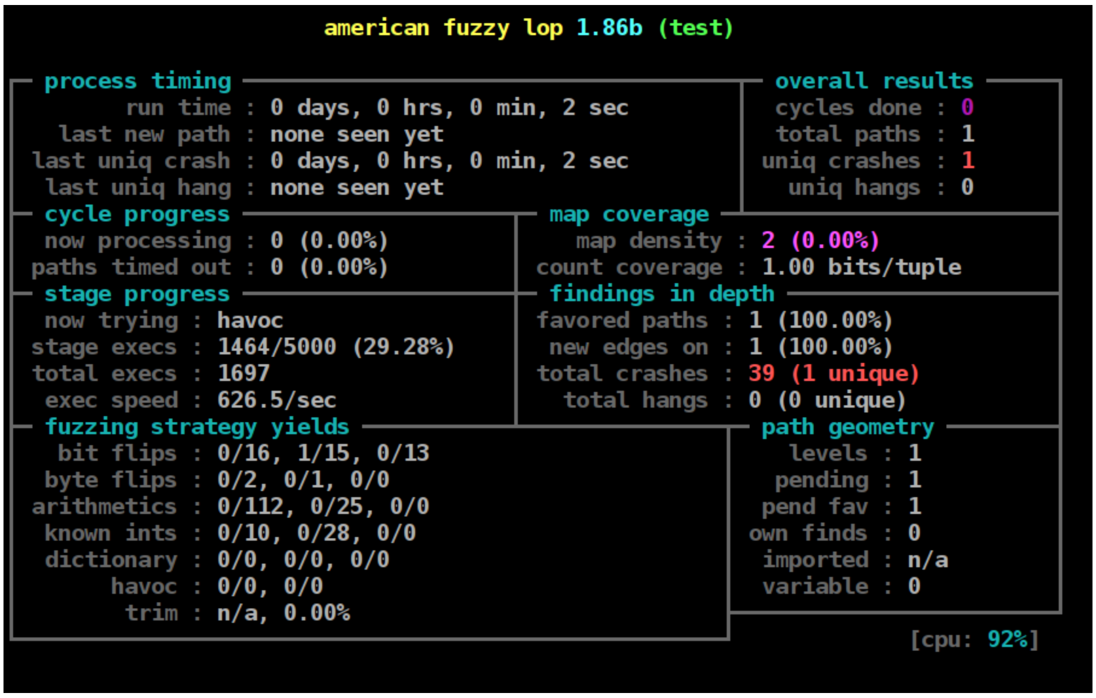
set to the empty word ϵ. For each node of the tree, starting from the root node, each right subtree contains access strings to accepting states while left subtrees contain access strings to rejecting states of M. Given a new word s', we simply start at the root nodes, then sift down the tree by executing a membership query for $s' \cdot \lambda_1$ and depending on if the query returns “yes” or “no” continuing with the left or right subtree until we reach a leaf node labeled with s. If $s' = s$ then the states are equivalent, otherwise the classification tree is updated to include another leaf node representing the newly learned distinct state s'. The main learning loop of the KV algorithm is shown in more detail in Listing 3.2. Following the initialization of the classification table in Lines 2-5, new states learned from counterexamples are repeatedly added until an equivalence query is successful in Lines 8-14. The $Update(C_T, c)$ function in Line 14 adds a new leaf to the C_T based on a counterexample c returned from an equivalence query.

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

Listing 3.2: KV algorithm

3.4 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 utilitis [20], it can be applied to test the reliability and security of any system that takes input. While originally just a simple random text-generation program, modern fuzzers are often more complex and boast a variety of features. Modern fuzzers mainly differ on the method of data generation and can be roughly categorized as generation-based or mutation-based fuzzers. Generation-based fuzzers generate their data from scratch, whereas mutation-based fuzzers modify, or “mutate” existing data. Additionally, one can categorize fuzzers based on how much knowledge they have on the SUT, or in other words, how much information on the expected input structure is known by the fuzzer. 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 (white-box testing), in other scenarios where no source code is public (black-box testing), other methods of guaranteeing meaningful coverage are required. One possible solution, usable



```
                    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%]
```

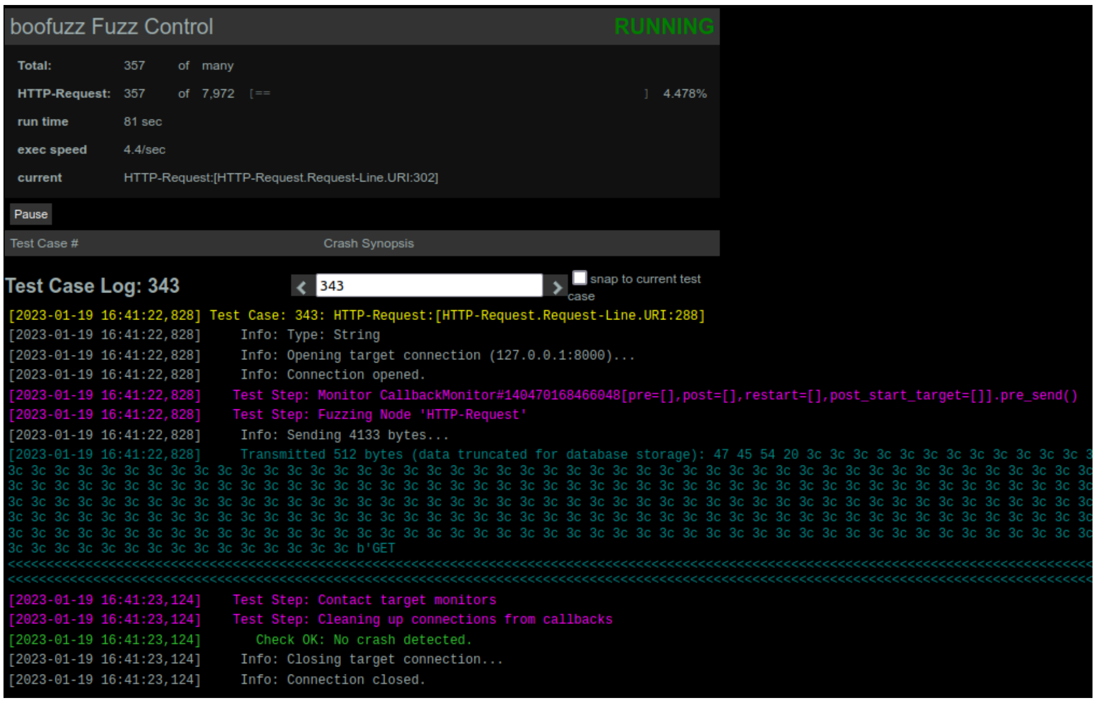
Figure 3.1: AFL fuzzer

for black-box scenarios, could be to use a model representation of the SUT to generate more relevant inputs and therefore better coverage. 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.

Two popular examples of fuzzers are american fuzzy lop (AFL) [34] and boofuzz [25]. AFL is a software fuzzer, written mainly in C, that uses genetic algorithms in combination with instrumentation to achieve high code coverage of the SUT. The instrumentation has to be added to the the target by compiling the SUT using a custom compiler provided by AFL. AFL has been successfully used to find bugs in many high-profile projects such as OpenSSH [1], glibc [2] and even linux memory management [3].

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

[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 3
3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c
3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c
3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c
3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c
3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c
3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c 3c b'GET
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
[2023-01-19 16:41:23,124]    Test Step: Contact target monitors
[2023-01-19 16:41:23,124]    Test Step: Cleaning up connections from callbacks
[2023-01-19 16:41:23,124]      Check OK: No crash detected.
[2023-01-19 16:41:23,124]     Info: Closing target connection...
[2023-01-19 16:41:23,124]     Info: Connection closed.
```

Figure 3.2: Boofuzz fuzzer

other unexpected behavior and the framework can furthermore be instructed to restart or reset the SUT when needed. In this paper, we use boofuzz primitives to generate our values for fuzzing, as detailed in Chapter 5.

3.5 IPsec

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

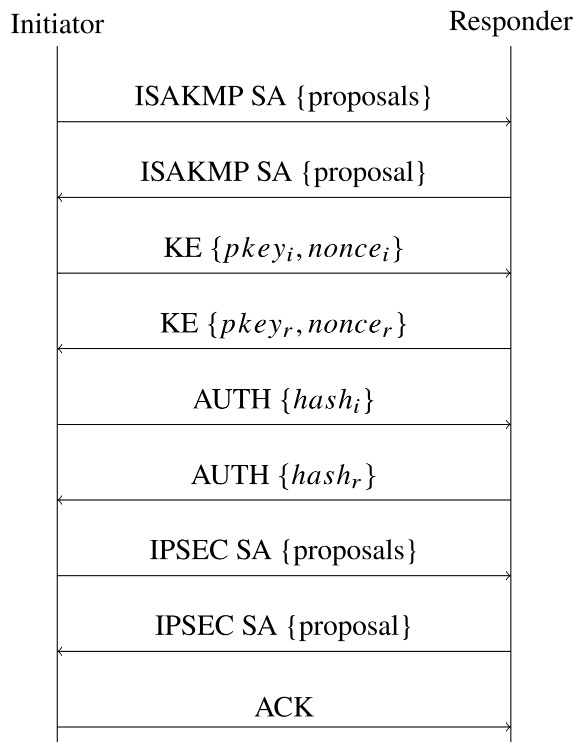


Figure 3.3: IKEv1 between two parties

The IKEv1 protocol works in two main phases, both relying on the Internet Security Association and Key Management Protocol (ISAKMP). Additionally, phase one can be configured to proceed in either Main Mode or Aggressive Mode. A typical exchange between two parties, an initiator (e.g. an employee connecting to a company network) and a responder (e.g. a company VPN server), using Main Mode for phase one and PSK authentication, can be seen in Figure 3.3. In contrast, aggressive mode reduces the number of packets in phase one to only three. While faster, this method is considered to be less secure, as the authentication hashes are sent in clear text. In phase one (Main Mode), the initiator begins by sending a Security Association (SA) to the responder. A SA essentially details important security attributes required for a connection such as the encryption algorithm and key-size to use, as well as the authentication method and the used hashing algorithm. These options are bundled in containers called proposals, with each proposal describing a possible security configuration. While the initiator can send multiple proposals to give the responder more options to choose from, the responder must answer with only one proposal, provided both parties can agree upon one of the suggested proposals. This initial communication is denoted as *ISAKMP SA* in Figure 3.3 and also exchanges initiator/responder cookies, tokens used as identifiers for the remainder of the connection. Subsequently, the two parties perform a Diffie-Hellman key exchange, denoted as *KE*, with the public key shorted to $pkey$, and send each other nonces used to generate a shared secret key `SKEYID` as detailed in Listing 3.3. Here, `PSK` refers to the pre-shared key, `Ni/Nr` to the initiator/responder nonce and `CKY-I/CKY-R` to the initiator/responder identifier cookie. The symbol "|" is used to signify concatenation of bytes, not a logical or. Note that IKEv1 allows using various different authentication modes aside from PSK, including public key encryption and digital signatures. `SKEYID` is used as a seed key for all further session keys `SKEYID_d`, `SKEYID_a`, `SKEYID_e`, with g^{xy} referring to the previously calculated shared Diffie-Hellman secret and `prf` to a pseudo-random function (in our case, HMAC). 0, 1 and 2 are constants used to ensure that the resulting key material is different and unique for each type of key. Following a successful key exchange, all further messages of phase one and two are encrypted using a key derived from `SKEYID_e` and `SKEYID_a` for authentication. Finally, in the last section of phase one *AUTH*, both parties exchange and verify hashes to confirm the key generation was successful. Once verification succeeds, a secure channel is created and used for phase two communication. If phase one uses Aggressive Mode, then only three packets are needed to reach phase two. While quicker, the downside of Aggressive Mode is that the communication of the hashed authentication material happens without encryption. This means, that using short PSKs in combination with Aggressive Mode is inherently insecure, as the unencrypted hashes are vulnerable to brute-force attacks provided a short key-size [4]. The shorter phase two (Quick Mode) begins with another SA exchange, labeled with *IPSEC SA* in Figure 3.3. This time, however, the SA describes the security parameters of the ensuing ESP/AH communication and the data is sent authenticated and encrypted using the cryptographic material calculated in phase one. This is followed by a single acknowledge message, *ACK*, from the initiator to confirm the agreed upon proposal. After the acknowledgment, all further communication is done via ESP/AH packets, using *SKEYID_d* as keying material.

```
# For pre-shared keys:
SKEYID = prf(PSK, Ni_b | Nr_b)

# to encrypt non-ISAKMP messages (ESP)
SKEYID_d = prf(SKEYID, g^xy | CKY-I | CKY-R | 0)

# to authenticate ISAKMP messages
SKEYID_a = prf(SKEYID, SKEYID_d | g^xy | CKY-I | CKY-R | 1)

# for further encryption of ISAKMP messages in phase two
SKEYID_e = prf(SKEYID, SKEYID_a | g^xy | CKY-I | CKY-R | 2)
```

Listing 3.3: IKE Keying

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

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

Compared to other protocols, IPsec offers a high degree of customizability, allowing it to be fitted for many use cases. However, in a cryptographic evaluation of the protocol, Ferguson and Schneier Ferguson and Schneier [9] 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 [17]. Nevertheless, IPsec-IKEv1 is still in wide-spread use to this day, with the largest router producer in Germany, AVM, still only supporting IKEv1 in their routers [13]. We use IPsec-IKEv1 with Main Mode and ESP in this paper and focus on the IKE protocol as it is the most interesting from an AAL and security standpoint.

Chapter 4

Model Learning

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

The goal of our setup is to learn a Mealy machine that models the SUL. We refer to it as the *learned model*, or simply *model*. 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 sequence of (multiple) inputs as a *run*. We refer to one execution of our learning program as one learning attempt. A successful learning attempt is one that resulted in a model of the SUL. As we are working with Mealy machines, the term *output query* is used instead of membership query.

consider n
entirely to
ation secti

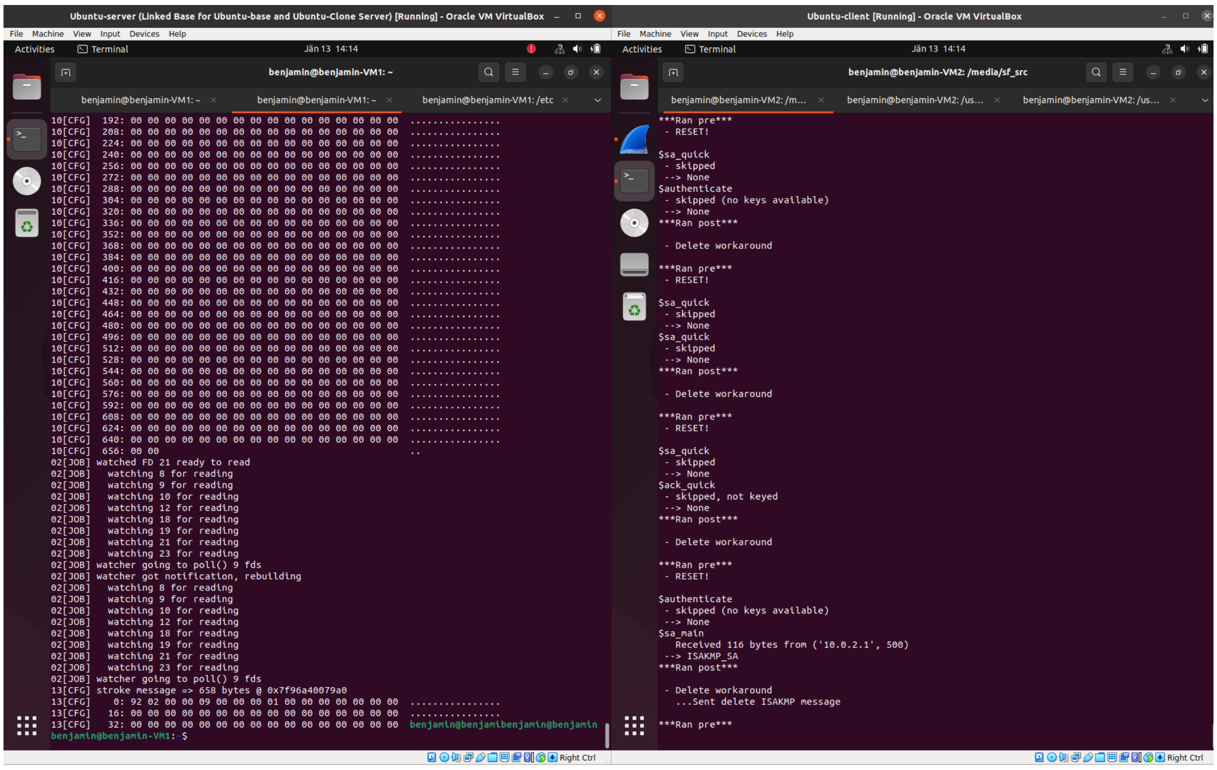
4.1 Environment Setup

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

appendix

this

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


Ubuntu-server (Linked Base for Ubuntu-base and Ubuntu-Clone Server) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Jän 13 14:14
benjamin@benjamin-VM1: ~
benjamin@benjamin-VM1: ~
benjamin@benjamin-VM1: ~
benjamin@benjamin-VM1: /etc
Right Ctrl
Ubuntu-client [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Jän 13 14:14
benjamin@benjamin-VM2: /media/sf_src
benjamin@benjamin-VM2: /m...
benjamin@benjamin-VM2: /us...
benjamin@benjamin-VM2: /us...
Right Ctrl

```
10[CFG]  192: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  208: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  224: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  256: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  272: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  288: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  304: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  320: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  336: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  352: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  368: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  384: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  400: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  416: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  432: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  448: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  464: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  496: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  512: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  528: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  544: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  560: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  576: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  592: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  608: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  624: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10[CFG]  656: 00 00                                            ..
02[JOB] watched FD 21 ready to read
02[JOB]   watching 8 for reading
02[JOB]   watching 9 for reading
02[JOB]   watching 10 for reading
02[JOB]   watching 12 for reading
02[JOB]   watching 18 for reading
02[JOB]   watching 19 for reading
02[JOB]   watching 21 for reading
02[JOB]   watching 23 for reading
02[JOB] watcher going to poll() 9 fds
02[JOB] watcher got notification, rebuilding
02[JOB]   watching 8 for reading
02[JOB]   watching 9 for reading
02[JOB]   watching 10 for reading
02[JOB]   watching 12 for reading
02[JOB]   watching 18 for reading
02[JOB]   watching 19 for reading
02[JOB]   watching 21 for reading
02[JOB]   watching 23 for reading
02[JOB] watcher going to poll() 9 fds
13[CFG] stroke message => 658 bytes @ 0x7f96a40079a0
13[CFG]    0: 92 02 00 00 09 00 00 00 01 00 00 00 00 00 00 00  ................
13[CFG]   16: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
13[CFG]   32: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  benjamin@benjamibenjamin@benjamin
benjamin@benjamin-VM1:~$
```

```
***Ran pre***
 - RESET!

$sa_quick
 - skipped
 --> None
$authenticate
 - skipped (no keys available)
 --> None
***Ran post***

 - Delete workaround

***Ran pre***
 - RESET!

$sa_quick
 - skipped
 --> None
$sa_quick
 - skipped
 --> None
***Ran post***

 - Delete workaround

***Ran pre***
 - RESET!

$sa_quick
 - skipped
 --> None
$ack_quick
 - skipped, not keyed
 --> None
***Ran post***

 - Delete workaround

***Ran pre***
 - RESET!

$authenticate
 - skipped (no keys available)
 --> None
$sa_main
   Received 116 bytes from ('10.0.2.1', 500)
 --> ISAKMP_SA
***Ran post***

 - Delete workaround
   ...Sent delete ISAKMP message

***Ran pre***
```

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

in conjunction with our custom mapper to communicate with and learn the model of an IPsec server. Figure 4.2 shows a typical learning attempt using two connected VMs. The right VM shows the output of an underway learning attempt, while the left one shows the corresponding Strongswan server logs.

4.2 Learning Setup

We used the Python automata learning library AALpy to learn our automata. It 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. However, learning automata with AALpy follows the same pattern, regardless of the type of automata.

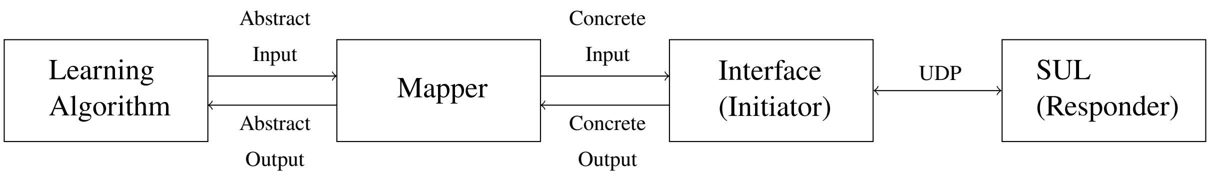


Figure 4.2: Automata Learning Setup

Figure 4.2 gives an overview of the learning process, adapted from Tappler et al. [33]. To begin, the learning algorithm sends abstract inputs chosen from the input alphabet to the mapper class, which

[4]TODO: github or supplementary material link

converts it to concrete inputs. The concrete inputs are then sent to the SUL, by means of a communication interface. In our case, the mapper class comprises the major portion of our work 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 4.1: Mapping protocol to input alphabet names

To begin learning an automaton with AALpy, we must first choose a suitable input alphabet encompassing the language known by the server, as well as the learning algorithm to be used. Our chosen input alphabet corresponds to the IKEv1 protocol messages shown in Figure 3.3. The protocol messages map to abstract inputs of the input alphabet as shown in Table 4.1. We use both the L^* and KV algorithms for learning with a state prefix equivalence oracle that provides state-coverage by means of random walks started from each state. The equivalence oracle is used by the chosen learning algorithm to test for conformance between the current hypothesis and the SUL, giving either a counterexample on failure, or confirmation that we have learned the SUL. This corresponds to an equivalence query. We also enabled several optional AALpy features including caching and non-determinism checks to improve the learning process. An overview of the relevant learning algorithm initialization code can be seen in Listing 4.1. Line 3 shows the used input alphabet, Line 4 the used equivalence oracle and Line 5 the used learning algorithm. Both the equivalence oracle and learning algorithm take the input alphabet and an object representing an interface to the SUL as parameters, were the SUL interface is defined as shown below in **??** and can execute inputs on, as well as reset the the actual SUL. The equivalence oracle is also passed as a parameter to the learning algorithm with a few additional optional parameters specifying the type of automaton to learn and enabling non-determinism checking and caching.

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

Listing 4.1: Equivalence Query code

In the SUL interface, we defined the *step* and *reset* methods as can be seen in Listing 4.2. *step*, seen in Line 3, is used to execute input actions. *reset*, show in lines 8-12, reverts the SUL to an initial state.

An output query is a sequence of inputs executed on the SUL. Every input sequence is executed starting from an initial state, hence the need for a *reset* method. Used in combination, *step* and *reset* allow asking output queries to the SUL. *pre* is called before each output query and *post* afterwards. The abstract input chosen from the input alphabet is passed on to the mapper class for further processing. Line 4 shows how a function, corresponding to the abstract input, is called in the mapper class and the return value (abstract output) is passed on to the learning algorithm.

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

Listing 4.2: SUL interface

The mapper class implements methods for each communication step in a typical IPsec-IKEv1 exchange, as described in Section 3, but referred to by their input alphabet name according to Table 4.1. This includes methods for *sa_main*, *key_ex_main*, *authenticate*, *sa_quick*, *ack_quick* packets. Additionally, the mapper class supports *DELETE* messages. The *DELETE* message is special in that it actually sends two packets which is required to delete all existing connections to the Strongswan server. It is critical for the correct functioning of *reset* that this input is executed correctly, hence it requires the SUL state be checked after execution, which is not feasible during learning. For these two reasons, it was mostly left out of the learning process. Furthermore, the mapper class contains a variety of helper functions used to handle the decryption and encryption of packets as well as parse received informational messages. Informational messages are mainly used in IPsec to return error codes when something goes wrong. To illustrate our mapper class, (simplified) excerpts from the *sa_main* method are shown in Listing 4.3. It shows how a Scapy packet is constructed out of many different individually configurable layers and fields, allowing for a high degree of flexibility and customizability. We can see in Line 5, how an ISAKMP transform is created, encompassing various security parameters. This is packed into a ISAKMP proposal packet first and then the resulting packet is packet into an ISAKMP SA packet in Line 6. The SA packet is appended to a generic top level ISAKMP packet in Line 8. In Line 9 we then send the ISAKMP packet to the SUL and receive its response (if any) back, already converted into a matching Scapy object. The response then undergoes a retransmission check and is then parsed and relevant data is used to update local values in lines 11-16.

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

```
# code excerpt from IPSEC_MAPPER.py

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

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

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

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

Listing 4.3: Excerpt of sa_main method code

used as the return value. Finally, the abstract error codes are returned to the learning algorithm which uses it update its internal data structures and improve its understanding of the SUL by updating the model.

4.3 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. We implement this using a combination of the ISAKMP *DELETE* request and general ISAKMP informational error messages. While *DELETE* alone works for established connections in phase two of IKE, we require informational error messages to trigger a reset in phase one, as delete does not work here sufficiently.

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

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. Each request, we store the response for use in the next message and update affected key material as needed. Most notably, the IVs are updated almost every request and differ between M-IDs. Informational requests also handle their IVs separately from other message types. For each request that we send, if available, we try to parse the response, decrypting it if necessary and resetting or adjusting our internal variables as required to match the server. To keep track of all the different M-IDs, we use a Python dictionary to map M-IDs to relevant keying and IV information. Usually, IVs are updated to the last encrypted block of the most recently sent or received message, though this behavior varies slightly between phases and for informational messages. Keeping track of IVs is required to continuously be able to parse encrypted server responses and extract meaningful information. Implementation of the mapper class, in particular encryption and decryption functionality, was hindered at times by unclear RFC-specifications, but this was overcome by manually comparing packet dumps and Strongswan logs to fix errors.

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 is the same for both responses. Our retransmission handling is covered in more detail in Section 4.4. 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 that case we decrypt the message using the corresponding parameters (as they are calculated and saved differently for informational messages), and return a code indicating the error being reported. Finally we catch and log unimplemented message types, but this case should not occur 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 minimum amount needed to still get deterministic results. Next, we categorized the server informational responses according to their severity and impact and then grouped the most common ones together under the same abstract response code. This decreased the amount of states that had to be learned at the cost of some informational loss. However, since any deviations or non-deterministic behavior would have been caught by the learning framework, we are confident that no important information was lost. Finally we switched out the L^* learning algorithm for KV, as KV can be more performative for learning environments where output queries are expensive operations. As IKEv1 is a networks protocol with quite a bit of communication in each phase and we additionally have to implement small timeouts to wait for the server, each individual output query can take several seconds. With hundreds of output queries required to learn the IPsec server, this results in a lot of time spent running the algorithm. Consequently, any decrease to the amount of output queries should, in theory, lead to an overall decrease in runtime. Since AALpy supports the KV algorithm, switching between the two learning algorithms is as easy as setting a simple flag as shown in line three of Listing 4.4 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 6.

```
# code excerpt from IPSEC_IKEv1_SUL.py

if kv:
learned_ipsec, info = run_KV(input_al, sul, eq_oracle, automaton_type='
    mealy', cex_processing='rs')
else:
learned_ipsec, info = run_Lstar(input_al, sul, eq_oracle=eq_oracle,
    automaton_type='mealy', cache_and_non_det_check=True)
```

Listing 4.4: Switching Learning Algorithms

4.4 Combating Non-determinism

Despite all the precautions taken to create a disturbance-free learning environment detailed in Section 4.1, the IPsec server still exhibited non-deterministic behavior, resulting in variance among the learned models. While the majority of learned models were identical, the outliers were significantly different, having differing amounts of states and transitions between them. To help decrease the remaining non-deterministic behavior, additional timeouts were added to all requests in order to give the server more time to correctly work through all incoming requests. This measure helped further decrease the amount of outlying automata learned, however it did not fully fix the issue. Examination of the outliers led to the discovery that all outlying behavior was concentrated around so-called retransmissions. Essentially, the IKE specification allows for previous messages to be retransmitted if deemed useful. A possible trigger could be the final message of an IKE exchange being skipped / lost. For example, if instead of an *AUTH* message, the server receives a phase two *IPSEC SA* message, the server would not know if it missed a message or if their was an error on the other parties side. According to the ISAKMP specification in RFC 2408 [19], 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, seem to trigger in a not-quite-deterministic fashion in phase two of IPsec IKEv1 protocol.

While interesting for fingerprinting, we wanted a deterministic model as a base case for model-based fuzzing, so we implemented checks in our mapper to allow the ignoring of retransmissions. The retransmission-filtering can be easily enabled or disabled through a simple flag and works by checking the message ID of incoming server responses against a list of previous message IDs (excluding zero, as it is the default message ID for phase one). If a repeated message ID is found, it is flagged as a retransmission and depending on the current filtering rule, ignored. With this addition, non-deterministic behavior no longer occurred, allowing the learning of a very clean model, as shown in Chapter 6, Figure 6.3. As an additional method of dealing with non-determinism but still keeping retransmissions, we can also catch non-determinism errors as they occur and repeat the offending query several times. If upon the first rerun the non-determinism does not occur again, we accept the existing value as the correct one and continue. If however, it persists for a set amount of repetitions with the same constant server response, we assume that the original saved response was incorrect and update it to the new one. With the non-determinism correcting added, the automata learning works without non-determinism errors and the learned models are consistent with one another.

Chapter 5

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 runs (sequences of inputs of the input alphabet), and present two different methods of generating runs to test, with an emphasis on reducing the total runtime of fuzzing to a reasonable amount.

less actual
those for t
ation. Exp
setup more
greater det
showcasin
process

5.1 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 potentially black-box scenario, some additional considerations and steps are required, as seen in Figure 5.1. 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 runs are passed to the fuzzer, choosing, at random, a single input of each run to fuzz, while still sending the other inputs surrounding the fuzzed input to the SUT as well. Additionally, the SUT is reset after each execution of a fuzzed run, ensuring that each fuzzed input is executed in exactly the same starting state of the SUT. Figure 5.1 shows an example of such a fuzzed run, with the fuzzed value being the third input of the run. 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 5.1. Each input that is sent to the SUT, depicted in red in Figure 5.1, is also executed on the Mealy machine in parallel, comparing the response codes of the SUT and Mealy machine. A mismatch between the response of the SUT and the expected one returned by the Mealy machine indicates, that a new state has been discovered. As the Mealy machine was created from the model of the SUT, any new state can be treated as interesting behavior worth further investigation. Run generation/selection, new state detection and fuzz input generation is all handled separately from the SUL interface, in a new script called *fuzzing* [1].

While this basic fuzzing procedure is rather straightforward, the questions remaining is how to choose or generate the runs that are to be fuzzed, how the reference model is created/learned and how the test data for fuzzed inputs in generated. The following subsections cover the creation of the reference model, as well as two methods of run selection/generation.

[1]TODO: github or supplementary material link

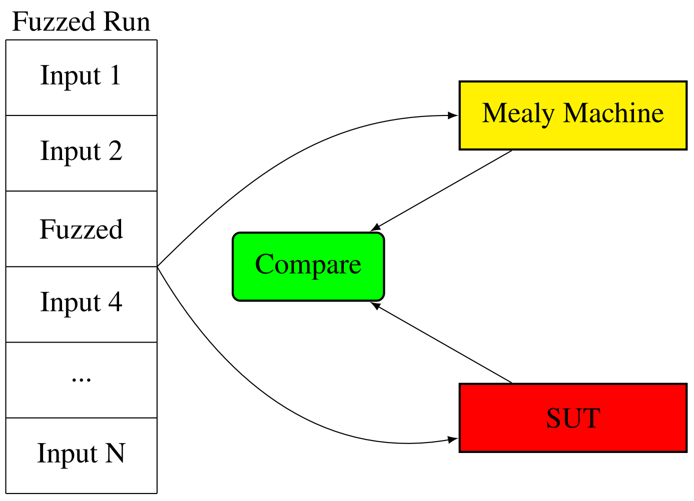


Figure 5.1: Overview of the fuzzing process.

5.1.1 Learning the Reference Model

The black-box determination of interesting inputs during fuzzing requires a model of the SUT to extract expected responses from. While a (deterministic) model of the SUL when exposed to expected inputs, had already successfully been learned, this proved to be not particularly useful for model-based fuzzing, as the majority of fuzzed inputs would result in an error message response from the SUT and therefore be treated as a new state. Instead of using the model of only expected behavior, a new model was learned, again using retransmission-filtering, but this time also with an expanded input alphabet. In addition to the previous input alphabet letters, an erroneous version of each input letter that maps to an IKE packet with some sort of error or malformation was added. An example of such a malformed packet could be an incorrect length field, a wrong hash value or simply an unsupported SA option. Adding these new inputs to our input alphabet doubles the size of our input alphabet and results in the updated input alphabet consisting of the abstract inputs *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*. This new model represents the behavior of the SUL when exposed to expected inputs, as well as malformed packets or other unexpected inputs that could arise during fuzzing. Since our mapper class was designed in such a way as to allow for easy manipulation of packets, this was an easy change to implement. Some additional server responses had to be parsed correctly, but all in all, not much had to be changed in our mapper class to allow for the sending of fuzzed packets.

5.1.2 Detecting New States

As inputs should be executable simultaneously on the SUT as well as on the reference model and have their respective responses compared in order to discover new states, i.e. new behavior. In order to be able to compare the outputs automatically, a Python representation of the learned reference model is required. Conveniently, AALpy is able to parse generated dot files into a corresponding Mealy machine object. This Mealy machine has a current state, as well as a `step` method, taking an abstract input as a parameter and returning the corresponding expected response based on the learned response when applying that input to the current state. Using this Mealy machine, a fuzzed input sent to the SUT can then easily be checked to see if it results in the same next state and response as it did on our reference model. By passing the same input to both the actual SUT and the Mealy machine representation of the reference model, this comparison becomes very simple and can be easily automated. If the two responses do not match, a new state and therefore hitherto unexplored behavior of the SUT will have been discovered.

Using the Mealy machine representation of our new reference model, we were able to filter out far more uninteresting information and focus on more unusual behavior than previously.

5.1.3 Test Data Generation

As our mapper class was designed in such a way as 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, forgoing the protocol definitions. To get relevant fuzz values for each field, every fuzzable field was mapped to a matching boofuzz primitive data type and then used that to generate our data. This ensures that each field is fuzzed with relevant data allowed by the protocol, e.g. string inputs are not suddenly used for length fields. While testing completely incompatible types of inputs would also be an interesting experiment in and of itself, we decided to focus on allowed but unexpected inputs, as these seemed the most likely to lead to undocumented/unexpected behavior. Additionally, supporting completely invalid types would have required a complete redesign of our mapper class. To summarize, we now have a rough fuzzing framework, consisting of a reference model to detect new states reached during fuzzing, as well as fuzz data generation, on a field type-by-field type basis. However, we are still missing a way of choosing which, or generating runs to insert the fuzzed test data into.

5.1.4 Run Generation

The final piece missing for our fuzzer is the run-generation phase in which we look to generate a set of runs consisting of input alphabet words, where one (or more) of the inputs will be chosen to be fuzzed. As the chosen inputs to be tested are embedded in a full run, this ensures, that the fuzzed input is always executed in the same state, provided the SUT is reset between runs. The difficulty lies in selecting or generating the runs to use for fuzzing in such a way, that the amount of new states discoverable through fuzzing is maximized, i.e. as much interesting new behavior is tested as possible. Effectively, this means that the goal is to achieve good coverage of the interesting behavior of the SUT. As the SUT is a reactive system, this task boils down to finding specific chains of inputs that lead to said new states. Naturally, one could achieve this by exhaustively fuzzing every possible field on every possible combination and concatenation of inputs, guaranteeing full coverage of the SUT. Unfortunately, due to the IPsec-IKEv1 protocol being rather complicated with tons of configurable fields, fuzzing even half the possible fields for a basic IKEv1 exchange would be an immense task that goes far beyond the scope and resources of this masters thesis. In fact, even the most simple packet of the exchange, the final *ACK* message, has at least nine distinct fields, while others have upwards of 50. Instead of trying all possible possibilities, several techniques to limit the amount of fuzzing to be done were implemented in a way that aims to still maximize the chances of discovering new states and potential bugs, while dramatically reducing the required runtime.

Firstly, instead of fuzzing every possible field of the protocol, the selection was narrowed down to 5-10 key fields from each packet. The fields were chosen based on our estimation of their impact and chance of leading to errors. Additionally, fields unique to specific messages were prioritized. We focused on length fields, SA proposals and hashes/keys, but also added general fields, such as the responder/initiator cookies. All the chosen fields were added as parameters to their respective mapper class methods and default to their usual values. Packets can then have all or just some of their fields fuzzed, as needed. When fuzzed, the fields are fed with input from a matching boofuzz data generator as explained in Subsection 5.1.3.

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

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

Additionally, two separate approaches for the generation/selection of relevant runs to use during fuzzing were implemented. The first method focuses on reusing existing runs from the model learning stage to achieve a more widespread coverage, while the second method generates a new run from scratch, that is designed to lead to as many new states as possible when fuzzed. The two methods are presented in more detail below.

5.1.4.1 Filtering

Our initial idea when thinking about how to generate our runs for fuzzing, was to go on random walks through the Mealy machine and mirror the messages sent to the SUT as well. 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 runs had already been generated during model learning, guaranteeing state-coverage (at least for the learned automaton), these runs were repurposed for fuzzing. The problem with this approach however, was that the resulting set of runs was rather large. So, in an effort to reduce the fuzzing space, an additional filtering phase was added, to reduce the number of runs to test before the actual in-depth fuzzing. In this initial filtering phase, each run is processed one by one, randomly designating one or more of its inputs as the fuzzing target. Next, each fuzzable field of that input is tested in the context of the run, with a greatly reduced set of fuzz values (3-5 values per field) and the results compared to the expected outcomes using our Mealy machine, as explained previously in Subsection 5.1.2. If new behavior is found, the run and choice of fuzzed input(s) passes the filtering and are saved. Runs in which no new behavior is discovered are discarded. This allows us to focus our resources on testing those configurations in which it is more likely to discover new behavior and therefore also bugs. Note, that by filtering out runs and by choosing the inputs to fuzz randomly and not exhaustively, the potential of missing some new states is increased. However, since many of the runs from learning are very similar, differing only in a specific suffix, this decreases the chance of discarding interesting runs, as chances are good that a run that contains at least a relevant subset of the discarded run will pass.

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

Finally, following the automatic and manual filtering, 175 runs of various lengths remained, compared to roughly ten times the number before filtering. The filtered list of runs can be found in Appendix and contains all the discovered runs that exhibited new behavior. While not exhaustive, this approach of run selection through filtering learning runs gives good coverage over a variety of different input scenarios. However, fuzzing 175 distinct runs takes a considerable amount of time (upwards of five days), due to the long length of some runs, as well as the high amount of fuzz value inputs. While the selection of runs could certainly be optimized further, for example by adding additional filtering steps, the runtime is likely to remain rather high and would not be suitable for quick scans during security testing. It is however suitable for more in-depth testing. Instead of our approach using existing runs and filtering, we also developed a promising method of run generation using scored mutation that is described in Subsection 5.1.4.2.

Diagram for fuzzing process

5.1.4.2 Mutation-based Testcase Generation

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

In its simplest form, mutation-based data generation refers to the generation of data, in our case runs, by repeatedly mutating a base case. In other words, runs are generated by repeatedly applying small changes to a base run (possibly empty). 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 run. The second mutation operation swaps an existing input in the run with its opposite version (so an erroneous version becomes a valid one, and vice versa). Our mutation base consists of either a random input, or a specified run 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 attempt to use fully random runs for fuzzing, we know that phase two is explored far less than phase one. Therefore, to aid the mutation process, the option of starting with phase one already completed was added.

The mutation-based run generation at this point, while functional, was still very basic and the runs generated were often of low interest. As the goal is to generate a run with the highest possible chance of reaching the largest amount of interesting states, scoring was added to the mutation process.Scoring means, that the generated mutations were given a score and only mutations with a higher or equal score compared to the previous high score were accepted. Intuitively, the score given should serve as a representation of how easy it is to find new states while fuzzing a given run. To calculate the score of a mutated run, each fuzzable field of every input in the run is tested using the minimal list of fuzzing values used previously in the initial filtering phase of the filtering-based approach. The amount of new states discovered while minimally fuzzing each input of the run is recorded and referred to as s_{new}. The score of a run s_{run} is calculated as the sum of the number of new states found for each input s_{new} divided by the number of inputs in the run, over all inputs of the run.

$$s_{\text{run}} = \sum_{0}^{n-1} \frac{s_{\text{new}}}{n} \tag{5.1}$$

Scoring does take some time, as at least five fields are tested per input, and every input in the run is minimally fuzzed compared to the random choice of input to fuzz employed by the previous method. However, it is still much faster than the initial filtering phase, provided the length of the mutated run does not approach to the number of runs needed to learn the model divided by five. Note, that while usually significantly faster than the filtering phase, scoring occurs after every mutation, so for long mutations (>30 mutations) the total runtime can exceed that of the previous filtering step. The key difference is, that after the mutations are completed, the result will be a single run that is more interesting than any previously generated run, as opposed to the 175 runs after filtering. To help further increase the likelihood of generating interesting mutations, the 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 to increase the likelihood of the last letter being flipped over random letters. These changes try to decrease the chances of wasting time changing existing interesting configurations instead of adding to them, but still leaves the possibility given enough mutations. The rational being, that by adding more inputs, potentially more unique states can be visited.

An excerpt of a mutation over 60 mutations can be seen in Listing 5.1. The base case, seen in Line 2, shows the initial starting point for future mutations. Next, in Line 5, `authenticate` is mutated into `authenticate_err`. As the total score does not decrease, the mutation is accepted, as mutations are encouraged. Mutation B in Line 8 mutates `key_ex_main` into `key_ex_main_err`, however this mutation is rejected, as it leads to a significant score decrease. Finally, Mutation C shows the insertion of an additional `key_ex_main` input in Line 11. This mutation is accepted, as it leads to a score increase.

Comparing the two methods, the filtering method takes far longer, but does explore a much higher number of states. In contrast, the mutation-based approach is much faster, and focuses on a small sample of states, but generates that sample to be as interesting as possible in regards to the amount of new states that can be discovered through it. Both methods found identical findings for the tested Strongswan IPsec server.

```
Base case, Score: 7.3
['sa_main', 'key_ex_main', 'authenticate']

Mutation A, Score: 7.3
['sa_main', 'key_ex_main', 'authenticate_err']

Mutation B, Score: 5.3 - Discarded
(['sa_main', 'key_ex_main_err', 'authenticate_err'])

Mutation: C, Score: 7.75
['sa_main', 'key_ex_main', 'key_ex_main', 'authenticate_err']
```

Listing 5.1: Mutations

Chapter 6

Evaluation

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

6.1 Learning Results

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

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

6.1.1 Learned Models

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

The first common model, presented in Figure 6.1 took approximately 52 minutes (3092 seconds) to learn with the *KV* algorithm, spread over seven learning rounds, and consists of 10 states. Of those 52 minutes, roughly half were used for state exploration / membership queries and the other half for conformance checking, with conformance checking taking slightly longer (1501 vs 1591 seconds). 171 membership queries were performed by the learning algorithm in 2047 steps, whereas 100 equivalence queries were performed for conformance checking in 1826 steps. In contrast, when learned with the L^* algorithm, model learning took almost 85 minutes (5094 seconds) over five learning rounds. Here, the split between state exploration and conformance checking was more distinct, with state exploration taking up approximately 68% of the total runtime and conformance checking only requiring the remaining 32% (3489 vs 1605 seconds). Notably, the time needed for conformance checking remained largely the same between the two algorithms, however the difference in state exploration / membership queries is quite large. We discuss this behavior in more detail including a statistical comparison of the two algorithms in Subsection 6.1.2.

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

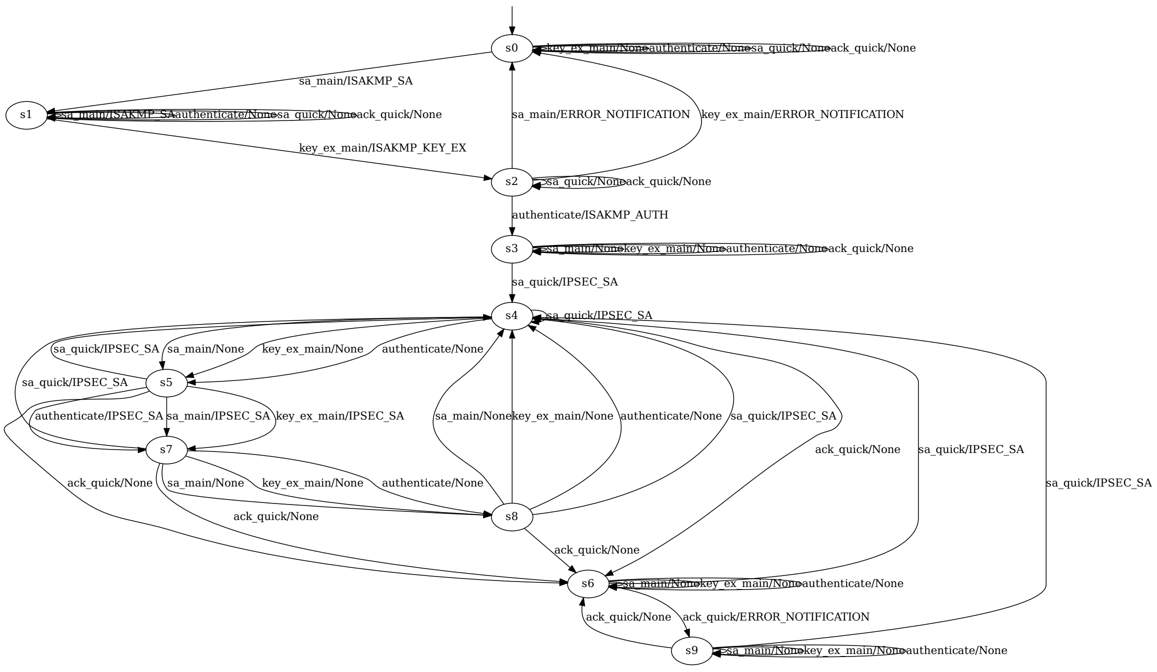


Figure 6.1: First commonly learned model with retransmissions.

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

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

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

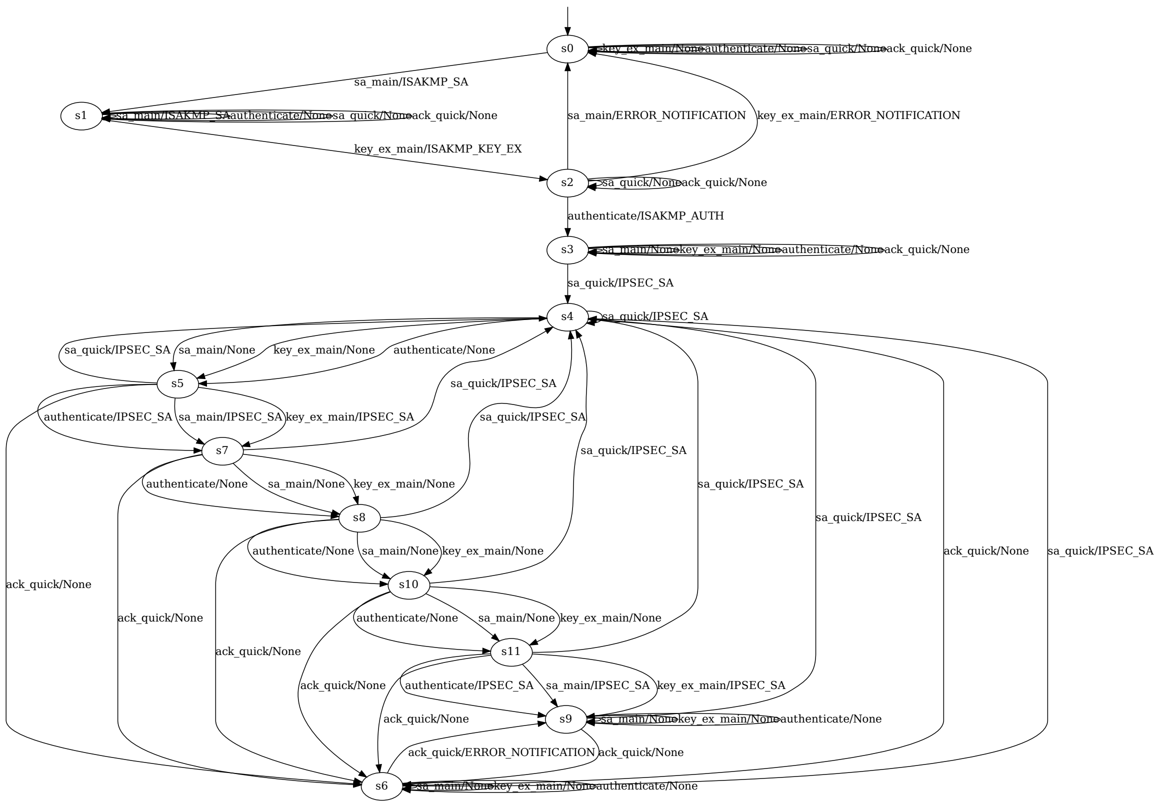


Figure 6.2: Second commonly learned model with retransmissions.

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

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

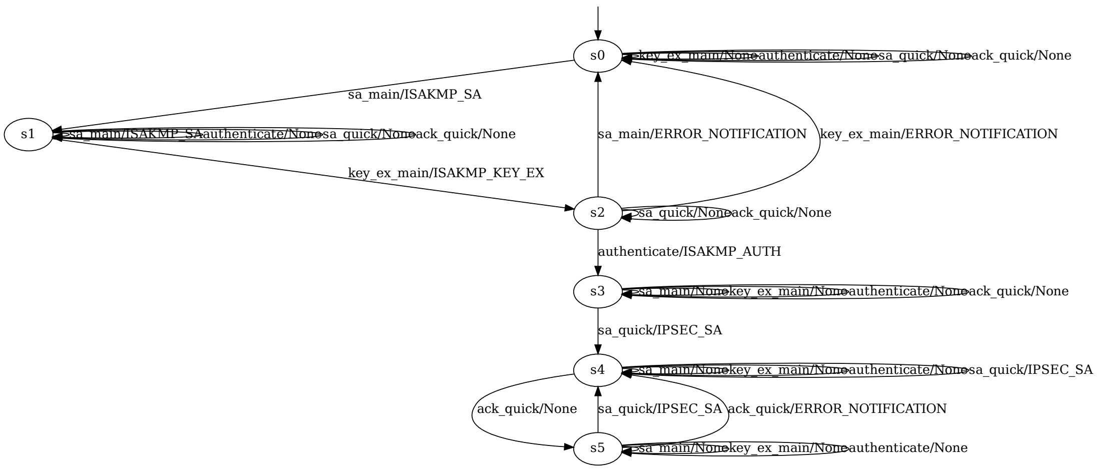


Figure 6.3: Clean model learned using retransmission filtering

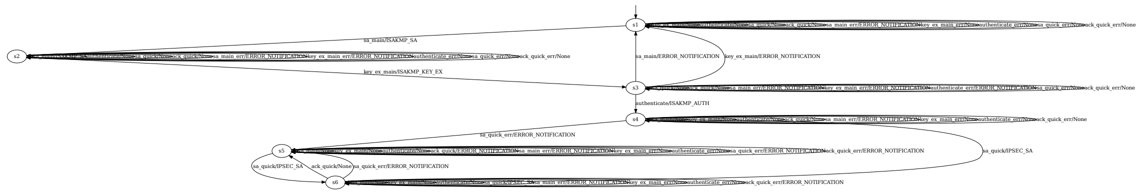


Figure 6.4: Model with malformed messages

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

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

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

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

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

Table 6.1: Comparison L^* and KV

6.1.2 Comparing KV and L^*

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

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

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

Redo with
20 runs. A
standard d
such statis

6.1.3 Library Error

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

6.2 Fuzzing Results

Fuzzing results

Compare mutation based fuzzing and filtering results / runtimes

Chapter 7

Conclusion

test

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

[13] AVM Computersysteme Vertriebs GmbH. *Connecting the FRITZ!Box with a company's VPN*. https://en.avm.de/service/vpn/tips-tricks/connecting-the-fritzbox-with-a-companys-vpn/. 2022.

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

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

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

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

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

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

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

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

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

[23] Oliver Niese. *An integrated approach to testing complex systems*. 2003.

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

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

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

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

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

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

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

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

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

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

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