Active automata learning of an IPsec IKEv1 Server using AALpy

Benjamin Wunderling

Institute of Software Technology, Graz Unversity of Technology, Graz, Austria
benjamin.wunderling@gmail.com
Supervised by Bernhard K. Aichernig

Abstract. Virtual Private Network (VPN) protocols are widely used to create a secure mode of communication between several parties over an insecure channel. A common use case for VPNs is to secure access to company networks. Therefore, errors in VPN software are often severe. IPsec is a VPN protocol that uses the Internet Key Exchange protocol (IKE). IKE has two versions, namely IKEv1 and the newer IKEv2. While IPsec-IKEv2 has been investigated in the context of automata learning, no such work has been performed for IPsec-IKEv1. This paper describes the IPsec-IKEv1 protocol and shows the steps taken to learn the state machine of an IPsec server. We present a learned model and discuss its potential applications for model-based fuzzing and fingerprinting of IPsec implementations.

Keywords: IPsec · Automata learning · AALpy · VPN.

1 Introduction

Virtual Private Networks (VPNs) are used to allow secure communication over an insecure channel. 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 [1]. This makes finding vulnerabilities in VPN software more critical than ever. IPsec is a VPN protocol and most commonly uses the Internet Key Exchange protocol (IKE) to share authenticated keying material between involved parties. Therefore, IKE and IPsec are sometimes used interchangeably. We will stick to the official nomenclature of using IPsec for the full protocol and IKE for the key exchange only. IKE has two versions, IKEv1 and IKEv2, with IKEv2 being the newer and recommended version [4]. However, despite IKEv2 supposedly replacing its predecessor, IKEv1 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 [9]. State models of protocol implementations are useful tools in testing. They can, e.g., be used to detect software implementations [14], or generate test cases automatically [15]. One method of generating such models 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 an automaton

describing the behavior of the SUL through its responses. This automaton is then compared with the SUL, adapting it if they show different behaviors. Guo et al. [10] investigated IPsec-IKEv2 using automata learning [10], however so far, no studies have focused on IKEv1 in the context of automata learning. We learn the state model of a sample IPsec-IKEv1 server using the active automata learning framework AALpy [12]. We configure AALpy to use L^* automata learning, and construct a custom Python interface between AALpy and the IPsec server. In this paper, we first introduce preliminary information on VPNs and automata learning in Section 2. Section 3 discusses other related work. In Section 4, we briefly introduce AALpy and our learning setup. Subsequently, we will present our custom interface between AALpy and the IPsec server, discussing design choices and implementation difficulties. Finally, we present the learned model and discuss its potential applications and future work in Sections 5 and 6.

2 Preliminaries

2.1 Mealy Machines

Mealy machines are finite state machines where each output transition is defined by the current state and an input. More formally, a Mealy machine is defined as a 6-tuple $M = \{S, S_0, \Sigma, \Lambda, T, G\}$, where S is a finite set of states, $S_0 \in S$ is the initial state, Σ is a finite set called input alphabet, Λ is a finite set called output alphabet, T is the transition function $G : S \times \Sigma \to \Lambda$ which maps a state and an element of the input alphabet to another state in S and G is the output function $T : S \times \Sigma \to S$ which maps a state-input alphabet pair to an element of the output alphabet Λ.

2.2 Automata learning

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

AAL began in 1987 with a paper by Dana Angluin, titled "Learning regular sets from queries and counterexamples" [3]. In this seminal paper, Angluin introduced the L^* algorithm which is still used for learning deterministic automata. L^* works using a Minimally Adequate Teacher (MAT) model in which a learner queries a teacher about a SUL. The teacher must be able to answer equivalence and membership queries posed by the learner regarding the SUL. Equivalence queries are used to check if a learned model accurately matches the SUL. Membership queries are used to check whether a word is included/accepted. The learner, using the responses to its queries, then updates its model of the

SUL. Learned models are commonly represented as Mealy machines, finite-state machines with outputs depending on the current state as well as inputs. The original L* paper learns deterministic finite automata, however the algorithm can be extended to apply to other modeling formalisms including Mealy machines [16].

2.3 IPsec

IPsec or IP Security, is a VPN layer 3 protocol used to securely communicate over an insecure channel. It is based on three sub-protocols, IKE, the Authentication Header (AH) 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. 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 [6] 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 [11]. 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 [9]. We use IPsec-IKEv1 with ESP in this paper and focus on the IKE protocol as it is the most interesting from an AAL and security standpoint.

The IKEv1 protocol works in two main phases, both relying on the Internet Security Association and Key Management Protocol (ISAKMP). A typical exchange between two parties, an initiator and a responder, can be seen in Figure 1. In phase one (Main Mode), the initiator sends a Security Association (SA) to the responder. A SA essentially details important security attributes 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. In comparison, the responder must answer with only one proposal, provided it supports one of the suggestions. Subsequently, the two parties perform a Diffie-Hellman key exchange and exchange nonces to generate a shared secret key. This secret key is used as a seed key for all further session keys. Following a successful key exchange, all further messages are encrypted. Finally, both parties exchange hashed authentication material (usually pre-shared keys or certificates) and verify the received hash. If successful, a secure channel is created and used for phase two communication. The shorter phase two (Quick Mode) begins with another SA exchange. This time, however, the SA describes the security parameters of the ensuing ESP/AH communication. This is followed by a single acknowledge message from the initia-

tor to confirm the agreed upon proposal. After the acknowledgment, all further communication is done via ESP/AH packets.

3 Related Work

Model learning of other network protocols like SSH [8], or TCP [7] has been performed in the past, with learned models being used for model checking the learned protocols. Both Novickis et al. [13] and Daniel et al. [5] learned models of the related OpenVPN protocol [13] and used the learned models for fuzzing. In a work by Pferscher and Aichernig [14], learned models were used to fingerprint Bluetooth Low Energy devices (BLE). Guo et al. [10] used automata learning to learn and test the IPsec-IKEv2 protocol. In contrast with our work, they used the LearnLib library for automata learning and utilized the learned model for model checking.

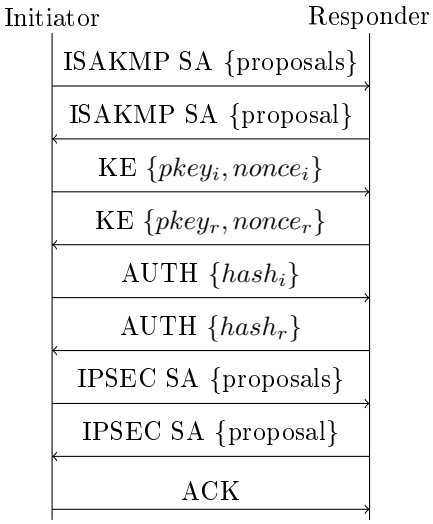

Fig. 1. IKEv1 between two parties

4 Learning IPsec

4.1 Environment Setup

We developed and tested our interface using two VirtualBox 6.1 Virtual Machines (VMs) running standard Ubuntu 22.04 LTS distributions. All communication took place in an isolated virtual network to eliminate possible external influences. During learning, all power saving options and similar potential causes of disruptions were disabled. The IPSec server was restarted before each learning attempt. We designated one VM as the initiator and one as the responder to create a typical client-server setup. The open source IPsec implementation Strongswan[1] was installed on the responder VM and set to listen for incoming connections from the initiator VM. The Strongswan server was configured to use pre-shared keys for authentication and default recommended security settings. Additionally, it was configured to allow unencrypted notification messages, which we used in our interface to reset the connection. To learn the model of the Strongswan server, run the *IPSEC_IKEv1_SUL* Python script which uses the learning library AALpy with our custom Python IPsec client implementation to communicate with and learn the model of the server.

[1] https://www.strongswan.org/

4.2 Learning Setup

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

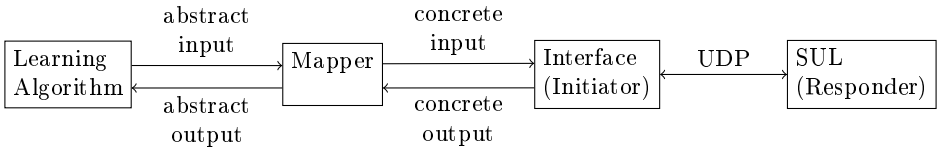

Fig. 2. Automata Learning Setup

Figure 2 gives an overview of the learning process, adapted from Tappler et al. [17]. To begin, the learning algorithm sends abstract inputs chosen from the input alphabet to the mapper class, which converts it to concrete inputs. The abstract inputs are then sent to the SUL, by means of a communication interface. In our case, the mapper class comprises the major portion of our work and converts the abstract words into actual IPsec packets that can be sent to the SUL Strongswan server via UDP packets. This separation between abstract and concrete in/outputs allows for easy future modifications to the message implementations, including fuzzing support, as well as increasing the readability of our code.

To begin learning an automaton with AALpy, we must first choose a suitable input alphabet encompassing the language known by the server, as well as the learning algorithm to be used. Our chosen input alphabet consists of the initiator-to-responder messages shown in Figure 1. We use the L^* algorithm for learning with an equivalence oracle that provides state-coverage by means of random walks. The chosen equivalence oracle is used by the 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. Additionally we defined a *step* and *reset* method. We use *step* to execute one input action from the current query and *reset* to revert the SUL to an initial clean state. We also enabled several optional AALpy features including caching and non-determinism checks to improve the learning process.

Our mapper class implements methods for each communication step in a typical IPsec-IKEv1 exchange, as described in Section 2. We use the Python library Scapy[2] to construct ISAKMP packets as required by the IKEv1 protocol. This approach allows us to change fields and values of generated packets at will,

[2] https://scapy.readthedocs.io/en/latest

opening the possibility of fuzzing for our future work. Parsing was made more difficult by the fact that Scapy does not support all the packets required by IPsec-IKEv1. To solve this problem, we implemented the missing packets in the Scapy ISAKMP class and used this modified version.

The IPsec packets generated by the mapper class are then passed on to our communication class, which acts as an interface for the SUL and handles all incoming and outgoing UDP packets. 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 responses received from the communication interface and returns an abstract output code representing the received data to the learning algorithm.

As messages will be sent in random order during learning, we require a robust framework that correctly handles en/decryption of messages. For key management, we simply store the current base-keys but keep track of Initialization 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. 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. This is required, to continuously be able to parse server responses and extract meaningful information.

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

Since occasional server errors occur, we catch non-determinism errors as they occur and repeat the offending query several times. If upon the first rerun the non-determinism does not occur again, we accept the existing value as the correct one and continue. If however, it persists for a set amount of repetitions with the same constant server response, we assume that the original saved response was incorrect and update it to the new one. With this non-determinism correcting added, the automata learning works with no errors and the learned automata are consistent with one another. Additionally, making use of timed waits after each communication to give the server time to respond also helps decrease the number of non-determinism errors that have to be caught.

5 Evaluation

The model, shown in Figure 3, was learned from a Linux Strongswan U5.9.5 server, with a typical P2P PSK configuration. The model took an average of 300 minutes to learn over 5 learning rounds and consists of 13 states. Of the 300 minutes, roughly 75% were used for state exploration or output queries and 25% for conformance checking. 786 output queries were performed in roughly 6600 steps. On average, three to four non-determinism errors were caught and fixed per learned model, however it should be noted that the learned automata for runs that required non-determinism fixing and those that did not are identical. We therefore assume it to be caused by potential race conditions occurring rarely on the server, causing inconsistent behavior.

A strong separation can be observed between states zero through three and four through twelve. This separation matches the separation of IKEv1 into two phases. Once phase one has completed, phase one messages should be ignored by the IPsec server and the learned automata reflects this behavior. Another interesting behavior is exhibited in states S6 and S10, where an input of `SA_main`, `authenticate` or `key_ex_main`, all phase one inputs, which are usually ignored in phase two, returns a valid and unexpected `IPSEC_SA` response. This behavior seems to be a specific chain of inputs that causes a retransmit of a previous response, which then does not match the input, which does in fact get ignored by the server. Some of the None responses are due to design choices in the mapper class implementation, as some impossible cases were simplified to save time. For example, if no key-exchange has yet occurred, the client will not be able to send sensible encrypted data, so the mapper class simply returns None. While observing the behavior of the server when exposed to completely non-sensible input is interesting from a fuzzing standpoint, as all specifications state that the encryption requires a prior keying procedure, we decided to ignore those few cases. However, for future work in the field of fuzzing, these edge-cases should be considered as well.

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.

To investigate interesting behavior seen on the learned model, we developed a small testing framework that allows one to execute individual runs very simply. This allowed us to quickly perform sanity-checks for our model and verify that individual runs match the behavior shown by the model.

6 Conclusion and Future Work

The learned model shows that our AALpy interface works as intended and can be used to learn the behavior of an IPsec server. We discussed the implementation

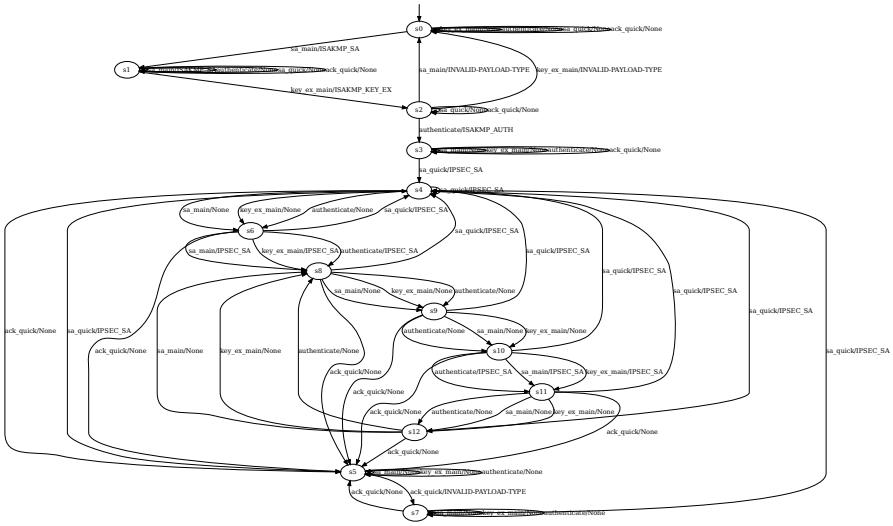

Fig. 3. Strongswan Server Automata

problems that arose and described our solutions to those problems. Particularly noteworthy are our contributions to the Scapy ISAKMP module and our solution for non-determinism, as well as our *reset* method. Looking towards potential use cases for our learned automata, our focus on keeping generated packet fields easily modifiable makes automata-based fuzzing a straightforward extension of our work. Future work could also go into improving the performance and error-handling of our IPsec interface. Additionally, comparing the learned model with those of other servers and other software versions could prove to be an interesting application area.

References

1. Abhijith, M., Senthilvadivu, K.: Impact of VPN technology on it industry during covid-19 pandemic. In: IJEAST (2020)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and computation **75**(2), 87–106 (1987)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/https://doi.org/10.1016/0890-5401(87)90052-6, https://www.sciencedirect.com/science/article/pii/0890540187900526
4. Barker, E., Dang, Q., Frankel, S., Scarfone, K., Wouters, P.: Guide to IPsec VPNs. https://doi.org/https://doi.org/10.6028/NIST.SP.800-77r1
5. Daniel, L.A., Poll, E., de Ruiter, J.: Inferring OpenVPN state machines using protocol state fuzzing. In: 2018 IEEE European Symposium On Security And Privacy Workshops (Euros&PW). pp. 11–19. IEEE (2018)
6. Ferguson, N., Schneier, B.: A cryptographic evaluation of IPsec (1999)
7. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: International Conference on Computer Aided Verification. pp. 454–471. Springer (2016)
8. Fiterău-Broştean, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F., Verleg, P.: Model learning and model checking of SSH implementations. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 142–151 (2017)
9. GmbH, A.C.V.: Connecting the FRITZ!Box with a company's VPN. https://en.avm.de/service/vpn/tips-tricks/connecting-the-fritzbox-with-a-companys-vpn/, accessed: 2022-09-09
10. Guo, J., Gu, C., Chen, X., Wei, F.: Model learning and model checking of IPsec implementations for Internet of Things. IEEE Access **7**, 171322–171332 (2019)
11. Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., Kivinen, T.: Internet key exchange protocol version 2 (IKEv2). Tech. rep. (2014)
12. Muškardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: AALpy: an active automata learning library. Innovations in Systems and Software Engineering pp. 1–10 (2022)
13. Novickis, T., Poll, E., Altan, K.: Protocol state fuzzing of an OpenVPN. Ph.D. thesis, PhD thesis. MS thesis, Fac. Sci. Master Kerckhoffs Comput. Secur., Radboud Univ (2016)
14. Pferscher, A., Aichernig, B.K.: Fingerprinting Bluetooth Low Energy devices via active automata learning. In: International Symposium on Formal Methods. pp. 524–542. Springer (2021)
15. Pferscher, A., Aichernig, B.K.: Stateful black-box fuzzing of Bluetooth devices using automata learning. In: NASA Formal Methods Symposium. pp. 373–392. Springer (2022)
16. Shahbaz, M., Groz, R.: Inferring mealy machines. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5850, pp. 207–222. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_14, https://doi.org/10.1007/978-3-642-05089-3_14
17. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing iot communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 276–287 (2017). https://doi.org/10.1109/ICST.2017.32