Inferring OpenVPN State Machines Using Protocol State Fuzzing

Lesly-Ann Daniel[1]

University of Rennes 1 - ENS Rennes, France

Abstract. OpenVPN is a widely used VPN solution, used to extend a private network over a public network while offering an extra layer of security. The reliability of such a security protocol is of the utmost importance but can be easily compromised by a vulnerability in the implementation. A technique called *protocol state fuzzing*, based on black-box fuzzing, can be used to infer a state machine from a protocol implementation. The inferred state machines provide a good insight into the implementation and can be used to detect any spurious behavior. Although OpenVPN is a widely used TLS-based VPN solution, there is no specification of the protocol, which makes it a particularly interesting target to analyze. In this paper, we apply protocol state fuzzing to the standard OpenVPN and the OpenVPN-NL implementations in order to infer state machines of the servers that focus on particular phases of the protocol. Finally we analyze those state machines, show that they can reveal a lot of information about the implementation which are missing from the documentation, and discuss the possibility to include those state machine in a formal specification.

Keywords: virtual private networks, OpenVPN, regular inference, LearnLib, protocol state fuzzing

Internship at the Institute for Computing and Information Science, Radboud University, Nijmegen The Netherlands.
From the 15th May, 2017 to the 13th August, 2017.
Under the supervision of Erik Poll and Joeri de Ruiter.

1 Introduction

Virtual Private Network solutions are widely used to establish secure data transmissions over insecure channels. They use a tunneling mechanism to provide an additional layer of confidentiality, authentication and integrity that is not supported by the underlying protocol. However, the security of a protocol can easily be compromised by a vulnerability in the implementation.

Formal methods can be used to automatically test protocol implementations for vulnerabilities. Among these methods, protocol state fuzzing is a technique that permits to infer a state machine from the implementation of a protocol [3, 10]. This approach uses black-box fuzzing on the order of well-formed messages and infers a model of the implementation from the output, as a Mealy Machine. A proper state machine should allow all the transitions defined by the grammar of the protocol and react appropriately to unexpected messages - for instance by ignoring the message or dropping the connection.

The inferred state machine provides a useful insight into the choices - and errors - made in the implementation. A manual analysis can be performed to detect any logical flaws and to check the compliance of the implementation with its specification. It can also reveal superfluous states and transitions which should be removed as a precaution. Furthermore, it gives a good overview of the sequence of messages (which is often not well specified), and can be used to automatically define a formal specification of the protocol [23].

This paper focuses on the TLS-based OpenVPN protocol [12]. Even though the OpenVPN protocol is a widely used Virtual Private Networks (VPN) solution, it has not been subject to a lot of research and there is no formal specification of the protocol. There is also no documentation about the sequence of messages leading to a successful connection nor on the conduct to adopt when receiving an unexpected message - even though this is essential for a security protocol.

We use regular inference, provided by the LearnLib library, to infer state machines of two different OpenVPN servers: the standard OpenVPN implementation which relies on the OpenSSL library, and the OpenVPN-NL implementation which relies on the PolarSSL library. For each of them we infer several states machines that focus on particular phases of the protocol. We manually analyze those state machines and show that they can give us a lot of information about the implementation that are not specified within the documentation. Finally, we discuss how state machines can be used to define a formal specification of the protocol.

1.1 Related Work

This section provides a short overview of the related work performed in the field of regular inference of security protocols. The regular inference itself is detailed in Sec. 3.

The automatic construction of system models from observation of their external behavior can be performed using regular inference (also known as automata learning or state machine learning). This technique has first been applied to security protocol by Shu and Lee [25], in combination with a validation process, in order to build a model of a physical implementation and check it against message confidentiality property on the fly.

Regular inference has been extended using predicate abstraction [3] to consider the influence of data on the control flow. Then, [19] proposed a systematic method to implement a test harness for LearnLib, including a mapper and a data monitoring part. Finally [23] discusses the possibility to use regular inference to infer protocol formal specification, and to define the session language i.e the sequence of messages, in the form of a protocol state machine.

Regular inference with LearnLib has been applied to analyze several security protocol [5, 4, 10, 9, 13]. This approach revealed new security flaws in several TLS implementations analyzed [10].

1.2 Overview

We first discuss the OpenVPN protocol in Sec. 2 and the regular inference in Sec. 3. Then we present our experimental setup in Sec. 4. The result of our analysis are presented in Sec. 5. Finally, we conclude in Sec. 6

2 The OpenVPN Protocol

This section describes the OpenVPN protocol and details some of the OpenVPN functionalities. For an in depth presentation of VPNs and OpenVPN, the reader can refer to [12]. The doxygen-generated documentation [1] and the security overview [2] can also be consulted for further details.

VPNs are used to extend a private network over a public network (e.g public Internet connection). This technology can be used by companies to connect geographically separated offices or to allow remote workers to access the company network. Usually, VPNs also ensure integrity, secrecy and authenticity of the data transferred, allowing remote users to securely access the private network.

2.1 OpenVPN Networking Principle

This subsection stands to introduce required knowledge about the networking concepts used in the OpenVPN protocol. We assume that the reader is familiar with the OSI model, otherwise they can refer to App. B.

At each layer N, the peers exchange Protocol Data Units (PDUs) which consist of two parts: the header and the payload. The header contains meta-data on the sender, recipient and general information for the transfer. The payload is the effective data, which can also be an other PDU (often a layer-$N+1$ PDU) that will be wrapped into the layer-N PDU. This process called encapsulation is used to pass information through the OSI layers and is the base of networking.

Encapsulation can also be used to provide a network service that the underlying network does not support or provide directly, such as encryption or authentication. This technique called *tunneling* is used by OpenVPN to provide extra security properties. Layer-2 frames (e.g Ethernet frames) or layer-3 packets (e.g IP packets) are wrapped into a VPN message and sent to the remote VPN peer. Therefore the entire effective message to transmit - including its meta-data like sender and recipient - is encapsulated within the VPN PDU and can benefit from its security properties. Figure 1 illustrates this tunneling process. Note that, as most VPNs can only handle one type of message, the OpenVPN protocol can handle both layer-2 frames and layer-3 packets.

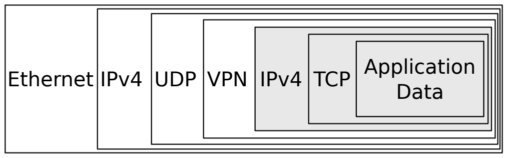

Fig. 1. Example of OpenVPN tunneling IP packets over an UDP channel. The gray message benefits from the security properties (encryption, integrity, authentication) provided by the enclosing OpenVPN protocol.

2.2 Security

Like most of the VPN solutions, OpenVPN guarantees the following properties over the data transmitted:

Confidentiality Prevents eavesdroppers from reading the private messages. This property is achieved by end-to-end encryption: if an attacker is sniffing the communication, all they can see is encrypted data.

Authentication Prevents unauthorized users from accessing the VPN. This property can be achieved by using key-pairs and certificates.

Integrity Ensures that the transmitted data was not tampered with. This property can be achieved by adding a message digest that is checked by the recipient.

The OpenVPN implementation offers a large choice of security options. The security of the OpenVPN protocol is based on the Transport Layer Security (TLS) protocol, and the implementation on the OpenSSL library, used for its TLS session negotiation, its encryption and authentication and its random number generation primitives. Confidentiality is ensured using the encryption primitives from OpenSSL, which offers a large variety of ciphers and key sizes. The default cipher is Blowfish with Cipher Block Chaining (CBC) and the default key size is 128-bits. Authentication is achieved by the use of pre-shared keys, TLS certificates or username/password. Message integrity is checked with an Hash-based Message Authentication Code (HMAC) added to the messages. The default hash function used is Secure Hash Algorithm 1 (SHA-1), albeit known as vulnerable to collision attacks [27, 26] and officially deprecated for digital signature generation since 2011 [7].

The OpenVPN implementation provides two main methods of key exchange: a pre-shared key and a TLS based mechanism. In both methods, each peer possesses four independent keys: HMAC-send, HMAC-receive, encrypt and decrypt.

Pre-shared key mode This mode uses symmetric encryption: the two peers agree on a static pre-shared key before the tunnel is started - by default both peers will use the same keys but the VPN can be configure to use the four keys independently. The authentication is straightforwardly provided by the ownership of the static key. The advantage lies in the simplicity of the method: there is no key negotiation required before the data tunneling can start. The main disadvantage is that the method does not provide a secure way way to exchange the keys. Furthermore, forward secrecy is not ensured: if the session-keys are compromised, all the past (and the future) communications are also compromised. A custom re-keying mechanism could be added to circumvent those issues, but it would add a layer of complexity which could lead to security flaws.

TLS mode This mode is based on TLS, which is one of the most important cryptographic security protocols (e.g. used in HTTPS, FTP, SMTP...) and has been subject to a lot of research [10, 22, 11, 16, 21, 14, 20, 17, 18]. A TLS session with bidirectional authentication is negotiated between the client and the server (i.e. both parties must present their own certificate). The OpenVPN implementation offers two key negotiation methods[1]. If key-method 1 is used, the keys are generated directly by the peers and exchanged in a secure way over the TLS connection. If key-method 2 is used, random material is generated, exchanged over the TLS connection, and the keys are computed from the random material using the OpenSSL Pseudorandom Function (PRF). In both cases, the keys are unidirectional (client and server keys are different) and the security of the communication relies on random source material from both parties.

The rest of the paper will focus on the TLS mode which is more complex and involves key exchange and re-keying, contrary to the pre-shared key mode which is straightforward.

2.3 OpenVPN Sessions

As we previously discussed, OpenVPN is a TLS-based VPN which relies on the OpenSSL library for its security primitives. The TLS session negotiation and the data tunneling are processed over independent channels with their own packet identifiers and keys. OpenVPN multiplexes this *data channel* and *control channel* over a single network stream (route for IP packets or bridge for Ethernet frames).

[1] https://build.openvpn.net/doxygen/html/key_generation.html

This network stream is not necessarily reliable since OpenVPN preferably uses UDP transport (e.g. Figure 1) over TCP[2].

Control channel The control channel is the channel used to set up the connection to the remote peer and to negotiate the *session-keys*, i.e. the keys that will be used to secure the data channel. A fully authenticated TLS session is initiated between the two peers and the session-key random material is exchanged in a secure way over the TLS connection (for both key-methods). Once the two peers have received the session-keys, the actual data tunneling can start. Appendix C details the regular sequence of messages leading to a successful connection.

Since TLS is designed to operate over a reliable channel, the control channel is provided with an extra reliability layer, referred to as the OpenVPN's reliability layer, which consists of a simple acknowledgement mechanism and is active in both UDP or TCP tunneling.

The OpenVPN implementation also offers the *–tls-auth* option to authenticate packets from the control channel, adding an HMAC signature to the control messages. This mechanism allows OpenVPN to quickly throw away unauthenticated packets, without wasting resources and thus protecting against Denial of Service (DoS) attacks.

Data channel The data channel is used to forward the actual data. The actual data (IP packet or Ethernet frame) to transmit is encrypted and MAC-ed with the previously negotiated session-keys, and tunneled (preferably over an UDP channel) without any extra reliability layer provided by OpenVPN. Note that a reliability layer can still be provided by the encapsulated protocol, e.g. tunneling of a TCP session will benefit from the TCP's reliability layer.

3 Protocol State Fuzzing

Protocol state fuzzing is defined in [10] as a technique that uses regular inference to infer a state machine from a protocol implementation. It uses black-box fuzzing on the order of well-formed messages, to infer a state machine which models the protocol implementation. In this paper, those state machines are represented as Mealy machines.

3.1 Mealy Machines

A Mealy machine is a finite state machine with output, in which a transition, based on the current state and input, will result in a change of state and produce an output. Mealy machines are deterministic i.e. for each input and current state, only one transition is possible, therefore they can only be used to model

[2] Because of the TCP's reliability layer collisions when tunneling TCP over TCP: http://sites.inka.de/sites/bigred/devel/tcp-tcp.html

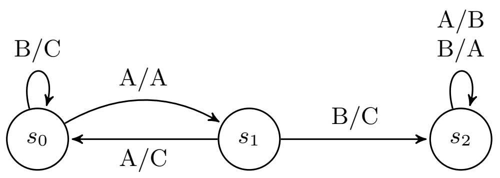
$$\begin{tikzcd}
s_0 \arrow[loop above, "B/C"] \arrow[r, bend left, "A/A"] & s_1 \arrow[l, "A/C"] \arrow[r, "B/C"] & s_2 \arrow[loop above, "A/B \; B/A"]
\end{tikzcd}$$

Fig. 2. A Mealy Machine with 3 states

deterministic systems. An example of a graphical representation of a simple Mealy machine is shown in Fig. 2. The transition from the state $s1$ to the state $s2$ labeled B/C means that if the state machine is in state $s1$ and receives an input B, then it will switch to state $s2$ and produce the output C.

In this paper, Mealy machines are used to model the behavior of the OpenVPN server. They describe how the server reacts in response to input messages: which output it will produce and how its state will be affected. The next part discusses how state machines can be automatically inferred from real-time systems.

3.2 Regular Inference

The state machine of the OpenVPN server is inferred using a technique called regular inference. It uses black-box fuzzing: well-formed packets are sent to the server and the output is used to infer a state machine. The regular inference primitives are provided by the LearnLib library [24]. The system which is analyzed, namely the OpenVPN server, is referred to as the System Under Learning (SUL) and its state machine is denoted M.

Learner The regular inference involves two actors: a *Learner* (the LearnLib library) and the SUL (the OpenVPN server). The *Learner* has no initial knowledge about M but is provided with an input alphabet upon which it will build queries and ask them to the SUL. A fundamental property that must be ensured is the independence of subsequent queries. Therefore, between each query, the SUL must be reset to its initial state. In our case, it is effectively done by killing the OpenVPN server process and starting a new one. There are two kind of queries:

Membership query What is the response to a sequence of input symbols ?
Equivalence query Is an hypothesized automaton H equivalent to M ?

The *Learner* is composed of two parts: the *learning algorithm* and the *equivalence algorithm.* The learning algorithm will keep sending membership queries to the SUL until it comes up with a strong hypothesis. Then the hypothesized state machine H is passed to the equivalence algorithm which will answer the equivalence query. If H is equivalent to M, then H is returned as the model of the SUL. Else, the equivalence algorithm returns a counterexample which comes to refine the hypothesis and is used to perform subsequent membership queries. The learning algorithm is resumed until the hypothesized automaton H is equivalent to M.

The learning algorithm used in this paper is a modified version of Angluin's L* algorithm [6] which produces Mealy machines. The equivalence algorithm is a modified version of the w-method [8]. It has been refined to cut off entire search branches based on the fact that, when the connection is closed by the server, it will remain closed. Therefore, we stop building queries over prefixes that end up with a closed connection.

The main problem that has not yet been mentioned is the communication between the *Learner* and the SUL, which is the role of the test harness.

Building a test harness The main challenge to infer a correct state machine is to prepare the application specific learning setup. This includes determining a suitable abstraction of input and output messages, and finding ways to manage concrete runtime data that influences the behavior of the target system [19]. The test harness thus consists of a mapper part and a monitoring part.

The *Learner* is provided with an abstract input alphabet upon which it will build queries and feed them to the SUL. However the SUL has his own real input alphabet, which is different from the Learner's input alphabet. The Learner will also expect the responses from the SUL to be translated into symbols over an abstract output alphabet. Thus, the test harness must contain a mapper which job is to translate the abstract input symbols from the learner into real input (i.e. OpenVPN packets), and real output into abstract output symbols. Note that the level of abstraction will affect the final learned model: a compromise must been made between the precision of the model and the learning complexity.

On the other hand, building correct system inputs is not trivial as they are based on concrete runtime data that influence the behavior of the system:

- The actual messages must be sent through the network.
- The responses from the server must be processed in order to recover important information (e.g. session-keys).
- The acknowledgement process must be handled.
- The security primitives must be implemented in the way expected by the server (e.g. valid authentication, encryption and signatures).

Handling those runtime data requires to understand the behavior of the system and to make decisions, concerning the semantic of the abstract input symbols, which will affect the final state machine.

4 Experimental Setup

In this paper we use regular inference techniques provided by the LearnLib library to infer state machines of two OpenVPN server. The server is running on a VMware virtual machine hosted on the same computer than the *Learner* and can be managed using Secure Shell (SSH). LearnLib provides the L* learning algorithm and the w-method equivalence algorithm.

4.1 Test Harness

The test harness implementation is based on the previous work of De Ruiter and Poll [10] on TLS protocol state fuzzing. The source code is available at https://framagit.org/leslyann/statelearner. In order to infer the state machine of the server, we need to build a test harness which will make the link between LearnLib and the OpenVPN server.

The test harness consists of a mapping component which provides the abstraction layer, and a monitoring part (basically a stateless OpenVPN client) which manages runtime data. The mapping part of the test harness is in charge of converting the abstract input messages from the learner into OpenVPN messages. The monitoring part is in charge of building those messages based on runtime data, such as session identifiers or session-keys. In the same way, the output messages from the server have to be be processed to extract the relevant information, and converted into output messages understandable by the learner. The messages must be well-formed to be accepted by the server and the security primitives must be implemented and used as expected by the server. Building a test harness thus requires a deep understanding of the OpenVPN implementation.

Since there is no formal specification of the OpenVPN protocol, low level information was not straightforward to get. We mainly relied on Wireshark traces, the doxygen-generated documentation [1], the security overview [2], and when inner depth analysis was needed, we used the OpenVPN source code and the server logs with maximum verbose output.

4.2 Nondeterminism Issues

The determinism of the SUL is of paramount importance since LearnLib can only learn state machines of deterministic systems. Nondeterministic behavior of the SUL at best produces a wrong model, at worse causes unexpected behavior of the learner, up to its non-termination. For instance, let us consider a nondeterministic response given to an equivalence query which raises a counterexample. Then the learning process is wrongly resumed and the response is wrongly added to the hypothesis, which will eventually raise more issues. The less frequent the nondeterministic behavior is, the harder it is to catch, which is very insidious because long learning phases can turn out to be unsuccessful because of one wrong query.

Detecting nondeterminism The first way to detect nondeterminism is to manually check the counterexamples found by the learner. Whenever nondeterministic behavior is suspected, the defective query can be run several times until a nondeterministic answer is caught. Then, the reasons of the defect can be examined further through the log file of the server and the Wireshark traces. This method can also be used to estimate the probability of such nondeterministic behavior.

Causes of nondeterminism In a network application such as we are analyzing, there are two causes of nondeterminism.

First, the UDP connection between the client and the server is not reliable, so packet loss may be a cause of nondeterminism. This kind of error is not expected in theory since the server is running on a virtual machine hosted on the same computer as the client. However, the actual behavior is slightly different as sometimes the connection drops and it fails the learning process. Appendix D.1 contains further details on the network-related causes of nondeterminism.

Second, there are multiple timeouts and delays on the server side, so the response to a query may vary depending on those timing-related events, which is seen as nondeterminism by the learner. A list of those timing-related events is available in App. D.2, along with the consequences on the learning time. The solutions we adopted to work around this timing-related nondeterminism often implies longer sleeping-times or timeouts on the client side. Choosing the appropriate timeouts and sleeping-times is a challenging issue. Under-approximating them may cause nondeterminism in the learning process and make it fail, but on the other hand, setting them too long would significantly slow down the learning process.

Managing nondeterministic behavior Nondeterministic behavior can be corrected to some extent. In the equivalence algorithm, if a nondeterministic answer to a query raises a counterexample, the output will be wrongly added to the model and the learning process will be wrongly resumed. In order to prevent this to happen, we modified the equivalence algorithm to reduce the probability of learning an nondeterministic output. Each time a counter-example is found by the equivalence algorithm, the query is processed again to check the concordance of the outputs. If both outputs are the same, we assume that the output is correct and that a counterexample has been found. Otherwise, the query is processed again to determine the correct output. If the output fails to converge at some point, an exception is raised for nondeterminism. This simple modification could be added to LearnLib as an option to detect nondeterminism, especially since the computational overhead is negligible.

4.3 Input Alphabet for Learning

In order to keep the learning complexity low, we only include messages that are intended to be sent from a client to a server and that would be accepted by the server configuration[3] and we abstract the acknowledgement mechanism. We also only include TLS messages required to establish a successful OpenVPN session. The full input alphabet is given in Sec. E.

An OpenVPN session is considered successful when the initialization sequence is complete and the data tunneling can actually start. The corresponding

[3] Thus, the ServerHardReset message and the messages for key-method 2 are not included. They were manually tried and resulted in a closed connection which is not really interesting anyway.

state is referred as Initialization Sequence Complete (ISC). To detect a successful data exchange, we use the OpenVPN tunnel to send a ping request to the server. If the exchange is successful the server will send back a ping response through the tunnel.

Depending on the input alphabet and on the monitoring part, the inferred state machine can change significantly. The learner was run with several input alphabets to infer various state machines and highlight various behaviors of the server (which are detailed in Sec. 5), while keeping the input alphabet rather small to reduce the learning complexity. Note that the complexity does not only depends on the size of the input alphabet, but also on the final number of states.

5 Results

We analyzed two different implementations of the OpenVPN protocol: OpenVPN 2.3.10 with library version OpenSSL 1.0.2g, referred to as OpenVPN, and OpenVPN-NL based on OpenVPN 2.3.9 with library version PolarSSL 1.2.19 [4], referred to as OpenVPN-NL. OpenVPN-NL is a stripped and hardened version of OpenVPN, intended for the Dutch government, which disallow the OpenVPN insecure configurations. The server has been configured to use key-method 1 and not the *tls-auth* option. Both UDP and TCP modes were analyzed and turned out to behave differently.

In order to keep the learning complexity low, we chose to split the analysis into several parts. Each part focuses on a particular phase of the protocol. The first part focuses on the OpenVPN session initialization, the second part on the TLS handshake and the last part on the re-keying process.

For each state machine, the sequence of messages leading to a successful OpenVPN tunnel, namely the *happy-flow*, is indicated with bold edges. The state '0' refers to the initial state, the state 'ISC' (Initialization Sequence Complete) is the state from which the data tunneling can actually start and the state 'X' refers to a closed connection.

5.1 The OpenVPN Session Initialization

From the documentation[5] and server logs, we can see that the OpenVPN implementation stores its OpenVPN sessions in three session slots. The first slot contains the *active session* (i.e. session which initialization sequence is complete and which can process DATA messages), the second slot contains the *untrusted session* being negotiated, and the last slot contains the old session[6]. Each OpenVPN session is initiated with a ClientHardReset message that has a unique session-identifier. However, the expected impact of the ClientHardReset on

[4] https://openvpn.fox-it.com/

[5] See https://build.openvpn.net/doxygen/html/group__control__processor.html#details for more details on the *tls_session* structures.

[6] Note that those session slots are an implementation choice and not a fundamental aspect of the OpenVPN protocol

the server is not specified in the documentation, this is why we tried to highlight it by building a state machine over three input symbols:

- CHRv1
- Tls:FullSession - which treats the entire TLS-based key exchange as one atomic step
- DataPingReq

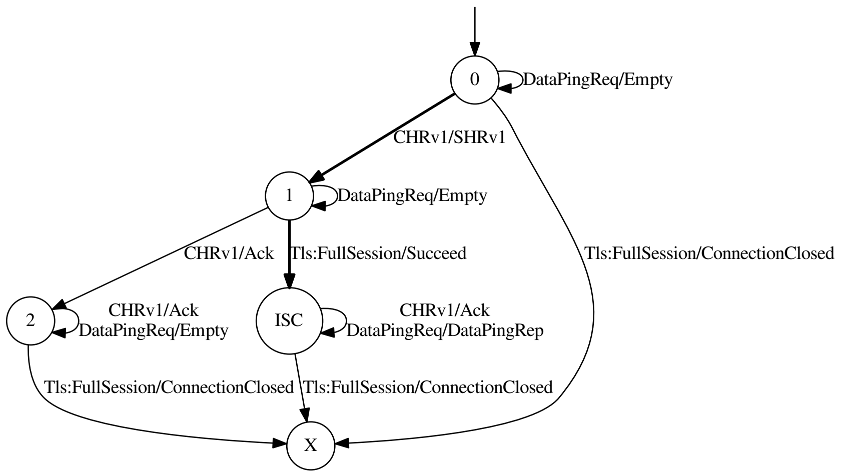

Fig. 3. State machine of an OpenVPN or OpenVPN-NL server running in TCP mode. Note that only one path can lead to a successful connection ($0 \rightarrow 1 \rightarrow$ ISC) and only the first CHRv1 (in state 0) triggers a SHRv1 and leads to a successful connection. The others CHRv1 eventually end up in a closed connection.

The state machines of the OpenVPN server and the OpenVPN-NL server are the same, which makes sense since the OpenVPN-NL implementation is based on the OpenVPN implementation. The TCP and the UDP modes differ in the way they handle the sessions, which is not specified in the documentation and (even if it does not seem insecure) is quite surprising.

Starting with a TLS message in TCP mode result in a closed connection (see $0 \rightarrow X$ in Fig. 3), while in UDP mode, the messages are just ignored by the server (see $0 \rightarrow 0$ in Fig. 4). This is because the UDP messages with an unknown session-id are ignored by the server.

In UDP mode, the session-keys can be renegotiated by sending a new ClientHardReset, which can be seen for instance in the dashed loop in Fig. 4. This is not possible in TCP mode since only the first ClientHardReset can result in a successful connection as shown in Fig. 3, the others eventually resulting in a closed connection. We found an explanation for this difference: in UDP mode the server has no way to know if the client is reset, contrary to TCP mode. Therefore, when the client reconnects and tries to initiate a new session by sending a

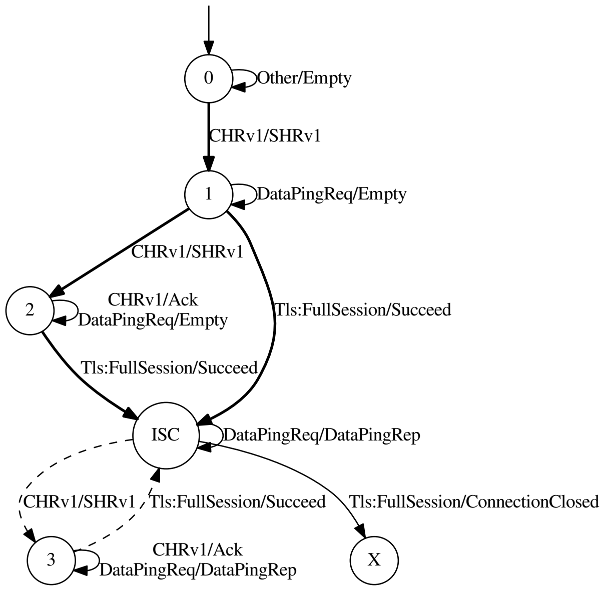

Fig. 4. State machine of an OpenVPN or OpenVPN-NL server running in UDP mode. Note that several paths can lead to a successful connection ($0 \rightarrow 1 \rightarrow$ ISC and $0 \rightarrow 1 \rightarrow 2 \rightarrow$ ISC). We can see from state 2 that several CHRv1 can lead to a successful connection. The dashed loop results in a session-keys renegotiation, initiated by a CHRv1.

new ClientHardReset, the server can process the ClientHardReset and the new session can be seamlessly renegotiated.

We also found that in UDP mode, two sessions can be under negotiation at the same time, but only if there is no active session. This can be seen in Fig. 4 from the path $0 \rightarrow 1 \rightarrow 2$ as the first two ClientHardResets trigger a response from the server, but after reaching the state ISC, only one ClientHardReset triggers a ServerHardReset (i.e. path ISC $\rightarrow$ 3). This is because when the active session slot is empty, it is used to store the first untrusted session (the others are stored in the second slot).

Figure 4 also shows that in UDP mode, a session initiated with a ClientHardReset message can succeed without triggering a ServerHardReset message. For instance following the path $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow$ ISC with the sequence of messages CHRv1/SHRv1 $\rightarrow$ CHRv1/SHRv1 $\rightarrow$ CHRv1/Empty $\rightarrow$ Tls:FullSession/Succeed, the active session-id will be the one introduced by the third CHRv1 message which triggers no response from the server. This behavior, which is quite confusing but not insecure, is based on the fact that the server only respond with a ServerHardReset when filling a new session slot. The first and second CHRv1 fill the first and second slots but the third ClientHardReset just overrides the second session in the second slot.

These differences also explain why in UDP mode two paths can lead to a successful session (i.e. $0 \to 1 \to 2 \to$ ISC and $0 \to 1 \to$ ISC in Fig. 4), while in TCP mode there is only one path ($0 \to 1 \to$ ISC in Fig. 3).

Finally from the server logs it seems like in TCP mode, the structure containing the second session is allocated when receiving the ClientHardReset but the corresponding ServerHardReset is never sent. Therefore this session is stuck in S_PRE_START state[7]. However, the subsequent TLS messages are processed by the server but the responses to the Tls:ClientHelloAll are not forwarded to the client. Finally if the TLS handshake is pursued, the Tls:CertificateVerify triggers an error for bad signature and the connection is dropped by the server.

5.2 The TLS Handshake

This part focuses on analyzing the details of the TLS sessions used to set up an OpenVPN connection. The TLS session negotiation was previously abstracted by the Tls:FullSession input symbol. In order to investigate the TLS sessions, this Tls:FullSession symbol is split into several symbols corresponding to the different TLS messages. The input alphabet consists of the following messages:

- wCHRv1
- Tls:ClientHelloAll
- Tls:ClientKeyExchange
- Tls:ClientCertificate
- Tls:ClientCertificateVerify
- Tls:ChangeCipherSpec
- Tls:Finished
- KeyNeg1
- DataPingReq

In order to make the state machine simpler, CHRv1 is replaced by an altered version: wCHRv1. First, we want to focus on one session only so the wCHRv1 keeps the previous session-id and TLS session parameters. Second, resetting the packet-id would raises some issues in regard to the acknowledgement mechanism. Indeed, the Control messages with a known packet-id are considered as replayed packets by the server, therefore the responses of the server after a wCHRv1 would depend on the number of control messages previously sent and the server could no longer be modeled as a finite Mealy machine. This is why the wCHRv1 does not reset the packet-id.

The differences between OpenVPN and OpenVPN-NL state machines are only related to the different TLS implementations and TLS cipher suites they use. As expected, the OpenSSL state machine included in the OpenVPN state machine and the PolarSSL state machine included in the OpenVPN-NL state machine are similar to those inferred in [10]. For example in the OpenVPN state

[7] See https://build.openvpn.net/doxygen/html/group__control__processor.html for more details on session states

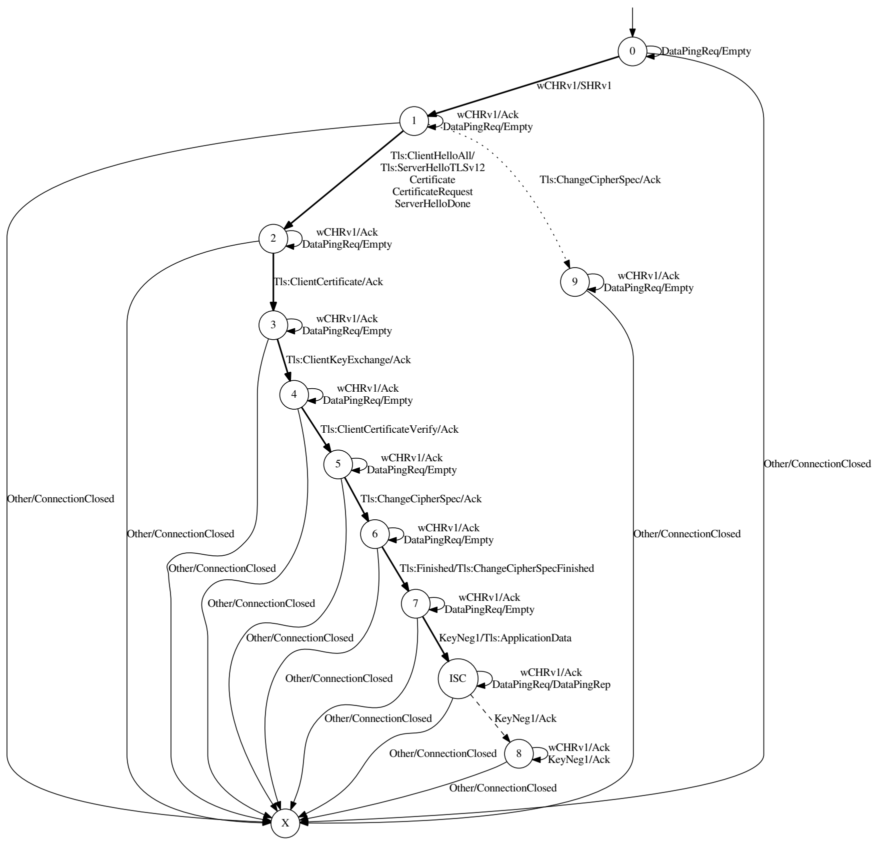

Fig. 5. State machine of an OpenVPN server running in TCP mode. The dotted edge is a characteristic of OpenSSL which does not return an error when receiving a ChangeCipherSpec before a ClientHello. The dashed edge shows that a second key negotiation message can be sent without triggering an error from the server.

machine, Fig. 5 shows that a ChangeCipherSpec sent before a ClientHelloAll does not directly result in a closed connection but goes into the dead-end state 9 where the TLS session can no longer succeed. This behavior is inherited from the OpenSSL implementation which does not return an error in this particular case. From Fig. 6 we can see that the PolarSSL library behaves differently since this particular sequence of messages results in a closed connection.

However, the OpenVPN-NL implementation is also more permissive in some other situations. When the TLS handshake is complete (in states 7, ISC and 8) and an extra TLS handshake message is sent, OpenVPN-NL will forward the Alert message (see the italic labels) instead of closing the connection.

OpenVPN uses TLS_RSA_WITH_AES_128_CBC_SHA cipher suite for the TLS session and OpenVPN-NL uses TLS_DHE_RSA_WITH_AES_256_CBC_SHA.

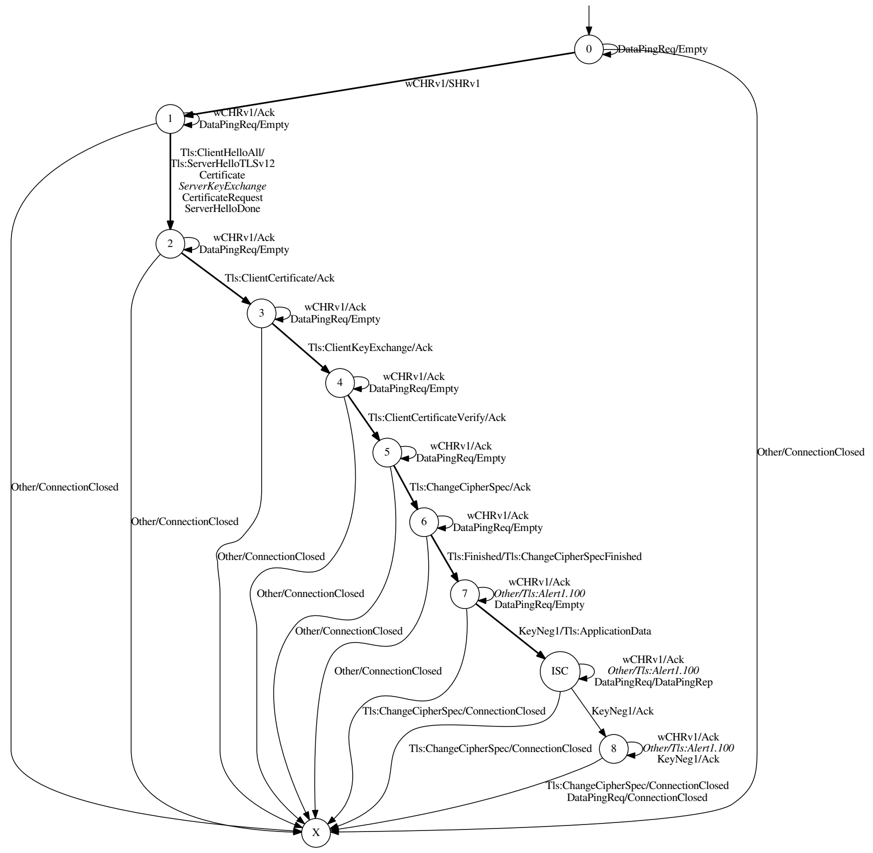

Fig. 6. State machine of an OpenVPN-NL server running in TCP mode. The italic alert labels show that the OpenVPN-NL implementation raises an alert when receiving extra TLS handshake messages after the handshake is complete, instead of closing the connection like OpenVPN. The italic ServerKeyExchange is related to the DH key exchange used in the TLS session.

This difference explains the extra ServerKeyExchange in the OpenVPN-NL state machine which is only sent in DH key exchange.

Both implementations allow the client to send several KeyNeg1 messages over the TLS session, but only the first one is processed while the others are ignored. In our test harness, we made the choice to generate and send fresh session-keys (used to encrypt and MAC Data messages) whenever we send a new KeyNeg1 message. This results in the extra state 8 which highlights the fact that when the server receives a Data message encrypted and MAC-ed with the wrong keys, it will drop the connection resulting in the DataPingRep/ConnectionClosed transition from state 8 to state X).

Finally the TCP and UDP models are similar except for some extra states that can be observed in Fig. 7, related to the acknowledgement process in UDP mode. Starting the communication with a Control message different from wCHR1 results in the dead-end states 2 and 3, because of the acknowledgement process. In state 3, the server has received a wCHR1 with a packet-id $n > 0$ and is now waiting for the messages with a packet-id lower than n. Therefore, the subsequent TLS messages are not processed. This is an other difference from TCP mode that is not specified in the documentation.

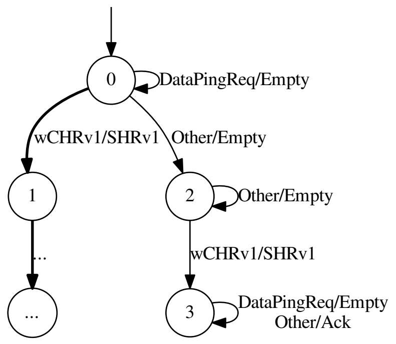

Fig. 7. Subset of the state machine of an OpenVPN or OpenVPN-NL server, focusing on the particularity of the UDP mode. The state 1 and its subsequent states are identical to the TCP versions in Figs. 5 and 6.

5.3 The Key Renegotiation Mechanism

The *–reneg-bytes*, *–reneg-pkts* and *–reneg-sec* options can be used to trigger automatic session-keys renegotiation after a certain number of bytes, packets or seconds. This key renegotiation mechanism is triggered by the client or the server with a SoftReset message. To focus on the effect of this SoftReset message, we inferred a state machine over the following input symbols:

– wCHRv1
– Tls:ClientHelloAll
– Tls:FullHandshake
– KeyNeg1
– DataPingReq
– SoftReset

The OpenVPN and OpenVPN-NL inferred state machine are similar, except for the forwarding of the TLS alerts and the extra ServerKeyExchange, mentionned in Sec. 5.2. As expected, Fig. 8 shows that the key renegotiation mechanism can only be triggered after the OpenVPN session is initiated, i.e. in states ISC and 4. The SoftReset messages sent before the ISC state end up in a closed connection which is a safe behavior to adopt in a security protocol.

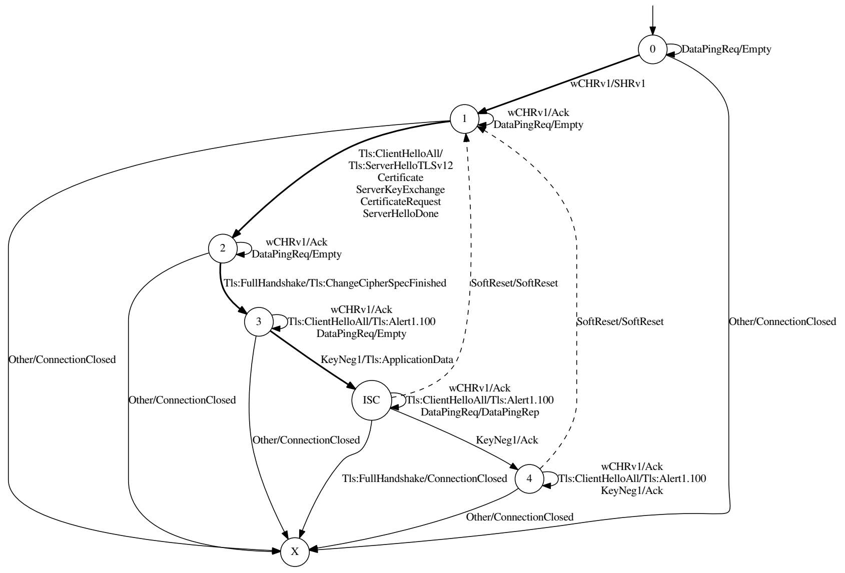

Fig. 8. State machine of an OpenVPN-NL server running in TCP mode, highlighting the key renegotiation mechanism. The dashed labels show the successful soft reset messages.

After a successful SoftReset message, the state machine goes to state 1 and the Data messages are no longer processed by the server (in states 1, 2 and 3 the Data messages are ignored). This is the result of a choice we made in the test harness. The *key_id* that identifies the session-keys of a particular Data message has been incremented by the SoftReset but the second pair of keys has not been negotiated yet. The server will ignore the subsequent Data messages with the wrong *key-id* and, as a result, the state machine is simpler since a successful SoftReset results in a transition to state 1 instead of creating some new state where a Data message using the old session-keys would trigger a response from the server.

The UDP mode is different from the TCP mode and is a good example of the limitations of LearnLib. When the active session-keys are renegotiated via a SoftReset and the OpenVPN initialization sequence fails, the active session state is set as Error, the session is set as the Lame_Duck[8] and a new session is initialized by the server. Then, the server waits for 56 seconds and sends a ServerHardReset. The test harness cannot differentiate this behavior from the regular no reply case, unless it waits 1 minute for a ServerHardReset each time there is no reply from the server. On the other hand, if the test harness does not wait, the ServerHardReset will eventually be caught as a reply to an other message and this will trigger nondeterminism in the learning

[8] See https://build.openvpn.net/doxygen/html/group__control__processor.html for more details on the sessions states

process. We are then stuck in a situation where a long timing-related event can not be suppressed, then trying to infer a state machine would definitely be time consuming.

5.4 Documentation Issues

During the construction of the test-harness, we encountered several complications that might be worth noting for future work on the OpenVPN protocol and that are listed here in decreasing order of importance.

– First, the sequence of messages leading to a successful tunneled data transmission is not explicitly documented, which makes it challenging for a developer to come up with a new OpenVPN implementation. Especially the behavior in case of erroneous messages is not specified, even though it is essential for a security protocol implementation to handle those error cases properly. The sequence of message could be added to the documentation as a protocol state machine similar to those presented in this paper. For instance, the one defined in Fig. 8 gives a good overview of the sequence of messages establishing an OpenVPN session.
– In the documentation, the expected behavior when receiving a HardReset or a SoftReset message is not explicit. It is not specified how the different fields of the messages must be handled and how the messages should affect the server and the client. Moreover, in the implementation it is not clear when a ClientHardReset is taken into account by the server since it does not always trigger a ServerHardReset. Finally the differences between the UDP and TCP modes are not mentioned in the documentation but are clearly visible in the inferred state machines.
– The padding algorithms used for encryption[9] are not specified in the documentation and it would be really helpful to have them documented in the *Data Channel Crypto Module*[10]. Moreover, in the *Data channel key generation* section[11], the process used by OpenVPN to generate key expansion in key-method 2 is only documented by a reference to the source code. It might be helpful to add some extra documentation about the key expansion function and the PRF.
– Finally, we reported a mistake in the security overview [2] and the documentation[12], concerning the order of the fields of the Key Negotiation message using key-method 1.

6 Conclusion

We presented an automatic analysis of two OpenVPN implementations using a technique called protocol state fuzzing: we used regular inference, which relies on

[9] We used the Java PKCS5Padding for Blowfish/CBC and AES/CBC.
[10] https://build.openvpn.net/doxygen/html/group__data__crypto.html
[11] https://build.openvpn.net/doxygen/html/key_generation.html
[12] https://build.openvpn.net/doxygen/html/network_protocol.html

black-box fuzzing, to infer state machines of the OpenVPN server and performed a manual analysis upon them. This approach is able to find logical flaws in the state machine of the implementation, but could not detect for instance, flaws triggered by malformed messages or the recent flaws found using fuzzing [15].

This approach gives a coarse analysis of the implementation and abstracts the fine details. First, the state machine is dependent on the the test harness which defines the input alphabet and semantic of the messages. For instance the abstraction we adopted intentionally conceals the acknowledgement mechanism, but also the smooth transition of the key renegotiation mechanism. Those concessions on the precision of the model are necessary to keep the learning complexity low and reduce the learning time. Second, the timing-related events which plays a great role in the protocol can not be modeled in a simple Mealy Machine and therefore must be abstracted. Modeling timing-related events would require a more complex model of timed-automaton with output, which can currently not be inferred from real systems. In addition, those timing-related events cause nondeterminism in the learning process which can only be handled by introducing timeouts and delays in the test-harness. They are the main bottleneck of the learning process and can explain the learning time ranging from about 40 minutes to 49 hours.

Building a test harness involves re-implementing an OpenVPN client, able to send correct messages to the server, and thus requires a deep understanding of the protocol. It is a difficult and time-consuming application-specific task, definitely more worthwhile if it can be reused to analyze many implementations or throughout the software evolution, such as it has been done for TLS [10, 9] or SSH [13].

The inferred state machines provide a useful insight into the decisions - and errors - made in the implementation. They can be used to easily spot superfluous states and transitions, which should lead to further analysis by experts. As part of a security evaluation, they can help to harden the implementation by simplifying the state machine, thus reducing the chance to subsequently find a vulnerability. They can also be used to automatically infer a specification from an implementation, that could be automatically updated throughout the software evolution.

The inferred state machines from the different implementations of the OpenVPN servers did not reveal any vulnerability, and comply to what would be expected from a security protocol. Whenever a TLS message ends up in a failure of the TLS handshake, the OpenVPN session initialization is aborted and the connection is closed, which is the safest conduct to adopt. The same conduct is adopted when the integrity check or decryption of a Data message raises an error. Otherwise, the incorrect OpenVPN messages are ignored by the OpenVPN server such as messages with unknown session-id (Sec. 5.1), the KeyNeg messages send after the session initialization (Sec. 5.2), or Data message with wrong key-id (Sec. 5.3). Finally, these results can sensibly increase the confidence in the tested OpenVPN implementations.

To conclude, we believe that this is a shame that the sequence of messages leading to a successful OpenVPN connection and the conduct to adopt when receiving an unexpected message are not specified. This information could easily be modeled by one (or several) protocol state machine such as we inferred. The documentation would really benefit from the addition of the expected state machine, like for instance the one defined in Fig. 8 which gives a good overview of the sequence of messages establishing an OpenVPN session. Alongside a prose specification could be added to describes the main timing-related events and more details on how to handle error cases such as unexpected or incorrect input messages.

Bibliography

[1] OpenVPN source code documentation. https://build.openvpn.net/doxygen/html/, . Accessed: 2017-06-22.

[2] OpenVPN security overview. https://openvpn.net/index.php/open-source/documentation/security-overview.html, . Accessed: 2017-06-22.

[3] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. *ICTSS*, 6435:188–204, 2010.

[4] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 673–686, 2010.

[5] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal models of bank cards for free. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 461–468. IEEE, 2013.

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

[7] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication*, 800:131A, 2011.

[8] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

[9] Joeri de Ruiter. A tale of the openssl state machine: A large-scale black-box analysis. In *Nordic Conference on Secure IT Systems*, pages 169–184. Springer, 2016.

[10] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium*, pages 193–206, 2015.

[11] Gregorio Díaz, Fernando Cuartero, Valentín Valero, and Fernando Pelayo. Automatic verification of the TLS handshake protocol. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 789–794. ACM, 2004.

[12] Markus Feilner. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd, 2006.

[13] Paul Fiterau-Brostean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of ssh implementations. 2017.

[14] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally composable security analysis of TLS. *ProvSec*, 5324:313–327, 2008.

[15] Vranken Guido. The OpenVPN post-audit bug bonanza. https://guidovranken.wordpress.com/2017/06/21/the-openvpn-post-audit-bug-bonanza/. Accessed: 2017-08-9.

[16] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C Mitchell. A modular correctness proof of ieee 802.11 i and TLS. In

Proceedings of the 12th ACM conference on Computer and communications security, pages 2–15. ACM, 2005.

[17] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-dhe in the standard model. In *Advances in Cryptology–CRYPTO 2012*, pages 273–293. Springer, 2012.

[18] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *Advances in Cryptology–CRYPTO 2013*, pages 429–448. Springer, 2013.

[19] Maik Merten, Malte Isberner, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automated learning setups in automata learning. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 591–607, 2012.

[20] Paul Morrissey, Nigel Smart, and Bogdan Warinschi. A modular security analysis of the TLS handshake protocol. *Advances in Cryptology-ASIACRYPT 2008*, pages 55–73, 2008.

[21] Kazuhiro Ogata and Kokichi Futatsugi. Equational approach to formal analysis of TLS. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 795–804. IEEE, 2005.

[22] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):332–351, 1999.

[23] Erik Poll, Joeri De Ruiter, and Aleksy Schubert. Protocol state machines and session languages: specification, implementation, and security flaws. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 125–133. IEEE, 2015.

[24] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71. ACM, 2005.

[25] Guoqiang Shu and David Lee. Testing security properties of protocol implementations-a machine learning based approach. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 25–25. IEEE, 2007.

[26] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. *IACR Cryptology ePrint Archive*, 2017:190, 2017.

[27] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Crypto*, volume 3621, pages 17–36. Springer, 2005.

A Acronyms

VPN	Virtual Private Networks
OSI	Open Systems Interconnection

SUL	System Under Learning
PDU	Protocol Data Unit
PRF	Pseudorandom Function
HMAC	Hash-based Message Authentication Code
SHA-1	Secure Hash Algorithm 1
CBC	Cipher Block Chaining
TLS	Transport Layer Security
DoS	Denial of Service
VM	Virtual Machine
SSH	Secure Shell
ISC	Initialization Sequence Complete
DH	Diffie-Hellman

B The OSI Model

A protocol is a well-defined set of rules that describes the interaction between communication entities in networks. They can be classified into layers according to their role in the communication using the Open Systems Interconnection (OSI) model. OSI defines seven independent abstraction layers which take care of a specific job and send the data to the adjacent layer. The OSI model scheme is described in Tab. 1.

	Layer	**PDU**	**Examples**
Host Layers	7.Application	Data	HTTP, FTP, POP, SMTP, IMAP, IRC, SSH, TELNET, BitTorrent...
	6.Presentation		
	5.Session		
Media Layers	4.Transport	Segment (TCP) Datagram (UDP)	TCP, UDP, SSL, ICPM...
	3.Network	Packet	IPv4, IPv6, IPSEC...
	2.Data Link	Frame	Ethernet, Wifi, Token ring, PPP...
	1. Physical Link	Bit	10BASE-T, RS-232, USB physical layer

Table 1. The seven OSI layers with examples of widespread protocols for each layer.

C OpenVPN Sequence of Messages

This section describes the regular sequence of messages exchanged between a client and a server initiating a OpenVPN connection in TLS-mode. This sequence of messages, called the *happy-flow*, is illustrated in Fig. 9. All the messages are acknowledged via the OpenVPN reliability layer, except the Data messages.

First, the client initiates the connection by sending a ClientHardReset message which contains an identifier referring to its new session, namely its

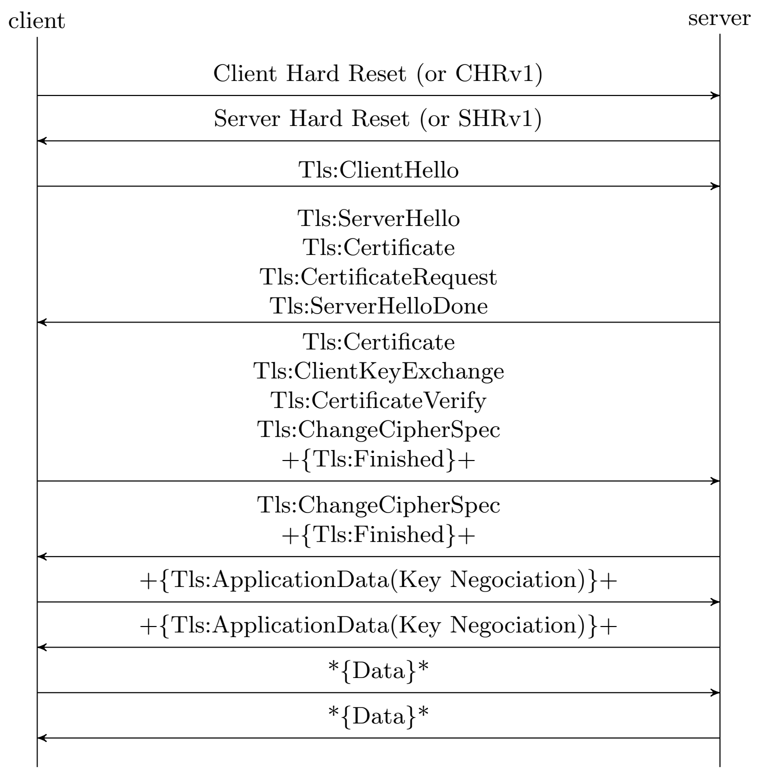

Fig. 9. A regular OpenVPN session using TLS mode, without DH cipher suite. A message secured with the TLS keys is denoted as +{msg}+, and a message secured with the VPN session-keys is denoted as *{msg}*. The ACK messages are omitted.

session-id. The server answers with a SERVERHARDRESET message which contains its own session-id.

Then, a fully authenticated TLS handshake is initiated between the two peers, which means that both must present their own certificate. Note that the TLS sequence of message can change depending on the selected cipher suite: the SERVERKEYEXCHANGE message is only sent for DH cipher suites. This difference can be seen in Fig. 5 and Fig. 6 since the default cipher suites used for the TLS sessions differ: contrary to OpenVPN, OpenVPN-NL uses a DH cipher suite and thus sends a SERVERKEYEXCHANGE.

When the TLS handshake is complete, the client generates its random material and sends it to the server in a TLS APPLICATIONDATA message, secured with the TLS keys, as illustrated in Fig. 10. The server uses the random material to recover the session-keys (depending on the key-method) and sends its own random material to the client.

The Key Negotiation messages also contains an option string which is used by the remote peer to check the consistency of the configuration options. The OpenVPN implementations are pretty liberal in the way they handle this option string: if the remote string does not match the local options, a warning is sent but the connection is not aborted - even with an empty string.

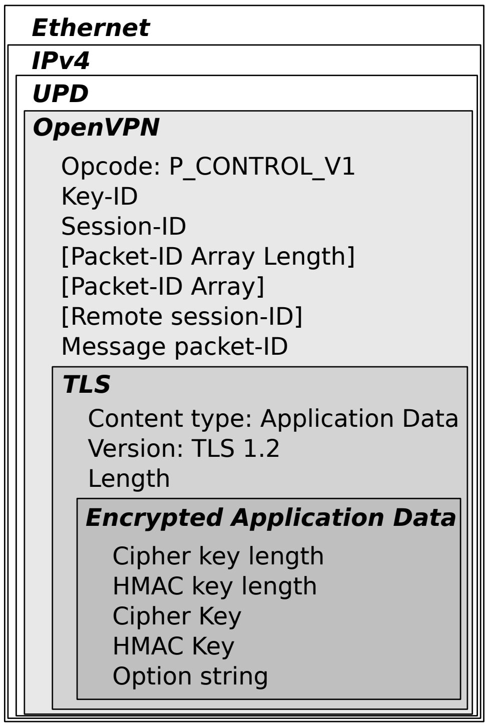

Fig. 10. An OpenVPN (TLS mode and key-method 1, without *–tls-auth*) frame exchanging the session-key material over a TLS session. Fields denoted as [field] are part of the acknowledgement mechanism and are optional.

Once both peers have received the session-keys, they can start exchanging the actual Data messages. The Data messages are IP packets or Ethernet frames, encrypted and MAC-ed used the session-keys.

D Causes of Nondeterminism in the Learning Process

This section details the causes of nondeterministic replies from the server and the techniques adopted to circumvent them. Subsection D.1 focuses on the network related causes such as packet loss and Subsec. D.2 focuses on the timing-related events.

The OpenVPN server is running on a VMware Virtual Machine (VM) hosted on the same computer as the test harness. The server can be started and killed by the test harness using SSH. This configuration has been adopted for its simplicity.

D.1 Network Related Causes of Nondeterminism

While using VMware with the NAT connection, sometimes for about five seconds, all the packets sent by the virtual machine (OpnVPN but also SSH packets) get lost inside the VM (before reaching the interface between the VM and the host). The solution is to switch to the *Host-Only* connection, which also prevent the VM from accessing the Internet.

Even after switching to the Host-Only connection, the long phases of learning raised some nondeterministic behavior because of packet loss. For example, during the automatic time synchronization of the VM (which eventually fails because it cannot access the Interent), the OpenVPN packets are significantly delayed. We took the following precautions to avoid packet loss:

– Use *Host-Only* connection on the VM,
– Turn off the Internet connection on the host,
– Disable automatic time synchronization on the VM,
– Disable automatic system updates on the VM.

D.2 Timing-related Causes of Nondeterminism

This section details the timing-related events that cause nondeterminism in the learning process, and the solutions adopted to circumvent those issues. The server is configured via a configuration file, which path is specified in the command line. The configuration file of the server includes a *–hand-window* option to set the TLS-based key exchange timer, and a *–verbosity* options to set the output verbosity of the server from 0 to 9.

TLS handshake window By default, the TLS-based key exchange must finalize within 60 seconds. Otherwise, the handshake will fail and the connection will be reset. The solution we adopted is to increase the *–hand-window* timer from 60 to 60000.

Restart of the connection The TLS handshake can fail to succeed because of an unexpected TLS message. In such a case, the connection is shut down and the server waits a 2 seconds delay before restarting the connection. In TCP mode, the client can detect the closed socket but in UDP mode, detecting the shutdown is not directly possible because ICMP packets cannot be caught. Two workarounds can be considered. First, whenever the server does not respond to the client, the client waits for 2.5 seconds. The inconvenience is that a closed connection can not be distinguished from no reply from the server, so in this case the client will wait for no reason. The second solution is to run a *nc* command[13] and scan the ports of the server in case of empty

[13] nc(1) - Linux man page: https://linux.die.net/man/1/nc

reply. In addition of being less time-consuming, this solution allows the client to detect a closed connection and to close its own.

Reset time of the server Because subsequent queries must be independent, each time the learner starts a new query, the server must be reset. It is effectively done by establishing a new SSH session, killing the old OpenVPN process and start a new one. This command might take some time to fully execute and, if the client starts sending messages to the server before it is ready, those messages will be lost. To prevent this, a one second delay has been added on the client side.

Server verbosity If the *–verbosity* level is too high, extra SSH messages containing server logs would be exchanged between the host and the VM, which would also slow down the VPN connection (or there would be an extra delay to redirect the server output stream to a file on the VM). Moreover, each reset of the server would prompt its configuration which would also be sent through the SSH tunnel. This operation can increase the reset time of the server which is already long enough. Thus, it is better to set the verbosity to zero in the server configuration file during the learning process, otherwise the timeouts have to be increased.

TCP and UDP timeouts The TCP and UDP timeouts must be carefully chosen to avoid nondeterminism. They must be long enough so the client is sure to receive the message, but short enough to speed up the learning process and not trigger the resend mechanism of OpenVPN because the server did not receive the acknowledgement.

These timeouts delays added in the test harness are the bottleneck of the learning process:

- The *nc* command with a 2 sec. timeout, whenever the server does not reply to the client in UDP mode.
- The 1 sec. delay each time the server is reset (between each queries).
- The TCP and UDP timeouts (respectively 800 and 100 ms) for each message sent to the server.

The learning time mainly depends on the input alphabet and the resulting state machine. The average learning time for the state machines in Sec. 5.1 is about 40 minutes, for the state machines in Sec. 5.3 it is about 3:50 hours, and for the state machines in Sec. 5.2 it is about 49 hours. Similar work on protocol state fuzzing of TLS implementations [10] ranged from about 9 minutes to over 8 hours. For approximately the same number of queries, our learning time is considerably longer because of the timeouts and delays introduced.

E Learner's Input Alphabet

This section describes the abstract alphabet of the learner and gives a short description of the semantic of each symbol (in other words, the impact of the symbol on the test harness). Table 2 contains the symbols referring to OpenVPN messages and Table 3 contains the symbols referring to TLS messages.

Message	Description
CHRv1	Send a ClientHardResetV1 message with fresh session-id, packet-id and TLS session.
wCHRv1	Send a ClientHardResetV1 message but keep the old parameters.
KeyNeg1	Generate fresh session-keys and send an ApplicationData TLS message containing the new session-keys for key-method 1.
SoftReset	Increment the key-id and send a SoftReset message.
DataPingReq	Send a ping request to the server through the OpenVPN tunnel
CHRv2	Send a ClientHardResetV2 message with fresh session-id, packet-id and TLS session.
KeyNeg2	Send a ApplicationData TLS message containing the session-key random material for key-method 2.

Table 2. Description of the OpenVPN input symbols.

Message	Description
Tls:ClientHelloAll	Send a ClientHello TLS message.
Tls:ClientKeyExchange	Send a ClientKeyExchange TLS message.
Tls:ClientCertificate	Send a Certificate TLS message.
Tls:ClientCertificateVerify	Send a CertificateVerify TLS message.
Tls:ChangeCipherSpec	Send a ChangeCipherSpec TLS message.
Tls:Finished	Send a Finished TLS message.
Tls:FullSession	Send all the TLS messages from Tls:ClientHelloAll to Tls:Finished, plus KeyNeg1
Tls:FullHandshake	Send all the TLS handshake messages from Tls:ClientKeyExchange to Tls:Finished

Table 3. Description of the TLS input symbols in their order of arrival in the initialization of the TLS session.