Inferring OpenVPN State Machines Using Protocol State Fuzzing

Lesly-Ann Daniel
University of Rennes 1 - ENS Rennes
Email: lesly-ann.daniel@ens-rennes.fr

Joeri de Ruiter
Radboud University
Email: joeri@cs.ru.nl

Erik Poll
Radboud University
Email: erikpoll@cs.ru.nl

***Abstract*—The reliability of a security protocol is of the utmost importance but can easily be compromised by a vulnerability in the implementation. A crucial aspect of an implementation is the protocol's state machine. The state machine of an implementation can be inferred by black box testing using regular inference. These inferred state machines provide a good insight into implementations and can be used to detect any spurious behavior. We apply this technique to different implementations of OpenVPN: the standard OpenVPN and the OpenVPN-NL implementations. Although OpenVPN is a widely used TLS-based VPN solution, there is no official specification of the protocol, which makes it a particularly interesting target to analyze. We infer state machines of the server-side implementation and 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 is missing from the documentation, and discuss the possibility to include state machines in a formal specification.**

I. Introduction

Virtual Private Network (VPN) solutions are widely used to establish secure data transmissions over insecure channels (e.g. a 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. VPNs use a tunneling mechanism to provide an additional layer to ensure confidentiality, authentication and integrity independent of the underlying protocol. The security of the protocols used to achieve this can easily be compromised by a vulnerability in the implementation.

To automatically detect some of these vulnerabilities we can use formal methods. Regular inference, or protocol state fuzzing, is a technique to infer a state machine from the implementation of a protocol [1]. The inferred state machine provides a useful insight into the choices and errors made in the implementation. It should allow all the transitions defined by the grammar of the protocol and react appropriately to unexpected messages, by ignoring the message or dropping the connection. The inferred state machine can be analyzed to detect any logical flaws and to check compliance of the implementation with its specification. It can also reveal superfluous states and transitions which should be removed as a precaution. Finally, 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 [2].

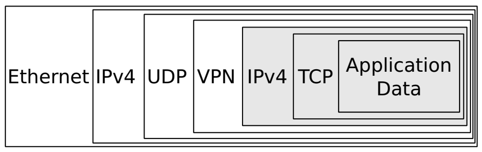

Fig. 1. Example of OpenVPN tunneling IP packets over an UDP channel. The gray message benefits from the security properties (confidentiality, integrity, authentication) provided by the enclosing OpenVPN protocol.

This paper focuses on the OpenVPN protocol [3], which is based on TLS. Even though it is a widely used VPN solution, it has not been subject to a lot of research and no formal specification exists of the protocol. Indeed, there is no documentation about the sequence of messages leading to a successful connection nor about the correct behavior in response to receiving unexpected messages - even though this is essential for a security protocol.

We use LearnLib [4] to infer state machines of two different OpenVPN servers: the standard OpenVPN implementation based on OpenSSL, and the OpenVPN-NL implementation based on PolarSSL. 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 show a lot of information about the implementation that cannot be found in any documentation. Finally, we discuss how state machines can be used to define a formal specification of the protocol.

The OpenVPN protocol is introduced in Section II and protocol state fuzzing in Section III. We present our experimental setup in Section IV and the results of our analysis in Section V. Finally, the related work is discussed in Section VI and we conclude in Section VII.

II. The OpenVPN Protocol

OpenVPN provides *tunneling* to provide *confidentiality*, *authentication* and *integrity* for the data transmitted. The entire message to be transmitted (IP packet or Ethernet frame, including its meta-data like sender and recipient) is encapsulated within an OpenVPN message, as illustrated in Figure 1. For an in-depth presentation of OpenVPN, see [3], the doxygen-generated documentation [5], or the security overview on the OpenVPN website [6].

DOI 10.1109/EuroSPW.2018.00009

OpenVPN offers a large choice of security options. It is based on the OpenSSL library, which is used for its TLS session negotiation, its encryption and authentication and its random number generation primitives. Two methods of key exchange are provided: a pre-shared key and a TLS-based mechanism. 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 more straightforward. In both methods, each peer possesses four independent and unidirectional *session-keys*: HMAC-send, HMAC-receive, encrypt and decrypt, used to encrypt and MAC the data messages.

The TLS mode is based on TLS, which has been subject to a lot of research [7, 8, 9, 10, 11, 12, 13, 14, 15]. A TLS session with bidirectional authentication is negotiated between the client and the server (i.e. both parties must present their own certificate) and is used to securely establish the session-keys. There are two methods of session keys establishment: in key-method 1, each peer generates their own cipher and HMAC keys and sends them to the other; while in key-method 2, the keys are computed by mixing random material from both parties using the TLS pseudo-random function (PRF). Once both peers have received the session keys, the data tunneling can start: the actual data (IP packet or Ethernet frame) to transmit are encrypted, MAC-ed and encapsulated within a DATA message. Figure 2 shows the normal sequence of messages leading to a successful connection.

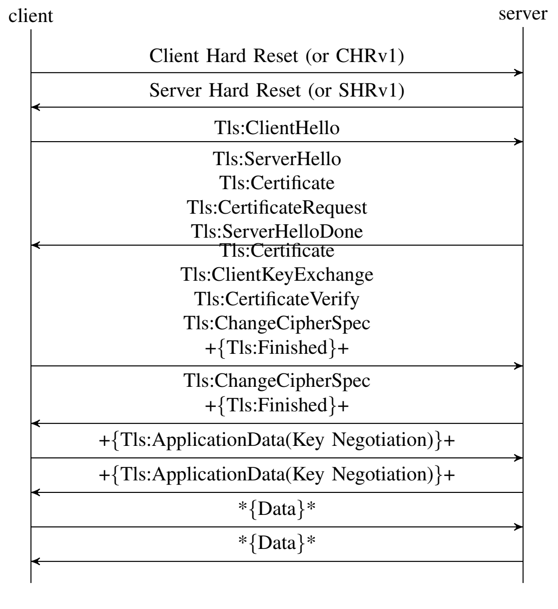

Fig. 2. A regular OpenVPN session using TLS mode, without DH cipher suite. A message secured with the TLS keys is denoted as +{msg}+, and a message secured with the VPN session-keys is denoted as *{msg}*. All the messages are acknowledged via the OpenVPN reliability layer, except the DATA messages. The ACK messages are omitted.

The TLS session negotiation and the data tunneling are processed over independent channels: the *control channel* and the *data channel*, with their own packet identifiers and keys.

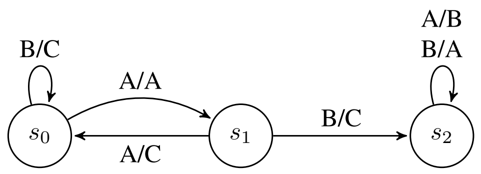

Fig. 3. A Mealy machine with 3 states

OpenVPN multiplexes the *control channel* and *data channel* over a single network stream which is not necessarily reliable. OpenVPN actually prefers to use UDP transport instead of TCP, due to the TCP reliability layer collisions when tunneling TCP over TCP[1]. Because 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 active in both UDP or TCP tunneling. Note that the data channel can still benefit from a reliability layer provided by the encapsulated protocol, e.g. tunneling of a TCP session will benefit from reliability data transfer offered by TCP.

The OpenVPN implementation also offers the *–tls-auth* option to authenticate packets from the control channel by adding an HMAC to the control messages. This mechanism allows OpenVPN to quickly throw away unauthenticated packets, without wasting resources and thus protecting against DoS attacks, and it reduces the attack surface for finding exploitable software bugs for any Man-in-the-Middle attacker.

III. Protocol State Fuzzing

Protocol state fuzzing is defined in [7] as a technique that uses regular inference (a.k.a. automata learning or state machine inference) to infer a state machine from a protocol implementation. Regular inference uses black-box fuzzing on the order of well-formed messages to automatically infer a state machine which models the implementation of a system, based on its external behavior. In this paper, those state machines are represented as Mealy machines.

A. 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. The Mealy machines we use are deterministic, i.e. for each input and current state, only one transition is possible. Figure 3 shows a graphical representation of a simple Mealy machine. 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.

We will use Mealy machines to model the behavior of the OpenVPN server. They describe how the server reacts in response to input messages: which output it produces and how its state is affected. The next part discusses how state machines can be automatically inferred.

[1]http://sites.inka.de/sites/bigred/devel/tcp-tcp.html

B. *Regular Inference*

The state machine of the OpenVPN server is inferred using *regular inference*, a technique based on black-box fuzzing where 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 [4]. The system which is analyzed, namely the OpenVPN server, is referred to as the *system under learning* (SUL) and its state machine is denoted by M.

The regular inference involves two actors: a *learner* (the LearnLib library) and a 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.

The learner is composed of two parts: the *learning algorithm* and the *equivalence algorithm*. The learning algorithm will keep sending *membership queries* (i.e. what is the response to a sequence of input symbols?) 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 an *equivalence query* (i.e. is the hypothesized state machine H equivalent to M?). As we cannot know for sure whether a hypothesis is equivalent to the implemented state machine, we need to approximate this check. If H is deemed equivalent to M, then H is returned as the model of the SUL. Else, the equivalence algorithm returns a counterexample which is used to refine the hypothesis and the learning algorithm is resumed until it finds a new strong hypothesis. The learning algorithm is resumed until the hypothesized state machine H is deemed equivalent to M.

The learning algorithm used in this paper is Niese's modified version of Angluin's L* algorithm which can be used to infer Mealy machines [1, 16]. The equivalence algorithm is a modified version of the W-method [17], refined to cut off entire search branches based on the fact that once a connection is closed by the server it will remain closed [7]. Therefore, we stop building queries over prefixes that end up with a closed connection. Finally the test harness (detailed in Section IV-A) controls the communication between the learner and the SUL.

IV. Setup

The servers we test run on a VMware virtual machine hosted on the same computer as the learner and can be started or reset via SSH. The L* learning algorithm and the W-method equivalence algorithm are both provided by LearnLib.

A. *Test Harness*

The main challenge to infer a correct state machine is to prepare the application specific learning setup, i.e. the *test-harness*. 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 [18]. Consequently, the test harness consists of a *mapper* and a *monitor*. The test harness implementation is based on the previous work of de Ruiter and Poll [7] on TLS. The source code is available at https://github.com/jderuiter/statelearner/tree/openvpn. Note that the test harness can be reused to analyze several versions of the protocol as long as the language of messages has not been changed.

The learner is provided with an abstract input alphabet upon which it builds queries intended for the OpenVPN server. However, the OpenVPN server expects actual messages and not the abstract symbols from the learner's input alphabet. Similarly, the learner will expect the responses from the server as abstract symbols from its output alphabet. Therefore, the test harness contains a mapper which translates the abstract symbols to the actual OpenVPN packets, and vice versa. 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.

The monitor is in charge of building correct system inputs, based on concrete runtime data that influence the behavior of the system; basically, it consists of a stateless OpenVPN client. For example, it sends the messages through the network, processes the responses to recover important information (e.g. session ids and keys), handles the acknowledgement process, and implements the security primitives in the way expected by the server (e.g. valid authentication, encryption and signatures). Managing this runtime data requires a deep understanding of OpenVPN to make decisions concerning the semantics of the abstract input symbols, which will affect the final state machine.

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 [5], and the security overview [6]. When more in-depth analysis was needed, we used the OpenVPN source code and the server logs with maximum verbose output.

B. *Nondeterminism Issues*

That the SUL is deterministic is of paramount importance since LearnLib can only learn state machines of deterministic systems. Nondeterministic behavior of the SUL can produce a wrong model, or cause unexpected behavior of the learner, incl. non-termination. Unfortunately, the OpenVPN server has some nondeterministic behaviour that we have to hide to the learner to be able to infer a state-machine. 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. When nondeterminism is suspected (e.g. because the learner does non terminate), we manually analyze the query cache to find the query with a nondeterministic answer. The defective query can be analyzed further to track down the cause of nondeterminism through the log file of the server and the Wireshark traces. When the source of nondeterminism is identified, we design a "trick" to work around it.

There are two main 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. We did not expect this kind of behaviour, since our server is simply running on a virtual machine on the same computer as the client. However, with some configurations of the VM (e.g. when using NAT connection, or during time-synchronization of the VM), we did experience that sometimes the connection dropped, which caused the learning process to fail. The solution is to adapt the configuration of the VM to circumvent these issues (e.g. turn to host-only connection, disable automatic time synchronization).

Second, there are multiple timeouts and delays on the server side, e.g. the reset time of the server, the time to process the messages, and the TPC and UDP timeouts. The response to a query may vary depending on those timing-related events, which is seen as nondeterminism by the learner. The solutions we adopted to work around this timing-related nondeterminism often implied longer sleeping-times or timeouts on the client side. This has a big impact on the learning time and constituted the main bottleneck of the learning process. 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 setting them too long can significantly slow down the learning process. For example, for set the UDP and TCP timeouts we started with low values and increased them until there were no more packet losses (100 ms for UDP and 800 ms for TCP).

In addition, in order to prevent a wrong counterexample to be added to the hypothesis after an equivalence query, we modified the equivalence algorithm to detect nondeterminism. Each time a counter-example is found by the equivalence algorithm, the query is processed again to check whether the outputs match. If both outputs are the same, we assume that the output is correct and that a counterexample has been found, otherwise, an exception is raised for nondeterminism. This simple modification could be added to LearnLib as an option to detect nondeterminism. The message replay was good enough in our situation because the probability of nondeterminism is very low (less than 1/100); however, it may not work for higher probabilities of nondeterminism. In general, nondeterminism can either be caused by a nondeterministic target, which therefore it cannot be modeled as a Mealy machine so it is out of the scope of our approach, or an unreliable environment (e.g. packet losses) which we can try to resolve by making the environment more reliable (e.g. increasing the timeouts or replaying the packets the appropriate number of time given the probability of packet loss).

C. Input Alphabet for Learning

In order to keep the learning complexity low, we only include messages that would be accepted given the server configuration and we abstract away the acknowledgement mechanism. So, the SERVERHARDRESET message and the messages for key-method 2 which result in a closed connection are not included. We also only include TLS messages required to establish a successful OpenVPN session. An OpenVPN session is considered successful when the initialization sequence is complete and the data tunneling can start. 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 providing different levels of abstraction, to infer various state machines and highlight different behaviors of the server (which are detailed in Section V). This also permits to lower the complexity of the learning process by reducing the size of the input alphabet and the final number of states in the model.

V. RESULTS

We analyzed two different implementations of OpenVPN: OPENVPN 2.3.10 using OPENSSL 1.0.2G, referred to as OpenVPN, and OpenVPN-NL[2] based on OPENVPN 2.3.9 using POLARSSL 1.2.19, referred to as OpenVPN-NL. OpenVPN-NL is a stripped and hardened version of OpenVPN, intended for Dutch government use, which disallows insecure configurations. The server is 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, 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.

A. The OpenVPN Session Initialization

From the documentation[3] 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. the 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. Note that those session slots are an implementation choice and not a fundamental aspect of the OpenVPN protocol. Each OpenVPN session is initiated with a CLIENTHARDRESET message (CHRV1) that has a unique session-id. However, the expected impact of the CHRV1 on the server is not specified in the documentation. Therefore we tried to highlight it by building a state machine over three input symbols: CHRV1 initiates a new session, TLS:FULLSESSION

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

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

treats the entire TLS-based key exchange as one atomic step, and DATAPINGREQ sends a ping request through the tunnel.

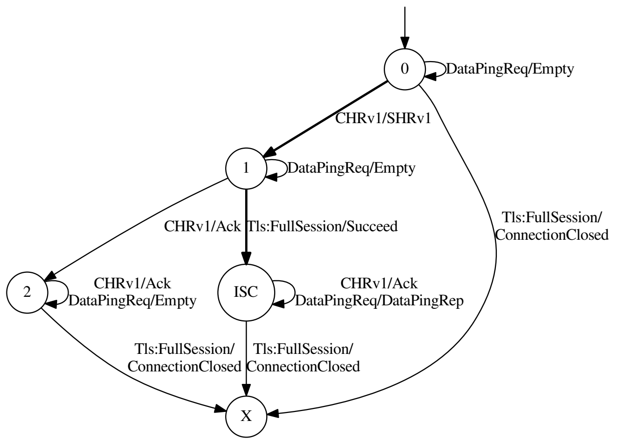

Fig. 4. State machine of an OpenVPN or OpenVPN-NL server running in TCP mode.

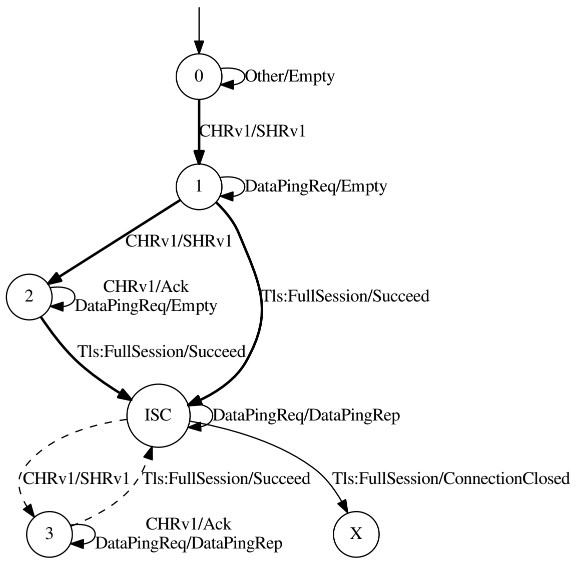

Fig. 5. State machine of an OpenVPN or OpenVPN-NL server running in UDP mode.

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 is quite surprising, even though it does not seem insecure.

Starting with a TLS message in TCP mode results in a closed connection ($0 \rightarrow X$ in Figure 4). Conversely, these messages are ignored in UDP mode ($0 \rightarrow 0$ in Fig. 5) since all 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 CHRv1, (the dashed loop in Figure 5). This is not possible in TCP mode since only the first CHRv1 can result in a successful connection, whereas the others eventually result in a closed connection, as can be seen in Figure 4. We found an explanation for this difference: in UDP mode the server cannot know if the connection is closed on the client side, contrary to TCP mode. Therefore, if the client reconnects and tries to initiate a new session by sending a new CHRv1, the server can process the CHRv1 and the new session can be seamlessly renegotiated.

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 Figure 5 from the path $0 \rightarrow 1 \rightarrow 2$ as the first two CHRv1 trigger a response from the server, but after reaching the state ISC, only one CHRv1 triggers a SHRv1 (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 5 also shows that in UDP mode, a session initiated with a CHRv1 message can succeed without triggering a SHRv1 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 SHRv1 when filling a new session slot. The first and second CHRv1 fill the first and second slots but the third CHRv1 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. $0 \rightarrow 1 \rightarrow 2 \rightarrow ISC$ and $0 \rightarrow 1 \rightarrow ISC$ in Figure 5), while in TCP mode there is only one path ($0 \rightarrow 1 \rightarrow ISC$ in Figure 4).

Finally from the server logs we observe that in TCP mode, the structure containing the second session is allocated when receiving the CHRv1 but the corresponding SHRv1 is never sent and the session is stuck in the S_PRE_START state[4]. However, the subsequent TLS messages are processed by the server though the responses to the TLS:CLIENTHELLOALL are not forwarded to the client. Finally if the TLS handshake is continued, the TLS:CERTIFICATEVERIFY triggers an error for a bad signature and the connection is dropped by the server.

B. The TLS Handshake

Next we focus on analyzing the details of the TLS sessions used to set up an OpenVPN connection. The whole TLS session negotiation was previously abstracted into a single TLS:FULLSESSION step. To investigate it in more detail, it is split into several steps corresponding to the different TLS messages: TLS:CLIENTHELLOALL, TLS:CLIENTCERTIFICATE, TLS:CLIENTKEYEXCHANGE, TLS:CLIENTCERTIFICATEVERIFY, TLS:CHANGECIPHERSPEC, TLS:FINISHED, and KEYNEG1.

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

In order to make the state machine simpler we change CHRv1 to wCHRv1 that focuses on only one session by keeping the previous session-id and TLS session parameters. Resetting the packet-id in wCHRv1 (as done in CHRv1) introduces an issue w.r.t. the acknowledgement mechanism because the CONTROL messages with a known packet-id are considered to be replayed packets by the server. Thus, the responses of the server after wCHRv1 would depend on the number of control messages previously sent and the server could no longer be modeled as a *finite* Mealy machine. For this reason, wCHRv1 does not reset the packet-id, unlike CHRv1.

Figure 6 shows the resulting state machines for OpenVPN and OpenVPN-NL. Most messages resulting in a closed connection have been removed for readability and the sequence of messages from TLS:CLIENTCERTIFICATE to TLS:FINISHED has been condensed into the TlsHsk state.

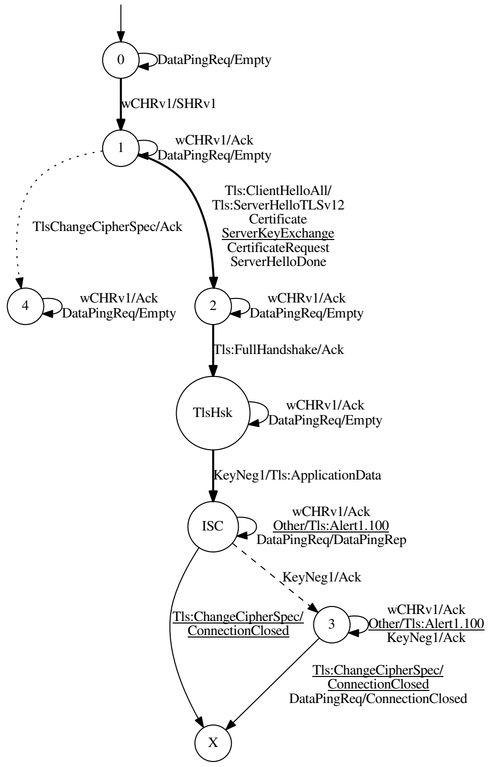

Fig. 6. State machine of an OpenVPN server. The dotted edges correspond to a transition specific to OpenVPN and the underlined messages are specific to OpenVPN-NL

The differences between the OpenVPN and OpenVPN-NL state machines are only due to the different TLS implementations and 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 [7]. For example, the OpenSSL implementation does not return an error when a CHANGECIPHERSPEC is sent before a CLIENTHELLOALL, hence the dead-end state 4 where the TLS session can no longer succeed, which is specific to OpenVPN, as OpenVPN-NL simply closes the connectionin this case.

The OpenVPN-NL implementation is more permissive in some other situations. When the TLS handshake is complete (in states ISC and 3) and an extra TLS handshake message is sent, OpenVPN-NL returns an ALERT (see the underlined labels), whereas OpenVPN closes the connection.

OpenVPN uses TLS_RSA_WITH_AES_128_CBC_SHA as its cipher suite, whereas OpenVPN-NL uses the cipher suite TLS_DHE_RSA_WITH_AES_256_CBC_SHA. This difference explains the extra SERVERKEYEXCHANGE in the OpenVPN-NL state machine which is only sent when using Diffie-Hellman (DH) key exchange.

Both implementations allow the client to send several KEYNEG1 messages over the TLS session, but only the first one is processed and the others are ignored. In our test harness, we made the choice to generate and send fresh session-keys (used to encrypt and MAC the DATA messages) when sending a new KEYNEG1 message. This results in the extra state 3 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 3 to X.

Finally there is a difference in TCP and UDP modes (Figure 7) because the acknowledgement process is not respected in TCP mode (which is not specified in the documentation). Starting the communication with a CONTROL message different from wCHR1 results in the dead-end states 2 and 3 because in state 2, the server receives a wCHR1 with a packet-id $n > 0$ and waits for the messages with a packet-id lower than n in state 3. Therefore, the subsequent TLS messages are not processed.

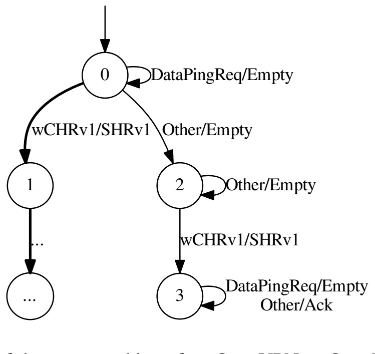

Fig. 7. Subset of the state machine of an OpenVPN or OpenVPN-NL server, focusing on the particularity of the UDP mode. State 1 and its subsequent states are identical to the TCP version.

C. The Key Renegotiation Mechanism

In OpenVPN, renegotiation of the session keys can be triggered automatically with a SOFTRESET message after a certain number of bytes, packets or seconds by either the client or the server. To focus on the effect of this SOFTRESET message, we infer a state machine using the following input symbols: wCHRv1, TLS:CLIENTHELLOALL, TLS:FULLHANDSHAKE (which contains the TLS messages from TLS:CLIENTKEYEXCHANGE to TLS:FINISHED), KEYNEG1, DATAPINGREQ and SOFTRESET. The inferred state machines for OpenVPN and OpenVPN-NL are similar, except for the responses containing TLS alerts and the extra SERVERKEYEXCHANGE message mentioned in Section V-B. As expected, Figure 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.

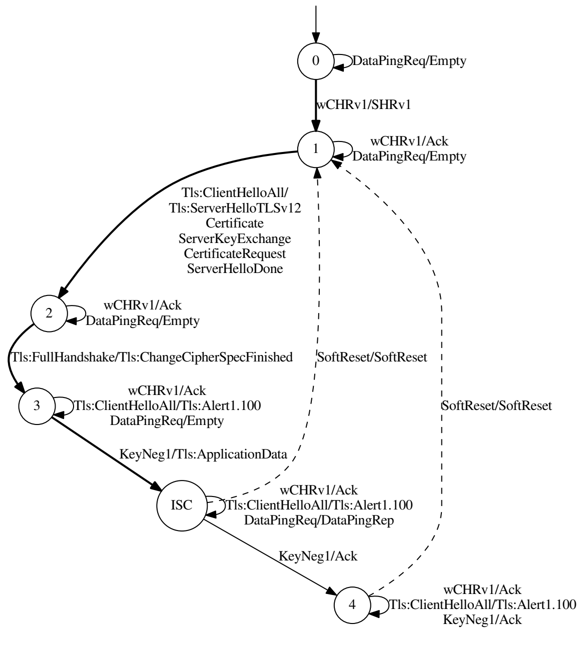

Fig. 8. State machine of an OpenVPN-NL server running in TCP mode, highlighting the key renegotiation mechanism. The dashed labels show the successful soft reset messages. Messages resulting in a closed connection have been removed for readability.

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, if initialization sequence fails then the active session state is set as ERROR and a new session is initiated by the server which waits for 56 seconds and sends a SERVERHARDRESET. The test harness cannot differentiate this behavior from the regular 'no reply' case unless it waits for 1 min to catch the SERVERHARDRESET each time there is no reply from the server. If the client does not wait, the SERVERHARDRESET will eventually be caught as a reply to another message, which introduces nondeterminism in the learning process. In this situation the long timing-related event cannot be suppressed and trying to infer a state machine would be too time consuming.

D. Documentation Issues

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

First, the sequence of messages leading to a successful tunnel 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. This sequence of messages could be added to the documentation as a protocol state machine similar to those presented in this paper. For instance, Figure 8 gives a good overview of the sequence of messages that establishes an OpenVPN session.

In the documentation the expected behavior when receiving a HARDRESET or SOFTRESET message is not made explicit. It is not specified how the different fields of the messages must be handled or 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[5] are not specified in the documentation and it would be helpful to have them documented in the *Data Channel Crypto Module*[6]. Moreover, in the *Data channel key generation* section[7], the process used by OpenVPN to perform key expansion in key-method 2 is only documented by a reference to the source code. It would be helpful to include some more documentation on the key expansion function and the pseudorandom function.

Finally, we reported a mistake in the security overview [6] and the documentation[8] which has not been corrected yet. The order of the fields of the KEY NEGOTIATION message in key-method 1 do not match the implementation: the documentation reports `cipher-key length`, `cipher-key`, `HMAC-key length` and `HMAC-key`, but the message actually starts with `cipher-key length` and `HMAC-key length`.

[5]We used the Java PKCS5PADDING for BLOWFISH/CBC and AES/CBC.
[6]https://build.openvpn.net/doxygen/html/group__data__crypto.html
[7]https://build.openvpn.net/doxygen/html/key_generation.html
[8]https://build.openvpn.net/doxygen/html/network_protocol.html

VI. Related Work in Protocol State Fuzzing

The idea of using regular inference to analyse implementations of security protocols dates back to at least Shu and Lee [19]. An extensive survey of this and other techniques to reverse engineer protocol implementations has been given by Narayan et al. [20].

Regular inference with LearnLib has been applied to analyze implementations of EMV payments cards [21], biometric passports [22], TLS [7, 23], and SSH [24]. Nearly always different implementations of the same protocol turn out to have different state machines, so regular inference can be used to fingerprint a particular implementation. In most cases the impact of fingerprinting is limited, but it can leak confidential information; for example, a comparison of e-passport implementations from ten different countries showed that the nationality can be determined from eacht implementation's fingerprint [25]. For several TLS implementations regular inference revealed new security vulnerabilities [7]; the FREAK attack on TLS [26] already showed security flaws caused by flawed implementations of the TLS state machine, which might have be found using regular inference.

Regular inference has been extended using predicate abstraction [27] to consider the influence of data on the control flow. In [18], Merten et al. proposed a systematic method to implement a test harness for LearnLib, including a mapper and a data monitoring part. There has also been research into inference of timed automata [28, 29]. Such techniques might be used to analyse protocol implementations including their timing behaviour and possibly avoid the problem of timing-related nondeterminism that we ran into.

Concerning OpenVPN, Vranken [30] developed fuzzers based on libFuzzer to analyze the OpenVPN implementation and found four important security vulnerabilities.

VII. Conclusion

We presented an automated analysis of two OpenVPN implementations using a technique called protocol state fuzzing, which uses regular inference to infer state machines of the OpenVPN server. This approach is able to find logical flaws in the state machine of implementations, but cannot detect, for instance, flaws caused by malformed messages, such as the recent OpenVPN flaws found using fuzzing [30]. We analyzed the inferred state machines manually, as they are relatively small; for bigger state machines, one could consider using a model checker to formally verify properties, as done in [24].

Our analysis abstracts from some of the finer details of the implementations. First, the state machine is dependent on the test harness which defines the input alphabet and semantics of the messages. Our test harness intentionally conceals the acknowledgement mechanism and the smooth transition of the key renegotiation mechanism. These concessions to 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 cannot be modeled in a simple Mealy machine and therefore must be abstracted. Modeling timing-related events would require a more complex model of timed-automata 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 essentially involves re-implementing an OpenVPN client, able to send correct messages to the server in any order. This is a difficult and time-consuming task for a specific protocol, so it is more worthwhile if the test harness can be reused to analyze many implementations, as has been done for TLS [7, 23] or SSH [24]. In the case of OpenVPN there are not as many implementations but the test harness can be reused to analyze the different versions.

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 then warrant closer analysis as they may introduce security flaws. In a security evaluation, they can be used to harden the implementation by simplifying the state machine, reducing the risk of vulnerabilities. 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 for the implementations of OpenVPN servers did not reveal any vulnerabilities, and they comply to what would be expected from a security protocol. Security-critical errors, such as failures in the TLS handshake and failed integrity checks or decryption of Data messages, always result in a closed connection. The servers also ignore incorrect messages, such as messages with an unknown session-id, KeyNeg messages sent after the session initialization, or Data message with a wrong key-id. So our results increase the confidence in the tested OpenVPN implementations.

It is a shame that the message sequence leading to a successful OpenVPN connection or the correct behavior when receiving unexpected messages is not specified clearer in the OpenVPN documentation. This information could easily be specified by one (or several) protocol state machine such as we inferred. The documentation would really benefit from the addition of such state machines, e.g. the one given in Figure 8 which gives a good overview of the sequence of messages used to establish an OpenVPN session. Alongside such a state machine, A prose specification alongside such a state machine coud then describe the main timing-related events and more details on how to handle error cases such as unexpected or incorrect input messages.

References

[1] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[2] E. Poll, J. de Ruiter, and A. Schubert, "Protocol state machines and session languages: specification, imple-

mentation, and security flaws," in *Security and Privacy Workshops (SPW)*. IEEE, 2015, pp. 125–133.

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

[4] H. Raffelt, B. Steffen, and T. Berg, "LearnLib: A library for automata learning and experimentation," in *Formal methods for industrial Critical Systems (FMICS'05)*. ACM, 2005, pp. 62–71.

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

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

[7] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations." in *USENIX Security Symposium*, 2015, pp. 193–206.

[8] L. C. Paulson, "Inductive analysis of the internet protocol TLS," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 3, pp. 332–351, 1999.

[9] G. Díaz, F. Cuartero, V. Valero, and F. Pelayo, "Automatic verification of the TLS handshake protocol," in *ACM Symposium on Applied computing*. ACM, 2004, pp. 789–794.

[10] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell, "A modular correctness proof of IEEE 802.11i and TLS," in *Computer and Communications Security (CCS'05)*. ACM, 2005, pp. 2–15.

[11] K. Ogata and K. Futatsugi, "Equational approach to formal analysis of TLS," in *International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE, 2005, pp. 795–804.

[12] S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, and J. Schwenk, "Universally composable security analysis of TLS." *ProvSec*, vol. 5324, pp. 313–327, 2008.

[13] P. Morrissey, N. Smart, and B. Warinschi, "A modular security analysis of the TLS handshake protocol," *Advances in Cryptology-ASIACRYPT 2008*, pp. 55–73, 2008.

[14] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 273–293.

[15] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the TLS protocol: A systematic analysis," in *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 429–448.

[16] O. Niese, "An integrated approach to testing complex systems," Ph.D. dissertation, Dortmund University, 2003.

[17] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, vol. SE-4, no. 3, pp. 178–187, 1978.

[18] M. Merten, M. Isberner, F. Howar, B. Steffen, and T. Margaria, "Automated learning setups in automata learning," *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pp. 591–607, 2012.

[19] G. Shu and D. Lee, "Testing security properties of protocol implementations-a machine learning based approach," in *International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE, 2007, pp. 25–25.

[20] J. Narayan, S. K. Shukla, and T. C. Clancy, "A survey of automatic protocol reverse engineering tools," *ACM Computing Surveys*, vol. 48, no. 3, pp. 40:1–40:26, 2016.

[21] F. Aarts, J. de Ruiter, and E. Poll, "Formal models of bank cards for free," in *Software Testing, Verification and Validation Workshops (ICSTW'13)*. IEEE, 2013, pp. 461–468.

[22] F. Aarts, J. Schmaltz, and F. Vaandrager, "Inference and abstraction of the biometric passport," *Leveraging Applications of Formal Methods, Verification, and Validation*, pp. 673–686, 2010.

[23] J. de Ruiter, "A tale of the OpenSSL state machine: A large-scale black-box analysis," in *Nordic Conference on Secure IT Systems*. Springer, 2016, pp. 169–184.

[24] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations," in *SPIN Symposium on Model Checking of Software (SPIN'17)*. ACM, 2017, pp. 142–151.

[25] H. Richter, W. Mostowski, and E. Poll, "Fingerprinting passports," in *NLUUG spring conference on security*, 2008.

[26] A. D.-L. Benjamin Beurdouche, Karthikeyan Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *IEEE Security and Privacy (SP 2015)*. IEEE, 2015, pp. 535–552.

[27] F. Aarts, B. Jonsson, and J. Uijen, "Generating models of infinite-state communication protocols using regular inference with abstraction." *ICTSS*, vol. 6435, pp. 188–204, 2010.

[28] O. Grinchtein, B. Jonsson, and M. Leucker, "Inference of timed transition systems," *Electronic Notes in Theoretical Computer Science*, vol. 138, no. 3, pp. 87–99, 2005.

[29] S. E. Verwer, "Efficient identification of timed automata: Theory and practice," Ph.D. dissertation, TU Delft, Delft University of Technology, 2010.

[30] G. Vranken, "The OpenVPN post-audit bug bonanza," https://guidovranken.wordpress.com/2017/06/21/the-openvpn-post-audit-bug-bonanza/, 2017, accessed: 2017-08-9.