The Closer You Look, The More You Learn

McMahon Stone, Chris; Thomas, Sam L.; Vanhoef, Mathy; Henderson, James; Bailluet, Nicolas; Chothia, Tom

DOI:
10.1145/3548606.3559365

Citation for published version (Harvard):
McMahon Stone, C, Thomas, SL, Vanhoef, M, Henderson, J, Bailluet, N & Chothia, T 2022, The Closer You Look, The More You Learn: A Grey-box Approach to Protocol State Machine Learning. in *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* Proceedings of the ACM Conference on Computer and Communications Security, Association for Computing Machinery , New York, pp. 2265-2278, 28th ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, United States, 7/11/22. https://doi.org/10.1145/3548606.3559365

The Closer You Look, The More You Learn: A Grey-box Approach to Protocol State Machine Learning

Chris McMahon Stone
cxm373@alumni.bham.ac.uk
University of Birmingham
Birmingham, U.K.

Sam L. Thomas
sam@binarly.io
BINARLY
Pasadena, California, USA

Mathy Vanhoef
mathy.vanhoef@kuleuven.be
imec-DistriNet, KU Leuven
Leuven, Belgium

James Henderson
jxh1012@alumni.bham.ac.uk
University of Birmingham
Birmingham, U.K.

Nicolas Bailluet
nicolas.bailluet@ens-rennes.fr
École Normale Supérieure
Rennes, France

Tom Chothia
T.Chothia@bham.ac.uk
University of Birmingham
Birmingham, U.K.

ABSTRACT

We propose a new approach to infer state machine models from protocol implementations. Our new tool, StateInspector, learns protocol states by using novel program analyses to combine observations of run-time memory and I/O. It requires no access to source code and only lightweight execution monitoring of the implementation under test. We demonstrate and evaluate StateInspector's effectiveness on numerous TLS and WPA/2 implementations. In the process, we show StateInspector enables deeper state discovery, increased learning efficiency, and more insight compared to existing approaches. Our method led us to discover several concerning deviations from the standards and vulnerabilities in IWD and WolfSSL, both of which were assigned CVEs.

CCS CONCEPTS

• Security and privacy → Security protocols; **Software reverse engineering**; **• Networks → Protocol correctness**.

KEYWORDS

Protocol Security, State Machine Learning, Reverse Engineering

ACM Reference Format:
Chris McMahon Stone, Sam L. Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet, and Tom Chothia. 2022. The Closer You Look, The More You Learn: A Grey-box Approach to Protocol State Machine Learning. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3548606.3559365

1 INTRODUCTION

Protocol state machine learning has been used to analyse many cryptographic protocols, e.g., TLS [16, 17], SSH [20], WPA [36], Bluetooth [30], DTLS [19] OpenVPN [15], and many others [5, 6, 18, 39]. A state machine shows how an implementation responds to any sequence of inputs it receives. The "happy flow" of a protocol is generally well described in the protocol's specification, however, most specifications do not clearly describe how implementations should handle unexpected, out of order messages, which can lead to errors. Vulnerabilities found via state machine learning include an implementation of TLS where sending a `ChangeCipherSpec` message to a server *before* a `ClientKeyExchange` leads to the server using an insecure default key [17], and an implementation of WPA/2 where an access point skips key authentication after three timeouts [36]. Identifying such bugs in application logic is a vital and complementary analysis to other techniques, like fuzzing, which test for vulnerabilities related to memory corruptions [27, 35].

State machine learning takes a set of input message sequences, sends them to a system under test (SUT) and records the results. Trying all possible combinations of messages is intractable, thus, automated learning systems use optimised algorithms such as L^* [7] or TTT [26] to perform learning. The learned machines are usually Mealy-machine automata [28, 34], whose transitions map from an input to the SUT, to the output received in response, and the resulting state. This kind of learning is purely *black-box* and relies entirely on I/O observations. It assumes that the SUT is deterministic, i.e., that the same sequence of inputs will always lead to the same protocol state. Due to this assumption, to be tractable, black-box learning can only determine if two states that exhibit the same I/O behaviour are equivalent up to a fixed bound. Past algorithms try all possible inputs sequences up to a fixed size, in the worst case this takes exponential time and if the depth is too small, they miss protocol behaviours. Because of this, past work has used small sets of core protocol messages and limited the depth of the search[1] (we show below that this can lead to missed attacks).

We present a, two stage, state machine learning method and supporting tool, StateInspector. In the first stage, we capture and analyse snapshots of the implementation's execution context while it runs the protocol to identify locations in memory that *define* the protocol state. In the second stage, we learn the complete protocol state machine by analysing how the implementation responds to input queries and then inspecting the SUTs memory to see what protocol state it is in. As we can now recognise each protocol state

[1] Looking at SSH, [20] uses between 5 and 9 inputs, saying: "we used only a subset of the most essential inputs, to speed up experiments" and for OpenVPN, [15] uses between 3 and 7 inputs, saying: "In order to keep the learning complexity low, we only include messages that would be accepted given the server configuration". [17] uses 8 core TLS inputs, whereas our methods can test a full set of 21 TLS inputs.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559365

as soon as we reach it, we no longer need to try all possible protocol inputs to a set depth.

As with past approaches, our method makes certain assumptions about the SUT. We discuss these in detail and empirically verify that they are reasonable and realistic for implementations of complex, widely deployed protocols, including TLS and WPA/2. Our evaluation demonstrates that for modestly sized input sets, as used in past work, our method is significantly more efficient. Further, it shows that StateInspector can handle larger, more realistic input sets and greater exploration depths than past work. In fact, we find that such input sets cause these methods—all based on black-box learning—not to terminate. Our experiments show that learning with larger input sets is crucial to finding *deeper* state machine vulnerabilities. To that end, StateInspector enabled us to find vulnerabilities in WolfSSL and IWD, which could not have been found with black-box learning methods. We also found issues of concern in OpenSSL and Hostapd. In summary, we make the following contributions:

(1) A new model inference approach, which leverages observations of run-time memory and program analyses.
(2) A demonstration that our approach runs faster and learns models that black-box approaches cannot.
(3) An evaluation on TLS and WPA implementations, finding two CVE vulnerabilities and several other issues.

We make StateInspector, our test harnesses, our test corpus, and our full results and data sets publicly available[2] [38].

2 BACKGROUND

Model Learning The models we learn take the form of a *Mealy machine*. For protocols, this type of model captures the states of the system which can be arrived at by sequences of inputs from a set I, and trigger corresponding outputs from a set O. We consider Mealy machines that are *complete* and *deterministic*. This means that for each state and input $i \in I$, there is one output and one possible next state. Mealy machine learning algorithms include L^* [7, 28, 34] and the more recent space-optimal TTT algorithm [26]. All of these algorithms work by systematically posing *output queries* from I^+ (i. e., non-empty strings of messages), each preceded by a special *reset query*. More advanced learning algorithms that learn automata with, e.g., side effects [41] or weights [40] have been proposed, but these have not been used on cryptographic protocols.

For a learning algorithm to interact with a SUT, it relies on a "test harness", which is a stand alone program that translate between an abstract and a concrete protocol message representation. For example, a TLS test harness will map the message `ClientKeyExchange` to a full TLS Client Key Exchange message including a encrypted key seed. A test harness will also be responsible for keeping track of key material and nonces, generating keys and performing encryption as needed. For protocols with good open source libraries, e.g., TLS [35], building a test harness is relatively straightforward, but for proprietary protocols, it may amount to implementing specialised clients or servers from scratch.

Protocol state machine learning algorithms pose combinations of inputs to a SUT until they find a hypothesis for the state machine which achieves specific properties, namely *completeness* and *closedness*. They then employ a secondary procedure which poses further *equivalence* queries. These queries attempt to find counterexamples to refine the hypothesis. In black-box learning, these can be generated in various ways, including random testing, or formally complete methods such as Chow's W-Method [12]. The W-Method is computationally expensive and essentially requires an exhaustive search of all input sequences up to some specified bound, with no guarantee of correctness beyond this bound. Therefore, finding an optimal input set and a bound small enough to ensure termination, yet large enough to produce useful models has been a key part of past work on state machine learning for security analysis.

Going from a state machine to an attack The state machine gives a detailed, abstract view of how a protocol implementation works, and it must be analysed to see if there is a flaw in the implementation logic. Cryptographic protocols will normally lead to an accepting state in which encrypted data can be exchanged, this state should be reached by carrying out the steps in protocol as given in the specification. Vulnerabilities can take the form of a non-standard path to this accepting state. It is normal for protocol state machines to have many states to handle errors, some of which may be recoverable from and some many not, however paths that reach the accepting state while bypassing key authentication steps are of particular interest. For instance, [17] found a path to the accepting state for TLS which bypassed the Client Key Exchange, and [36] found a path to the accepting state for a WPA implementation that would bypass a message authentication step if three timeouts in a row occurred.

Binary Program Analysis Our grey-box analyses are performed at the assembly language level (c.f. white-box analyses, where source code information is utilised) and are based on two techniques: taint propagation and symbolic execution. Taint propagation tracks the influence of *tainted* variables by instrumenting code and tracking data at run time. Any new value that is derived from a tainted value will also carry a taint.

Symbolic execution computes constraints over *symbolic* variables based on the operations performed on them. It tracks the relation between variables and for a branch condition that depends on a symbolic variable, it can query a SMT solver to obtain suitable assignments that enable us to explore all feasible branches.

3 MOTIVATION AND RELATED WORK

In Table 1, in addition to StateInspector, we summarise five key approaches whose central aim is to learn a state machine model of a given protocol implementation. Each approach learns this model for different reasons (i. e., fuzzing, reverse-engineering, or conformance testing), and their assumptions also differ (i. e., black or grey-box access, known or unknown target protocols, and active or static trace-based learning).

Limited Coverage Learned model coverage in passive trace-based learning approaches (e. g., [14, 21]) is entirely determined by the quality of pre-collected packet traces, and consequently is most appropriate for learning unknown protocols. On the other hand, since active methods can generate protocol message sequences on-the-fly, their coverage capability is self-determined. All state-of-the-art active methods (black-box [16, 17, 19, 20, 36] and grey-box [11, 31]) use automata learning algorithms (e. g., L* or TTT) for their

[2] https://github.com/ChrisMcMStone/state-inspector

Table 1: Comparison of protocol model learning approaches. Style refers to active generation of inputs sequences, or passive trace replays. Requirements denote: (P)rotocol Implementation, (A)bstraction functions for I/O, (T)races of protocol packets, and (F)uzzer. Learned information categorised as input (I), input to protocol state (I2PS), and input to concrete state (I2CS).

		Goals	Style	Requirements				Crypto-	What is learned?			State-classifier
				P	A	T	F	protocols	I	I2PS	I2CS	
Black-box	Model learning [17, 36]	Model extraction	Active	●	●	○	○	●	○	●	○	I/O
	Pulsar [21]	State-aware fuzzing	Passive	○	○	●	●	○	○	●	○	I/O
Grey-box	Prospex [14]	Protocol RE	Passive	○	●	●	○	○	○	●	○	Control-Flow-I/O
	MACE [11]	Input discovery	Active	○	●	●	○	○	●	●	○	I/O
	AFLnet [31]	State-aware fuzzing	Active	○	●	●	●	◐	●	●	○	I/O
	StateInspector	**Model extraction**	**Active**	●	●	○	○	●	○	●	●	**Memory**

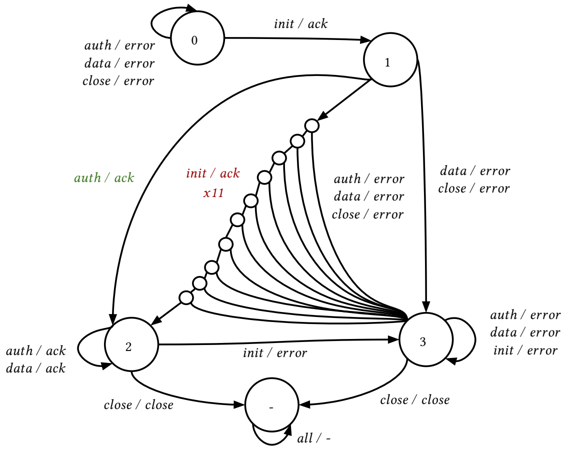


Figure 1: State machine with a deep-state backdoor that can only be activated by sending 12 consecutive *init* messages.

query generation and model construction, and such algorithms are known to have two key limitations. First, queries to a target system are constructed with a fixed, pre-defined set of inputs (i. e., protocol messages). This means that behaviour triggered with inputs outside of this set cannot be explored. To help alleviate this issue, two approaches integrating automata learning with grey-box analyses have been proposed—MACE [11], which uses concolic execution, and AFLnet [31], which uses fuzzing. Despite some success when applied to text-based protocols, these and other existing model learning techniques suffer from a second issue, namely, limited exploration with respect to the *depth* of the state machine.

Consider the state machine in Figure 1. In this simple protocol, a *backdoor* exists that allows us to transition from state 1 (not authenticated) to state 2 (authenticated) by supplying an input of 11 `init` messages, as opposed to an `auth` message[3]. To learn this state machine, state-of-the-art model learning with the TTT-Algorithm [26] and modified W-Method [17] fails to terminate after 1M queries—essentially operating by brute-force. This is because the number of queries needed is polynomial in the number of messages, states and exploration depth. Moreover, to identify this backdoor, we must set a maximum exploration depth of at least 11. In practice, however, since query complexity explodes with a high bound, most works use a much lower bound (e. g., 2 in [17, 20]), or use random testing which lacks completeness. This query explosion is also exacerbated by large input sets. Hence, for systems where time-spent-per-query is noticeable (e. g., a few seconds, as is the case with some protocols), it is apparent that learning such systems requires a more efficient approach.

Limited Optimisations Similar to the problem of limited coverage, current approaches also lack sufficient information to optimise learning. That is, I/O observations do not enable learning effort to be focused where it is likely to be most fruitful. Consider de Ruiter et al.'s application of black-box learning to RSA-BSAFE for C TLS [17]. Their learner cannot detect that the server does not close the connection upon handshake termination, and, therefore, exhausts its exploration bound, posing tens of thousands of superfluous queries.

Prospex's grey-box approach [14] potentially avoids this issue by using execution traces of message processing code to (indirectly) help classify states. However, it cannot capture states that only manifest as changes to memory, e. g., counter increments per message read, and its coverage is limited due to the use of trace-replays. Consequently, like MACE [11] and AFLnet [31], it is not applicable to protocols with complex state or replay-protection (i. e., security protocols).

Limited Insight Methods that use only I/O observations may over-abstract the true state machine to such a degree that their output model prevents practitioners from obtaining a sufficiently detailed understanding of an implementation. Further, as active learning methods produce models that depend on how they map between concrete and abstract messages, their learned models will be incorrect if the mapping is incorrect, which can be challenging to identify. This mapping is done using a so-called test harness, which is a highly flexible implementation of the target protocol (denoted by **P** in Table 1). A harnesses' main function is to send and interpret protocol messages in an arbitrary order.

This difficulty is highlighted in the black-box analysis of TLS performed by de Ruiter et al. [17]. The authors' acknowledge that their state machines for OpenSSL 1.0.1g and 1.0.1j are incorrect due to differing assumptions between the implementation and their test harness (page 11, Section 4.8). Unfortunately, determining whether a model is incorrect due to a flawed test harness, or whether there is a bug in the SUT is difficult based on a model learned using just I/O. In general, we must manually analyse both the harness and implementation to determine the root cause. Conversely, our method provides insight into how state defining memory correlates with I/O behaviour, and can therefore be used to detect such bugs

[3] This backdoor is similar to a flaw found in a WPA/2 implementation [36], which permitted a cipher downgrade after 3 failed steps of the handshake.

with significantly less effort. To demonstrate this, we provide an analysis of the aforementioned flaw in Section 6 and Appendix F.

Some recent work has extended learning with taint analysis to learn register automata from source code [25, 33]. Such automata are more insightful than Mealy machines, since they attempt to model conditions on state data for transitions between states. Also working on source code, Hoque et al. [24] have used symbolic execution to learn a state machine. In contrast, our method works from the protocol binary alone. A different approach is taken by Pacheco et al. [29] who learn state machines from the text of the specification document, which obviously cannot find implementational vulnerabilities.

Research Questions Based on the above discussion we form the following research questions to guide the design and and provide an evaluation criteria for our approach, StateInspector:

RQ1: Can we identify the protocol state of the SUT by analysis of the memory?

RQ2: Can we combine this memory analysis with automata learning methods to learn state machines more efficiently?

RQ3: Does this provide more insight into the SUTs behaviour, letting us use larger input sets, and finding deeper states and behaviours that past learning methods cannot?

4 OVERVIEW

We depict a high-level overview of our approach in Figure 2. Our method combines insights into the runtime behaviour of the SUT (provided by the execution monitor and concolic analyser) with observations of its I/O behaviour (provided by the test harness) to learn models that provide comparable guarantees to traditional black-box approaches. We now state the assumptions we make and provide an overview of our tools operation. Our first assumption is

Assumption 1. *The protocol state machine of the SUT is finite, and can be represented by a Mealy machine.*

Justification: This is a standard assumption for all state machine learning methods, without this the learning process cannot terminate. However we show with our Hostap 2.8 case study in Section 6, that if the state machine is infinite then the process can be stopped at any time to return a correct sub-automaton of the infinite state machine, which can still be useful for analysis.

In all concrete implementations, each state of this Mealy machine will correspond to a particular assignment of values to specific "state-defining" memory locations, allocated by the implementation. We use two terms to discuss these locations and the values assigned to them, which we define as follows:

Definition 1 (Candidate State Memory Location). *Any memory location that takes the same value after the same inputs, for any run of the protocol.*

Definition 2 (A Set of State-Defining Memory). *A minimal subset of candidate state memory locations whose values during a protocol run uniquely determine the current state.*

We refer to a member of any state-defining memory set as *state memory.* Next we assume that we know the input language needed to interact with the SUT and hence generate the state machine:

Assumption 2. *All protocol states can be reached via queries built up from the inputs known to our testing harness I^+.*

Justification: This is a standard assumption of black-box state machine learning. Our method is best for testing implementations of well known protocols. If some inputs are missing from our input alphabet then we will learn a correct sub-automaton of the state machine that does not include the missing inputs.

Under these assumptions, we now describe the operation of our approach. Our learner uses a test harness to interact with the SUT. This test harness is specific to each protocol analysed and tracks session state, and is responsible for communicating with the SUT and translating abstract representations of input and output messages (for the learner) to concrete representations (for the SUT). In the first phase of learning, the learner instructs the test harness to interact with the SUT to exercise normal protocol runs, trigger errors, and induce timeouts multiple times. We capture snapshots of the SUT's execution context (i. e., its memory) for each of these runs using an execution monitor. We perform subsequent analysis of the snapshots in order to determine a set of candidate memory locations that can be used to reason about which state the SUT is in. We make the following assumption about this memory:

Assumption 3. *A set of state-defining memory, for all states, is allocated on the heap and used along a normal protocol run, during error handling, or timeout routines.*

Justification: We have examined implementations of TLS, WPA, EAP protocols [1, 2], SSH [3] and OpenVPN [4] and found that storing state memory on the heap is standard. If state memory was stored on the stack, or via the program counter, our current implementation would wrongly identify different states as the same, and then terminate issuing a warning when different behaviour was detected. We discuss how our method could be expanded to handle state on the stack or via the program counter in Section 6.5. We may miss candidate locations if they do not change during any known bootstrap flow. Fortunately, our results and analysis of implementation code indicate that this assumption holds for most protocols

Any memory location found to have the same value for each sequence of inputs, for each run, is considered candidate state memory. Any location that takes the same value in all states is discarded. Assumption 3 ensures that this reduced set of locations will be a set of state-defining memory, however, this set may also contain superfluous locations.

The next phase of learning uses the candidate set to construct a model of the protocol state machine. During this process, we identify the set of state-defining memory for each state. Changes in a location, such as a message counter, that tracks the state but has no effect on it, could lead to our method wrongly thinking it has discovered new states. To merge such states we use taint analysis and symbolic execution to find out which candidate locations are not used in conditionals and do not lead to writes to other candidate state definition locations. This allows us to determine if two states arrived at from the same input sequence should be considered equal, even if their memory differs, as we only need to consider the values of state-defining locations when performing this so-called equivalence check. It also allows us to effectively ignore any

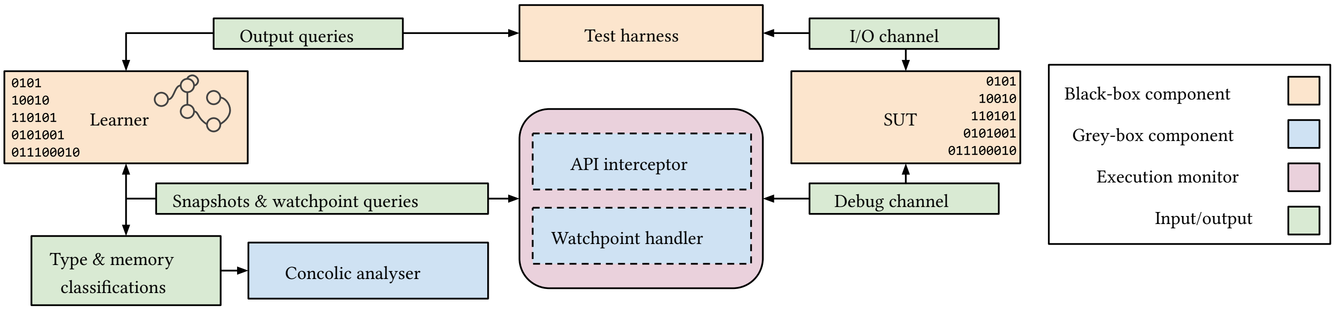


Figure 2: Grey-box State Learning Architecture.

superfluous locations in our candidate set since they will not be considered state-defining for any state.

To learn the state machine, we queue all queries from I^+ to fulfill the *completeness* property for states in a Mealy machine. Namely, that all inputs are defined for each state. We perform these queries iteratively in increasing length. As in the first phase, we take snapshots of the execution state of the SUT on each I/O action. Each time a distinct state, according to its I/O behaviour and defined state memory, is discovered, we queue additional queries from this state in order to determine all states reachable from it. We continue building the automaton in this way until no new states are found.

As well as state memory, it is possible that our candidate set contains locations with assignments that appear to be state-defining for all states identified from the set of posed inputs, but are actually not. For instance, if a protocol implementation maintains a counter of the messages received, it might be mistaken for state memory, leading to a self loop in the state machine being conflated with an infinite progression of seemingly different states. Recognising such memory will prevent our framework from mistaking a looping state from an infinite series of states, and so ensures termination. To determine if such memory exists and if it can be ignored, we perform a *merge check*. This allows us to replace a series of states repeated in a loop, with a single state and a self loop. This check is only performed between states that have the same observed input and output behaviour, and are connected at some depth (i. e., one is reachable from the other). We make the following assumption about this depth:

Assumption 4. *The depth that we check for possible merge states is larger than the length of any loop in the state machine of the implementation being tested.*

Justification: If loops within a state machine exist beyond the configured depth bound, and the memory across these states is always changing (e. g., a counter), then StateInspector will duplicate states. In both of these cases, like black-box methods, we provide a means to bound exploration. We choose to bound by a configurable time, but this can also be done by exploration depth. In the worst case, the resulting models will be no less representative than those learned with black-box methods.

When performing a merge check, we consider two states equal if the values assigned to their state-defining locations are equal. Therefore, for each location that differs, we must determine if it behaves as state memory for each state (we discuss the conditions for this in Section 5.4). To do so, we use a novel analysis (implemented in the concolic analyser in Figure 2). This analysis identifies a memory location as state-defining when this location influences decisions about which state we are in. Concretely, this is encoded by checking if the location influences a branch that leads to a write to any candidate state memory. That is, state-defining memory will control the state by directly controlling writes to (other) state memory, and non-state-defining memory will not. This leads to our final assumption:

Assumption 5. *Any state-defining memory will control a branch, along an execution path triggered by messages from the set I^+, and it will lead to a write to candidate state memory within an a priori determined number of instructions.*

Justification: This restates the definition of state memory by capturing how such memory is used by a program to control which inputs lead to which outputs and new states. We discuss the selection of the above bound for the number of instructions ths might take in Section 5.4.2, and Section 6.5. but note that this is a configurable parameter of our algorithm.

We complete learning when no further states can be merged, and no new states (that differ in I/O behaviour or state memory assignment) can be discovered by further input queries. At this point, StateInspector outputs a representative Mealy machine for the SUT. We note that if any of these assumptions do not hold we can still approximate the state machine of the SUT by imposing a time bound on the learning, this is discuss further in Section 6.

5 METHODOLOGY

The learner component of Figure 2 orchestrates our model inference process, as implemented by StateInspector. It learns how the SUT interacts with the world by observing its I/O behaviour via the test harness, and learns how it performs state-defining actions at the level of executed instructions and memory reads and writes, via the execution monitor (Section 5.1). Inference begins with a series of bootstrap queries in order to identify candidate state memory, i. e., M (Sections 5.2, 5.3). Once completed, the model refinement proceeds with the aim of identifying each state by its I/O behaviour and set of state-defining memory.

This core model refinement algorithm works in an iterative loop. It starts by taking the outputs from the bootstrap phase, ❶ an estimation of M and ❷, an initial set of identified states, and then queries input sequences of incrementing lengths n. Queries are formed by taking each input from our alphabet and appending it to the input sequence needed reach each existing state. All queries of length n are then posed to the SUT, and states are identified from their memory contents at the locations defined in M. Using this information the model the model is updated, and a state merging check (Section 5.5) is performed complemented by a process to handle uncertain memory in M (Section 5.4). With this iteration of model refinement now complete, n is incremented and the process repeats until model completeness is achieved.

We show a full sketch of our algorithm in Appendix A.

5.1 Monitoring Execution State

To monitor a SUT's state while it performs protocol related I/O, we take snapshots of its memory and registers at carefully chosen points. As many protocols are time-sensitive:

(1) We avoid inducing overheads such that the cost of each query becomes prohibitively high.
(2) We do not interfere with the protocol execution, by, for example, triggering timeouts.
(3) We only snapshot at points that enable us to capture each input's effect on the SUT's state.

To perform a snapshot, we momentarily pause the SUT's execution and copy the value of its registers and contents of its mapped segments to a buffer controlled by the execution monitor. To minimise overheads, we buffer all snapshots in memory, and only flush them to disk after the SUT has finished processing a full query. Whilst snapshots do cause a worst-case overhead of 20% time-per-query, this does not impact the behaviour of any protocols analysed. Regarding disk requirements, snapshots were at most 500kb each. For large state machines which required 1000s of queries, we actively deleted snapshots during learning to optimise disk space usage.

To identify when to perform snapshots, we infer which system calls (syscalls) and fixed parameters a SUT uses to communicate with its client/server. We do this by generating a log of all system calls and passed parameters used during a standard protocol run. We then identify syscall patterns in the log and match them to provided inputs and observed outputs. The number of syscalls for performing I/O is small, hence the number of patterns we need to search for is also small, e. g., combinations of `bind`, `recvmsg`, `recvfrom`, `send`, etc. When subsequently monitoring the SUT, we hook the identified syscalls and perform state snapshots when they are called with parameters matching those in our log.

5.1.1 *Mapping I/O Sequences To Snapshots.*

To map I/O sequences to snapshots, we maintain a monotonic clock that is shared between our test harness (which logs I/O events) and our execution monitor (which logs snapshot events). We construct a mapping by *aligning* the logged events from each component, as depicted in Figure 3. When doing so, we face two key difficulties. First, implementations may update state memory before or after responding to an input, hence we must ensure we take snapshots to capture both possibilities (i. e., case ❸ in Figure 3). Second, some parts of a query may not trigger a response, hence we must account for the absence of *write*-like syscalls triggering a snapshot for some queries (i. e., cases ❷ and ❹).

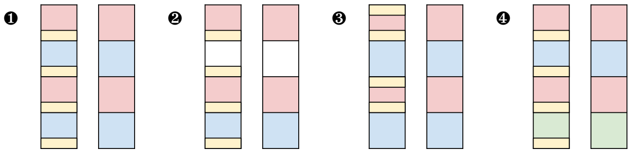


Figure 3: Mapping I/O sequences to snapshots. We depict four cases ❶-❹, snapshot events are shown on the left and I/O on the right. We indicate read/input events in red, write/output events in blue, state changes in yellow, and connection close events in green. ❶ shows the case where snapshots and state changes are trivially aligned with I/O. ❷ shows the case where a state change occurs without a corresponding output event. ❸ shows the case where state changes occur after output events. ❹ shows the case where a state change occurs after the last input event, with no output before the connection is closed.

For all scenarios, we trigger *input* event snapshots after *read*-like syscalls return, and *output* event snapshots just before *write*-like syscalls execute. We take additional snapshots for *output* events before each *read*-like syscall. This enables us to always capture state changes, even if no corresponding *write*-like syscall occurs, or the state change happens after the output is observed.

If there is no output to the final input of a query q, then we instruct the learner to pose an additional query with an extra input, $q||i$ for all $i \in I$. We expect there to be a *read* event corresponding to the final input of one of these queries—thus providing us with a snapshot of the state after processing the penultimate input. If the SUT does not perform such a read, i. e., it stops processing new input, as in case ❹, we detect this by intercepting syscalls related to socket closure, and inform the learner that exploration beyond this point is unnecessary.

5.2 Identifying Candidate State Memory

Snapshot Generation Our first goal is to learn a set of candidate state memory (Definition 1) M. To form an initial approximation of M, we perform bootstrap queries against the SUT, this produces a set of memory snapshots where all of the memory which tracks state is defined and used (Assumption 3). Our bootstrap queries take the form, $BF = \{b_0, b_1, ...b_n\}$, where $b_i \in I^+$. The first of these queries b_0 is set as the *happy flow* of the protocol. This is the expected normal flow of the protocol execution, which we assume prior knowledge of. For example, in TLS 1.2, $b_0 = (ClientHelloRSA, ClientKeyExchange, ChangeCipherSpec, Finished)$. The other queries in BF are specific mutations of this happy flow, automatically constructed with the intention of activating error states, timeout states, and retransmission behaviour. Each of these queries b_x is derived by taking all prefixes p_x of the happy flow b_0, where $0 < |p_x| \leq |b_0|$, and for all inputs in I, and appending to p_x each $i \in I$ a fixed number of times T such that $b_x = p_x||i^T$.

Every bootstrap query is executed at least twice, so that we have at least two snapshots for each equivalent input sequence, which we require for the next step of our analysis. We can reuse all of the bootstrap queries for refining our model in subsequent phases. When possible, we also attempt to alternate functionally equivalent

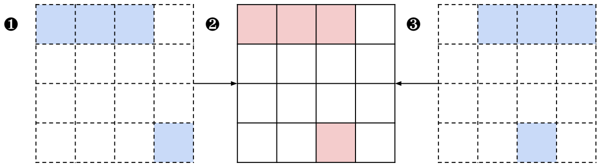


Figure 4: Allocation alignment across different executions. Each large block represents the memory layout of a different protocol run; coloured squares represent state-defining locations. Allocation alignment computes a mapping of allocations of a given run (blue squares) to a *base* configuration (red squares). Using this mapping, we can *diff* snapshots that have different configurations, e. g., ❶ and ❸, by first mapping them onto a common configuration, i. e., ❷.

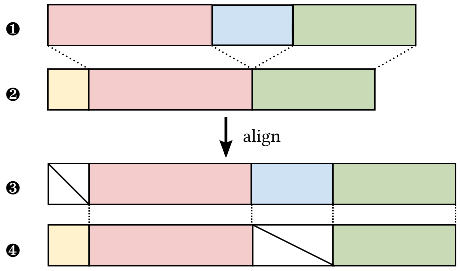


Figure 5: Logs alignment using Gotoh's algorithm. ❶ and ❷ represent two logs to align; coloured rectangles represent their longest common sub-sequences of allocations. ❸ and ❹ represent the aligned versions of ❶ and ❷; crossed out rectangles represent gaps inserted for unmatched subsequences of allocations.

input parameters to the SUT across bootstrap flows. We do this to maximise the potential number of locations we eliminate due to not holding the same value at equivalent states (Assumption 3). For example, for TLS implementations, we execute some bootstrap flows with different server certificates, which enables us to eliminate memory that would otherwise have to be identified as non-state-defining using our concolic analyser, described below.

Handling Dynamic Memory Protocols will generally allocate state memory on-demand, using e.g., `malloc`. This presents a challenge when identifying state-defining locations, as logically equivalent allocations may not reside at the same offsets across different protocol executions. To address this non-determinism, we compute a mapping of each execution's allocations to a single *base* configuration. This enables us to analyse all snapshots together with respect to a common configuration, rather than pair-wise. Figure 4 visualises our approach.

We first construct an allocation log for each execution that records a timestamp, the stack pointer (i. e., callee return address), and call parameters, of every call to a memory allocation function (e. g., `malloc` and `free`). Then, to derive a mapping between two logs, we use Gotoh's algorithm [22] to find a *global alignment* of the two sequences of allocations, as shown in Figure 5. As Gotoh's algorithm is designed to match biological sequences rather than allocation sequences, we make adaptations to the input parameters of the algorithm to make it suitable for alignment of allocation logs. We match the allocations on their size and calling context (recorded stack pointer)—the set of all $(size, context)$ pairs define the input alphabet of the algorithm (in contrast to $\{A, T, C, G\}$ for biological applications). We also set the input scores in such a way as to avoid inserting unnecessary gaps in the alignment and as to prevent mismatches (otherwise allowed by the algorithm)—we do not want to align two allocations whose sizes or calling contexts are different (non-equivalent across executions). We fully explain our adaptations and choice of parameters in Appendix C.

We choose our *base* configuration, or log, as the largest *happy flow* log, under the assumption that it will contain all allocations related to state-defining locations for any possible session.

Snapshot Diffing Following mapping each bootstrap snapshot's allocations onto the *base* log, we *diff* them to obtain our candidate set M. Each element $m \in M$ corresponds to the location of a contiguous range of bytes of dynamically allocated memory, which we represent by: an allocation address, an offset relative to the start of the allocation, and a size.

We perform diffing by first grouping all snapshots by their associated I/O sequences. Then, for each group, we locate equivalent allocations across snapshots and identify allocation-offset pairs which refer to byte-sized locations with the same value. We then check that every identified location also contains the same value in every other I/O equivalent snapshot, and is a non-default value in at least one snapshot group (Assumption 3). This gives us a set of candidate state memory locations. We note that as this process is carried out at the *byte* level, we additionally record all bytes that do not abide by this assumption. This enables us to remove any incorrectly classified bytes once we have established the real bounds of individual locations (Appendix D).

5.3 Minimising Candidate State Memory

Given our initial set of candidate state memory locations, we reduce M further by applying the following operations:

(1) **Pointer removal:** we eliminate any memory containing pointers to other locations in memory. We do this by excluding values that fall within the address-space of the SUT and its linked libraries.
(2) **Static allocation elimination:** we remove any full allocations of memory that are assigned a single value in the first snapshot which does not change throughout the course of our bootstrap flows.
(3) **Static buffer elimination:** we remove any contiguous byte ranges larger than 32 bytes that remain static.

Operations 2 and 3 are used to filter locations corresponding to large buffers of non-state-defining memory, for example, in OpenSSL, they eliminate locations storing the TLS certificate.

Type-Based Minimisation Since state-defining locations are often only meaningful when considered within the address bounds of their full type e. g., four consecutive bytes as a 32-bit integer, throughout learning we additionally run a rudimentary type inference algorithm procedure. This is described in Appendix D.

5.4 Handling Uncertain State Memory

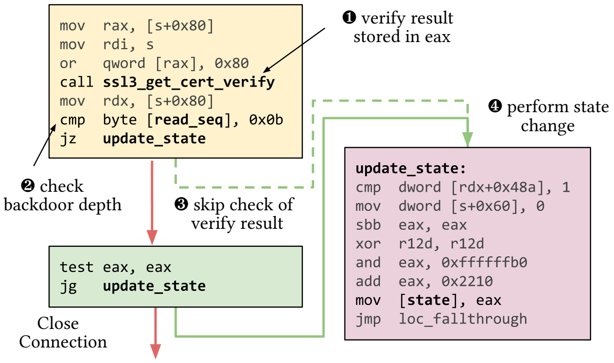
Forming our candidate set M by computing the differences between snapshots gives us an over-approximation of all of the locations used to discern the SUT's state. However, an implementation may not use all of these locations to decide which state it is in for every state. When testing if two memory configurations for the same I/O behaviour should be considered equivalent, we must identify if differing values at candidate locations imply that the configurations correspond to different states (i. e., the differing locations are state-defining), or if the values they take are not meaningful for the particular state we are checking. For a given state, by analysing how locations actually influence execution we can tell state-defining locations from those that are not, as in all cases, non-state-defining locations will have no influence on how state is updated. Since this kind of analysis is expensive, the diffing phase is crucial in minimising the number of candidate locations to analyse—typically reducing the number to tens, rather than thousands.

5.4.1 Properties Of State-Defining Memory. To confirm a given location is state-defining, we attempt to capture execution traces of its location behaving as state-defining memory. We summarise these behaviours below, which we base on our analysis of various implementations:

(1) **Control-dependence:** writes to state memory are control-dependent on reads of state memory, e.g., a state `enum` flag read forming the basis of a decision for which code should process an incoming message, and the resulting state machine transition defined by a write to state memory.
(2) **Data-dependence:** non-state memory that is conditionally written to due to dependence on a read from state memory may later influence a write to state memory.
(3) **State-defining and state-influential locations:** for a particular state, its state-defining memory locations (Definition 2) will always be read from before they are written to (to tell which state one is in), and subsequent writes to state memory will be control- or data-dependent upon the values of those reads (Assumption 5). For state-influencing locations, e. g., input buffers, this property will not hold—while the contents of a buffer may *influence* state, it will not directly *define* it, and will be written to before being read.

5.4.2 Discerning State-Defining Locations. If a location holds more than one value when performing a merge check, we apply an additional analysis to determine if it is state-defining. First, we identify execution paths that are control-dependent on the value stored at the location. Then, we check if any of those paths induce a write to known state memory. If so, we classify the location as state-defining.

We base our analysis on a variation of byte-level taint propagation combined with concolic execution. We apply it in sequence to state snapshots taken on reads of the candidate location *addr*. We start analysis from the instruction *pc* performing the read that triggered the state snapshot. Our analysis proceeds by tainting and symbolising *addr*, then tracing forwards until we reach a branch whose condition is tainted by *addr*. If we do not reach a tainted conditional statement within W instructions, we do not continue analysing the path. At the conditional, we compute two assignments for the value at *addr*—one for each branch destination. We then symbolically explore each of the two paths until we have processed W instructions or we reach a return instruction. If we observe a write to known state memory on either path, we consider *addr* to be state-defining.



```
mov  rax, [s+0x80]
mov  rdi, s
or   qword [rax], 0x80
call ssl3_get_cert_verify
mov  rdx, [s+0x80]
cmp  byte [read_seq], 0x0b
jz   update_state
```

```
test eax, eax
jg   update_state
```

```
update_state:
cmp  dword [rdx+0x48a], 1
mov  dword [s+0x60], 0
sbb  eax, eax
xor  r12d, r12d
and  eax, 0xffffffb0
add  eax, 0x2210
mov  [state], eax
jmp  loc_fallthrough
```

Figure 6: OpenSSL client verification bypass. At ❶ we check the client certificate signature, at ❷ we check if the `read_seq` counter is equal to 11, if so, we bypass the check of the result of client certification verification via ❸. At ❹ we perform a state update that is control-dependent on `read_seq`'s value. StateInspector identifies the dependence and dynamically adjusts the learning depth to discover the deep-state change.

Figure 6 depicts an example location that requires such analysis. The counter `read_seq` is used to implement a sneaky backdoor that bypasses client certificate verification in OpenSSL. Our analysis finds that `read_seq` taints the branch `jz update_state`, and thus, leads to a state change. We therefore classify it as state-defining. We select an analysis bound of $W = 512$ based on the observation that reads from memory involved in comparisons tend to have strong locality to the branches they influence. Since we analyse multiple state snapshots for each *addr*, generated for every possible input in I, we reduce the chance for missing locations used in state-defining ways outside of our bound.

5.5 State Merging

We use the values of our candidate state memory to identify when two different sequences of inputs resulted in the same protocol state. However, if there is superfluous memory in our candidate set, such as, e.g., a message counter, this might stop us from identifying states as being the same, and in the worst case lead to non-termination. Therefore, we introduce a *merge check* that attempts to merge states when all queries in the queue of a given length have been performed.

This check operates by identifying pairs of *base* and *merge* states, which can be merged into a single state. These pairs are selected such that two properties hold: ❶ the *merge* state must have *I/O equivalence* to the *base*, and ❷ the *merge* state should be reachable from the *base*. I/O equivalence signifies all input-output pairs are equal for the two states, to a depth D. The intuition is that if I/O differences have not manifested in D steps, they may in fact be the same state, and we should therefore check if the differences in their memory are state relevant. The base-to-merge state reachability must also be possible in D steps (Assumption 4). We enforce this property based on the observation that in security protocol implementations, state duplication is more likely with states along the same *path*. So-called *loopback* transitions often occur where inputs are effectively ignored with respect to the state, however their processing can still result in some changes to memory, resulting in duplication for our memory classified states. Loops in models between multiple states are less common, but will be checked by our learner provided the number of states involved is less than or equal to D.

Our merge check determines whether the memory being used to distinguishes the states is state-defining. Algorithm 2 (see Appendix) shows a sketch of our approach. In summary, for each location of differing memory, and for each input message (to ensure completeness), we monitor the execution of the SUT. We supply it with input messages to force it into the state we wish to analyse we then begin to monitor memory reads to the tracked memory location. We follow this by supplying the SUT with each input in turn and perform context snapshots on the reads to the memory location, which we call a *watchpoint hit*. We then supply each of these snapshots as inputs to our concolic analyser (detailed in the next section), which determines if the SUT uses the value as state-defining memory (Assumption 5). If our analysis identifies any location as state defining, then we do not perform a merge of the two states. Conversely, if all tested locations are reported as not state-defining, then we can merge the *merge* state into the *base* state, and replace the transition with a self loop.

5.6 Implementation

Our resulting implementation consists of a Java-based learner, a ptrace-based [8] execution monitor, and test harnesses in Python and Java. Our concolic analyser is built using Triton [32] and IDA Pro [23]. In total, StateInspector consists of over 10kloc across three different languages. We provide a complete breakdown of the code in Appendix E.

6 EVALUATION

We tested StateInspector against implementations of two of the most widely used cryptographic protocol—TLS and WPA/2. Our results are then compared against *the* state-of-the-art, model learning focused technique from Table 1—black-box model learning. Other works in this Table are either 1) underpinned by this same black-box learning algorithm but lack support for cryptographic protocols (MACE [11] and AFLnet [31]), or 2) are limited to core state and transition modelling due to their non-active nature (Pulsar [21] and Prospex [14]).

All of our test harness code and protocol state machine diagrams are available in our artefact submission [38]. For each state machine we verified a sample of the paths against the SUTs and we ensured that the protocol paths as described in the specification were possible in the state machine. In all cases the logic of the protocol included the correct specified protocol flow and the state machines corresponded with the observable behaviour of the SUT.

6.1 TLS

We evaluated our tool on the TLS 1.2 server implementation of OpenSSL, RSA-BSAFE-C and GnuTLS, and both the server and client implementations of Hostap's internal TLS library. We learn these models with two separate input sets, one containing only the core TLS messages (as in de Ruiter et al. [17]), and another with a much larger input set, including client and server authentication messages and multiple key exchanges (as provided by the TLS-Attacker test harness [35]). Table 2 lists the time taken to learn all models, as well as the identified *candidate state memory* details, query statistics and number of states identified. We list the number of queries required to learn the same models with state-of-the-art black-box learning, i. e., the TTT algorithm [26] with the modified W-Method equivalence checking of [17] with a conservative equivalence checking depth of 3. We cut off black-box experiments after 3 days if they show no sign of approaching termination.

6.1.1 *WolfSSL.*

When testing WolfSSL TLS server v4.8 with StateInspector and our extended input alphabet, we obtained the state machine in Figure 11 (appendix). The expected "happy flow", marked in green, goes from s0, s2,...,s7; s7 is the *accept state* that allows us to send encrypted application data, indicating the protocol ran successfully. Most unexpected messages lead to a single error state s1. However, we see that the server accepts all of the messages normally sent by a server (the red transitions from s3, s4, s5 & s6 to s2), which can also then lead to the *accept state*.

We would expect server messages sent to a server to be ignored (a self loop) or result in a transition to an error state. However, we noticed that these messages transition to a new state that can reach the *accepting* state. That is, a client can send a `ServerHello` message to the server and set the server nonce to a value of its choosing. Thus, a client can completely control the encryption key. Unfortunately, this does not directly lead to a MITM attack, as inserting a message into a TLS session results in a transcript mismatch and termination of the protocol run.

We found similar behaviour in the WolfSSL client: it accepts and processes all client messages apart from `ClientKeyExchange`. Since StateInspector can map from uses of state memory to source-code locations, we could quickly determine that the client performs a check that explicitly rejects `ClientKeyExchange`—annotated with a comment saying it needs to be stopped for "security". Investigating further, we found that if the client was compiled with support for the PSK cipher-suite, we could send a `ClientHello` to the client, it would accept the message, and (incorrectly) set a flag that marks it as a server. As the flag is set after sending a `ChangeCipherSpec` message, the client can be tricked into believing the protocol has finished, which forces it to loop forever, and, thus, leads to a DoS attack. We reported the vulnerability (CVE-2021-44718) to the WolfSSL authors and they released a fix in v5.1. We note that traditional black-box learning does not terminate for the client or server (we stop after 3 days), and therefore would not have allowed us find these issues. In contrast, StateInspector took less than an hour.

6.1.2 *OpenSSL.*

We tested OpenSSL versions ranging from 2014 to 2020. Of particular interest is the OpenSSL server running version 1.0.1g. Our state machine for this implementation found a previously discovered [17] critical vulnerability: specifically, if a `ChangeCipherSpec` (`CCS`) message was sent before the `ClientKey Exchange`, the server would compute the session keys using an empty master secret. This vulnerability shows up clearly in the

Table 2: Model learning results on TLS 1.2 and WPA/2 servers. TLS experiments are repeated with two alphabets, core and extended. Times specified in hours and minutes (hh:mm). Experiments which did not terminate: ■.

	Protocol	Implementation	Classifying Mem.		Mem.	I/O	Total	I/O Mem.	Watchpoints		Total Time	Black-Box	
			Locations	Allocations	States	States	Queries	Queries	Queries	Hits		Queries	Time
Core Alpha.	TLS 1.2	RSA-BSAFE-C 4.0.4	88	14	11	9	128	109	19	4	00:06	204k	■
	TLS 1.2	Hostap TLS	159	32	11	6	194	160	34	17	00:24	971	01:48
	TLS 1.2	OpenSSL 1.0.1g	126	16	19	12	488	280	208	116	00:18	5819	00:42
	TLS 1.2	OpenSSL 1.0.1j	125	16	13	9	291	165	126	23	00:10	1826	00:13
	TLS 1.2	OpenSSL 1.1.1g	126	6	6	6	88	86	2	0	00:02	175	00:02
	TLS 1.2	GnuTLS 3.3.12	172	7	16	7	265	129	36	42	00:10	1353	00:21
	TLS 1.2	GnuTLS 3.6.14	138	7	18	8	973	452	522	121	00:36	3221	00:52
	TLS 1.2	WolfSSL 4.8	163	33	6	6	126	90	30	0	00:07	320	00:12
Ext. Alpha.	TLS 1.2	RSA-BSAFE-C 4.0.4	165	7	14	7	918	688	230	635	01:08	133k	■
	TLS 1.2	OpenSSL 1.1.1g	467	66	11	11	535	524	11	0	01:05	1776	01:25
	TLS 1.2	OpenSSL 1.0.1g	554	110	40	20	2454	1860	594	609	04:21	20898	21:05
	TLS 1.2	GnuTLS 3.3.12	185	5	23	19	1427	1274	153	128	02:05	66k	■
	TLS 1.2	GnuTLS 3.6.14	157	6	13	11	730	673	57	37	01:32	66k	■
	TLS 1.2	WolfSSL 4.8	173	33	9	9	479	449	30	0	01:05	73671	■
	WPA/2	Hostap 2.8†	138	3	24	6	1629	804	825	261	03:30	2127	03:30
	WPA/2	IWD 1.6	135	8	5	5	264	126	138	597	01:02	3000+	■

† For Hostap, we stopped both learners at ~200 minutes, as we found that its state machine is infinite and would prevent both approaches terminating.

state machine as a path to the exchange of encrypted data state, that does not include the ClientKeyExchange message.

The state machine we learnt was different from the one found in [17] using black box learning, we found that there was an error in the test harness used in [17]: when two ClientHello messages are send by the test harness, the test harness will use the first nonce to generate the key whereas the server will use the second. As the test harness cannot then communicate with the server it wrongly believes that the server is in an error state (this error was acknowledged in [17]). Our direct memory analysis methods are able to tell that the resulting state is not an error state, and so produce the correct state machine. Appendix F shows the state machine and provides a more detailed discussion of this issue.

When testing client authentication with our extended alphabet, using the -Verify flag, we found that a client would halt on a invalid certificate but the server would not. This shows up in the state machine as a path to the accepting state of the server using an invalid certificate input. Searching the documentation we found an additional flag -verify_return_error that must be used if we want OpenSSL to halt when it receives an invalid client certificate. A recent Github issue[4] describes the same misleading configuration, and resulted in a patch to the OpenSSL client implementation[5]. This arguably makes matters worse as now the server and client behaviour is inconsistent. We have reported this issue to OpenSSL and they have added fixing it as a post version 3.0.0 milestone.

6.1.3 *RSA-BSAFE-C.*

We selected this implementation as it serves to demonstrate that our method is applicable not just to open-source implementations, but also to closed-source.

From our experiments, we found that each time the server sends an Alert message it performs a partial socket shutdown. Specifically, it calls shutdown(n, SHUT_RD) on the connection with the test harness. This means that it no longer processes inputs, however, this is not detectable by the test harness. For our approach this is not a problem; we detect that no progress can be made using our execution monitor (by hooking shutdown) and prevent further inputs from this point. For black-box learning, which is entirely dependent on the test harness, the socket closure is not detected, and so learning continues exploring beyond the receipt of an Alert message. As shown in Table 2, this leads to many superfluous queries, and, as a result, the learner fails to terminate within 3 days for either alphabet. In comparison, our algorithm is able to learn the same models with 128 queries in 6 minutes for the core TLS functionality, and in ~1 hour for a more complex model capturing client authentication and alternative key exchanges. Notably, this latter test revealed that repeated ClientHello's are only permitted when the server is configured with forced client authentication.

6.1.4 *GnuTLS.*

Our tests on the TLS server implementations in GnuTLS 3.3.12 and 3.6.14 showed substantial changes between the two versions. In particular, each version required us to hook different syscalls for snapshotting. State count also differed, especially so when testing with the extended alphabet. Analysis of the models revealed slightly different handling of the Diffie-Hellman key exchanges, which in the older version resulted in a path of states separate from RSA key exchange paths.

The difference in learning performance between the two approaches, in the case of the extended alphabet, was profound. Black-box learning failed to terminate after 3 days and over 60k queries. We found that this was due to multiple states and inputs warranting empty responses. Consequently, black-box learning exhausted its exploration bound, trying all possible combinations of the troublesome inputs at the affected states. In contrast, as shown in Table 2, StateInspector is able to handle such cases much more effectively. This is in part because some groups of inputs are found to result in equivalent snapshots, and when the state equivalence is not immediately evident, our merging strategy quickly finds memory differences are inconsequential.

6.1.5 *Hostap-TLS.*

We tested Hostap's TLS library as both a client and server. Although this library is described as experimental, it is

[4] https://github.com/openssl/openssl/issues/8079

[5] https://github.com/openssl/openssl/pull/8080

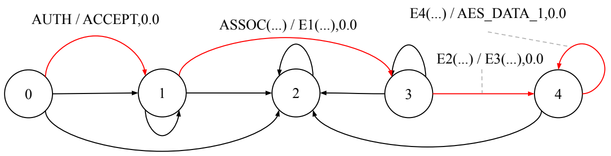


Figure 7: IWD Version 1.8 state machine (unpatched)

used in resource-constrained Wi-Fi devices [13] and Wi-Fi clients in embedded Linux images created using Buildroot [9].

Our state machine showed that, surprisingly, TLS alerts are always sent in plaintext, which might leak information about the connection. Further, some frames from the extended alphabet, such as Heartbeat requests or responses were not supported, and sending them resulted in desynchronisation of the TLS connection. We therefore do not include learning results for this implementation with the extended alphabet.

More worryingly, the model showed that against a client, the `ServerKeyExchange` message can be skipped, meaning an adversary can instead send `ServerHelloDone`. The client will process this message, and then hits an internal error when sending a reply because no RSA key was received to encrypt the premaster secret. When Ephemeral Diffie-Hellman is used instead, the client calculates g^{cs} as the premaster secret with s equal to zero if no `ServerKeyExchange` message was received. Because the exponent is zero, the default math library of Hostap returns an internal error, causing the connection to close, meaning an adversary cannot abuse this to attack clients. Nevertheless, this does illustrate that the state machine of the client does not properly validate the order of messages, and if Hostapd switch to a different math library it might become vulnerable to attack.

6.2 WPA/2's 4-Way Handshake

We tested two widely used Linux WPA implementations: IWD (iNet Wireless Daemon) and Hostap. To learn the state machine, we start with an input alphabet of size 4 that only contains messages which occur in normal handshake executions. Our test harness automatically assigns sensible values to all of the fields of these handshake messages. To produce larger input sets, we also tried non-standard values for certain fields. For example, for the replay counter, we tried the same value consecutively, set it equal to zero, and other variations. To this end, we created two extended alphabets: one of size 15 and another one of size 40.

To detect key reinstallation bugs [43], we let the SUT send an encrypted dataframe after completing the handshake. In the inferred model, resulting dataframes encrypted using a nonce equal to one are represented using `AES_DATA_1`, while all other dataframes are represented using `AES_DATA_n`. A key reinstallation bug, or a variant thereof, can now be automatically detected by scanning for paths in the inferred model that contain multiple occurrences of `AES_DATA_1`.

6.2.1 IWD. We show part of the state machine we learn for IWD version 1.8 in Figure 7. AUTH and ASSOC are the initial wi-fi messages and E1 to E4 are the steps of the WPA 4-way handshake. Simplifying this state machine we only labelled the happy flow and merge the different configuration options for the handshake messages into one action. We provide the full state machine in our submitted artefact.

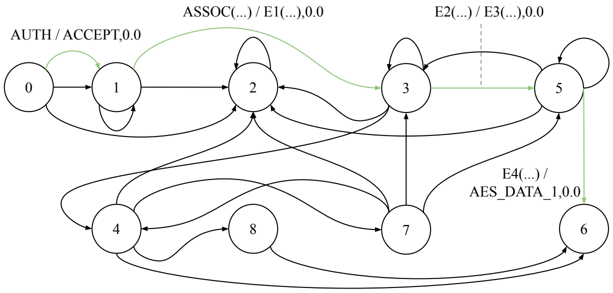


Figure 8: IWD Version 1.9 state machine (patched)

The key aspect of this state machine is that the last message of the handshake (E4) can be replayed in the state that sends encrypted data (state 4).This means an adversary can replay message E4, after which it will be processed by IWD. As shown by the WPA KRACK attack [43] this triggers a key reinstallation. Note that this is not a default key reinstallation, i. e., we confirmed that existing key reinstallation tools cannot detect it [42]. The discovered reinstallation can be abused to replay, decrypt, and possibly forge data frames [43]. We reported this vulnerability and it has been patched and assigned CVE-2020-17497 [37]. When running our tool on the patched version of IWD, the state machine enters a new state after receiving message 4, which confirms that the vulnerability has been patched. The state machine is shown in Figure 8, here we see that it is no longer possible to send multiple E4 messages. We also see some additional states, as before state 2 is a live error state, but now states 4 and 7 allow the protocol to securely recover from repeated AUTH and ASSOC messages.

With black-box testing, the key reinstallation is only found when using a small input alphabet *and* when the test harness always sends handshake messages with a replay counter equal to the last one used by the AP. If the harness instead increments the replay counter after sending each handshake message, the key reinstallation can only be discovered when using an extended alphabet. However, in that case black-box learning takes much longer: after 3 hours the learner creates a hypothesis model but it is unable to verify this hypothesis within 8 hours (at which point we terminated the learner). This shows that our method handles larger input alphabets more efficiently, especially when queries are slow, resulting in more accurate models and increasing the chance of finding bugs.

6.2.2 Hostapd. One obstacle with Hostapd is how the last used replay counter in transmitted handshake messages is saved: our learner initially includes this counter in its candidate state memory set. When trying to merge states with different replay counter values, our concolic analysis determines this value to be state defining. Indeed, Hostapd checks whether incoming frames include a particular value. However, with each loop of the state machine, the expected value changes, and hence the learner prevents the merge. This results in a violation of Assumption 1, meaning that the true protocol state machine of Hostapd is infinite and therefore no learning method can find it. We can approximate the state machine by including a time bound on the learning. Because of this we also do not explore a bigger alphabet. For a fair comparison, we used the same time bound in our method and in black-box learning. The models which were produced with both approaches within ~210 minutes were equivalent. Under these conditions, we note that both resulting state machines followed the standard and contained no surprising transitions.

6.3 Deep State Protocol Bugs

We note that states in protocols beyond typical learning bounds (i.e., 2) are not only theoretical—a state machine flaw in a WPA/2 router [36] found a cipher downgrade was possible after 3 failed initiation attempts (which would have been missed with configured bounds less than 3, as in [17]).

When using StateInspector an implementation of our motivating example protocol from Section 3, Figure 1, we found that it could learn the state machine and find the authentication bypass with 149 queries in just over a minute, with our concolic analyser taking 20 seconds of that time. In comparison, a black-box learner using TTT and the modified W-method failed to terminate its analysis after 3 days.

To provide a more realistic test-case, we modified OpenSSL version 1.0.1j to add an extremely simple (4 line) backdoor (Appendix G, Listing 1). The backdoor hijacks existing state data to implement a simple client authentication bypass, activated by after receiving n unexpected messages (see Figure 6 of Section 5.4.2).

To learn this model, we used an extended alphabet of 21 messages, including core TLS functionality, client certificates, and various data frames. With $n = 5$, black-box learning configured with a bound equal to n, fails to identify the state after 3 days of learning and ~100k queries. StateInspector, on the other hand, identifies the main backdoor-activation state in under 2 hours and ~1.2k queries and the second activation variant (with two preceding `ClientHellos`, as opposed to the usual one) in under 3 hours and ~2.5k queries. Doubling the depth n to 10 results in 3.2k and 4k queries respectively—an approximately linear increase. Black-box learning predictably failed to locate the even deeper state due to its exponential blow up in this particular scenario.

6.4 Analysis of False Positives/Negatives

Our method found issues with four of the implementations we looked at. We verified that all of these issues are real and we have included PoC attack code and patches for the IWD key reinstallation attack and the WolfSSL DoS attack, both of which have CVEs. The inconsistent client and server certificate checking, OpenSSL has been confirmed by OpenSSL and they have added a fix as a milestone. We verified that Hostap-TLS does sent plaintext alerts sent in plaintexts, and that we can force a client to use g^0 as a premaster secret, however this does not result in an attack, because the math library used returns an error and halts the protocol.

The black-box model learning method [12, 26], which we compare our method with, checks candidate state machines against the implementation and only terminates once it correctly models the implementation to a depth of 3 steps, following the shortest path to any state. In our tests, whenever the black-box method terminated, our method produced the same state machine, therefore we can be sure that our state machines also correctly model the implementation to a depth of at least 3 steps.

For the cases in which the black-box method did not terminate, we examined each state machine in turn. For RSA-BSAFE-C with the core library, we tested all paths around the state machine to depth 3, and verified that it correctly models the implementation behaviour. Testing all paths for the larger input alphabet would be computationally difficult, and with a lack of source-code or ground truth for this implementation, further assessment could not be carried out. With GnuTLS and WolfSSL however, we compared the state machine (all of which are available on our website [38]) to the implementation source code. All paths without loops will already have been verified by the learning method, and those with loops in these state machines are self loops indicating no state change. We ensured that these self-looping transitions do not lead to any change in state memory by inspecting the source code responsible for this observed behaviour.

For GnuTLS, we additionally needed to check that application data sent early was discarded, and for GnuTLS that a client hello, key exchange and certificate message sent after a change cipher spec messages are also ignored. Therefore, we can conclude that the state machines from our experiment in Table 2 correctly predict the behaviour of the implementation to a depth of at least three steps from any state, i.e., no misidentified states.

Examining the state machines learnt, we see that false positives at a depth greater than 3 would only be possible if our method had mistaken a state change for a self loop. As we show with our deep state example, our method is able to detect the progression of state much more effectively than black-box methods, and so is less likely to make this mistake.

We may get false negatives if input messages are missing from our alphabet of possible inputs, hence our assumption that all inputs to the protocol are known. If inputs were missing we would learn a correct sub-automata of the state machine. In future work, we hope to be able to use symbolic execution and the memory fingerprint of the protocol to automatically discover new state changing inputs.

6.5 Limitations

Errors due to the test harness: A poor quality test harness may also lead to false results. If the test harness produced badly formed protocol messages, we would mistakenly show a correct protocol message leading to an error state. If the test harness sends well formed messages that are mistaken for another message we might get incorrectly labelled and/or missing transitions.

We found an example of such a mistake in a test harness in [17], an error in the nonce handling in the test harness leads to an incorrect transition to an error state. In Section 6.1.2 we showed that our method could detect the correct state of the implementation and highlight the error, suggesting that our method is more robust against errors in the test harness than standard blackbox learning.

An incorrect message response passed to the learner will lead to the learner detecting non-determinism and halting with an error message. A good test harness will need to ensure a reliable connection to the SUT, e.g. handling network errors and avoiding timeouts.

Risk of Overfitting: When looking at other protocols, it is possible we would get false results if we have overfitted the design of our method to our case studies. We note that we developed our method just by looking at OpenSSL, and applied it to the other TLS versions and to WPA with minimial alteration. This gives us some confidence that we are not overfitting to particular protocol versions. Our method largely is independent of the size and shape of the state machine, so these should not be an issue. StateInspector is designed to look for program state on the heap; it is possible that other protocol implementations might store their state on the stack, or by using the program location.

Stack memory could be considered part of our candidate state memory and for an implementation in which different locations in the code represent the different states, we could add the program counter to the candidate state memory. We note that it is not enough to add the program counter at just the time of the memory snapshot, as would often be a general method to read messages off a socket, rather we would have to traverse the call stack and record the last few return addresses. Our original design of our tool did include this feature, however we removed it, because it slowed down the analysis, and all of our case studies used the heap for state memory. As well as our TLS and WPA examples, we have looked at open source implementations of EAP protocols [1, 2], SSH [3] and OpenVPN [4] and confirmed they use heap memory for the protocol state. While simpler protocols might use stack memory or program counters, it seems cryptographic protocol implementations primary use heap memory for state.

Finally, a parameter we set during testing was the look ahead distance, to see if candidate state memory that was loaded into a register affected control flow, which we set to 512. Registers are limited, and so compilers will not generally load values into registers and a long time before they are used, so this distance is unlikely to be greater for other protocols.

6.6 Other Possible Designs

Comparison with A/B testing: An alternative approach to testing implementations is be a form of A/B testing, in which messages are sent to two different implementations of the same protocol to look for differences. This type of testing would not reveal the state machine of the implementation, so there would be no way to see if two queries lead to the same state. Consequently, it would require many more messages, and no way to know when the entire state machine had been explored. As a case study we consider the patched and unpatched IWD implementations, from Figures 8,7. With A/B testing, two different responses would be noted if the tester performed the handshake followed by two repeated E4 messages. This would lead to identification of the vulnerabilities. However, systematically finding this would require all possible queries to be tried, and as these include timeouts, each query takes on average 10 seconds. At a depth of 5, and our alphabet of 15 messages, this would take 87 days, compared to our methods run time of 1 hour. Deeper vulnerabilities, such as our deep state example of a WPA/2 vulnerability from [36] that is at a depth of 8, could not be found through systematic A/B testing of all paths.

Our experiments show that the majority of differences between the state machines are due to none security issues, such as error handling, therefore the A/B method would return many false positives. For instance, before reaching the vulnerabilities when comparing patched and unpatched IWD, A/B testing would also find 32 differences when traversing states 4, 7 and 8 of the patched version (Figure 8) none of which are security critical.

Justifying the use of symbolic analysis: To test if our memory snapshotting method would work without the taint analysis, we re-ran our case studies with the symbolic execution and state merging turned off. Out of these 9 (the GnuTLS and OpenSSL versions) failed to terminate. Inspecting the candidate state memory, we found that this included a message counter, which increased with every message received. Without symbolic execution this would look like a new protocol state was reached after every message. Symbolic execution can tell that the message counter never affects the control flow, and so removes it from the candidate state memory set.

Other design choices: It would also have been theoretically possible to find the state memory without the memory diffing, by using taint analysis on all memory to find which controlled the outputs. However, running taint analysis on so much memory is unlikely to be tractable. As memory allocation functions (e.g. malloc) may return different addresses for each run, we used Gotoh's algorithm [22] to alignment memory between runs. An alternative to this would have been to rewrite the memory allocator to return the same values for each run.

7 CONCLUSION

Black-box testing is fundamentally limited because it can only reason about how a SUT interacts with the outside world. On the one hand, this means that a large number of queries are required to learn state machines even for simple protocols, and on the other, because we have no insight into *how* the SUT processes inputs, we cannot optimise queries to trigger or avoid certain behaviours.

In this study we have proposed a new grey-box approach, StateInspector, which overcomes many of these issues. Our method outperforms black-box learning across all tests. In six of our tests, black-box learning did not terminate, this includes the two case studies that resulted in CVEs being assigned. We have demonstrated that black-box learning performs particularly badly with larger alphabets of inputs. Past work on learning state machines tackled this poor performance by ensuring termination with small alphabets of core messages. Our work on the other hand, makes it possible to consider a full library of messages, which has enabled us to discover high impact vulnerabilities and numerous concerning deviations from the standards.

REFERENCES

[1] [n. d.]. EAP Protocol, wpa_supplicant source code. https://w1.fi/cgit.

[2] [n. d.]. EAP Protocols, FreeRadius source code. https://github.com/FreeRADIUS/freeradius-server.

[3] [n. d.]. OpenSSH source code. https://github.com/openssh/.

[4] [n. d.]. OpenVPN source code. https://github.com/OpenVPN/openvpn.

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

[6] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. 2010. Inference and abstraction of the biometric passport. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 673–686.

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

[8] Erik Bosman. 2020. ptrace-burrito. Retrieved 3 September 2020 from https://github.com/brainsmoke/ptrace-burrito.

[9] Buildroot Association. 2020. Buildroot. Retrieved 3 September 2020 from https://buildroot.org/.

[10] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy*.

[11] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. 2011. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery.. In *USENIX Security Symposium*, Vol. 139.

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

[13] CodeApe123. 2020. Hostapd porting and use. Retrieved 9 January 2021 from https://blog.csdn.net/sean_8180/article/details/86496922.

[14] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 110–125.

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

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

[17] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations.. In *USENIX Security*, Vol. 15. 193–206.

[18] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*. Springer, 454–471.

[19] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2523–2540.

[20] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model Learning and Model Checking of SSH Implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN 2017)*. ACM, 142–151. https://doi.org/10.1145/3092282.3092289

[21] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.

[22] Osamu Gotoh. 1982. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* 162, 3 (1982), 705–708. https://doi.org/10.1016/0022-2836(82)90398-9

[23] Hex-Rays. 2020. IDA Pro. Retrieved 3 September 2020 from https://www.hex-rays.com/products/ida/.

[24] Md. Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. 2017. Analyzing Operational Behavior of Stateful Protocol Implementations for Detecting Semantic Bugs. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*. IEEE Computer Society, 627–638.

[25] Falk Howar, Bengt Jonsson, and Frits Vaandrager. 2019. Combining black-box and white-box techniques for learning register automata. In *Computing and Software Science*. Springer, 563–588.

[26] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification*. Springer, 307–322.

[27] Manuel Mendonça and Nuno Neves. 2008. Fuzzing wi-fi drivers to locate security vulnerabilities. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*. IEEE, 110–119.

[28] Oliver Niese. 2003. *An integrated approach to testing complex systems*. Ph. D. Dissertation. Universität Dortmund.

[29] Maria Leonor Pacheco, Max von Hippel, Ben Weintraub, Dan Goldwasser, and Cristina Nita-Rotaru. 2022. Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 51–68.

[30] Andrea Pferscher and Bernhard K Aichernig. 2021. Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning. In *International Symposium on Formal Methods*. Springer, 524–542.

[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[32] QUARKSLAB. 2020. Triton. Retrieved 3 September 2020 from https://triton.quarkslab.com/.

[33] Timo Schrijvers, FW Vaandrager, and NH Jansen. 2018. Learning register automata using Taint Analysis. *Bachelors Thesis* (2018).

[34] Muzammil Shahbaz and Roland Groz. 2009. Inferring Mealy Machines. *FM* 9 (2009), 207–222.

[35] Juraj Somorovsky. 2016. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1492–1504.

[36] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. 2018. Extending automated protocol state learning for the 802.11 4-way handshake. In *European Symposium on Research in Computer Security*. Springer, 325–345.

[37] Chris McMahon Stone, Sam L. Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet, and Tom Chothia. 2020. IWD: CVE-2020-17497.

[38] Chris McMahon Stone, Sam L. Thomas, Mathy Vanhoef, James Henderson, Nicolas Bailluet, and Tom Chothia. 2022. StateInspector. https://github.com/ChrisMcMStone/state-inspector.

[39] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. 2017. Model-based testing IoT communication via active automata learning. In *2017 IEEE International conference on software testing, verification and validation (ICST)*. IEEE, 276–287.

[40] Gerco van Heerdt, Clemens Kupke, Jurriaan Rot, and Alexandra Silva. 2020. Learning Weighted Automata over Principal Ideal Domains. In *Foundations of Software Science and Computation Structures*, Jean Goubault-Larrecq and Barbara König (Eds.). 602–621.

[41] Gerco van Heerdt, Matteo Sammartino, and Alexandra Silva. 2020. Learning Automata with Side-Effects. In *Coalgebraic Methods in Computer Science*, Daniela Petrişan and Jurriaan Rot (Eds.). 68–89.

[42] Mathy Vanhoef. 2021. KRACK Attack Scripts. Retrieved 30 January 2020 from https://github.com/vanhoefm/krackattacks-scripts.

[43] Mathy Vanhoef and Frank Piessens. 2017. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In *Proceedings of the 24th ACM Conference on Computer and Communication Security*. ACM.

A MAIN GREY-BOX STATE LEARNING ALGORITHM

Algorithm 1: Grey-box state machine learning.

```
In: $SUT$: system under test; $H$: test harness;
    $bootstrapQueries$: set of boostrap queries to initiate
    learning.
Out: $M$: identified state memory; $A$: learned automata.
$M \leftarrow empty$                // candidate state memory.
$brs \leftarrow empty$              // bootstrap query results.
// perform bootstrap queries
foreach $q \in bootstrapQueries$ do
    /* ❶ generate snapshots of the SUT for each query, ❷ use
       the snapshots to perform allocation log alignment and
       memory diffing, ❸ identify candidate state memory
       (Sections 5.1 and 5.2).                                 */
    $br \leftarrow execQuery(SUT, H, q)$
    $brs \leftarrow append(brs, br)$
    $M \leftarrow updateStateMem(M, br)$
end
$A \leftarrow emptyAutomata$        // initial (empty) model.
foreach $br \in brs$ do
    // bootstrap initial model (Section 5.2).
    $A \leftarrow processBootstrapResult(A, br)$
end
$M \leftarrow minStateMem(SUT, H, M)$        // Section 5.3.
$n \leftarrow 1$                    // current maximal query length.
repeat
    $qs \leftarrow generateQueries(n)$       // up to length $n$.
    // execute queries and update model.
    foreach $q \in qs$ do
        $A \leftarrow execQueryUpdate(SUT, H, A, M, q)$
    end
    // execute state merge check (Section 5.5).
    $A \leftarrow checkStateMerge(SUT, H, A, M)$
    // handle uncertain memory (Section 5.4).
    $M \leftarrow handleUncertainStateMem(SUT, H, M)$
    $n \leftarrow n + 1$
until $modelComplete(A, M, n)$
return $(A, M)$
```

B ALGORITHM TO GENERATE WATCHPOINT HIT SNAPSHOTS

Algorithm 2: Generation of watchpoint hit snapshots.

```
Input: $SUT$: the system under test; $H$: test harness;
       $(addr, size)$: memory to test; $p$: prefix query to arrive
       at merge state; $mergeState$: candidate merge state; $d$:
       merge depth.
Out: $W$: list of $watchpointHitExecutionDumps$
foreach $(input, s) \in reachableFrom(mergeState, d)$ do
    if $outputFor(input, s) \in disabled$ then
        $continue$;
    end
    $initialiseWatchpointMonitor(SUT, addr, size)$;
    $w \leftarrow execWatchpointQuery(SUT, H, p \mid\mid input)$;
    $W \leftarrow append(W, w)$;
end
return $W$;
```

C SETTING PARAMETERS FOR GOTOH'S ALGORITHM

Gotoh's algorithm is a dynamic programming algorithm that uses scores to find an optimal alignment for two sequences, i. e., the one with the highest score. It requires four input parameters that are used to score elements pair-wise:

(1) *match*: score used if two elements are aligned and match.
(2) *mismatch*: score used if two elements are aligned and do not match.
(3) $gap_{opening}$: score used if a gap is inserted or 'opened'. Inserting a gap results in an element from one sequence being aligned with 'nothing' and the other sequence being shifted right by one element.
(4) $gap_{enlargement}$: score if used an already 'open' gap is extended.

To compute the score of an alignment for two sequences, we sum the classification scores (i. e., *match, mismatch, gap opening*, or *enlargement*) of pairs of elements or inserted gaps, without reordering. For example, using the sequences in Fig. 9, we calculate our alignment score as:

$$match + mismatch + gap_{opening} + gap_{enlargement}$$

A B - -
A A B C

Figure 9: Example of sequence alignment. From left to right, we classify the pairs of elements as: *match* (A, A), *mismatch* (B, A), *gap opening* (-, B), and *gap enlargement* (-, C).

To avoid inserting many small gaps in an alignment, we must set $gap_{opening} < gap_{enlargement}$. This inequality ensures that opening a gap is more penalising than extending an already existing one. This forces gaps to be grouped, which results in an optimal alignment.

To derive mappings from allocation logs, we prevent any allocation mismatch when aligning two sequences. Therefore, we force the algorithm to insert a gap (of any size) instead of choosing a mismatch. For example, suppose we want to align two sequences of allocations of length n and m, with $n >= m >= 1$. The largest gap we could insert is of size n (when there is absolutely no match). Thus, to prevent any mismatch, we must set:

$$mismatch < gap_{opening} + (n-1)gap_{enlargement}$$

This inequality ensures that a mismatch is always more penalising than inserting a gap of maximal length. As a result, the algorithm will always choose to insert a gap instead of mismatching two allocations.

D CANDIDATE STATE MEMORY TYPE-INFERENCE

The values stored at state-defining locations are often only meaningful when considered as a group, e. g., by treating four consecutive bytes as a 32-bit integer. Since we perform snapshot diffing at a byte-level granularity, we may not identify the most-significant bytes of larger integer types if we do not observe their values changing across snapshots. As learning the range of values a location can take is a prerequisite to determine some kinds of termination behaviour, we attempt to monitor locations with respect to their intended types.

To learn each location's bounds, we apply a simple type-inference procedure loosely based upon that proposed by Chen et al. [10]. We perform inference at the same time as we analyse snapshots to handle uncertain state memory locations (Section 5.4). During this analysis, we simulate the SUT's execution for a fixed window of instructions, while doing so, we analyse loads and stores from/to our candidate state memory. To perform inference, we log the prefixes used in each access, e. g., in `mov byte ptr [loc_1], 0`, we record `byte` for `loc_1`. Then, following our main snapshot analysis, we compute the maximal access size for each location and assign each location a corresponding type. To disambiguate overlapped accesses, we determine a location's type based on the minimal non-overlapped range. Following type-inference, we update our model and the state classifications maintained by our learner using the new type-bounds discovered.

E ADDITIONAL IMPLEMENTATION DETAILS

In this appendix, we provide a breakdown of each major component of StateInspector's implementation.

State snapshotting. We perform memory and program state snapshots using ptrace-burrito [8]. We primarily use syscall-based hooks to trigger our snapshotting mechanism. Each snapshot consists of the program's registers and its entire virtual address space including, all heap regions, its stack, and all mapped sections.

Watchpoint queries. We use ptrace-burrito to set watchpoints, and use its hooking facilities to capture program state snapshots on each watchpoint hit. Each watchpoint snapshot contains the same information as a regular state snapshot, as well as information about the watchpoint that triggered the snapshot.

Taint & concolic execution. We use Triton [32] to build our taint and concolic execution-based analysis. To provide function and control-flow information to our tool we use IDA Pro [23], which provides function and basic block boundary information, and function-level control-flow graph information. To infer type information, we use Triton's read/write hooks to provide information on memory accesses.

F OPENSSL TEST HARNESS STATE MEMORY ANALYSIS

Table 3: The sets of differing memory for states on suspected and confirmed alternate paths to successful handshake completion, when compared to the legitimate *happy flow* state 4 (s4) memory.

State IDs	Addresses of differing memory to s4
s14 s15, s17, s13	{0x555555a162a0, 0x4b0, 0x13}
	{0x555555a162a0, 0x4b0, 0x120}
	{0x555555a162a0, 0x4b0, 0x0}
	{0x555555a0e6b0, 0x160, 0x0}
	{0x555555a0e6b0, 0x160, 0x10}
	{0x555555a0e6b0, 0x160, 0x44}
	{0x555555a0e6b0, 0x160, 0x68}
	{0x555555a0e6b0, 0x160, 0x6c}
	{0x555555a0e6b0, 0x160, 0xc0}
	{0x555555a0e6b0, 0x160, 0xc8}
	{0x555555a0e6b0, 0x160, 0xd2}

Using the learning output of StateInspector, we carried out an investigation into what the differing memory in Table 3 actually constituted. We were able to do this by referring to the logs produced by the tool, which print the source code details of any watchpoint hits made at reads of the tested memory. All considered states shared the two differing addresses listed at the top of the table. The first of these represented a counter referred to as the `sequence number`. This value increments for each message received after the `ChangeCipherSpec` message and is used for constructing MACs. The second address in the table refers to the frame type of the just received message. This value is determined as not state defining because from any given state it is always written to before being read, i. e., it may be state influencing, but not defining (c.f., Section 5.4.1). The third address in the table pertains to a flag which determines if the client has attempted more than one `ClientHello` message. Only prior to the receipt of a `ClientKeyExchange` is this flag ever read (and therefore used to define the state). Consequently, it is not considered state defining for any of the states listed in the table. The remaining memory refers to arbitrary certificate data which we also found was not state defining. In particular, we were unable to detect any reads of the memory from the states considered.

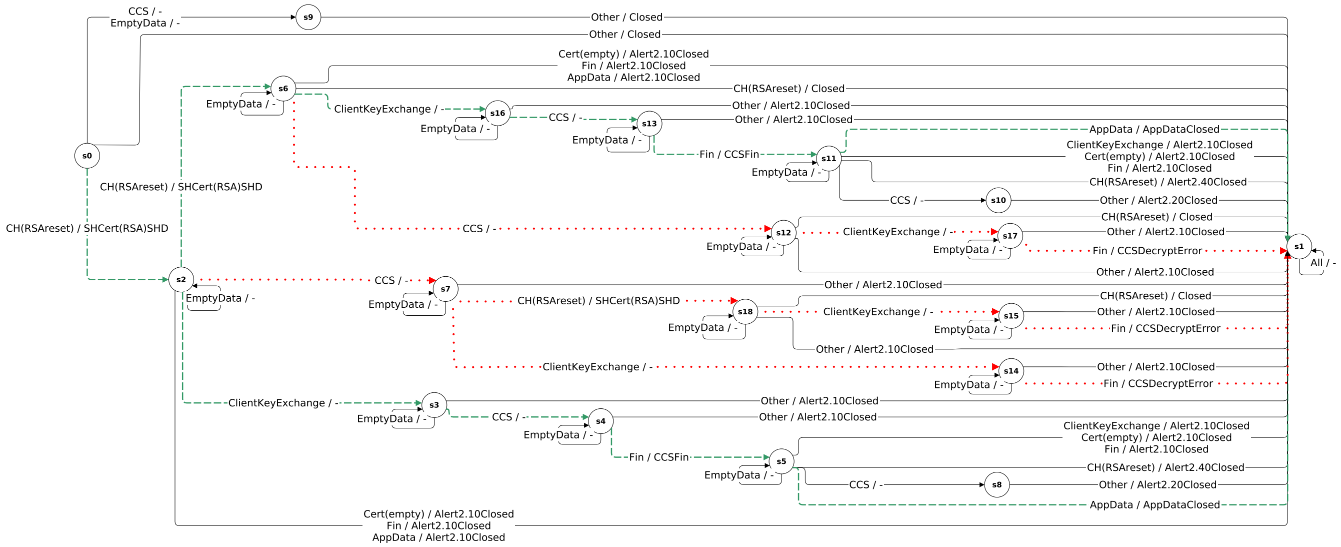


Figure 10: Learned model for OpenSSL 1.0.1g

G OPENSSL CLIENT AUTHENTICATION BYPASS BACKDOOR

Listing 1: A four line modification to OpenSSL 1.0.1j, which introduces a client authentication bypass at a configurable depth (BACKDOOR_DEPTH). In the first part, we hijack a counter, the read_sequence counter, and conditionally increment it each time an unexpected data frame is read whilst waiting on the ClientCertificateVerify signature. In the second part, if this counter reaches the specified depth value, invalid ClientCertificateVerify signatures are ignored, and the handshake proceeds to the ChangeCipherSpec state. After this point, the read_sequence is automatically reset by the implementation for use in later MAC calculations.

```
@@ /ssl/s3_pkt.c:139 ssl3_read_n()
         SSL3_BUFFER *rb;
+++ // Backdoor pt 1: Set-up
+++ if(s->state == SSL3_ST_SR_CERT_VRFY_A)
+++     ++(*s->s3->read_sequence);

         if (n <= 0) return n;
         rb = &(s->s3->rbuf);

@@ /ssl/s3_srvr.c:683  ssl3_accept()
         ret=ssl3_get_cert_verify(s);
--- if (ret <= 0) goto end;
+
+++ // Backdoor pt 2: Activation
+++ if(*s->s3->read_sequence != BACKDOOR_DEPTH
+++     && ret <= 0) {
+++     goto end;
+++ } else {
         ...
         s->state=SSL3_ST_SR_FINISHED_A;
         s->init_num=0;
+++ }
```

H MISLEADING OPENSSL DOCUMENTATION

When testing the built-in server of OpenSSL, we found that configuring it to enforce client authentication is rather error-prone. In particular, after an initial reading of the documentation for how to do so, we came across the option below:

```
-verify int, -Verify int
  The verify depth to use. This specifies
  the maximum length of the client certi-
  ficate chain and makes the server requ-
  est a certificate from the client. With
  the -verify  option a certificate is
  requested but the client does not have
  to send one, with the -Verify option
  the client must supply a certificate or
  an error occurs.
```

Following this, we used the -Verify option when starting the built-in server. Surprisingly though, our learned models indicated that the server accepted invalid client certificates. At first glance, this looked like a severe vulnerability, however, upon closer inspection of the documentation we found an extra option was required:

```
-verify_return_error
  Verification errors normally just print
  a message but allow the connection to
  continue, for debugging purposes. If
  this option is used, then verification
  errors close the connection.
```

Our online investigation revealed we were not alone in our confusion. A recent Github issue[6] describes the same misleading configuration, and resulted in a patch to the OpenSSL client implementation[7]. Unfortunately, a similar patch was not written for the server implementation, resulting in a confusing disparity between the command-line options for the client and server. We have reported this issue to the OpenSSL developers.

[6] https://github.com/openssl/openssl/issues/8079
[7] https://github.com/openssl/openssl/pull/8080

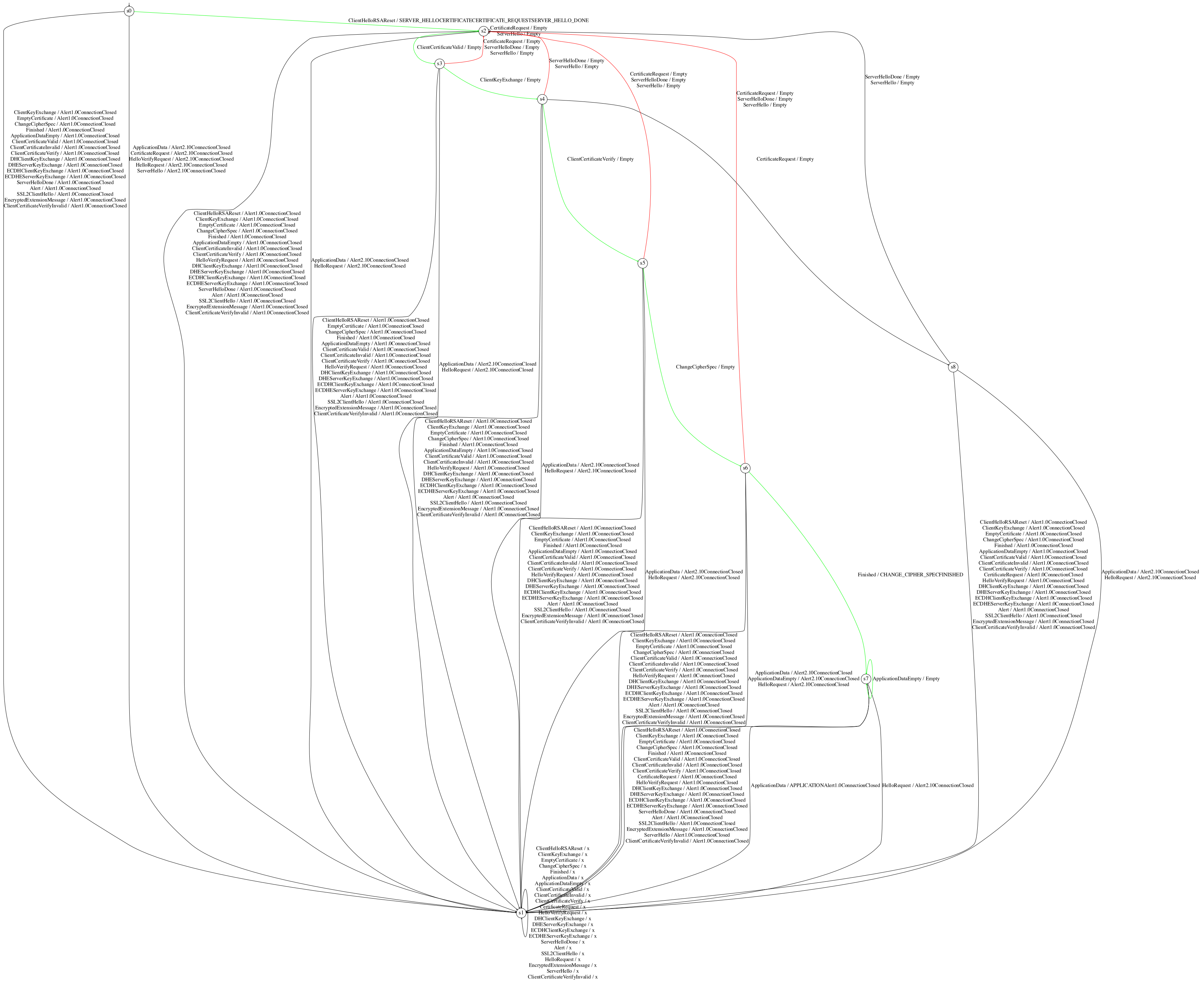

s0
s1
s2
s3
s4
s5
s6
s7
s8
ClientHelloRSAReset / SERVER_HELLOCERTIFICATECERTIFICATE_REQUESTSERVER_HELLO_DONE
CertificateRequest / Empty
ServerHello / Empty
ClientCertificateValid / Empty
CertificateRequest / Empty
ServerHelloDone / Empty
ServerHello / Empty
ServerHelloDone / Empty
ServerHello / Empty
ClientKeyExchange / Empty
CertificateRequest / Empty
ServerHelloDone / Empty
ServerHello / Empty
CertificateRequest / Empty
ServerHelloDone / Empty
ServerHello / Empty
ServerHelloDone / Empty
ServerHello / Empty
ClientCertificateVerify / Empty
CertificateRequest / Empty
ChangeCipherSpec / Empty
Finished / CHANGE_CIPHER_SPECFINISHED
ApplicationDataEmpty / Empty
ClientKeyExchange / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
ChangeCipherSpec / Alert1.0ConnectionClosed
Finished / Alert1.0ConnectionClosed
ApplicationDataEmpty / Alert1.0ConnectionClosed
ClientCertificateValid / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
ClientCertificateVerify / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
ServerHelloDone / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / Alert2.10ConnectionClosed
CertificateRequest / Alert2.10ConnectionClosed
HelloVerifyRequest / Alert2.10ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ServerHello / Alert2.10ConnectionClosed
ClientHelloRSAReset / Alert1.0ConnectionClosed
ClientKeyExchange / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
ChangeCipherSpec / Alert1.0ConnectionClosed
Finished / Alert1.0ConnectionClosed
ApplicationDataEmpty / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
ClientCertificateVerify / Alert1.0ConnectionClosed
HelloVerifyRequest / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
ServerHelloDone / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / Alert2.10ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ClientHelloRSAReset / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
ChangeCipherSpec / Alert1.0ConnectionClosed
Finished / Alert1.0ConnectionClosed
ApplicationDataEmpty / Alert1.0ConnectionClosed
ClientCertificateValid / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
ClientCertificateVerify / Alert1.0ConnectionClosed
HelloVerifyRequest / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / Alert2.10ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ClientHelloRSAReset / Alert1.0ConnectionClosed
ClientKeyExchange / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
ChangeCipherSpec / Alert1.0ConnectionClosed
Finished / Alert1.0ConnectionClosed
ApplicationDataEmpty / Alert1.0ConnectionClosed
ClientCertificateValid / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
HelloVerifyRequest / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / Alert2.10ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ClientHelloRSAReset / Alert1.0ConnectionClosed
ClientKeyExchange / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
Finished / Alert1.0ConnectionClosed
ApplicationDataEmpty / Alert1.0ConnectionClosed
ClientCertificateValid / Alert1.0ConnectionClosed
ClientCertificateVerify / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
HelloVerifyRequest / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / Alert2.10ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ClientHelloRSAReset / Alert1.0ConnectionClosed
ClientKeyExchange / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
ChangeCipherSpec / Alert1.0ConnectionClosed
ClientCertificateValid / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
ClientCertificateVerify / Alert1.0ConnectionClosed
HelloVerifyRequest / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / Alert2.10ConnectionClosed
ApplicationDataEmpty / Alert2.10ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ClientHelloRSAReset / Alert1.0ConnectionClosed
ClientKeyExchange / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
ChangeCipherSpec / Alert1.0ConnectionClosed
Finished / Alert1.0ConnectionClosed
ClientCertificateValid / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
ClientCertificateVerify / Alert1.0ConnectionClosed
CertificateRequest / Alert1.0ConnectionClosed
HelloVerifyRequest / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
ServerHelloDone / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ServerHello / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / APPLICATIONAlert1.0ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ClientHelloRSAReset / Alert1.0ConnectionClosed
ClientKeyExchange / Alert1.0ConnectionClosed
EmptyCertificate / Alert1.0ConnectionClosed
ChangeCipherSpec / Alert1.0ConnectionClosed
Finished / Alert1.0ConnectionClosed
ApplicationDataEmpty / Alert1.0ConnectionClosed
ClientCertificateValid / Alert1.0ConnectionClosed
ClientCertificateInvalid / Alert1.0ConnectionClosed
ClientCertificateVerify / Alert1.0ConnectionClosed
CertificateRequest / Alert1.0ConnectionClosed
HelloVerifyRequest / Alert1.0ConnectionClosed
DHClientKeyExchange / Alert1.0ConnectionClosed
DHEServerKeyExchange / Alert1.0ConnectionClosed
ECDHClientKeyExchange / Alert1.0ConnectionClosed
ECDHEServerKeyExchange / Alert1.0ConnectionClosed
Alert / Alert1.0ConnectionClosed
SSL2ClientHello / Alert1.0ConnectionClosed
EncryptedExtensionMessage / Alert1.0ConnectionClosed
ClientCertificateVerifyInvalid / Alert1.0ConnectionClosed
ApplicationData / Alert2.10ConnectionClosed
HelloRequest / Alert2.10ConnectionClosed
ClientHelloRSAReset / x
ClientKeyExchange / x
EmptyCertificate / x
ChangeCipherSpec / x
Finished / x
ApplicationData / x
ApplicationDataEmpty / x
ClientCertificateValid / x
ClientCertificateInvalid / x
ClientCertificateVerify / x
CertificateRequest / x
HelloVerifyRequest / x
DHClientKeyExchange / x
DHEServerKeyExchange / x
ECDHClientKeyExchange / x
ECDHEServerKeyExchange / x
ServerHelloDone / x
Alert / x
SSL2ClientHello / x
HelloRequest / x
EncryptedExtensionMessage / x
ServerHello / x
ClientCertificateVerifyInvalid / x

Figure 11: WolfSSL Server 4.8 State Machine (can be zoomed in, the PDF is available at [38])