Bachelor thesis
Computing Science

Radboud University

MM1T Learner for the 4-Way Handshake Protocol

Author:
Andrea Minichová
andrea.minichova@ru.nl

Supervisor/ first assessor:
Prof. Dr. F. W. Vaandrager
f.vaandrager@cs.ru.nl

Second assessor:
Dr. E. Poll
e.poll@cs.ru.nl

July 4, 2022

Abstract

Model learning is a technique which has been shown useful in software verification. One challenge researchers have faced is learning time-related behavior. In this research, we investigate a proposed solution, the use of Mealy Machines with 1 Timer (MM1Ts). We apply the MM1T model learner on the Wi-Fi 4-way handshake protocol. We compare the MM1T learner to an existing Wi-Fi learner, which efficiently learns the protocol through handling time behavior in a separate learning stage. We show that the MM1T learner is comparable to the Wi-Fi learner in the number of queries posed during learning. We conclude that the MM1T learner can be a potential solution for learning real-life applications such as network protocols.

Keywords: Model learning, Mealy machines, MM1Ts, IEEE 802.11, Wi-Fi, 4-Way handshake protocol

Contents

Chapter 1

Introduction

1.1 Model Learning

Model learning is a technique for inferring a state machine representation of a system. Current learning algorithms build upon a concept that was first proposed by Angluin in her seminal paper [1]. Essentially, learning is done by iteratively feeding the system input sequences, observing the corresponding output sequences, and adjusting the model accordingly. Since the knowledge of input-output relations is sufficient, this allows for learning models of systems whose inner workings remain unknown. These are so-called *black box systems*.

Black-box model learning in combination with subsequent model checking is used for software verification. The learning algorithm automates sending in (potentially malicious) input. After obtaining a model, model checking techniques are used to verify the system's behavior under different flows. This allows for spotting even the inconspicuous software flaws, some of which might be less important while others could pose significant risks to functional correctness or security.

Many case studies have demonstrated the importance of model learning for verifying security and network protocols. When learning the TLS protocol [2], de Ruiter et al. discovered flaws in 3 out of 9 tested implementations. Due to their work, it was possible to fix these bugs in future updates. DTLS protocol violations (including 4 serious security violations) were found and also prompted fixes [3]. Some TCP implementations in Windows and Linux systems were found non-conforming to the standards [4] which was later confirmed by Linux developers. Similar findings were brought by research into learning the SSH protocol [5].

Model learning is also recognized by the industrial sector. In [6], three

industrial case studies were conducted across various sectors: web (portfolio compression service from TriOptima), automotive (break-by-wire system from Volvo Technology), and financial (an access server from Fredhopper). The research uncovered errors in all three. In [7], model learning assisted the process of introducing a new hardware component in interventional radiology systems at Philips. The authors compared the models of the old and the new implementation and found issues in both. Discovering these issues in the early stage of development saved costs of having to fix them in the later stage.

1.2 Learning Time-Related Behavior

As we have seen, model learning has shown considerable potential for verifying real-life applications. However, there is still important work necessary to better harness this. One of the challenges researchers have to face is tackling time-related behavior. The algorithms currently in use do not allow for directly learning systems that implement clocks, such as previously discussed network protocols. In this context, clocks are used for e.g. packet retransmissions or dropping the connection after a certain period of inactivity.

One solution is to alter or artificially suppress time-related behavior. For example, while learning the TCP protocol [8], the authors adjusted the network adapter to ignore all retransmissions and kept the queries short enough to avoid possible protocol resets. When learning the SSH protocol [5] timeout periods were manually set to overcome time-induced non-determinism. In TLS learning timeouts are manually identified and multiple responses are mapped to a single output [2].

While such approaches reduce the state space of the final model, they also disregard potentially important behavior. With the intention to cover time-related behavior, researchers designed new classes of models and altered the learning algorithms.

One of such classes is timed automata [9] which incorporate clock variables and transitions with clock guards and resets. The problem with timed automata is that the clocks are not directly observable by learning algorithms. Deterministic event-recording automata (DERAs) [10] [11] solve this by introducing clock as part of the alphabet. Transitions then reset the clock based on the given input. Unfortunately, learning DERAs was shown too high in complexity due to the difficulty of inferring guards. Moreover, it is not expressive enough for learning certain network protocols.

DERAs which allow some transitions to not reset the clock [12] solve the

latter issue, although they simultaneously introduce complexity by having to infer which transitions reset the clock. A new learning algorithm was designed for deterministic one-clock timed automata (DOTAs) where resets are inferred by brute force [13]. This algorithm, however, also suffers from high complexity.

1.3 Learning the 4-Way Handshake with the MM1T Learner

Recently, a new class of models was proposed by Vaandrager et al.: Mealy Machines with 1 Timer (MM1Ts) [14]. The authors showed MM1Ts to be a good solution for learning time-related behavior.

The core principle of MM1Ts is the usage of timers instead of clocks. The difference between a timer and a clock is that the clock's value is increasing, whereas the timer's value is decreasing. Timers allow for discarding guards, thus improving the complexity whilst preserving a sufficient model expressivity. Additionally, the authors introduced a wrapper, which translates input/output sequences of so-called timed words into input/output sequences that can be processed by the existing learning algorithms. As a result, MM1Ts can be learned using functionalities from the commonly used LearnLib library [15].

MM1Ts open up a new possibility for learning network protocols with one timer. To further explore this potential, we conduct a case study on learning the 4-way handshake protocol which is used for establishing a Wi-Fi connection.

To this end, we first use a Wi-Fi learner tool designed by Stone et al. [16]. To learn the protocol efficiently, the authors divided the learning process into two stages. The first stage learns a base model excluding time behavior and the second stage extends the base model with transitions capturing timeouts and retransmissions. In this paper, we will use the term Wi-Fi learner to refer to this tool.

Using the Wi-Fi learner, we learn model representations of the handshake implementations in several routers. Then we use the acquired models to simulate the routers' behavior. We use the MM1T learner to learn MM1T representations of the simulated routers. Next, we compare the two learners in the number of queries required for learning the protocol. Finally, we provide a possible explanation behind our findings.

1.4 Document Structure

The remainder of this paper is structured as follows: The technical definitions of the concepts required to understand our work are provided in Chapter 2. We outline the experiment setup in Chapter 3. Next, we present our findings and a justification for them in Chapter 4. The relevant literature is covered in Chapter 5. Finally, we conclude the paper and suggest potential further research in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we dive into the technical details of the concepts used throughout our research. We introduce Mealy machines in Section 2.1, and the learning framework in Section 2.2. Next, we extend the explanation to MM1Ts in Section 2.3 and the learning framework for MM1Ts in Section 2.4. We describe the 4-way handshake protocol in Section 2.5. Finally, we explain how the Wi-Fi learner works in Section 2.6.

2.1 Mealy Machines

Both learners used in this research rely on the concept of Mealy machines [17]. Due to their expressivity, Mealy machines are in general a suitable class of models for learning real-life applications.

Mealy machines distinguish between an input and output alphabet. The output symbols are determined by both the current state and the input. There is always a unique output for a specific input-state combination, meaning that Mealy machines are *deterministic*. The transition function is fully defined for all symbols in all states since Mealy machines are always *input-enabled*.

Definition 1. *Mealy machine $\mathcal{M}$ is a 6-tuple defined as $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ where*

- *I is a finite input alphabet,*
- *O is a finite output alphabet,*
- *Q is a finite set of states,*
- *$q_0 \in Q$ is the initial state,*
- *δ is the transition function $Q \times I \to Q$,*

- *and* λ *is the output function* $Q \times I \to O$.

When a Mealy machine is in a state $q \in Q$ and receives an input $i \in I$, it transitions to the state $q' \in Q = \delta(q, i)$ and produces an output $o \in O = \lambda(q, i)$.

Example 1. We present a simple coffee machine example from [18]. The coffee machine consists of a button and two appliances, namely a water tank and a coffee pod. Before a cup of coffee can be made, the machine needs to be clean and the appliances need to be present. If this is satisfied, the machine prepares coffee upon pressing the button. Any mishandling causes the machine to reach an error state from which it cannot recover.

Following is the Mealy specification of the described coffee machine:

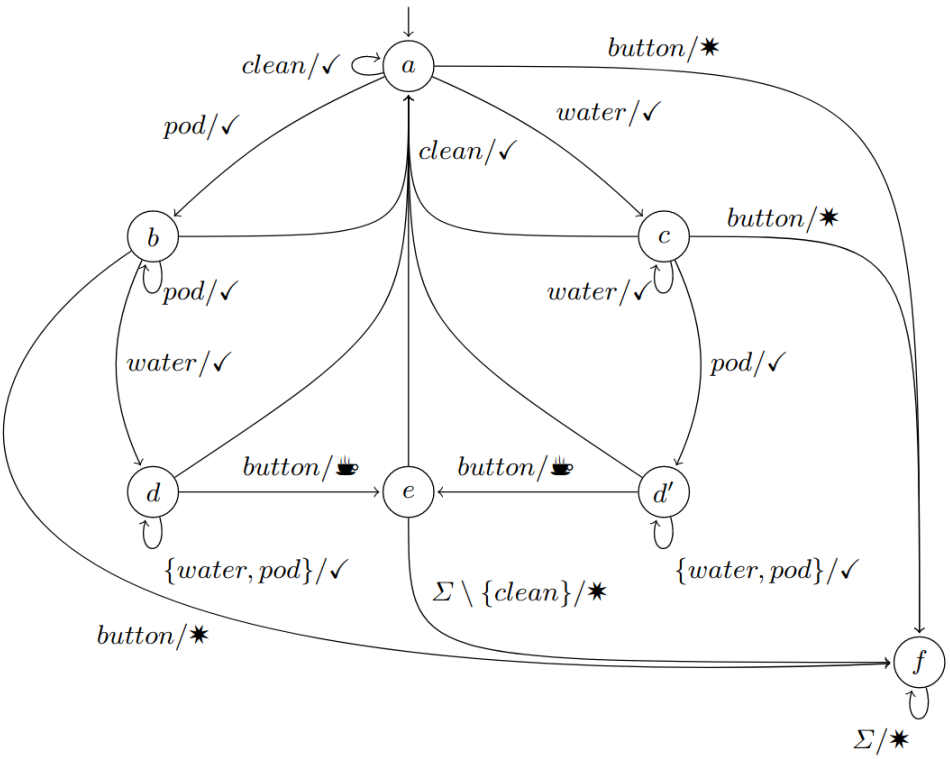

Figure 2.1: Mealy specification of a coffee machine, image from [18].

$\mathcal{M}_{CM} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ where

- $I = \{water, pod, button, clean\}$,
- $O = \{\checkmark, ☕, ✷\}$,

- $Q = \{a, b, c, d, d', e, f\}$,
- $q_0 = a$,
- the transition function and the output function are defined according to the diagram in Figure 2.1.

States d and d' can be reached through the same inputs but in a different order. Whenever an error occurs, the transitions lead to the sink state f.

2.2 MAT Framework

Most active learning algorithms (learning by posing queries) follow the MAT framework developed by Angluin [1]. The framework consists of a *learner* and a *Minimally Adequate Teacher* (MAT). The learner attempts to learn the model of the *System Under Learning* (SUL). The teacher knows the model, as it has direct access to the SUL. That is, it can feed the SUL input sequences, receive the corresponding output sequences and bring the SUL back to the initial state. As a result, the teacher can answer the queries of the learner who eventually learns the state machine representation of the SUL.

Assume the learner wants to learn a model - in our setting a Mealy machine $\mathcal{M}$. Based on the provided alphabet, it constructs a hypothesized Mealy machine $\mathcal{H}$. Learning is achieved through two types of queries:

1. **Membership query (MQ)**: Learner poses an input sequence. Teacher responds with corresponding output sequence. Learner updates $\mathcal{H}$ accordingly.
2. **Equivalence query (EQ)**: Learner asks whether the hypothesis is correct i.e., whether $\mathcal{H} \approx \mathcal{M}$. Either of two answers is possible:
   (a) Teacher responds with `no` and provides a counterexample that differentiates $\mathcal{M}$ and $\mathcal{H}$. Learner updates $\mathcal{H}$ accordingly.
   (b) Teacher responds with `yes`. This indicates that $\mathcal{H} \approx \mathcal{M}$, meaning the learner has learned $\mathcal{M}$ and the process terminates.

As Peled et al. observed, the MAT framework can be used for black-box learning of hardware and software components [19][20]. In this context, the SUL is restarted each time a new membership query is presented. Equivalence queries are approximated using a *Conformance Testing Tool* (CT) [21]. CT poses a stream of test queries (TQs), which work similarly to MQs. If one of the TQs answers with `no`, the answer to the equivalence query is `no`, otherwise it is `yes`.

Model learning applications often implement an additional *mapper* component which is placed in between the SUL and the learner. The mapper translates the input and output of the SUL into an alphabet recognized by the learner. Additionally, the mapper serves for abstraction, potentially improving the complexity of the learning algorithm. Refer to Section 2.6.2 to see an example mapper, i.e., the mapper of the Wi-Fi learner.

Figure 2.2 summarizes the learning process of black-box systems within the MAT framework. For a more thorough introduction into model learning, refer to [22] or [23].

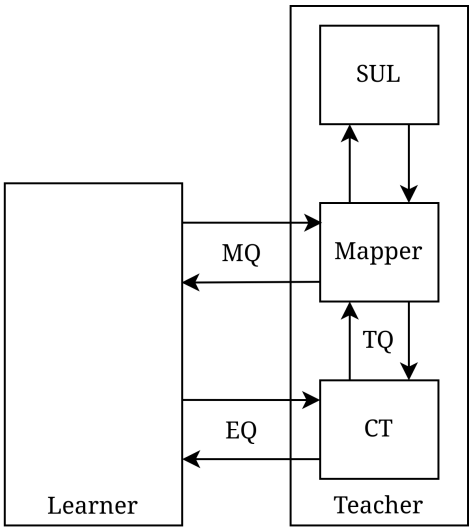

Figure 2.2: Learning process under the MAT framework, image from [22]

2.3 MM1Ts

As previously stated, MM1Ts are Mealy machines with one timer. There is always at most 1 timer in use at every state and each transition can either (re)set the timer to a certain integer value, stop the timer, or time out.

Following is the MM1T definition [14]. Here, we write $f : X \rightharpoonup Y$ to denote that f is a partial function from X to Y. We write $f(x) \downarrow$ if the result is defined for x, and $f(x) \uparrow$ if it is undefined.

Definition 2. *A* Mealy machine with a single timer (MM1T) *is defined as a 7-tuple* $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda, \tau)$*, where*

- I *is a finite set of* inputs*, containing a special element* timeout*,*
- O *is a set of* outputs*,*

- $Q = Q_{off} \cup Q_{on}$ *is a finite set of* states*, partitioned into subsets where the timer is off and on, respectively,*
- $q_0 \in Q_{off}$ *is the* initial state*,*
- $\delta : Q \times I \rightharpoonup Q$ *is a* transition function*, satisfying*

$$\delta(q, i) \uparrow \quad \Leftrightarrow \quad q \in Q_{off} \wedge i = \mathsf{timeout} \tag{2.1}$$

  (inputs are always defined, except for timeout *in states where timer is off),*

- $\lambda : Q \times I \rightharpoonup O$ *is an* output function*, satisfying*

$$\lambda(q, i) \downarrow \quad \Leftrightarrow \quad \delta(q, i) \downarrow \tag{2.2}$$

  (each transition has both an input and an output),

- $\tau : Q \times I \rightharpoonup \mathbb{N}^{>0}$ *is a* reset function*, satisfying*

$$\tau(q, i) \downarrow \quad \Rightarrow \quad \delta(q, i) \in Q_{on} \tag{2.3}$$
$$q \in Q_{off} \wedge \delta(q, i) \in Q_{on} \quad \Rightarrow \quad \tau(q, i) \downarrow \tag{2.4}$$
$$\delta(q, \mathsf{timeout}) \in Q_{on} \quad \Rightarrow \quad \tau(q, \mathsf{timeout}) \downarrow \tag{2.5}$$

  (when a transition (re)sets the timer, the timer is on in the target state; when it moves from a state where the timer is off to a state where the timer is on, it sets the timer; if the timer stays on after a timeout, it is reset).

Let $\delta(q, i) = q'$ *and* $\lambda(q, i) = o$*. We write* $q \xrightarrow{i/o,n} q'$ *if* $\tau(q, i) = n \in \mathbb{N}^{>0}$*, and* $q \xrightarrow{i/o,\perp} q'$ *or just* $q \xrightarrow{i/o} q'$ *if* $\tau(q, i) \uparrow$*.*

Example 2. We present an MM1T specification of the coffee machine introduced in Example 1. We extend the original example with a timer in state f. All transitions with the error output now start a timer of 10 seconds. When the timer reaches a timeout, the coffee machine is restarted and the user has a new chance to prepare a cup of coffee.

The input alphabet now contains a new symbol `timeout`. The output alphabet remains unchanged. The set of states Q is divided into subsets Q_{off} and Q_{on} with $Q_{on} = \{f\}$ and $Q_{off} = Q \setminus \{f\}$. We can verify that properties 2.1 through 2.5 hold.

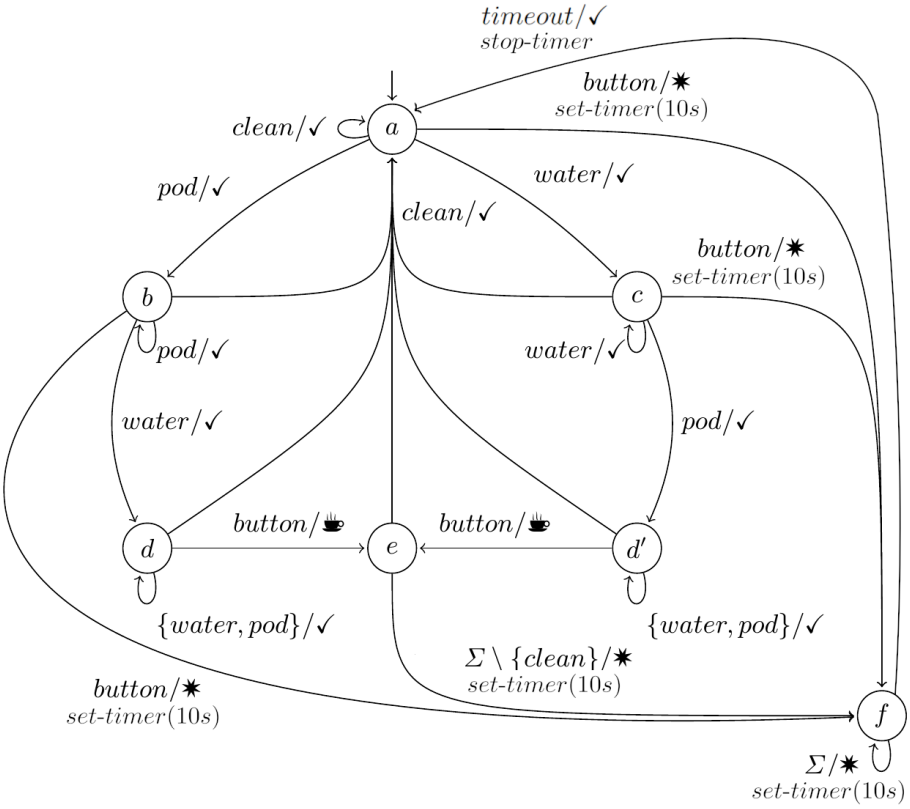

Figure 2.3: MM1T specification of the coffee machine

.

2.4 Learning MM1Ts

2.4.1 Timed Semantics

The behavior of an MM1T can be described through *timed semantics*. A timed word collects labels of transitions in a finite flow of the MM1T. Moreover, it describes the time elapsed between different transitions. Timed semantics is fitting for describing real-world systems.

Take for example transitions $a \to c \to f \to a$ from Figure 2.3. An example timed word over the transitions $w' = (5, water, ✓)\ (1, button, ✷)\ (10, \texttt{timeout}, ✓)$. Word w' can be read as follows: after an initial delay of 5 seconds, the user presents the water tank and the machine accepts this behavior (checkmark). After 1 second, the user presses the button, to which the machine responds with an error. After 10 seconds, a timeout is triggered and the machine accepts this behavior and transitions back to the initial state.

We write $\mathcal{M} \approx_{timed} \mathcal{N}$ to express that MM1Ts M and N are timed-equivalent i.e., the machines have the same set of timed words.

2.4.2 Reduction

Mealy machines can be converted into MM1Ts and vice versa. Mealy machine is just an MM1T in which the timer is off in all states. Conversely, MM1Ts can be represented as Mealy machines. New `timemout` self-loops are added for each state in Q_{off} to make the Mealy machine input enabled. The `timeout` input symbol gets associated with a dummy output symbol `nil`. The outputs of the other transitions are pairs consisting of the original output and a timer value.

We write $\mathsf{MM1T}(\mathcal{N})$ to represent a conversion of Mealy machine $\mathcal{N}$ to an MM1T and $\mathsf{Mealy}(\mathcal{M})$ to represent a conversion of MM1T $\mathcal{M}$ into a Mealy machine. A property presented in *Theorem 1* allows reducing the problem of learning MM1Ts to the problem of learning Mealy machines. This means that MM1T learning can be easily achieved through efficient learning algorithms for Mealy machines, such as L^*_M [24] and TTT [25].

Theorem 1. Suppose $\mathcal{M}$ is an MM1T with inputs I and outputs $O \subseteq \Omega$ for some set Ω, and $\mathcal{N}$ is a Mealy machine with all states reachable, inputs I and outputs $\hat{O} \subseteq (\Omega \times (\mathbb{N}^{>0} \cup \{\bot\})) \cup \{\mathsf{nil}\}$. Suppose $\mathsf{Mealy}(\mathcal{M}) \approx \mathcal{N}$. Then $\mathsf{MM1T}(\mathcal{N})$ is an MM1T and $\mathcal{M} \approx_{timed} \mathsf{MM1T}(\mathcal{N})$.

2.4.3 MAT Framework in MM1T Learning

To achieve the reduction, an adaptor is placed between the SUL (which behaves like MM1T $\mathcal{M}$) and the learner. From the perspective of the learner, the adaptor acts like $\mathsf{Mealy}(\mathcal{M})$. The learner then poses equivalence and membership queries like usual. To answer these queries, the adaptor translates them into timed queries.

Assume the learner wants to learn MM1T $\mathcal{M}$. It constructs a hypothesized Mealy $\mathcal{H}$ and poses two types of timed queries:

1. **Timed membership query (TMQ):** Learner supplies a sequence of inputs with precise timing. Teacher replies with timed words, including timeouts and the precise timing.
2. **Timed equivalence query (TEQ):** Learner asks whether the hypothesis $\mathcal{H}$ accepts the same timed words as $\mathcal{M}$ i.e., whether $\mathcal{M} \approx_{timed} \mathcal{H}$. Either of two answers is possible:
   (a) Teacher responds with `no` and provides a counterexample (timed word).

(b) Teacher responds with `yes` and the process terminates.

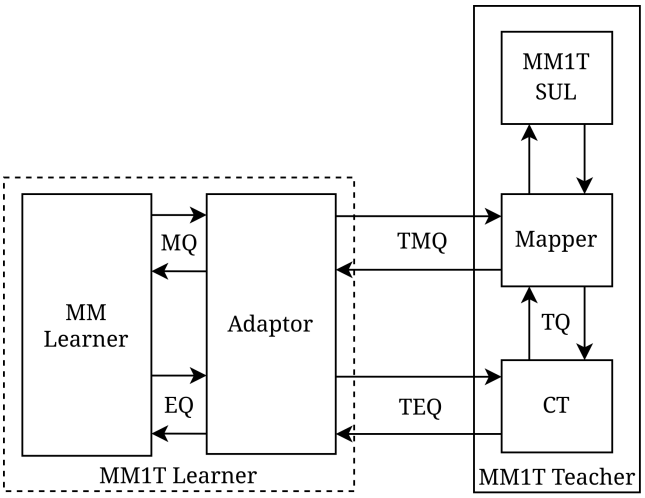

Figure 2.4: MAT framework for MM1T learning, image from [14]

.

Once the learner succeeded to learn Mealy $\mathcal{H}$ that is equivalent to $\mathsf{Mealy}(\mathcal{M})$, we know by *Theorem 1* that $\mathcal{M} \approx_{timed} \mathsf{MM1T}(\mathcal{H})$. This means that the learned MM1T is timed equivalent to $\mathcal{M}$. To look more closely at the theory and the exact implementation of the adaptor, we refer the reader to [14].

2.5 IEEE 802.11 4-Way Handshake

The 4-way handshake protocol is specified in Section 12.7.6 of the IEEE 802.11 standard [26]. Following this protocol, a client connects to an Access Point (AP) which provides a Wi-Fi connection.

2.5.1 Network Discovery

The client first needs to become aware of the network. This is accomplished either by sending out probe requests or by scanning beacon frames that are broadcast by the AP. Once a beacon frame or probe response is intercepted, the network becomes visible to the client.

Both frames contain information such as the AP's Service Set Identifier (SSID) and the Robust Security Network Element (RSNE). RSNE includes the supported cipher suites (CCMP and/or TKIP) and the version of the Wi-Fi Protected Access (WPA) in use.

2.5.2 Authentication and Association

Once the client chooses to connect to a network, it first needs to go through the authentication and association process. During authentication, (unless already installed) both ends generate the Pairwise Master Key (PMK) from the SSID and the network's passphrase. Note that despite the name, the real authentication only happens in the handshake. During association, the client chooses the cipher suite and the WPA version. Based on this choice, the AP either accepts or refuses the connection.

2.5.3 4-Way Handshake

After successful association, the 4-way handshake takes place. Here, the client and the AP authenticate each other and establish the encryption keys. The Pairwise Transient Key (PTK) is used for encrypting unicast traffic, whereas the Group Temporal Key (GTK) is used for encrypting multicast and broadcast traffic. These encryption keys are derived from master keys, namely the PMK and the Group Master Key (GMK). Once associated, the PMK resides on both stations so it does not need to be sent over.

The handshake consists of exchange of 4 messages encapsulated within EAPOL-Key frames:

- **Message 1**: AP sends its nonce, *ANonce*, to the client.
  - Client generates its nonce, *SNonce*, and derives the PTK from 5 values: SNonce, ANonce, both ends' MAC addresses, and the PMK.
- **Message 2**: Client responds with its nonce - SNonce. Message includes the Message Integrity Check (MIC) computed over the body of the EAPOL frame using the PTK.
  - AP derives the PTK as it now also knows the 5 necessary values for the computation.
  - AP verifies the MIC, and compares the RSNE to the one chosen during association.
  - AP generates a fresh random GMK.
  - AP computes the GTK from PTK, GMK and MIC.
- **Message 3**: AP sends the encrypted GTK to the client, along with the MIC and RSNE.
  - Client verifies the MIC and the RSNE.
  - Client extracts the GTK.

- **Message 4**: Client sends a confirmation to the AP that the keys have been installed. The frame is again protected by MIC.

Figure 2.5 visualizes the protocol together with the preceding network discovery, authentication and association stages.

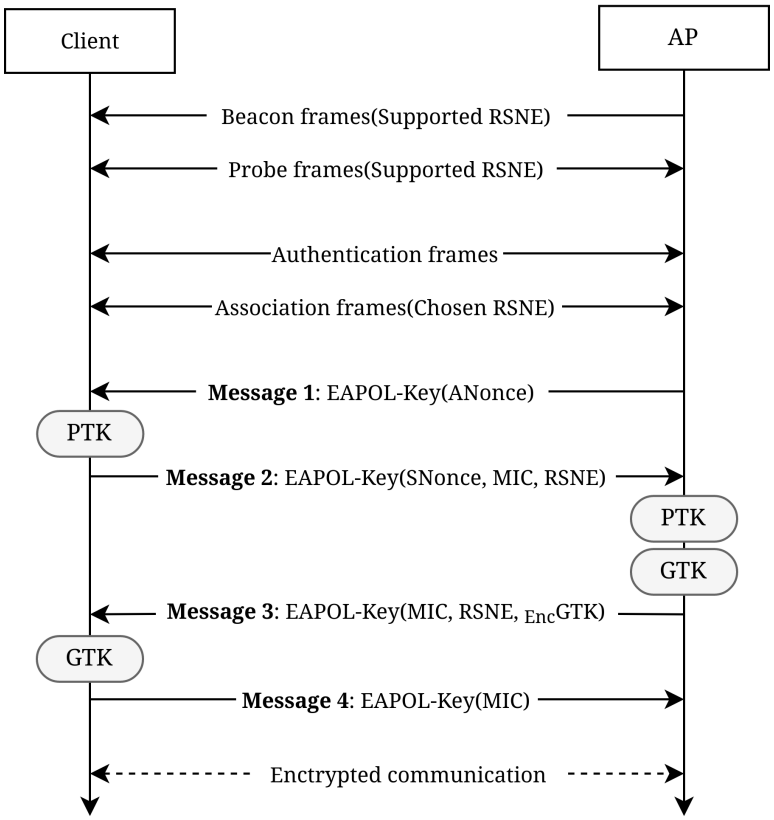

Figure 2.5: 4-way handshake

2.6 Wi-Fi Learner

2.6.1 Handling Time-Related Behavior

The Wi-Fi learner presented in [16] can handle the time-related behavior of the 4-way handshake while sustaining a reasonable complexity. This is achieved by separating learning into two stages.

To separate the stages, the modeler first defines two upper-bound timeout values - a small timeout TO_s and a big timeout TO_b. TO_s represents the normal response time i.e., the time frame in which packets are expected to arrive. This could be between 0.5 and 2 seconds. In case TO_s has passed,

we might expect retransmission. TO_b can be set to a higher value, such as 20 seconds. It prevents the learner from waiting endlessly in case there is no response from the AP. When TO_b is reached, the teacher concludes that the connection was dropped.

Following are the two learning stages:

- **Stage 1:** The learner aims to learn a base model which excludes any time-related behavior. Packets that pass TO_s are saved into a cache for stage 2. If retransmission is observed, all inputs are disabled, meaning learning of the given path is interrupted and the protocol is restarted.

  Equivalence checking is done through the $W-method$ [27]. This stage finishes rather quickly as the time-related behavior (which drastically increases query execution time) is ignored.

- **Stage 2:** Learning starts with posing queries from the cache obtained from stage 1. After this, learning continues per usual, allowing delay actions to explore time-related behavior.

  During equivalence checking, the learner sends from each state delay actions to explore all timeouts. The final model then undergoes the last round of equivalence checking.

The algorithm was designed under the assumption that there is only one clock in operation at all times. This is a reasonable assumption given the flow of the protocol. This assumption also invites the possibility of using the MM1T learner, as we will do in this research.

2.6.2 Mapper

A mapper is placed between the learner and the teacher. Abstract input symbols queried by the learner are translated into viable packets, which are then sent to the router (SUL). The response packets are captured by the mapper, translated into abstract output symbols, and sent to the learner. The abstract input/output alphabet is defined as follows:

- **Input**: $i \in I :=$ `MsgType(Params)` where `MsgType` is either of {`ASSOC, E2, E4, DATA, ENC_DATA, DELAY`} representing association request, EAPOL message 2, EAPOL message 4, unencrypted data, encrypted data and a delay action, respectively. These are the messages sent out by the client.

  The first three message types allow different `Params` values. These are defined in Table 2.1. This is the adjusted version of Table 1 from

[16] to only include the input tested in our research. `ASSOC` only allows for the `RSNE` parameter, whereas `E2` and `E4` allow for any.

- **Output**: $o \in O :=$ `MsgType(Params)|Timestamp` where `MsgType` is either of {`E1, E3, ENC_DATA_UNKNOWN, TIMEOUT, Deauth`} representing EAPOL message 1, EAPOL message 3, encrypted data, timeout (inactivity), and deauthentication, respectively. These are the messages sent out by the AP. `Timestamp` indicates the time elapsed since the last received message. The first two messages allow `Params` from Table 2.1.

The symbols `ENC_DATA` and `ENC_DATA_UNKNOWN` serve to confirm that the handshake was successful. That is, the client and AP can send encrypted data back and forth.

Parameter	Tag	Values	Description
Key descriptor	`KD`	`WPA1, WPA2`	Indicates the EAPOL Key type
Cipher suites	`CS`	`MD5, SHA1`	Ciphers and hash functions used for encrypting the Key Data field and calculating the MIC.
RSNE	`RSNE`	`cc, ct, tc, tt`	The chosen ciphersuite combination of TKIP (`t`) and CCMP (`c`) for the group and unicast traffic respectively

Table 2.1: Parameters of the input and output symbols

Chapter 3

Research

This chapter introduces our research, starting with a basic overview in Section 3.1. In Section 3.2 we describe how we set up the learners. We explain how the simulation was achieved in Section 3.3.

3.1 Overview

To test the usage of the MM1T learner for the 4-way handshake protocol, we utilize the Wi-Fi learner tool for two different reasons. One is to use the models learned with this tool to simulate the routers for MM1T learning. This allows us to facilitate our research without the need to re-implement the mapper (Section 2.6.2). The other is to draw comparisons between the two learners. Figure 3.1 visualizes the process.

3.2 Set up

The Wi-Fi learner tool together with documentation is provided online [1]. For the mapper, we use an older version (commit `6254930` on the provided GitHub repository) as this version is in line with the research paper [16]. Packet injection and sniffing are achieved using two external Wi-Fi interfaces. We provide more details on the hardware used in the Appendix (Section A.1).

We initialize the learner with L^* learning algorithm and a *W-method* equivalence oracle of maximum depth 1, as it was done in [16]. For the exact meaning of these terms, refer to LearnLib's documentation. We set TO_s to 2.0 seconds and TO_b to 6.0 seconds. This setting is agreeable to all tested routers. The input alphabet is: {`ASSOC(RSNE=cc)`, `E2(|KD=WPA2|RSNE=cc|CS=SHA1|)`, `E4(|KD=WPA2|CS=SHA1|)`, `DELAY`, `DATA`, `ENC_DATA` }.

[1] https://chrismcmstone.github.io/wifi-learner/

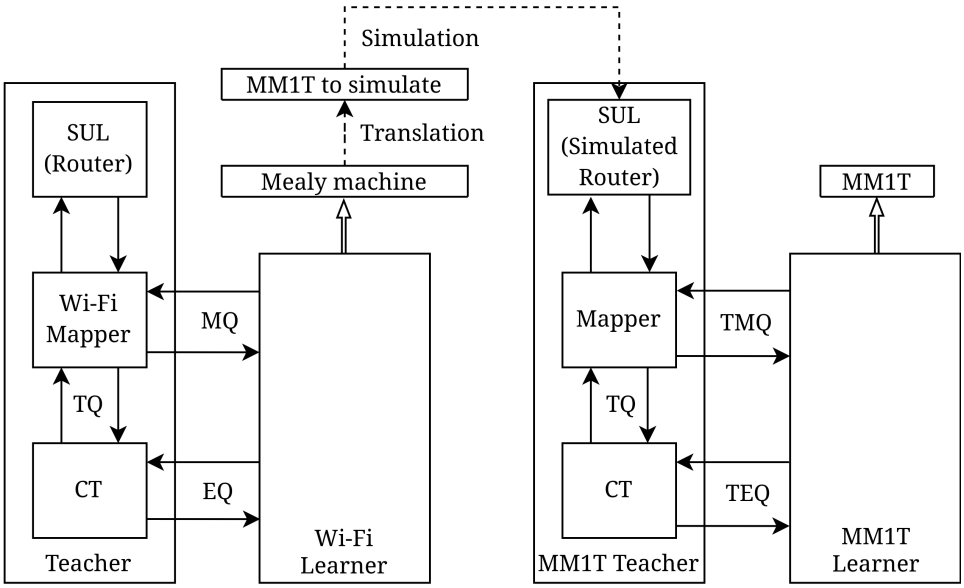

Figure 3.1: Experiment overview

The MM1T learner implementation (including the MM1T simulation) is also provided online [2]. We use the same learning algorithm and equivalence oracle as for the Wi-Fi learner.

3.3 Simulation

In order to simulate a router, we translate its Mealy representation into an MM1T representation following semantics accepted by the simulator. Both the output of the Wi-Fi learner and the input for the simulator is a `.dot` file. This file format serves for representing graphs through nodes, edges, and their corresponding labels. The translation between the two `.dot` files is achieved through rewriting the edge labels.

Although the Mealy machines acquired from the Wi-Fi learner do not contain clock values, they can be inferred implicitly from the model. The initial state does not set the timer because the protocol starts with an immediate action from the client (e.g., sending the association request). Moreover, for MM1Ts it holds by definition that $q_0 \in Q_{off}$. Thus, we remove the transition with `DELAY` input from the initial state. We note that apart from the initial state and the final state, each state has a timer on. That means every transition (re)sets the timer. This is visible through packet retransmissions or deauthentication frames after a delay, that are observed from every state.

[2] https://extgit.iaik.tugraz.at/scos/scos.sources/LearningMMTs

From the timestamp of these packets, we can tell what value the timer was set to in the origin state. In other words, what time delay triggered the retransmission or deauthentication. In case the delay triggers the `TIMEOUT` output, we set the timer in the previous transition to TO_s. This is because we expect the output `inactive` in the MM1T after TO_s seconds, so this is treated like an MM1T timeout. All transmissions towards the final state stop the timer.

All the timestamps are removed, as these are not part of the MM1T alphabet. The `TIMEOUT` output symbol of the Mealy alphabet is not related to `timeout` input symbol of the MM1T alphabet. `TIMEOUT` shows that no packet was received within TO_s seconds. To represent this behavior in an MM1T, we introduce a new output symbol `inactive`. The `timeout` input symbol replaces the `DELAY` input symbol since such input always results in a timeout in the Mealy models.

An overview of the difference in semantics is represented in Table 3.1.

Time behavior	**Translation**	**Mealy machine**	**MM1T simulation**	**MM1T model**
Delay input	Rewrite symbol	`DELAY`	`to[x]`	`timeout`
No response output	Rewrite symbol	`TIMEOUT`	`inactive`	`inactive`
Set timer	Infer from model	(implicit)	`x:=1000`	`set-timer(1.0s)`
Stop timer	Add to final transitions	(implicit)	`stop[x]`	(implicit)

Table 3.1: Rewriting rules from Mealy machine to MM1T simulation and the resulting MM1T symbols

To summarize: We rewrite the `DELAY` input into `to[x]` which will show as `timeout` in the final MM1T. We rewrite the `TIMEOUT` output into `inactive`. As previously mentioned, the timer values are inferred from the output after applying `DELAY`, and the corresponding timestamp. Setting the timer to, for example, 1 second is written as `x:=1000` and shows as `set-timer(1.0s)` in the MM1T. Lastly, we note that the final transitions stop the timer, for which we write `stop[x]`. This does not show in the learned MM1T as it is implicit.

Chapter 4

Results

In this chapter, we present our results. First, we discuss the Mealy models learned by the Wi-Fi learner in Section 4.1. Next, we relate the models to those obtained by the MM1T learner in Section 4.2. Finally, we compare the two learners in the number of queries in Section 4.3 and discuss the findings in Section 4.4.

4.1 Mealy Models

Using the Wi-Fi learner, we have obtained Mealy models of the 4-way handshake implementations on 6 different routers: three TP-Links (TL-WR841N, TL-WR940N, AC1750) a Netgear (WNR834B), and two ASUS routers (RT-N12D1, RT-AC66U).

To demonstrate the behavior of the handshake, we present the model of TP-Link TL-WR841N in Figure 4.1. The green transitions show the normal flow of the handshake. The red and orange transitions demonstrate that the router responds with deauthentication frames in case unexpected packets are received or it ignores such packets. The blue transitions demonstrate 2 retransmissions of EAPOL message 1 in 1-second intervals until the router gives up and sends a deauthentication frame. Similarly, the purple transitions show 3 retransmissions of EAPLOL message 3 in 1-second intervals.

To see the models of the remaining routers, see Appendix section A.2. We note that the ASUS routers have equivalent models, thus also the same implementation. The remaining models differ from each other in small detail. For example, compared to the TP-Link discussed above, Netgear has a much more minimal implementation as it only retransmits EAPOL messages once. However, the most minimal implementation can be seen in AC1750 as it does not make use of deauthentication frames - it rather ignores unexpected packets.

Similar implementation differences can be seen across all models, but in all cases, we only notice the desired behavior. This might be different if we used the Wi-Fi learner with the full input alphabet which deliberately tests the security of the handshake implementation, but this is out of the scope of our research.

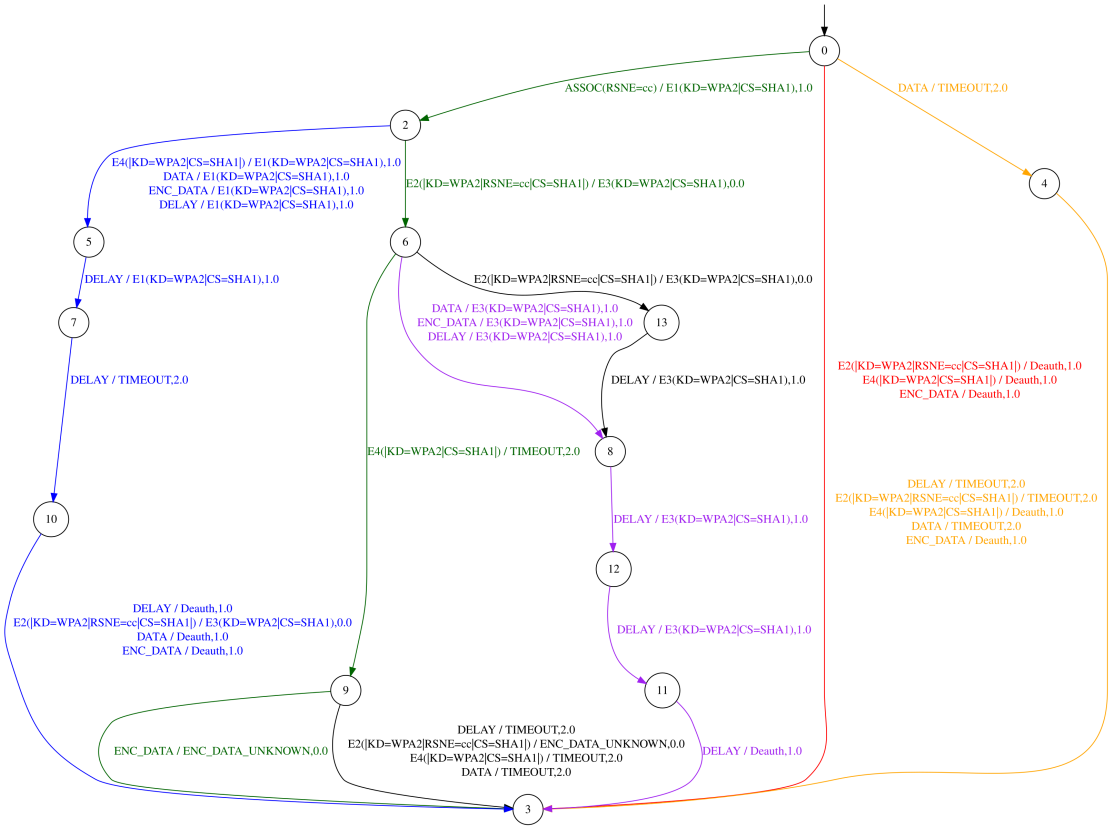

Figure 4.1: Mealy model of TP-Link TL-WR841N

Apart from AC1750 and TL-WR940N, we were able to verify that the handshake was successful on all routers. This can be seen in the final transition where encrypted data is sent back and forth after the AP received EAPOL message 4. We noticed that AC1750 does not support CCMP ciphersuite for group traffic, so we adjusted the alphabet to reach the normal flow of the handshake.

4.2 MM1T Models

We successfully learned MM1T models of all simulated routers. We verify that the obtained models are correct as they describe the same behavior present in their Mealy versions, plus they are equivalent to their simulated SULs. We present an example of TP-Link TL-WR841N in Figure 4.2.

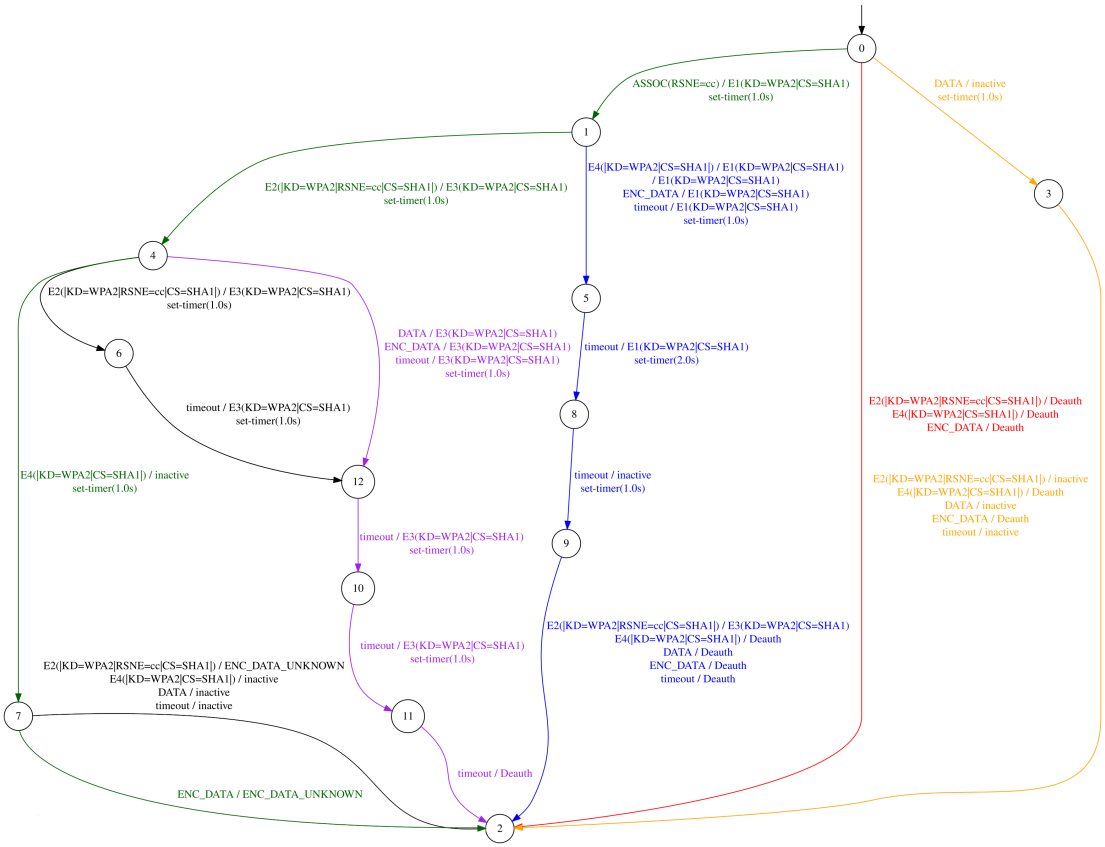

Figure 4.2: MM1T model of TP-Link TL-WR841N

We highlight the same transitions as in Figure 4.1. The time-related behavior is now represented through timers and is equivalent to that of the Mealy version.

4.3 Comparison

We provide comparisons of the number of queries posed to the SUL by the two learners. We ran the Wi-Fi learner 10 times per router and averaged the number of queries. The average standard deviation is 198 - the number

of queries per run differs by 14% percent on average. The MM1T learner was only run once per router, as the number of queries was the same for every run. We note that we used a maximum depth of 2 for the equivalence oracle while learning TL-WR841N, which influenced the number of queries used. This setting was necessary for the MM1T learner to reach a correct model with all states and transitions.

The numbers are summarized in the table below. For each router, we list the number of states in the model, the average number of membership queries, equivalence queries and total queries posed by the Wi-Fi learner (excluding error correction), and the number of membership queries, equivalence queries and total queries posed by the MM1T learner.

		Wi-Fi learner			**MM1T learner**		
Router	**#S**	**#MQ**	**#EQ**	**Σ**	**#MQ**	**#EQ**	**Σ**
TL-WR841N	13	1161	10740	11901	764	24414	25178
TL-WR940N	11	985	1448	2433	535	2616	3151
AC1750	10	507	1332	1839	426	1260	1686
WNR834B	10	608	1303	1911	535	2831	3366
RT-N12D1	10	678	1356	2034	535	3041	3576
RT-AC66U	10	722	1298	2020	535	3041	3576

Table 4.1: Comparison of the number of queries posed by each learner

4.4 Discussion

Although the resulting numbers are rather comparable, we observe that on average the Wi-Fi learner outperforms the MM1T learner.

It is important to note that the Wi-Fi learner deals with a non-deterministic SUL and performs error correction. Taking numbers from [16], the error rate on average accounts for 3% of all queries. We do not expect our error rate to be any higher, as we use a reasonably reliable network. Including error correction would bring the number of queries slightly closer but not enough to close the gap.

The Wi-Fi learner implements WPA-specific optimizations which gives it an advantage over the MM1T learner. For example, the protocol gets restarted upon receiving a deauthentication frame, meaning further queries are disabled. The MM1T learner would not be aware of this behavior and would try more queries after receiving the `Deauth` output.

Wi-Fi learner further uses a cache as a lookahead oracle. E.g., knowing that input sequence {`E2, ASSOC`} results in output disabling, it can look this up in the cache to conclude that the sequence {`E2, ASSOC, E2`} would have the same effect. As it was mentioned in [16], this optimization was responsible for greatly reducing the number of queries, as most queries could be looked up in the cache instead of being posed to the SUL.

We regard the two-stage learning as another potential efficiency-improving factor for the Wi-Fi learner. It was already discussed in Section 2.6.1 that such a solution reduces learning time, but we also suspect it might have reduced the number of queries needed. We have seen in previous work [28] that using a *subalphabet* before learning the full model has indeed decreased the number of queries in learning.

Even though the Wi-Fi learner is optimized and tailored specifically for the 4-way handshake, the (general) MM1T learner could compete against it. We conclude that the MM1T learner does fulfill the potential to learn the 4-way handshake protocol efficiently. We note that the learner has also outperformed other approaches ([13] and [29]) to learn time-related behavior on different benchmarks. Refer to [14] for exact results.

Chapter 5

Related Work

So far, there has not been further research into learning MM1Ts. General research into learning time-related behavior has been covered in Section 1.2. This chapter thus focuses on work related to verification of the 4-way handshake, some of which uses similar concepts as in our research.

First, we would like to point attention to a grey-box learning tool, *STATEINSPECTOR* [30]. As the authors point out, black-box learning can only describe how the SUL interacts with the outside world. Since we have no insight into how the SUL processes input, we cannot further optimize queries. *STATEINSPECTOR* overcomes these issues through observations of runtime memory and open-source code. The tool was used on different 4-way handshake implementations and discovered a high-impact vulnerability in one of them.

He et al. used a finite-state verification tool *Murφ* to analyze the handshake [31]. Provided a state machine, the tool exhaustively searches all execution sequences. Rather than learning a model automatically and checking it manually, the model is created manually and the program explores all the paths automatically. Using this approach, the authors found and analyzed an effective DoS attack on EAPOL message 1 in the 4-way handshake protocol.

Vanhoef et al. used a so-called *model based testing technique* [32]. First, they define a model representing a normal flow of the handshake. They use this model to automatically generate tests attacking possible vulnerabilities in the protocol, either through corrupting a certain parameter or the whole message. Since all tested implementations showed to be the same, the authors discovered new fingerprinting techniques. We note that this work efficiently verifies handshake implementations without deliberate use of model learning.

Tamarin is yet another verification tool that was used on handshake implementations [33]. The authors create a state machine from a subset of functionalities to be tested and run the *Tamarin* prover to automatically analyze a variety of attacks on the handshake. They discovered some models to be vulnerable to KRACK attacks. These models are in line with the official specifications, which hints that some properties in the standard are insufficient. Again, this research uses the concept of state machines but no active learning is present.

Chapter 6

Conclusions

This research investigated the potential use of the MM1T learner for the 4-way handshake protocol. We compared this learner to the efficient two-stage Wi-Fi learner in the number of queries posed to the SUL. We observed that the MM1T learner is a potential competitor to the Wi-Fi learner.

We note that using simulated routers in our work is a limitation, as it can only approximate the real-life behavior of the SUL. Implementing the MM1T learner to interact with a real router would be an improvement to our research. Such learner could then also implement further WPA-specific optimizations to improve complexity.

Our work tests the learners using one setup only, namely the L^* learning algorithm and the *W-method* equivalence oracle. We recognize this as another limitation since the results might differ given another setup. Future work could put the learners through additional testing with various setups.

Since the MM1T learner succeeded in learning the handshake efficiently, it invites the idea of using the learner for further case studies. It could be then compared to existing model-learning implementations (e.g., [2], [4], [5]) or it could be used for protocols which were not investigated in model-learning research thus far.

We also recognize the two-stage learning process as a significant contribution to the research. We suggest that this technique be explored further, for example, by applying it in other case studies.

Finally, we propose combining the MM1T learner with a general two-stage learner. Since both learners perform well on their own, it would be interesting to investigate the impact of combining the two concepts into a general, two-stage MM1T learner.

Bibliography

[1] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 2:75, pp. 87–106, 1987.

[2] J. De Ruiter and E. Poll, "Protocol State Fuzzing of TLS Implementations," in *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, (USA), p. 193–206, USENIX Association, 2015.

[3] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS Implementations Using Protocol State Fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2523–2540, USENIX Association, Aug. 2020.

[4] P. Fiterau-Brostean and F. Howar, "Learning-Based Testing the Sliding Window Behavior of TCP Implementations," in *FMICS-AVoCS*, 2017.

[5] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "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, (New York, NY, USA), p. 142–151, Association for Computing Machinery, 2017.

[6] L. Feng, S. Lundmark, K. Meinke, F. Niu, M. A. Sindhu, and P. Y. H. Wong, "Case Studies in Learning-based Testing," in *Testing Software and Systems* (H. Yenigün, C. Yilmaz, and A. Ulrich, eds.), (Berlin, Heidelberg), pp. 164–179, Springer Berlin Heidelberg, 2013.

[7] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report," in *Integrated Formal Methods*, (Cham), pp. 311–325, Springer International Publishing, 2016.

[8] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining Model Learning and Model Checking to Analyze TCP Implementations," in *Computer Aided Verification* (S. Chaudhuri and A. Farzan, eds.), (Cham), pp. 454–471, Springer International Publishing, 2016.

[9] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[10] O. Grinchtein, B. Jonsson, and M. Leucker, "Learning of Event-Recording Automata," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* (Y. Lakhnech and S. Yovine, eds.), (Berlin, Heidelberg), pp. 379–395, Springer Berlin Heidelberg, 2004.

[11] O. Grinchtein, B. Jonsson, and P. Pettersson, "Inference of Event-Recording Automata Using Timed Decision Trees," in *CONCUR 2006 – Concurrency Theory* (C. Baier and H. Hermanns, eds.), (Berlin, Heidelberg), pp. 435–449, Springer Berlin Heidelberg, 2006.

[12] L. Henry, T. Jéron, and N. Markey, "Active Learning of Timed Automata with Unobservable Resets," in *Formal Modeling and Analysis of Timed Systems* (N. Bertrand and N. Jansen, eds.), (Cham), pp. 144–160, Springer International Publishing, 2020.

[13] J. An, M. Chen, B. Zhan, N. Zhan, and M. Zhang, "Learning One-Clock Timed Automata," in *Tools and Algorithms for the Construction and Analysis of Systems* (A. Biere and D. Parker, eds.), (Cham), pp. 444–462, Springer International Publishing, 2020.

[14] F. Vaandrager, R. Bloem, and M. Ebrahimi, "Learning Mealy Machines with One Timer," in *Language and Automata Theory and Applications*, (Cham), pp. 157–170, Springer International Publishing, 2021.

[15] M. Isberner, F. Howar, and B. Steffen, "The Open-Source LearnLib: A Framework for Active Automata Learning," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), (Cham), pp. 487–495, Springer International Publishing, 2015.

[16] C. McMahon Stone, T. Chothia, and J. de Ruiter, "Extending Automated Protocol State Learning for the 802.11 4-Way Handshake," in *Computer Security* (J. Lopez, J. Zhou, and M. Soriano, eds.), (Cham), pp. 325–345, Springer International Publishing, 2018.

[17] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[18] B. Steffen, F. Howar, and M. Merten, *Introduction to Active Automata Learning from a Practical Perspective*, pp. 256–296. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[19] D. Peled, M. Vardi, and M. Yannakakis, "Black Box Checking," *Journal of Automata, Languages and Combinatorics*, vol. 7, pp. 225–246, 01 2002.

[20] A. Groce, D. Peled, and M. Yannakakis, "Adaptive Model Checking," *Logic Journal of IGPL*, vol. 14, 10 2002.

[21] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[22] F. Vaandrager, "Model Learning," *Commun. ACM*, vol. 60, p. 86–95, jan 2017.

[23] S. Ali, H. Sun, and Y. Zhao, "Model learning: a survey of foundations, tools and applications," *Frontiers of Computer Science*, vol. 15, pp. 1–22, 2021.

[24] M. Shahbaz and R. Groz, "Inferring Mealy Machines," in *FM 2009: Formal Methods* (A. Cavalcanti and D. R. Dams, eds.), (Berlin, Heidelberg), pp. 207–222, Springer Berlin Heidelberg, 2009.

[25] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), (Cham), pp. 307–322, Springer International Publishing, 2014.

[26] "IEEE Standard for Information technology: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pp. 1–3534, 2016.

[27] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, 1978.

[28] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying Automata Learning to Embedded Control Software," in *Formal Methods and Software Engineering* (M. Butler, S. Conchon, and F. Zaïdi, eds.), (Cham), pp. 67–83, Springer International Publishing, 2015.

[29] B. K. Aichernig, A. Pferscher, and M. Tappler, "From passive to active: Learning timed automata efficiently," in *NASA Formal Methods* (R. Lee, S. Jha, A. Mavridou, and D. Giannakopoulou, eds.), (Cham), pp. 1–19, Springer International Publishing, 2020.

[30] C. M. Stone, S. L. Thomas, M. Vanhoef, J. Henderson, N. Bailluet, and T. Chothia, "The Closer You Look, The More You Learn: A Grey-box Approach to Protocol State Machine Learning," *ArXiv*, vol. abs/2106.02623, 2021.

[31] C. He and J. C. Mitchell, "Analysis of the 802.11i 4-Way Handshake," in *Proceedings of the 3rd ACM Workshop on Wireless Security*, WiSe '04, (New York, NY, USA), p. 43–50, Association for Computing Machinery, 2004.

[32] M. Vanhoef, D. Schepers, and F. Piessens, "Discovering Logical Vulnerabilities in the Wi-Fi Handshake Using Model-Based Testing," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, (New York, NY, USA), p. 360–371, Association for Computing Machinery, 2017.

[33] R. R. Singh, J. Moreira, T. Chothia, and M. D. Ryan, "Modelling of 802.11 4-Way Handshake Attacks and Analysis of Security Properties," in *Security and Trust Management* (K. Markantonakis and M. Petrocchi, eds.), (Cham), pp. 3–21, Springer International Publishing, 2020.

Appendix A

Appendix

A.1 Hardware

To run the Wi-Fi learner, we need two external wireless interfaces which support monitor mode: one for packet injection, the other for packet sniffing. The details on the interfaces used in our research and some useful commands are provided in the tables below:

Type of Interface	**Name**	**Driver**	**Installation**
Sniffing	Rolio Wifi Adapter	rtl88x2bu	manual
Injecting	ASUS USB-N13	rtl8192cu	PnP
Injecting	Sitecom 300N WL-302	rt2800usb	PnP

Table A.1: Wireless interfaces used in the experiment

Command	**Effect**
`airmon-ng`	shows each interface, its driver and chipset
`iw <iface> set <mode>`	changes the mode of the interface
`iw dev`	shows information about each interface, including the mode
`airodump-ng <iface>`	performs packet sniffing (useful to check that this is supported)
`aireplay-ng --test <iface>`	performs packet injection (useful to check that this is supported)

Table A.2: Useful commands for working with interfaces

A.2 Mealy Models

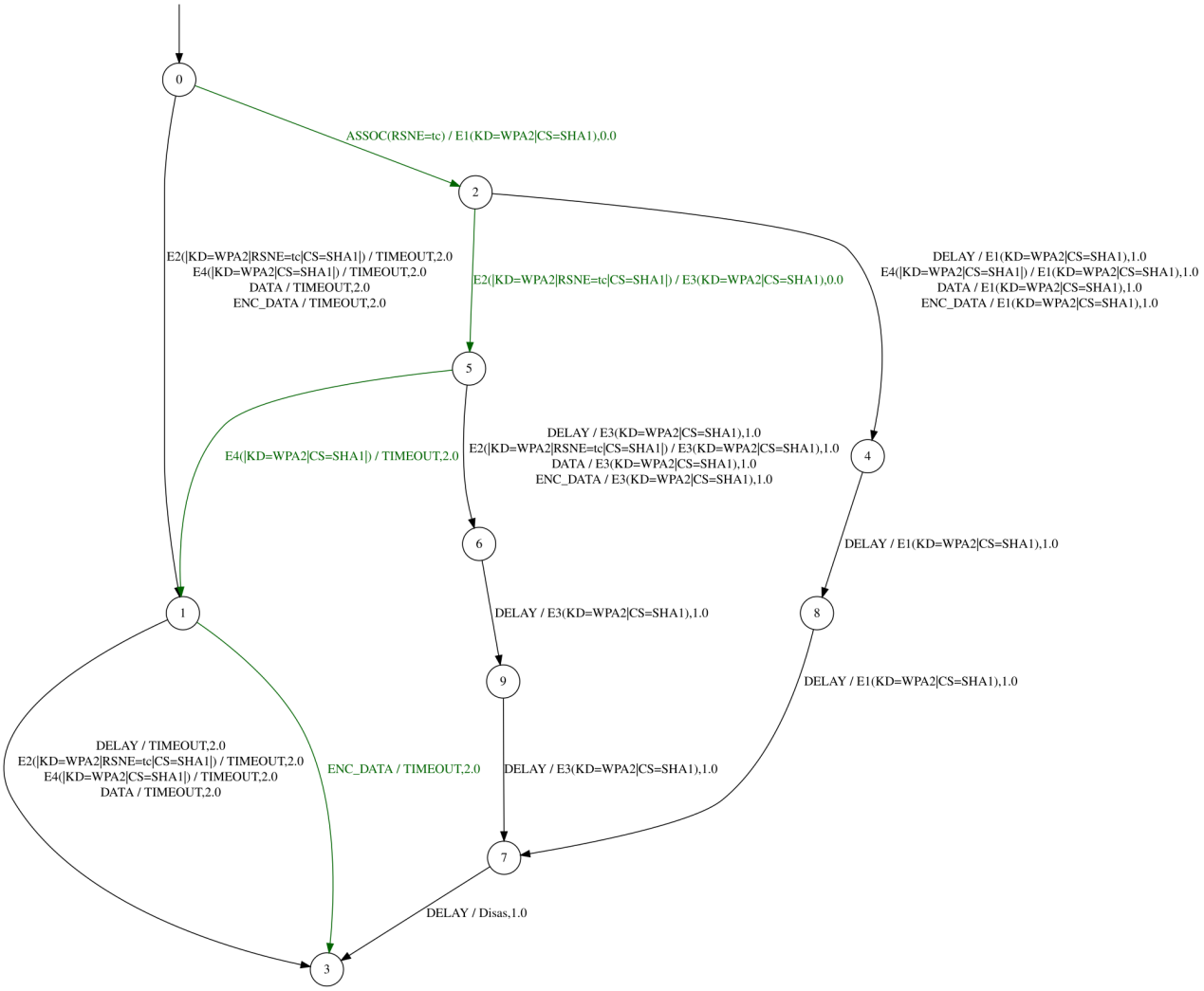

Figure A.1: TP-Link AC1750

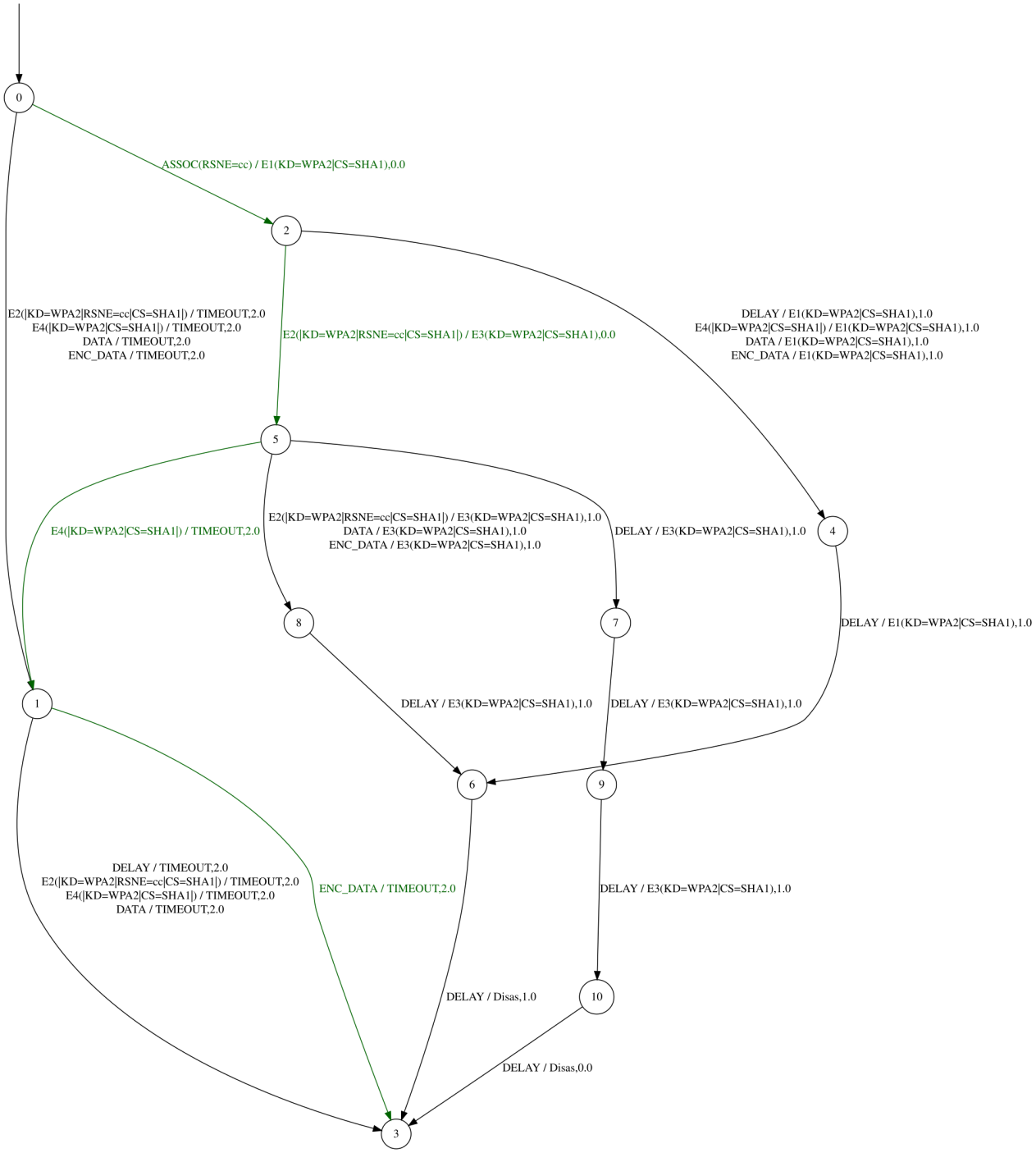

Figure A.2: TP-Link TL-WR940N

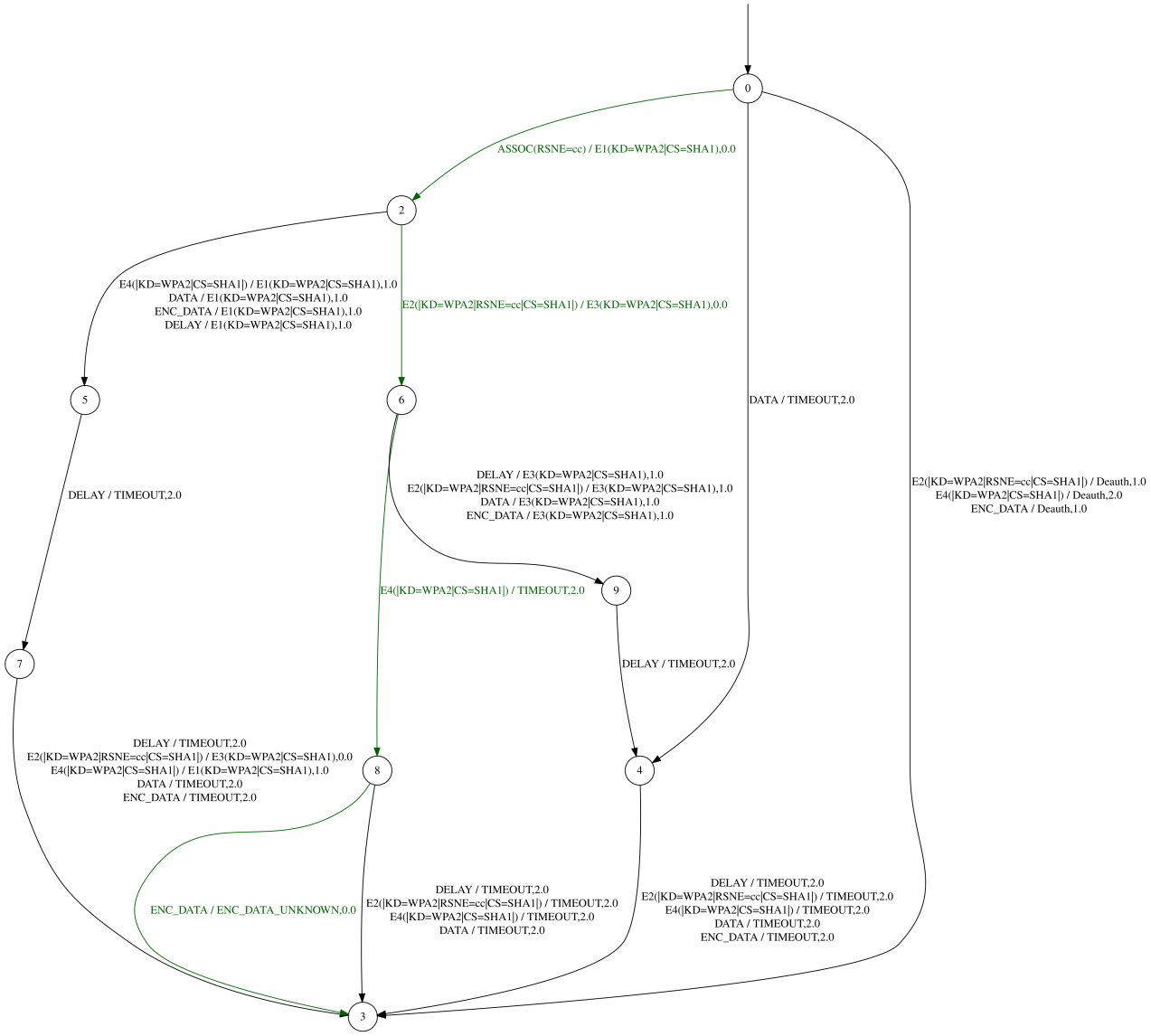

Figure A.3: Netgear WNR834B

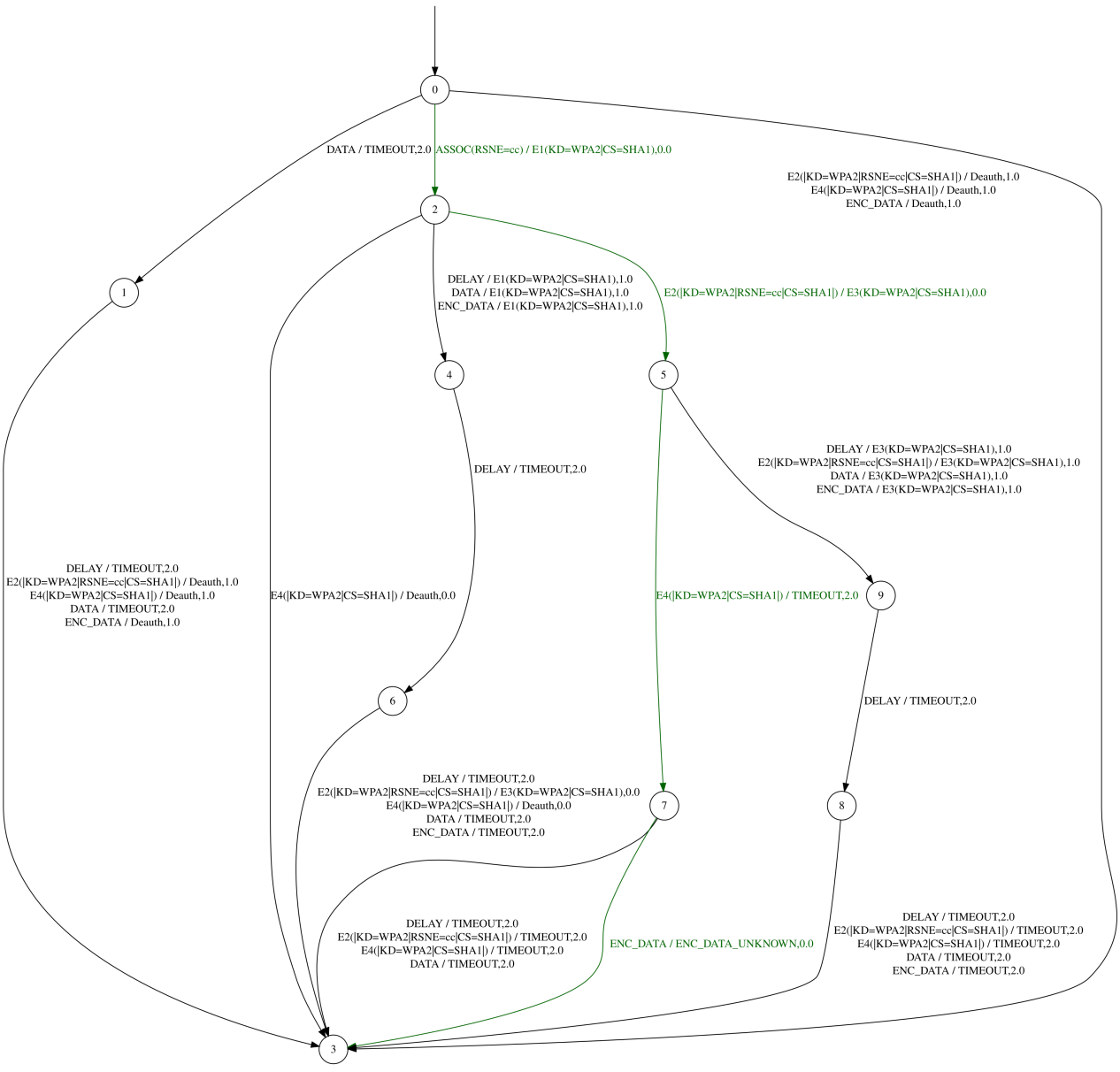

Figure A.4: ASUS RT-N12D1 and ASUS RT-AC66U