Bachelor thesis
Computer Science

Radboud University

State machine inference of Thread networking protocol

Author:
Bart van den Boom
S4382218

First supervisor/assessor:
Dr. ir. Joeri de Ruiter
joeri@cs.ru.nl

Second supervisor:
Dr. Ilya Kihzvatov
I.Kizhvatov@science.ru.nl
Second assessor:
Dr. ir. Erik Poll
e.poll@cs.ru.nl

June 29, 2018

Abstract

Thread is a recently deployed communication protocol stack specifically designed for home automation. Part of the protocol stack is Mesh Link Establishment (MLE), which is extended for Thread. The aim of this paper is to obtain useful insights in the behaviour of the MLE implementation of OpenThread. To do so, we inferred state machines by using active Mealy machine inference. These behavioural models were then compared to the specification as given by Thread, resulting in the discovery of anomalies present in the OpenThread implementation.

Contents

Chapter 1

Introduction

The internet of things (IoT) is on the rise. According to a study by Gartner, the estimated number of connected devices will rise to as much as 26 billion by 2020 [20]. The internet of things is defined as "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols" [13]. These objects or end-devices all have sensing, actuating, processing and communication capabilities. IoT devices range from Radio-frequency identification (RFID) tags, mobile phones to sensors, actuators and such used in homes and industry.
IoT provides a wide range of potential applications, for example wireless sensor networks (WSN) which typically consist of a large number of low-cost, low-power multifunctional nodes with sensing and data processing capabilities. According to a research conducted in 2001 [2], WSNs could possibly aid in military, healthcare, security, environmental and home use-cases.
IoT devices are usually connected through cellular technologies such as 3G/4G or using a gateway to form a local area network to connect to the internet [14]. This connectivity to the internet makes the IoT architecture susceptible to attacks and other security drawbacks which come with any internet connected device. Additionally, data collected and processed by IoT devices such as the energy usage profile of a household or medical data is regarded sensitive, thus it is essential that communication and operation happens over a secure and reliable channel.
Some questions regarding these security aspects remain unanswered because IoT devices are built upon different, sometimes proprietary communication standards used to connect to the internet, which in itself is regarded a hazardous environment. Although security should always be a priority, whether this is the case is an open issue due to the nature of these devices, which have limited wireless range and storage ability due to their low manufacturing cost and design which aims to have the longest possible battery life.
An example of a security flaw in IoT technology is brought to light by academic research on the ZigBee IoT protocol targeted at Philips Hue lights

[21]. This paper showed the large scale effects which go along with faulty implementations of IoT protocols. In this attack, attackers were able to spread a computer virus, a worm, through a large number of interconnected Philips Hue lights and take control of the device and cause irreversible damage.

Numerous protocols like Bluetooth, BLE, ZigBee, Z-Wave and 6LoWPAN have been proposed [23] to standardise the communication of IoT devices, but have never seen widespread adoption or serve a niche market. The latest development is the Thread networking protocol stack [25]. Thread is used in a specific type of network topology called a mesh network which is described in Section 2.1 and is based on existing communication standards, making it compatible with all hardware supporting IEEE 802.15.4 wireless technology. OpenThread [17] is an open-source implementation of the Thread protocol stack which will be the focus of this thesis.

The OpenThread implementation is still subject to changes and testing by the OpenThread community. Testing is done in the form of continuous integration (i.e., automatically running a set of tests on various platforms whenever a new build is ready). Another form of testing conducted by the OpenThread community is fuzz testing, in which the implementation is tested against randomly formed network packets to possibly cause unwanted behaviour such as memory leaks.

The goal of this thesis is to obtain useful insights in the workings of the OpenThread implementation and shed a light on possible security threats. We will accomplish this by deriving a behavioural model of the Mesh Link Establishment protocol (see Section 2.2.2) present in OpenThread. A method called state machine inference will be used to derive a model of the MLE protocol implementation. In this approach, the OpenThread implementation of MLE will act as a black box and by observing and manipulating network traffic, the underlying model can be constructed. We will then use this model to see how strictly it follows the specification as given by Thread [25]. The way OpenThread implements the MLE protocol has not been analysed using this technique and could possibly reveal anomalies in regard to the specification, or worse, security threads.

The L* learning algorithm forms the basis to derive the state machine. The algorithm makes use of an alphabet which abstracts all MLE specific messages. A mapper will need to be created to translate the alphabet into MLE-specific messages and vice versa. We will explain the relevant background information of Thread in Chapter 2 and the required knowledge of the techniques used in Chapter 3. After that we will discuss the modelling scope and implementation specifics in Chapter 4. Lastly the derived model will be analysed in Chapter 5 and conclusions will be drawn in Chapter 6.

Chapter 2

Thread

Thread is a recently deployed networking-standard meant for IoT devices, especially home automation. Developed by Google's Nest, it's purpose is to standardize device-to-device communication of IoT devices. Already existing wireless technologies such as Z-Wave and EnOcean fail to deliver the low power, resilience, IP-based and secure operation that Thread brings. The desire to interconnect IoT devices which speak different protocols is fulfilled by the Thread protocol stack by combining the best of current systems.
Thread brings IPv6 capabilities to IoT devices through the 6LoWPAN standard and is built on existing hardware supporting the IEEE 802.15.4 wireless standard which defines the operation of low-rate wireless personal area networks. Zigbee, a comparable network-standard also delivers IPv6 via 6LoWPAN but unlike Thread, it is not IPv6 addressable, which enables Thread to have cloud access. Also included is support for various network interfaces, like Bluetooth and Wi-Fi.
Thread implements the network and transport layer of the OSI-model [26] and thus forms a base on which applications can be built. The Thread-protocol is used in a mesh-network (see Section 2.1) that allows nodes to self-configure and fix routing problems. It partly accomplishes this by letting nodes inside a network change it's role accordingly to its physical topology. It must be noted that the role doesn't specify the physical capabilities of the device but the virtual state. The physical capabilities are specified by two types of Thread devices:

1. **Full Thread Device (FTD)**:
   Which can have any of the following roles:

   (a) **Leader**: Only one leader can be active in a Thread network at a time, the leader assigns router addresses and allows new router requests.

   (b) **Router**: Provides routing services and permits devices to join the Personal Area Network (PAN), routers may never be a sleepy device.

   (c) **Border Router**: A specific type of router that forms the connection between IEEE 802.14.5 PAN and other interfaces like Wi-Fi or Ethernet. A Thread network typically contains one or more border routers.
   (d) **Router-Eligible End Device (REED)**: REEDS are devices which have the capability of becoming a router, but due to network topology or conditions these devices are not acting as routers.
   (e) **Full End Device (FED)**: Normal Thread end device, only sending and requesting data via its parent node, which can be a REED or Router.

2. **Minimal Thread Device (MTD)**: This type of Thread device is meant for devices with limited memory capabilities.
   (a) **Minimal End Device(MED)**: Devices that do not have any routing capabilities and must communicate with their parent node.
   (b) **Sleepy End Device**: Devices that do not have any routing capabilities and must communicate with their parent node. Additionally these devices are in a sleep-state most of the time to save on battery consumption.

2.1 Mesh networks

IoT networks using WiFi typically use a star topology, where a single router provides a connection between all connected devices. Other network technologies like Bluetooth use a Point-to-Point topology. Thread networks use a mesh network topology, where every node is connected to as many other nodes as possible to efficiently route data throughout the network, thus forming a "mesh". Each topology having their own advantages and disadvantages; Thread is mostly targeted at sensor networks used in the home and thus, most suited a mesh topology.
Mesh networks dynamically self configure and fix routing problems and thus make radio systems more reliable by allowing nodes to have routing capabilities, dramatically increasing the range in which sensor nodes can be placed. This is very useful because of the characteristics and limitations in terms of wireless range of the IEEE 802.15.4 physical layer which Thread is built on. The topology of a typical Thread network can be seen in Figure 2.1.

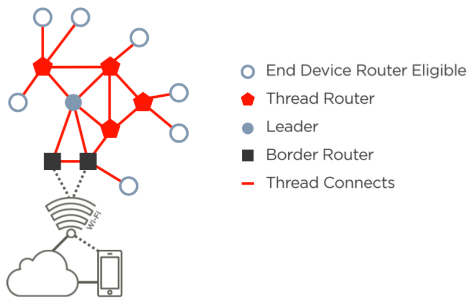


Figure 2.1: Mesh topology of Thread [10]

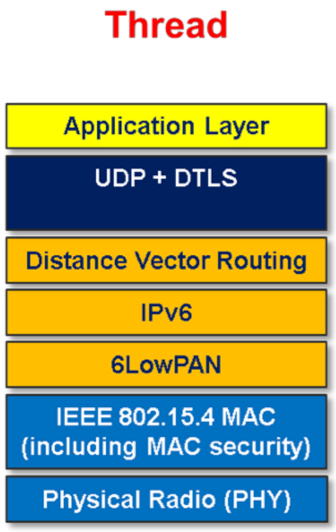


Figure 2.2: Thread Stack

2.2 Thread stack

IEEE 802.15.4, which Thread is built on is a technical standard which defines the operation of low-rate wireless personal area networks (LR-WPAN). The emphasis lies in extremely low manufacturing cost and technological simplicity. As mentioned earlier, Thread operates below the application layer and implements the Transport and Network layer of the OSI model [26]. IEEE 802.15.4 MAC (Medium Access Control) is used for message handling and congestion control. This layer implements CSMA (Carrier-Sense Multiple Access) to verify the absence of network packets on a channel before transmitting to that channel, frame retries and acknowledgement frames to ensure reliable communication. MAC security [1] functionality is used to provide integrity and confidentiality to messages at higher layers of the software stack. ICMP is supported for error messages and UDP is used for delivering IP packets between devices. DTLS (TLS over UDP) is used for authenticating a joining (untrusted) device. Lastly the Constrained Application Protocol (CoAP) is used as a replacement for HTTP for communication with the application which is built on Thread.

[1]To avoid confusion with the Message Authentication Code security principle, also abbreviated with MAC, "MAC security" from now on refers to the security suite present in IEEE 802.15.4 MAC.

2.2.1 IPv6

Historically, IoT devices have used a constrained proprietary version of IP for its communication due to the low memory footprint and computational capabilities of the devices. Thread however has native IPv6 connectivity and addressability which enabled devices to access services in the cloud, or interact with the user through Thread mobile applications. Additionally, it makes it more straightforward for developers to create applications because IPv6 is the standard for communication between all devices connected to the internet.

6LoWPAN

To deliver IP connectivity to a constrained Thread device, the 6LoWPAN adaptation layer is used. This is needed because of the minimum MTU (Maximum Transmission Unit) of IPv6, which is 1280 bytes and IEEE 802.15.4 only supports a maximum physical packet size of 127 bytes. Moreover, IEEE 802.15.4 links are usually error-prone. To overcome these problems 6LoWPAN uses several techniques such as header compression and packet fragmentation. Another important feature of 6LoWPAN is link-layer forwarding. This is a way of forwarding packets in a mesh network with very small overhead. 6LoWPAN, as an adaptation layer, sits between layer 2 and 3 of the OSI-model.

2.2.2 MeshCoP

The MeshCop protocol is used for commissioning new devices. Commissioning is the process of adding an untrusted radio device to a network. Thread uses the MeshCoP (Mesh Commissioning Protocol) protocol to do so. The process is invoked by a human administrator, authenticating a new device to the Thread network as eligible for joining, followed by the commissioning of the device with the network wide master Pre Shared Key (PSK). After the commissioning process, the untrusted device is authenticated and has knowledge of the network wide parameters over a secure channel [25].

2.2.3 MLE

In IEEE 802.15.4 networks, it is not uncommon to find links between devices which are fast and reliable one way and slow and lossy the other way. This is called an asymmetrical link. To configure links in a mesh network it is necessary to know if the link is reliable, Mesh Link Establishment (MLE) therefore, allows a node to broadcast the quality of a wireless link before configuring that link. MLE also allows nodes to periodically synchronize with other nodes, sharing network parameters to adapt to any changes in the topology, detect neighbouring devices and provide a secure MLE session,

on top of MAC security. This session is then used to configure changes in roles and topology [25].
MLE is extended for use with Thread and operates alongside the UDP layer and thus can't really be placed in the OSI-model. Because of its importance in the core operation of Thread and more specifically OpenThread, the implementation of MLE will be the focus of this study.

2.2.4 Security

Thread makes use of a network-wide key from which derived keys are used for MAC security and encryption of MLE messages. The IEEE 802.15.4 MAC-2006 security suite, which uses a 128-bit AES block cipher in CCM mode is used for encrypting link layer packets. When an untrusted device joins a Thread network a DTLS handshake is performed. During the joining of new devices and key agreement an elliptic curve variant of J-PAKE (Password Authenticated Key Exchange) using the NIST P-256 elliptic curve is used. EC-JPAKE is based on elliptic curve Diffie-Hellman for key agreement and Schnorr signatures as a Non-interactive Zero-knowledge proof to authenticate two peers and to ensure a shared secret between them based on a chosen passphrase. A (D)TLS handshake is used for EC-JPAKE to make the key exchange suitable for the unsecure UDP communication channel which is used by Thread.

Chapter 3

Automated state machine learning

This chapter will cover all required information regarding techniques, algorithms and the notion of a Mealy machine used in this thesis.

3.1 Mealy machines

Mealy machines will be used to represent the state machine of the System Under Test (SUT). Mealy machines very similar to finite state automata, but instead of only accepting a certain input string, an output string is given when a state transition is triggered. This means Mealy machines are very suited for describing the behaviour of a communication protocol because they have states which correspond to those in a protocol and transitions between states combined with input and output messages which behave deterministically and depend on the previous state and input just like in a protocol.
Mathematically a Mealy machine is a tuple $\mathcal{M} = \langle \mathcal{I}, \mathcal{O}, \mathcal{Q}, q_0, \rightarrow \rangle$, where

1. $\mathcal{I}, \mathcal{O}$ and $\mathcal{Q}$ are non-empty sets *input alphabet*, *output alphabet*, and *states*, respectively,
2. $q_0 \in \mathcal{Q}$ is the *initial state*, and
3. $\rightarrow \subseteq \mathcal{Q} \times \mathcal{I} \times \mathcal{O} \times \mathcal{Q}$ is the *transition relation.*

At any point in time the machine is in some state $q \in \mathcal{Q}$ and it is possible to give the machine an input $i \in \mathcal{I}$. The machine then deterministically selects a transition relation $q \xrightarrow{i/o} q'$, produces output symbol $o \in \mathcal{O}$, and jumps to the next state q'. Graphical representations of Mealy machines are often used to better understand its behaviour. A graphical representation of a simple mealy machine can be seen in figure 3.1. Transition relations are shown here as edges between nodes. The edge $0 \xrightarrow{A/C} 0$ for instance where for an input $A \in \mathcal{I}$ and output $C \in \mathcal{O}$, denotes a transition from the starting state q_0 to itself.

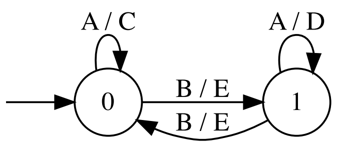


Figure 3.1: Graphical representation of a Mealy machine

3.2 Mealy machine inference

In this section we present Mealy machine inference. We will look at different methods and explain active Mealy machine learning. The goal of Mealy machine inference is to learn a Mealy machine denoting the behaviour of a software system, in our case, a communication protocol.

3.2.1 Approach

When applying state machine inference two approaches are commonly used, active and passive [7]. In a passive approach, a model of a software system is derived using observed system behaviour. This data can be for instance logs of a communication channel of a protocol run or execution traces, i.e., sequences of operations. This method has its limitations because in a typical protocol session not all possible protocol messages may be used and only valid protocol runs in general can be observed. Another limitation is when a SUT uses an encrypted communication channel where identical protocol messages may appear different.
An active approach is more suitable for the purpose of state machine inference because not only observed data (which may be a subset of the full protocol language) is taken into account but all implemented messages and message sequences. Active state machine learning makes use of a learning algorithm that queries an implementation of a protocol and observing responses from which a model can be inferred. State machine inference is a powerful tool for obtaining new insights of the implementation of the SUT. The obtained model can be compared against other implementations or checked against a specification for possible illegal state transitions. This way, security vulnerabilities present in the SUT can be exposed.

3.2.2 Active learning of Mealy machines

The active Mealy machine inference process consists of a Learner and a Teacher entity. The Learner tries to determine the underlying state machine $\mathcal{M}$ of a black-box software system, for instance a protocol. It accomplishes this by asking questions to the Teacher. The Teacher is an entity which acts like an oracle that can answer three types of questions about the SUT. The first kind of query is called a membership query, which consists of a question whether given an input symbol $i \in \mathcal{I}$ or sequence $u \in \mathcal{I}^*$ will return an output symbol $o \in \mathcal{O}$ or sequence and thus is part of the language.
The second type of query is called an equivalence query, which is a query consisting of a hypothesis $\mathcal{H}$. The Teacher will use this model constructed by the Learner to answer the question whether state machine $\mathcal{H}$ is equivalent to $\mathcal{M}$ via an equivalence approximation algorithm (see Section 3.2.3). The Teacher will answer either by *yes* or give a counterexample in the form of one or more input symbols $i^* \in \mathcal{I}$ which lead to a different output symbol $\mathcal{O}$. This counterexample is then used by the Learner to fine-tune the hypothesis, if possible. The third type of query is the reset query which resets the SUT to its initial state.
In this paper, Angluin's L* algorithm [3] is used to infer a state machine of the Thread networking protocol. In a Mealy machine inference setting, the learning algorithm must have knowledge of both the input alphabet and the output alphabet of the SUT. The way this is achieved is through an abstraction layer (mapper), which translates the concrete inputs and outputs of the SUT into abstract form.

3.2.3 Equivalence approximation algorithm

When presented an equivalence query, the Teacher tries to determine if a given hypothesis $\mathcal{H}$ is equivalent to an implementation by using membership queries. In practice, there is unlikely to be such a Teacher because testing whether a black-box software system behaves according to a given state machine is hard. The solution often applied is to use an approximation by sending randomly generated membership queries for each equivalence query. If after a set number of random generated membership queries no counterexample is found, there is confidence that the hypothesis is correct. The algorithm used for constructing these membership queries is the *randomwords* algorithm.

Chapter 4

Method

In this section we will elaborate on the methods used in the Mealy machine inference process and we will determine the scope of learning and experimental setup.

4.1 Scope of learning

The goal of this thesis is to infer a state machine from the OpenThread implementation of the Mesh Link Establishment protocol. As OpenThread is still under development, the version of OpenThread used is commit: $977cd00$. The version of Thread implemented by OpenThread is Thread 1.1.1.
Most protocols follow a client-server architecture, so when deriving a state machine, the behaviour of either a client or server is modelled. Thread does not follow this architecture because capable nodes can be configured dynamically to be either an end-device, or for example, a Leader or Router. Because of this, we will first derive the internal state machine of a Thread Leader node using only messages typically directed towards a Leader node. Afterwards we will use the full MLE message set as alphabet to possibly trigger the SUT to take on other roles than a Leader.

4.2 Experimental setup

OpenThread is normally ran on designated hardware platforms. Moreover, state machine inference of similar IoT protocols is generally done using such hardware platforms, where radio messages are captured and injected in the physical proximity of a device. The decision was made to run an instance of OpenThread in a virtual environment to allow executing code used for inference more straightforward and because OpenThread supports running instances in a POSIX environment. In this environment the IEEE 802.15.4 radios of dedicated OpenThread devices are emulated. POSIX emulation is also used by the OpenThread community for testing purposes and is representative of how the protocol stack works. The setup which was used is depicted in Figure 4.1.

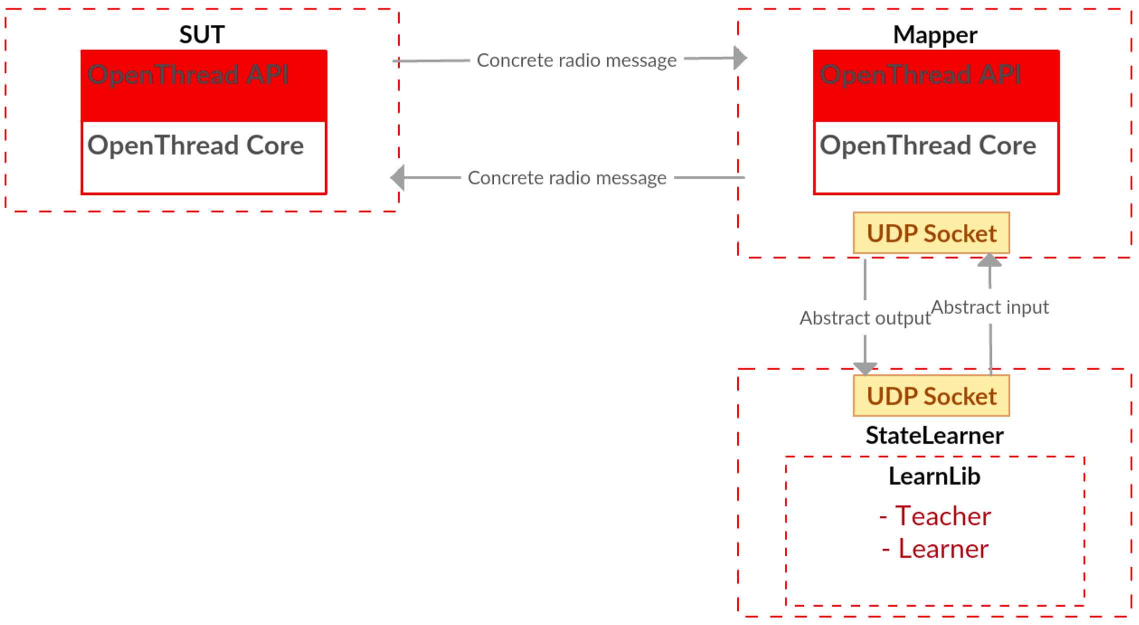


Figure 4.1: Experimental setup

4.2.1 Learning process

The inference process works as follows. First the StateLearner consults the learning algorithm which message is to be sent to the SUT, or if a reset operation is needed on the SUT. This abstract input is communicated to the mapper. The mapper constructs the concrete radio message corresponding to the abstract input or instructs the SUT to do a reset operation. The concrete radio message is then handed to the SUT where it is processed and if applicable, a response radio frame is sent back to the mapper. The mapper dissects this concrete radio message and maps it to an abstract output which is sent back to the StateLearner. This process is repeated until the learning algorithm has constructed its final state machine.
A timer is kept each time a concrete radio message is sent to the SUT for the event of the absence of a reply. When the timer exceeds a certain value the mapper sends a timeout signal back to the StateLearner to indicate an empty response. The event of multiple response packets is also handled by this timer.

4.3 Implementation

The OpenThread codebase is written in C++ and contains various useful testing and diagnosis modules. Because of this, we made use of existing code in the testing library to construct the SUT and parts of the mapper. The testing library of OpenThread exposes APIs of OpenThread and various function and buffer pointers to configure an OpenThread instance on the POSIX platform and to trigger events. The testing platform provides the following relevant capabilities:

- **Controlling instances via the instance API**, allowed us to initialize an OpenThread instance and disable unneeded features like CSMA and acknowledgement frames, which only create overhead and delays in the learning process. The instance API also lets us reset the SUT which is a prerequisite in the learning process.
- **Controlling the alarm module**, which is responsible for timing within the protocol. Delays for transmitting frames could be minimized to speed up the responses given by the SUT and thereby the learning process.
- **Controlling the radio** was used to trigger events like a receive on the virtual radio using a network packet generated by the mapper. Other functions were used to process incoming frames on the SUT as soon as the radio was in a receive state.

The implementation of the SUT and mapper with included instructions on how to run the state machine inference process with StateLearner has been made public on GitHub [1].

SUT

The SUT consists of the standard FTD OpenThread implementation ran from the testing library given by OpenThread. The instance is configured as a black box with a few adaptations to facilitate state machine learning. Communication protocols like Thread make use of timers to avoid packet collisions or determine a signal strength. Additionally, processes like Parent Attaching (see 4.4) rely heavily on timers. Due to the effect of timers, the state (transitions) of the SUT are influenced both by previously received packets, but also on the time passed. This is unwanted behaviour as it introduces non-determinism where Mealy machine inference assumes a deterministic state machine of the SUT. To eliminate the effect of timers, the timer constants in for example the Parent attaching process and the random delays when sending responses to certain queries have decreased drastically

[1] `https://github.com/bartvdenboom/Thread-state-machine-inference`

from several seconds to milliseconds on the SUT instance. This causes the Parent Attaching process, which happens each time the SUT is reset, to fail quickly such that the SUT reaches the Leader role. Timers related to timing out children have also been increased to avoid polls if children nodes are still alive. The Link Advertisement messages which are sent periodically by all Thread nodes have been disabled on the SUT node because these messages are typically only used to check if communication links are still reliable and complicate the learning process.

StateLearner

The Learner and Teacher entity originate from an existing implementation called StateLearner [6]. StateLearner supports multiple learning algorithms including the L* algorithm and various equivalence algorithms from LearnLib [19]. StateLearner acts as a wrapper of the underlying learning process and outputs and accepts abstract messages (represented as parts of the alphabet) which are to be translated by the mapper. The Teacher entity needs to be able to answer membership queries to the $\mathcal{M}$ of the SUT. Because the $\mathcal{M}$ is not known, the sequences of input messages (membership queries) are sent directly to the OpenThread implementation. Because the StateLearner uses abstract messages, a mapper has to be constructed to act as a translator between the Teacher and SUT. The *randomwords* algorithm used to generate membership queries used by the equivalence algorithm is configured to have a minimum and maximum word length of 5 and 25 respectively. The number of equivalence queries to the Teacher is set to 2000.

Mapper

The way the mapper is constructed is through a second OpenThread instance which is only used to form valid Thread network packets and dissect incoming network packets. This was convenient as the codebase is open-source, so existing functions to form certain network packets and dissect incoming packets could be used from the API.
The OpenThread instance used for forming network packets has been adapted heavily to only sent packets when instructed. An MLE message can have many possible configurations depending on the state of the sending device. To avoid introducing non-determinism, the contents of constructed messages are mostly fixed. Only those message fields used for replay protection (see Section 4.4, **Parent Attaching**) and sequence numbers are dynamically set to not break MLE functionality. Rather than going trough the process of commissioning, the network keys and parameters are set to be the same on the SUT and mapper instance to be able to start the MLE session. Lastly, network packets which are formed by the mapper are made such that they are conform the Thread specification [25].

4.4 MLE messages and processes

The set of MLE messages which are used in the inference process of the MLE state machine are derived from the specification given by Thread [25]. Thread specifies MLE Command Types and TLV (Type Value Length) Types. Command Types denote the type of message and a TLV Type is an encoding of network data like routing information, link strength and timeout values. How TLVs are encoded can be seen in Table 4.1. An MLE message of a certain Command Type typically contains more than one TLV. Further distinction can be made within MLE Command Types as the containing TLVs and control bits can vary in some cases.
MLE messages can be unicast or multicast. Unicast messages are addressed to one device. Multicast messages are addressed to a multicast address. This address can be for instance the Link-Local All Routers address, so only Routers on the same Thread Network receive this message.
The set of distinct messages together with the processes which they interact with can be seen below. For each process, a sequence diagram is given following the specification. The abbreviations used in the sequence diagrams below can be found in Table 4.5.

Table 4.1: TLV encoding

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Type							S	Length								Value...															

- **Network Discovery**
  Network Discovery is used by Thread devices to search for existing Thread networks to join. The process consists of two a two-message exchange. The Discovery Request message is a multicast message addressed to the Broadcast PANID (Personal Area Network Identifier) indicating that a device is scanning for Thread networks to join on a certain channel.
  The Discovery Request message contains the Discovery Request TLV. It is used to obtain the correct channel and PANID of a Thread network. In our experimental setup these values are fixed and already known but the processing of this message could still influence the state of the SUT.
  In response to a Discovery Request message, a unicast Discovery Response message message is addressed to the device where the Discovery Request originates from. It contains TLVs used in the MeshCoP protocol like network name, PANID and ports used in the commissioning process. Although the Discovery Request and Discovery Response messages are part of the MeshCoP protocol and not MLE, it is useful

to include because these messages are sent and processed using the MLE protocol framework.

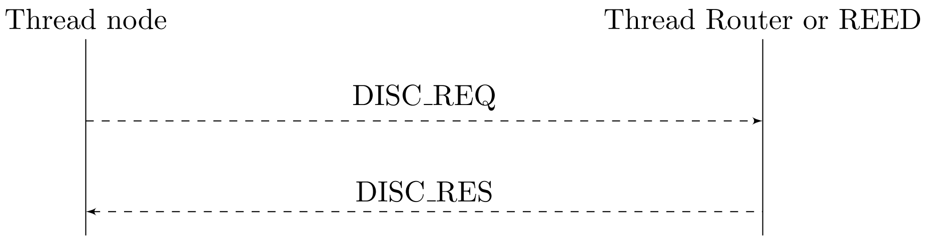


Figure 4.2: Network Discovery

- **Parent Attaching**
  After a device is commissioned, it will initiate the parent attachment process. The process consists of a handshake consisting of four messages (Parent Request, Parent Response, Child Request, Child Response). A new device will first send a Parent Request message after knowing the correct channel and PANID. This message is addressed to the Link-Local All Routers multicast address. It contains various TLVs, for example the Challenge TLV, which is a randomly chosen byte string. Some messages include this Challenge TLV to provide replay protection; any response message must contain the same byte string in a Response TLV. Another interesting TLV is the Scan Mask TLV. The Scan Mask TLV indicates whether only Routers or Routers and REEDs should respond. A connection with a router should first be attempted but if no reply is given within a given number of seconds, or the replies from routers do not contain a useful link quality, then a second Parent Request is sent with a different Scan Mask TLV to let REEDS also respond to this request. Because of this, two kinds of messages are included in the alphabet, a Parent Request to Routers only and a Parent Request to Routers and REEDS.

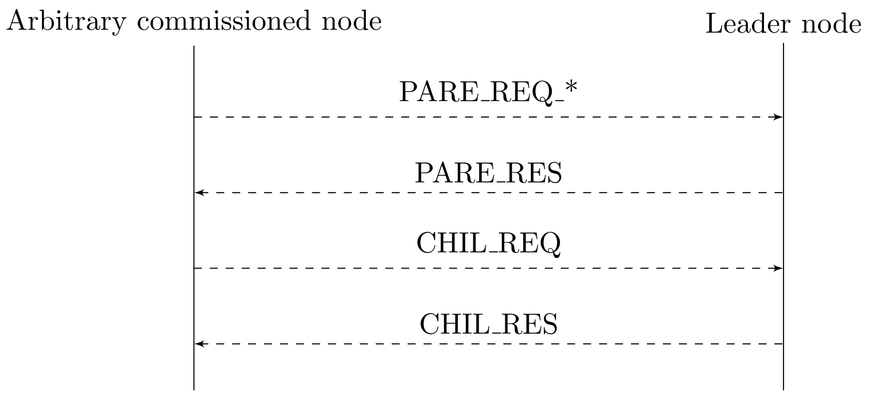


Figure 4.3: Parent Attaching process [2]

[2]An asterisk denotes any symbol such that the resulting message can be found in the specified alphabet (see Table 4.5) can be used. Any response message containing an asterisk denotes the same symbol as used in the request.

- **Announcements**
  The Announce message is a multicast message. The message contains a Channel TLV, PANID TLV and Active Timestamp TLV. An interesting TLV is the Active Timestamp TLV, which purpose is to denote the version the senders Operational Dataset. The Operational Dataset is a network wide set of network parameters. Two types of Announce messages can be distinguished. The first type of Announce message is used to announce changes in the Operational Dataset. The other Announce message, where the so called "U"-bit is set is used when an end device becomes "orphaned". This means it has lost previous connections due to changes in the physical proximity of the device or missed network data updates like a master key change.
  When an Announce message has been received the Active Timestamp present in the Announce is compared to the last known local Active timestamp. If the Active Timestamp present in the Announce message is newer than the local timestamp, the receiving device tries to attach to the sender as in the Parent Attaching process. After attachment it will send its own Announce message. If the Active Timestamp is older than the local version or when the "U"-bit is set, the receiving device will answer with an Announce message.

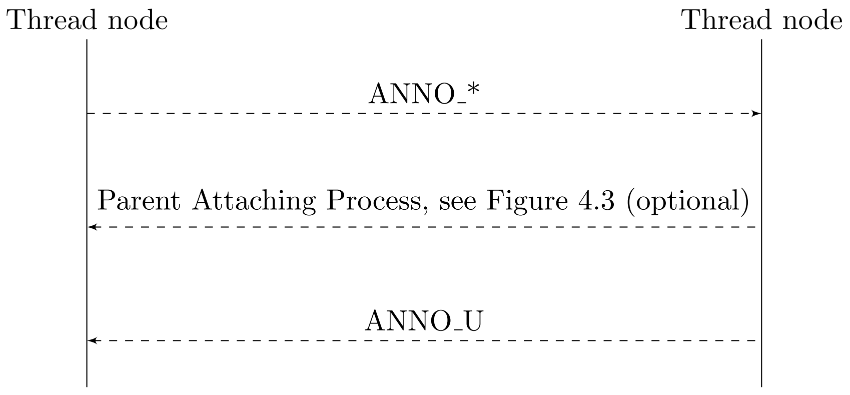


Figure 4.4: Processing Announcements

- **Child Update**
  The Child update process is performed when a synchronization of values between parent and child is needed. This happens when a child has to notify the parent of its changed parameters or after a reset. A Child update request can be sent by a Parent to a Child or vice versa. A Child Update Request can contain a Challenge TLV which is expected to be present in the Child Update Response in the Response TLV. The TLV request TLV can also be included to request certain

TLVs of the receiver. One variation is that the Parent may send a Child Update Request message containing only a Source Address TLV and a Status TLV containing the value 'error'. This message informs the Child that the sender is no longer its Parent. The Child Update Response message is a response to the Child Update Request Message and contains, if requested the Response TLV and TLVs requested by either the Parent or Child.

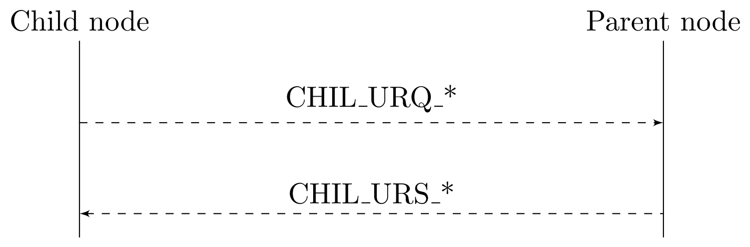


Figure 4.5: Child Update Process

- **Router ID Assignment**
  When a REED changes to a Router role it must obtain a Router ID. The REED sends a Address Solicit message to the Leader. The Leader will response with a Address Solicit Response message containing the assigned Router ID and Router Address. These two messages are also not part of the MLE message types, but necessary to include nonetheless as they are sent and processed through the MLE protocol.

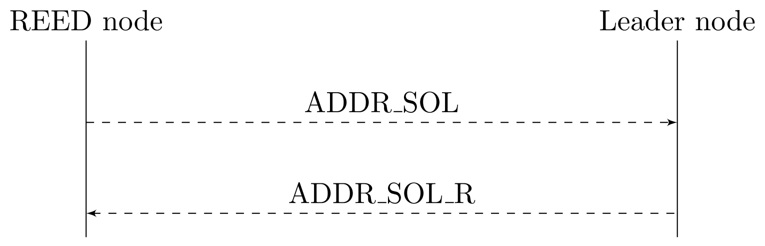


Figure 4.6: Router ID Assignment

- **Router ID Release**
  When a Router is redundant and wishes to return to a REED role it must first reconnect to the network as a Child and then send a Address Release Notification to the Leader containing its no longer used Router ID. In response, the Leader will send an empty acknowledgement message.

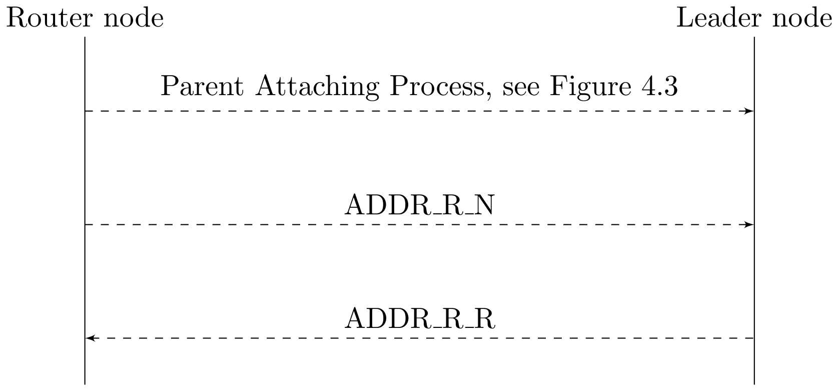


Figure 4.7: Router ID Release

- **Link Synchronization**
  Link synchronization is a two-message exchange on each direction of the link. It is performed as part of the Parent attaching process, when a REED becomes a router and whenever there is a mismatch of network data across a link. A Link request message is sent by the initiator and contains a Challenge TLV, TLV request TLV and the initiators current network data. A distinction can be made between different Link Request messages.
  The first type of Link Request is a multicast message to the Link-Local All Routers address and is sent when a device first becomes a router. The second type of Link Request is sent whenever a Router hears from a neighbouring Router for which it has no frame counter, for example when the first type of Link Request message is received. The only difference is this message has a unicast destination address. The third type of Link request is a multicast message sent to all Link-Local Routers whenever a Router is reset to obtain the latest network data.
  The Link Accept message is a response to the Link Request message and thus contains requested TLVs and a Response TLV. Two types of Link Accept messages can be distinguished. The first one being a response to a Link Request sent after a Router is first initialized. The second type of Link Accept is a response to a Link Request sent after

a Router has been reset.
An optimization of the bidirectional link synchronization process is implemented; instead of sending 4 messages, two times a Link Request and two times a Link Accept, the initiator of the link synchronization will first send a Link Request which is answered with a Link Accept and Request message. The initiator will then finish the link synchronization with a Link Accept message. The Link Accept and Request message will both have a Response TLV and a new Challenge TLV.
Lastly a Link Reject message is sent in response to a Link Request message whenever it is rejected by the receiver.

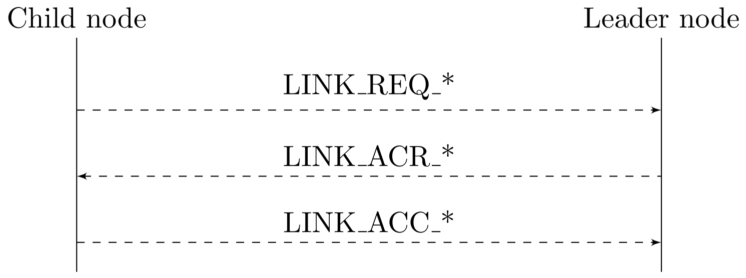


Figure 4.8: Link Synchronization process (bidirectional)

- **Network Parameter Dissemination**
A device which does not have the latest set of network data because it has just joined the network or other reasons can obtain these by sending a unicast data request specifying the TLVs needed in a TLV request TLV. The neighbour will answer by sending the requested TLVs in a data response message. A Data Response message must also sent by a device which has jus received received network data after a Parent Attachment or when becoming a Router.

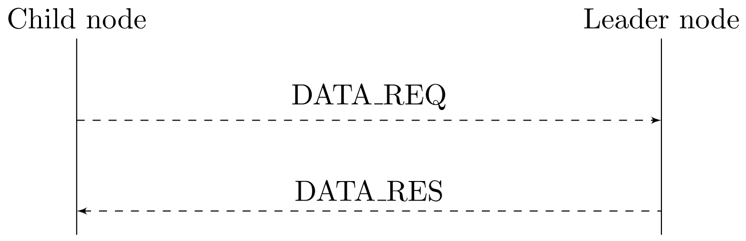


Figure 4.9: Network Parameter Dissemination

4.5 Learner Alphabet

The final set of abstract messages which is used by the Statelearner as input alphabet is as follows. Note that these abstract messages are also used as output alphabet.

Table 4.2: Full StateLearner alphabet

Abstract input	**Meaning**
DISC_REQ	Discovery Request Message.
DISC_RES	Discovery Response Message.
PARE_REQ_R	Parent Request to Routers only Message.
PARE_REQ_C	Parent Request to Routers and REEDS.
PARE_RES	Parent Response Message.
CHIL_REQ	Child ID Request Message.
CHIL_RES	Child ID Response Message .
ANNO_U	Announce Network Data Message.
ANNO_O	Announce Network Data Message for orphaned children.
CHIL_URQ_C	Child Update Request Message sent from a Child.
CHIL_URQ_P	Child Update Request Message sent from a Parent.
CHIL_URQ_R	Child Update Request to notify its no longer a Parent.
CHIL_URS_C	Child Update Response Message sent to a Child.
CHIL_URS_P	Child Update Response Message sent to a Parent.
CHIL_URS_R	Child Update Response to notify its no longer a Parent.
ADDR_SOL	Address Solicit Message.
ADDR_SOL_R	Address Solicit Response Message.
ADDR_R_N	Address Release Notification Message.
ADDR_R_R	Address Release Response Message (empty acknowledgement).
LINK_REQ_I	Link Request Message sent after the Parent Attaching process or whenever a device becomes a Router.
LINK_REQ_N	Link Request Message sent when a Router hears from a device for which it has no stored framecounter.
LINK_REQ_R	Link Request Message sent after a Router Reset.
LINK_ACC_I	Link Accept Message sent after receiving a LINK_ACR_I or LINK_REQ_I.
LINK_ACC_R	Link Accept Message sent after receiving a LINK_ACR_R or LINK_REQ_R.
LINK_ACR_I	Link Accept and Request Link message send when a LINK_REQ_I has been received.
LINK_ACR_R	Link Accept and Request Link message send when a LINK_REQ_R has been received.
LINK_REJ	Link Reject Message.
DATA_REQ	Data Request Message.
DATA_RES	Data Response Message.

Chapter 5

Results

In this chapter, we will analyse the inferred state machines of a Thread Leader node. First we will look at a state machine using the message set from Table 5.1 as input alphabet. These messages have been selected from the specification and consist of messages which are normally directed at a Leader or Router node. Messages for Router nodes are included because these are also handled by the Leader which inherently is also a Router.
Afterwards, we will analyse the inferred state machine of a Thread Leader node using the full message set from Table 4.5 as alphabet. This set of messages also contains those not typically directed at a Thread Leader node to possibly change the role of the SUT or change its internal state.

Table 5.1: Input alphabet

Abstract input	**Meaning**
DISC_REQ	Discovery Request Message.
PARE_REQ_R	Parent Request to Routers only Message.
PARE_REQ_C	Parent Request to Routers and REEDS.
CHIL_REQ	Child ID Request Message.
ANNO_U	Announce Network Data Message.
ANNO_O	Announce Network Data Message for orphaned children.
CHIL_URQ_C	Child Update Request Message sent from a Child.
CHIL_URS_C	Child Update Response Message sent from a Child.
ADDR_SOL	Address Solicit Message.
ADDR_R_N	Address Release Notification Message.
LINK_REQ_I	Link Request Message sent after the Parent Attaching process or whenever a device becomes a Router.
LINK_REQ_R	Link Request Message sent after a Router Reset.
LINK_ACC_I	Link Accept Message sent after receiving a LINK_ACR_I or LINK_REQ_I.
LINK_ACC_R	Link Accept Message sent after receiving a LINK_ACR_R or LINK_REQ_R.
LINK_REJ	Link Reject Message.
DATA_REQ	Data Request Message.
DATA_RES	Data Response Message.

The resulting state machine using the partial alphabet from Table 5.1 can be found in Appendix A where also a cleaned version can be found in Appendix A.2. The state machines using the full message set, including a cleaned version can be found in Appendix A.3 and A.4 respectively. The cleaned versions are made by my omitting state transitions like $q \xrightarrow{i/TIMEOUT} q$ where $q \in \mathcal{Q}$ and $i \in \mathcal{I}$. Here is $\mathcal{Q}$ the set of *states* and $\mathcal{I}$ denotes the *input alphabet*. By doing this, the messages which did not result in any state transition nor any response message are omitted and assumed to have no influence on the state machine of the SUT. This is done to make the state machine more readable.

5.1 Partial message set

To analyse the state machine of the OpenThread Leader node, we will compare state machine from Appendix A.2 with the expected behaviour of the processes as documented in Section 4.4 and the Thread specification as a reference [25].

5.1.1 Network Discovery

According to the specification every Route or REED should process incoming DISC_REQ messages at all times and answer with a DISC_RES message. This behaviour can also be observed in the state machine: every state has a DISC_REQ / DISC_RES transition. The DISC_REQ message does not change the state of the state machine, which is in accordance with Section 8.4.4.1.1.3 from the Thread specification. This section states that a Router or REED should keep a minimal state when processing DISC_REQ messages and should discard this state as quickly as possible.

5.1.2 Parent Attaching

The Parent Attaching process as seen in the inferred state machine does not deviate from the specification as given by Thread. We can see that from the initial state the Leader node responds to every PARE_REQ_* containing the correct Scan Mask TLV by sending a PARE_RES message and transitioning to a next state. From this state, the CHIL_REQ is answered once with the a corresponding CHIL_RES. The observed behaviour is expected as the Parent Attaching process consists of a handshake of four messages. The Scan Mask TLV in both the PARE_REQ_C and PARE_REQ_R message match those of a Leader device.

Interesting behaviour

Although not specified in the specification, it is apparent from the state machine that any commissioned node can initiate the Parent Attaching process an infinite number of times, even if its role is already that of a Child. Also, the number of Parent Requests followed up by a single Child Request does not seem to matter. As a consequence, a situation could exist where too much internal resources are allocated to handle these messages.

5.1.3 Announcements

As stated in the specification, an Announce message containing a newer Active Timestamp compared to the local Active Timestamp, like the ANNO_U message is answered by the receiving device attaching to its sender as a Child. After which an Announce message should be sent to the new Parent. This attachment process, allowing the SUT to become a Child could not be inferred in the state machine because the timers in the SUT have been configured to answer as quickly as possible and thus skipping some states in the Parent Attaching process. This means we were not able to observe the behaviour of a successful Parent Attach initiated by the SUT.
Nevertheless we can see that the SUT sends a sequence of messages denoted "Parent Attach *" which indicate that it's trying to attach to its sender after receiving a ANNO_U message.

Interesting behaviour

The specification also states that Announce messages from orphaned children (with the "U"-bit set) and Announce messages containing an older Active Timestamp compared to the local Active Timestamp are answered by an Announce message. In the inferred state machine we can see that the ANNO_O message is handled the same as a ANNO_U message, each having identical state transitions.
This means any received Announce message received by a Leader will be processed as having a later Active Timestamp.
This is unwanted behaviour because the Leader will then transition to a Child, passing on its Leader role to the sender because there can only be one Leader. Furthermore, after the Parent Attaching the new Child will send an Announce message. If this last Announce message is handled the same by the new Leader, an infinite loop of Announce messages and Parent Attaching messages could exist.
Note that the attachment of the SUT to the mapper instance could not be inferred due to the aforementioned configuration of timer values in the SUT. If such an infinite loop does not exist and the two devices change roles, the other connected devices will notice this and form new network partitions which will later merge into one, causing no damage to the network as stated

in the specification in Section 5.16.5.
The failed attempt by the SUT to attach to the mapper instance has an adverse effect on the Link Synchronization process. When a ANNO_* message is sent during bidirectional Link Synchronization, it causes a transition to the initial state of the state machine. In other cases, it causes the SUT to become unresponsive to LINK_REQ_ messages.
As stated by the specification in Section 8.4.2.2, a Leader must obtain its local Active Timestamp as part of the Operational Dataset from a dedicated Commissioning via the MeshCoP protocol. The Leader will then disseminate the Operational Dataset through the network. In our Thread network, this process does not take place due to the absence of a Commissioning device. Thus, the Active Timestamp is absent in both instances.
Nonetheless, the handling of ANNO_* messages without the presence of an Active Timestamp is still remarkable and theoretically could lead to the aforementioned infinite loop and causing a device to become unresponsive to LINK_REQ_* messages. In practice however, every Thread device is added to a network using a commissioning device such that the absence of an Active Timestamp is not likely to happen.

5.1.4 Child Update

The state machines show that the behaviour of a node receiving a CHIL_URQ_C are conform the Thread specification. As expected, two types of CHIL_URS_* messages are given as a response to a CHIL_URQ_C message. A CHIL_URS_R message is given as a response to a non-Child. Secondly, a CHIL_URS_C message is sent to a Child.

Interesting behaviour

However, it must be noted that the specification is unclear about when a CHIL_URQ_C should be answered with a corresponding CHIL_URS_C. It states "Regardless of why a Child Update Request was sent, the recipient always performs ..." but also "The Child Update Request and Child Update Response messages are used to synchronize values across the link between a Parent and a Child."
This implies that attachment of Parent and Child must have been completed before a CHIL_URQ_C message is answered with a CHIL_URS_* message. It becomes apparent from the inferred state machine that a CHIL_URQ_C is answered by a CHIL_URS_R regardless of prior Parent Attachment. This message is sent to notify a prior Child it must reattach due to an expired Child Timeout timer. In the state machine we can observe a CHIL_URS_R message is sent regardless of prior attachment.

5.1.5 Router ID Assignment and Release

Only a REED device, which by specification is attached to the Leader should send a ADDR_SOL message to obtain a Router ID. This is handled correctly by the implementation as we can only see a ADDR_SOL_R as a response in states where the Parent Attachment process has completed. The handling of a second ADDR_SOL or ADDR_R_N message is handled as expected, by not giving any response.
Where a ADDR_SOL request message has a dedicated response message type, namely ADDR_SOL_R, the ADDR_R_N does not. The empty acknowledgement frame does not indicate that the process has completed according to the specification.

Interesting behaviour

As seen in the inferred state machine, the Leader node will answer with an empty acknowledgement frame ADDR_R_R as soon as it has attached once to a Child (state 2). According to the specification, a Router who wishes to become a REED must first reconnect as a Child before sending a ADDR_R_N message.
State 9 denotes the case where a Router, i.e., the Router ID Assignment has already taken place, tries to reconnect as a child by sending a PARE_REQ message. After completing this Parent Attaching process we would expect to see a state in which the ADDR_R_N is handled exclusively by the Leader, according to the specification. However, after completing the Parent Attachment in state 9, the resulting state is state 2. State 2 is reached after the first Parent Attachment process. This means any connected Child can send a ADDR_R_N message and get an empty acknowledgement frame in response. This has no further consequences besides the fact that internal resources must be allocated by the Leader to answer these queries.

5.1.6 Link Synchronization

From the state machine we can see that unidirectional Link Synchronization is performed in two situations. After Parent Attaching, as indicated by a LINK_ACC message as response to a LINK_REQ_I. Secondly, after receiving a LINK_REQ_R message when the Router ID assignment process and Link Synchronization has already taken place. Bidirectional Link Synchronization happens after the Router ID Assignment has taken place denoted by a LINK_ACR message as a response.
The specification states Link Synchronization can be unidirectional or bidirectional, depending on if the receiver has a valid frame counter for the sender. Thus, the observed behaviour is according to the specification. The LINK_REJ message does not have influence the state machine of the SUT. Apart from the existence of LINK_REJ message, the specification does not

specify any uses or behaviour when processing or sending this message.

Interesting behaviour

More interesting behaviour when processing LINK_ACC_I and LINK_ACC_R messages can be observed from the state machine. When Link Synchronization is needed, for example in state 6 and 9, an unsolicited LINK_ACC_I or LINK_ACC_R causes the SUT to detach from the Child as seen by an empty response to a DATA_REQ message and a CHIL_URS_R response to a CHIL_URQ_C message.
Secondly, a LINK_ACC_* which is preceded by or followed up by the event of a failed Parent Attachment due to a received ANNO_* message causes the state machine of the SUT to transition to its initial state which is behaviour undefined in the specification. This has no far-reaching consequences as the sender can reattach as a Child afterwards.
Furthermore, the specification states under which circumstances Link Synchronization must be performed. From the state machine it becomes apparent that regardless of these conditions Link Synchronization can be performed if the sender has completed the Parent Attaching process without interference of the effects of the ANNO_* message from Section 5.1.3.

5.1.7 Network Parameter Dissemination

In states where the the SUT received DATA_REQ messages from an attached Child, DATA_REQ messages are handled by sending a DATA_RES message in response. The DATA_RES message is not expected to change the state of the SUT as it is used in after certain processes to ensure the sender has received the latest network data correctly. When changes happen in the network data, like during the Link synchronization process, DATA_REQ messages are not answered by a DATA_RES. This is to be expected according to the specification. However, after receiving an unsolicited LINK_ACC_* message the SUT also becomes unresponsive to DATA_REQ messages until Link Synchronization has taken place.

5.1.8 Conclusions

Based on the inferred model, some interesting behaviour of MLE processes was observed compared to the specification. The handling of Announce messages has shown a dependency that the MLE protocol has on the MeshCoP protocol. Also, cases exist where responses to unsolicited messages are given where they are not expected. Such behaviour is observed when handling PARE_REQ, CHIL_URQ_C and ADDR_R_N messages. Also, observations show that under specific conditions, a Leader can become unresponsive to LINK_REQ messages or lose a link to a Child.

We can also conclude that the SUT has not lost it's Router role by querying it using messages from Table 5.1 due to the fact the SUT always answered a PARE_REQ_R with a corresponding PARE_RES message.

5.2 Adding all messages

By using the full MLE message set from Table 4.5 we can investigate how a OpenThread Leader implementation handles more unexpected messages and possibly change its role. As seen in Section 5.1.6 we have seen that unexpected LINK_ACC_* messages influence the state of the Leader. The handling of unexpected messages is not described in detail in the Thread specification. Nonetheless, the number of states present in Figure A.4 should be roughly the same as those in Figure A.2 if unanticipated messages, especially unicast messages are handled correctly by discarding them.

5.2.1 Observations

Inferring a state machine of the Thread Leader using the full MLE message set yields in the same number of states. However some messages not present in the partial message set have caused extra state transitions in the state machine.
The CHIL_URS_R message, used to inform a Child that the sender is no longer its parent causes undefined behaviour. Depending on the state of the SUT two possible anomalies can be observed. In states 4 and 8, a CHIL_URS_R message results in a transition to the initial state. In states 2, 3 ,7 and 10 a CHIL_URS_R message will give a transition to state 5.
The difference between these two sets of states is that in states 4 and 8, a previously received ANNO_* message has influenced the Link Synchronisation process as documented in Section 5.1.3. States 2, 3 ,7 and 10 are states where the Parent Attaching process has completed. States where also Router ID assignment has taken place are not affected by the handling of this message.
The fact that this behaviour exists is remarkable due to the fact that the CHIL_URS message is only meant to be sent FROM a Parent and not TO a Parent. As a consequence, the handling of this message under the mentioned circumstances causes the need for a reattachment to the Leader.
Another interesting observation can be made when comparing occurrences of the bidirectional Link Synchronization with the sequence diagram from Figure 4.8. From the state machine we can observe that the bidirectional Link Synchronization, started by a LINK_REQ_I can also be completed by sending a LINK_ACR message instead of a LINK_ACC_I. Doing so, will give an additional LINK_ACC message as a response.
Furthermore, after completing a bidirectional Link Synchronization the SUT

will send LINK_ACC messages indefinitely as a response to LINK_ACR messages. For example, this behaviour can be seen in state sequence 9 $\rightarrow$ 13 $\rightarrow$ 17. As the Link Synchronization process uses Challenge and Response TLVs to counter replayed messages, this is unexpected behaviour. Another observation is that unsolicited LINK_ACR_* messages are handled the same as unsolicited LINK_ACC_* messages as documented in Section 5.1.6.
The LINK_REQ_N is transmitted when a Router hears from a new Router for which it has no stored frame counter. Two types of responses are given when this messages is processed. Just after completing the Parent Attaching process, a LINK_ACC response is given. This is an anomaly as this message should only be sent by a Router. States in which the mapper instance has been assigned a Router ID, bidirectional Link Synchronization starts as expected.

5.2.2 Conclusions

Compared to the state machine inferred using the partial message set from Table 5.1, some anomalies in behaviour have been introduced by using the full MLE message set as input alphabet. Interesting behaviour was caused by sending CHIL_URS_R, LINK_ACR_* and LINK_REQ_N messages under specific conditions which have caused extra state transitions in the resulting state machine. Additionally, the presence of Challenge and Response TLVs in Link Synchronization messages has no effect on replayed messages as observed from the state machine. Despite this, the number of states in the state machine has stayed the same as compared to the state machine from Section 5.1. We have also seen that the majority of the additional messages did not result in the behaviour of the state machine.
One aspect of MLE we have not been able to infer is that of a device changing its Leader role to a lower role due to the fact that many processes in MLE are dependent on timers. As timers interfere with the state machine inference process by introducing non-determinism, timing values had to be adapted to facilitate the inference process. Because of this, we have not been able to let the SUT take on other roles than Leader.
Conclusively, we can say that by inferring a state machine using the full MLE message set as input alphabet, we have discovered additional anomalies in the handling of MLE messages by a Leader.

5.3 Security Considerations

In cases where possible flooding of an OpenThread device could take place, some measures could be taken. The specification mentions that some security considerations should be taken when handling DISC_REQ messages within REEDS and Routers to prevent Denial of Service (DoS) situations. These considerations include prioritization of other MLE messages, keeping a minimal state when processing and discarding this state as quickly as possible and setting a quantifiable limit to allocating internal resources for handling DISC_REQ messages. These security considerations could be extended for the PARE_REQ, CHIL_URQ_C and ADDR_R_N messages.
Furthermore, the MLE implementation requires an Operational Dataset from the MeshCoP protocol to successfully handle ANNO_O messages. It would be better to check if the dependency is fulfilled before handling such messages. Otherwise a theoretical situation where two or more devices attaching to each other and sending ANNO_U messages indefinitely could exist.

Chapter 6

Conclusions

In this paper, we have shown that we can use active Mealy machine inference on the Mesh Link Establishment protocol as implemented by OpenThread. Two behavioural models were inferred and checked by hand if they were in accordance with the specification as given by Thread. After analysis of these models it became clear that the OpenThread implementation deviates from the specification in a few ways. Especially when handling unsolicited messages, the inferred state machines have showed some odd behaviour.
In practice, we have discovered that a dependency of MLE has on the MeshCoP protocol could theoretically break MLE functionality. Additionally we have shown countermeasures regarding replayed messages are not implemented correctly in OpenThread. Lastly, we discovered circumstances in which links were unintentionally broken causing a reattachment.
We did not succeed in inferring the complete behaviour of the MLE protocol. This is due to timer values within the protocol which had to be configured to facilitate Mealy machine inference.
Nevertheless, the used method has proven to be effective at obtaining useful insights in the robustness and possible security threats present in the OpenThread implementation of the MLE protocol. We have proposed to extend the security considerations already present in the Thread specification to overcome some of these problems in Section 5.3.

Chapter 7

Related Work

7.1 IoT protocol Analysis

Many studies have been conducted in the field of IoT protocols on several levels. Where some studies highlight the challenges and open issues related to IoT [24, 18], other investigate the security of implementations of IoT on different levels. The aforementioned study from 2017 [21] highlighted how a vulnerable implementation of certain cryptographic primitives of the ZigBee IoT protocol stack could possibly cause large scale damage. Technologies from which IoT protocol stacks are built upon, like the underlying IEEE 802.15.4 physical layer or the 6LoWPAN adaption layer are also subject to studies which have shown that there exist possible attacks and vulnerabilities [22, 16].

7.1.1 Thread

The Thread standard is relatively new, some of the fundamentals like the Mesh Link Establishment protocol [15] is still in draft status. Because of this not a lot of research has been conducted on this protocol stack. However, a recent paper conducted electromagnetic side-channel vulnerability analysis on a device running OpenThread. [8].

7.2 State machine learning

Extracting state machines from network protocols as a way to reverse-engineer a protocol specification has been conducted before [4, 7, 12]. In [4], a system was proposed to automatically extract a generalized state machine from a server program that processed network input. Using a passive approach, by monitoring the execution traces, they were able to extract generalized state machines of certain server protocol implementations. Automated active state machine learning, as implemented in this paper, has been used to derive state machines of implementations of various protocols TLS [7] and TCP [9] or those used in EVM bank cards [1]. Derived state machines of protocols are often used as a tool to compare protocol implementations against each other like in [7]. Such state machines are also used to check if a protocol implementation follows the specification like in [9] where an automated model checker has been used to do this.

7.2.1 FSM-Fuzzing

Another example of the usefulness of an inferred state machines describing protocol behaviour is by fuzzing on these state machines (FSM-fuzzing). Fuzzing is a technique where a protocol implementation is tested against random byte strings as input to trigger possible unwanted behaviour. Using the state machine of such an implementation ensures every possible state of the implementation can be fuzz-tested. FSM-fuzzing has been used as a final stage of passive state machine learning to trigger possible unwanted behaviour which could be investigated using the passive approach [5, 4]. This method has proved to generate higher quality testing cases to find real world vulnerabilities [5].

Chapter 8

Future Work

The research conducted in this paper can be proceeded in several ways. Some possibilities are discussed below.
As we are trying to say something about the robustness and security of the OpenThread implementation of the Thread protocol stack we can continue investigating this by applying the same method on other protocols within the implementation like CoAP or MeshCop. Additionally, the model learned of the implementation of the MLE protocol of OpenThread could be compared to other implementations of Thread when they arrive.
Also, checking the derived state machine against the specification by hand is tedious work for more complex protocols and may be prone to human error. Because of that, automated model checking of inferred models could also be done and has been conducted in similar research [9]. Mealy machine inference can also be extended by adding fuzzing. The purpose of fuzzing is to randomize the contents of message fields sent by the mapper to uncover unseen behaviour by our research and possibly reveal more security vulnerabilities.
Moreover, Mealy machines as a behavioural model of the MLE protocol lack in expressivity due to the influence of timers within the MLE protocol. In our research we eliminated the influence of timers but methods for inferring timed protocol state machines exist [11] and can be extended for use with MLE. Lastly, we could take a different approach to determine the security of the Thread protocol stack by focussing on other aspects like the cryptographic security of the OpenThread implementation or hardware platforms running OpenThread, which has proven to be a flaw similar protocols in the research conducted on the Philips Hue lights [21].

Bibliography

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

[2] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

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

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

[5] Baojiang Cui, Shurui Liang, Shilei Chen, Bing Zhao, and Xiaobing Liang. A novel fuzzing method for zigbee based on finite state machine. *International Journal of Distributed Sensor Networks*, 10(1):762891, 2014.

[6] Joeri de Ruiter. statelearner. `https://github.com/jderuiter/statelearner`, 2017.

[7] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *USENIX Security*, volume 15, pages 193–206, 2015.

[8] Daniel Dinu and Ilya Kizhvatov. Em analysis in the iot context: Lessons learned from an attack on thread. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):73–97, 2018.

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

[10] Sujata Neidig Grant Erickson. Navigating the ecosystem [webinar]. `https://www.threadgroup.org/Portals/0/documents/resources/Webinar_NavigatingTheEcosystem.pdf`, 2016.

[11] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 379–395. Springer, 2004.

[12] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations.

[13] D INFSO. Networked enterprise & rfid infso g. 2 micro & nanosystems, in co-operation with the working group rfid of the etp eposs, internet of things in 2020, roadmap for the future [r]. *Information Society and Media, Tech. Rep*, 2008.

[14] Vasileios Karagiannis, Periklis Chatzimisios, Francisco Vazquez-Gallego, and Jesus Alonso-Zarate. A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing*, 3(1):11–17, 2015.

[15] R Kelsey. Mesh link establishment draft-kelsey-intarea-mesh-link-establishment-06. Technical report, Internet-Draft. 2014. url: https://www.ietf.org/archive/id/draft-kelsey-intarea-mesh-link-establishment-06.txt.

[16] Konrad-Felix Krentz and Gerhard Wunder. 6lowpan security: Avoiding hidden wormholes using channel reciprocity. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, pages 13–22. ACM, 2014.

[17] Nest Labs. Openthread. `https://github.com/openthread/openthread`, 2017.

[18] Rwan Mahmoud, Tasneem Yousuf, Fadi Aloul, and Imran Zualkernan. Internet of things (iot) security: Current status, challenges and prospective measures. In *Internet Technology and Secured Transactions (ICITST), 2015 10th International Conference for*, pages 336–341. IEEE, 2015.

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

[20] Janessa Rivera and Rob van der Meulen. Gartner says the internet of things installed base will grow to 26 billion units by 2020. *Stamford, conn., December*, 12, 2013.

[21] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. Iot goes nuclear: Creating a zigbee chain reaction. *Weizmann Institute of Science, Tech. Rep*, 2016.

[22] Syed Muhammad Sajjad and Muhammad Yousaf. Security analysis of ieee 802.15. 4 mac in the context of internet of things (iot). In *Information Assurance and Cyber Security (CIACS), 2014 Conference on*, pages 9–14. IEEE, 2014.

[23] Glenn Schatz. The complete list of wireless iot network protocols.

[24] Sabrina Sicari, Alessandra Rizzardi, Luigi Alfredo Grieco, and Alberto Coen-Porisini. Security, privacy and trust in internet of things: The road ahead. *Computer networks*, 76:146–164, 2015.

[25] Thread Group, Inc. *Thread Specification*, 2 2017. Rev. 1.1.1.

[26] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.

Appendix A

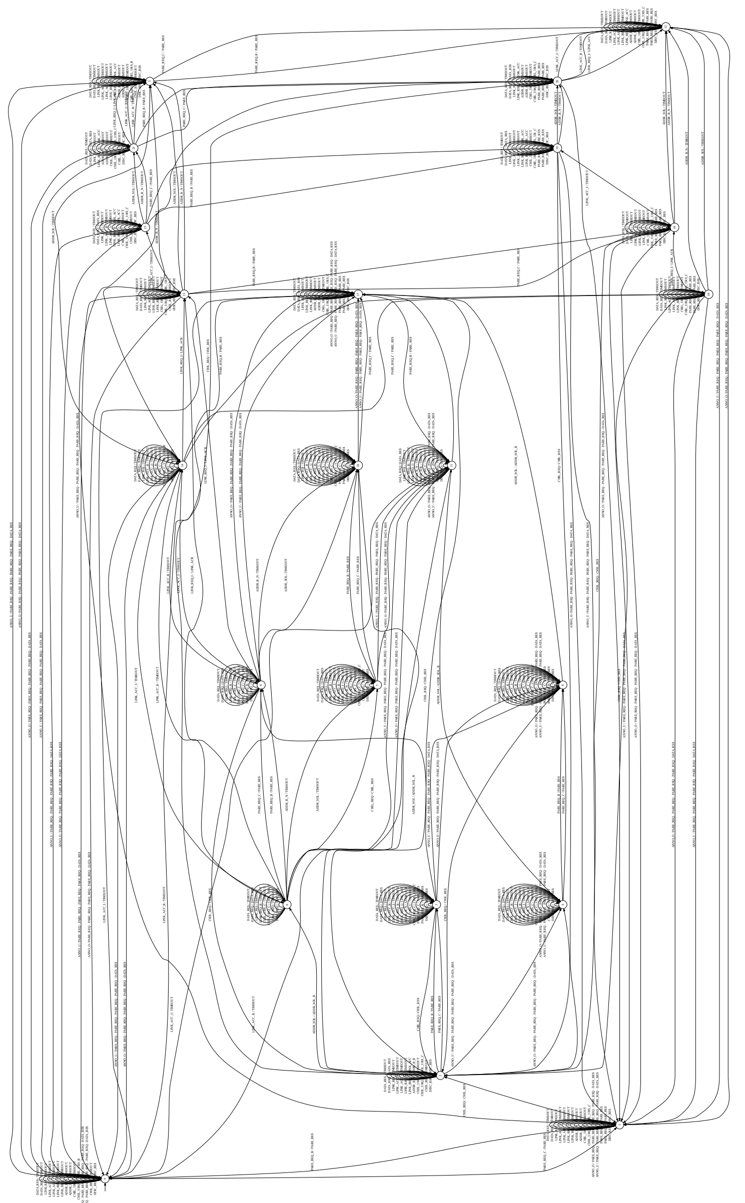


Figure A.1: Raw output from StateLearner

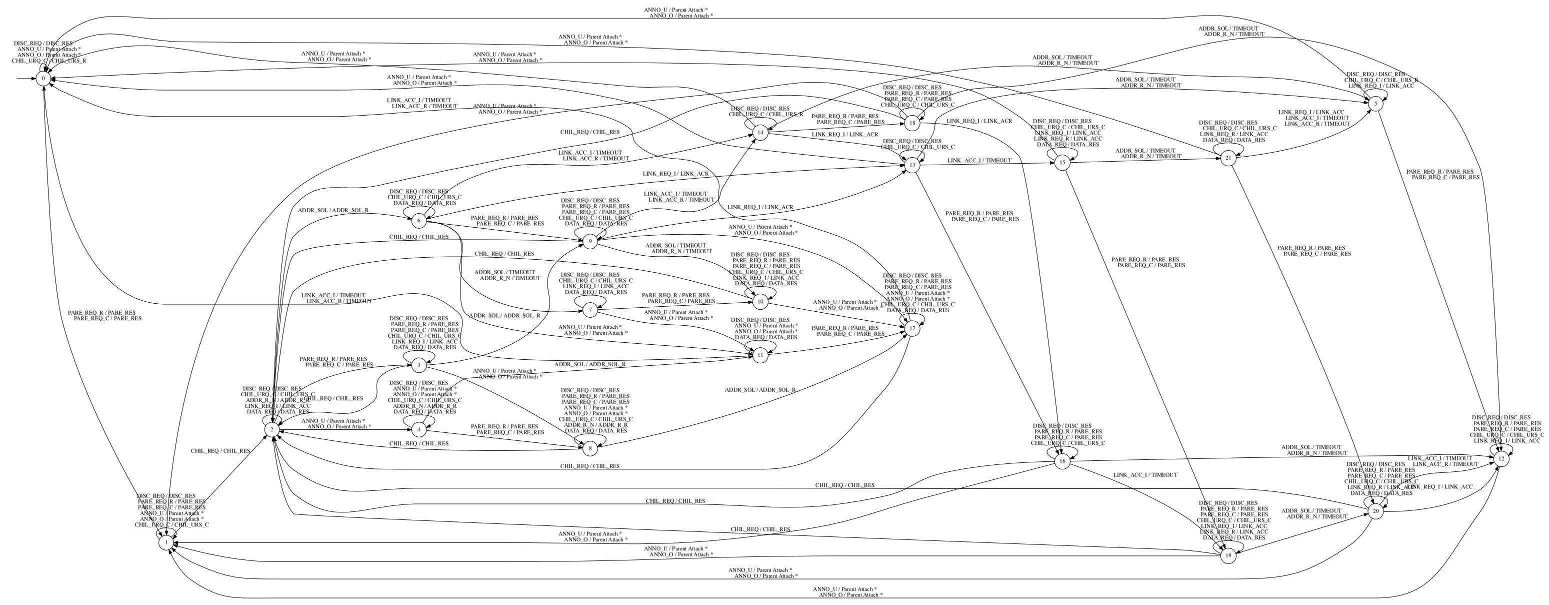
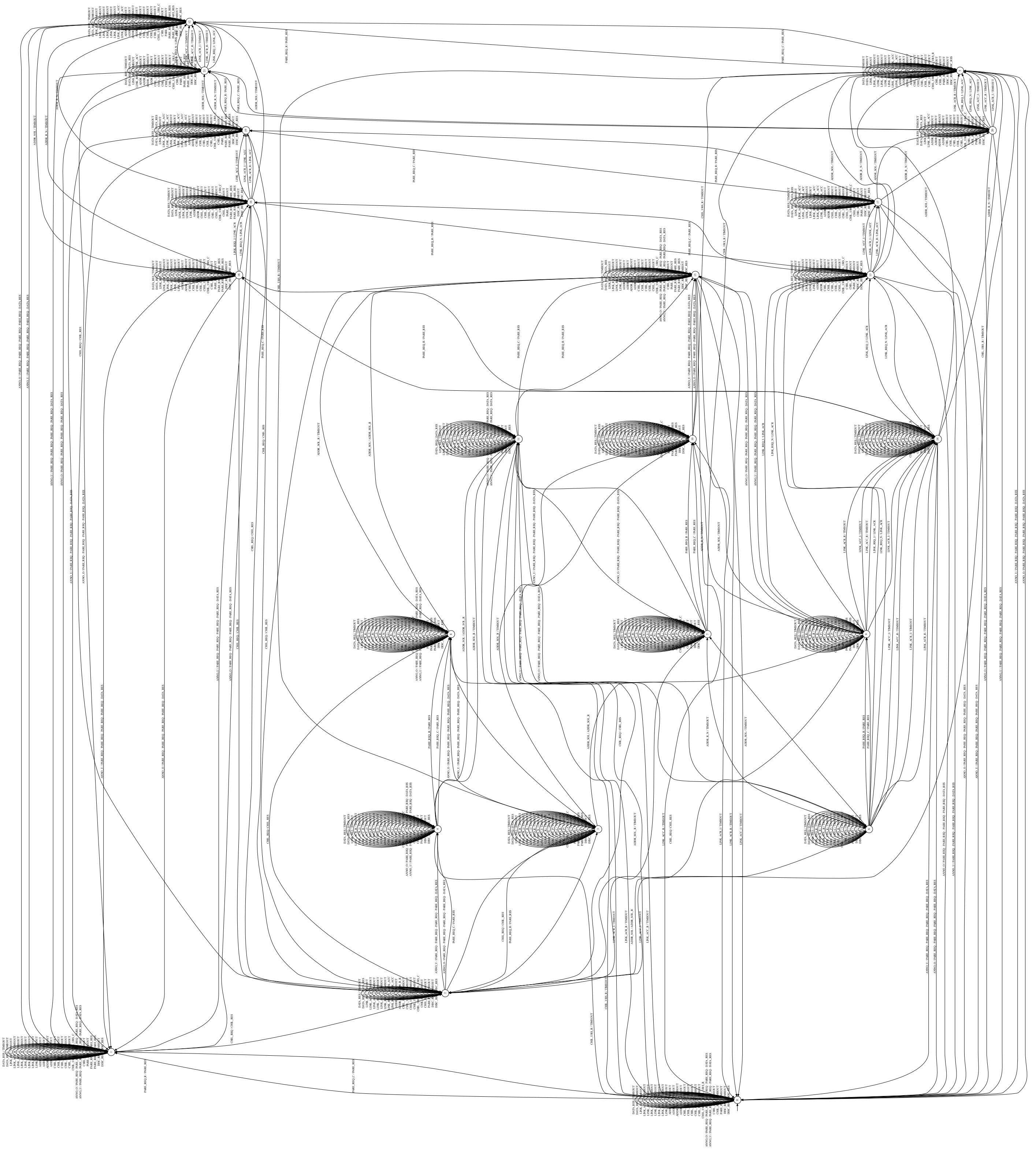


Figure A.2: Cleaned output from StateLearner

Upon receving a ANNO_O or ANNO_U message, the SUT sends out the following sequence of messages: "PARE_REQ - PARE_REQ - PARE_REQ - PARE_REQ - DATA_RES". As this sequence can be identified as a device trying to attach it is abbreviated with "Parent Attach *"



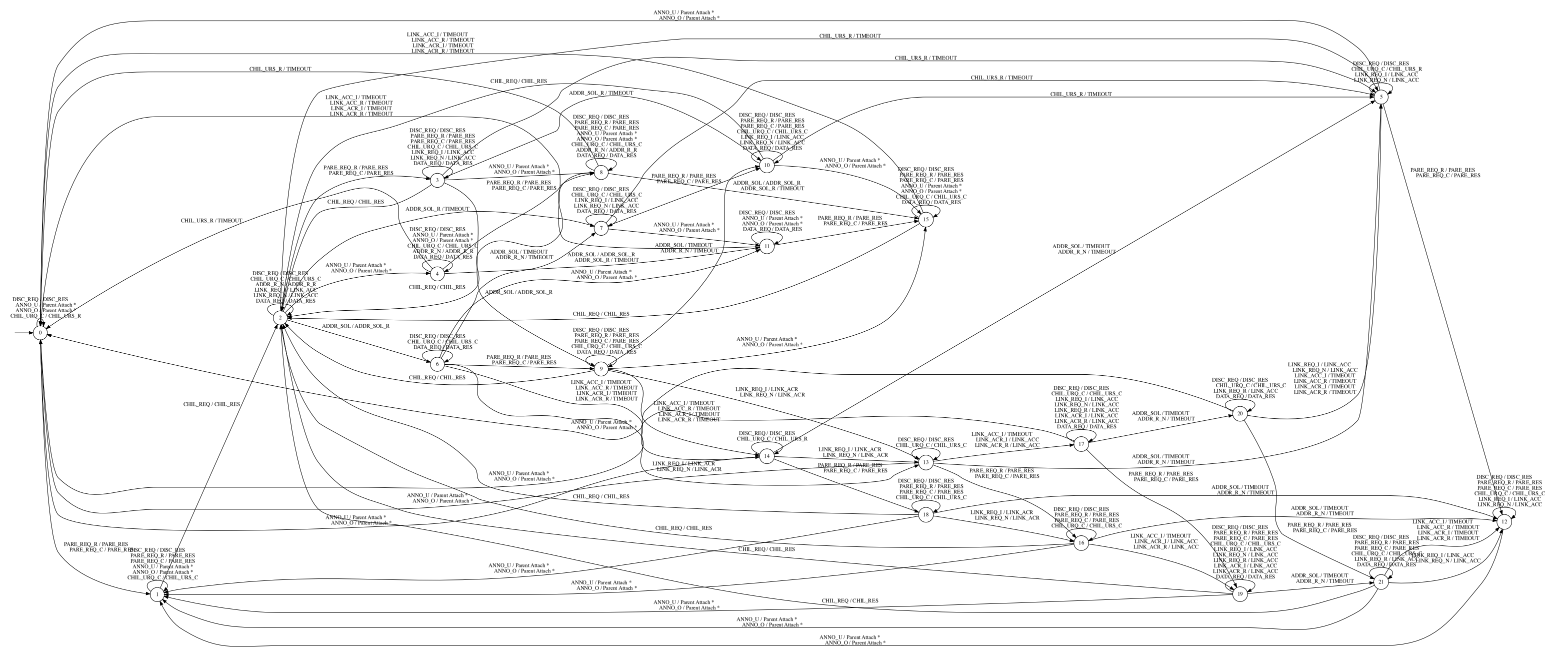


Figure A.4: Cleaned output from StateLearner