Summary

The modern world is relying increasingly on software solution for not only simple problems such as smart-housing, but also critical systems such as healthcare and banking. Thus it is extremely important that no critical bugs or unexpected behaviour, can be found in these critical systems. Therefore testing applications for functionality, and correct behaviour is a prominent part of a developers job when creating new applications, and recently an alternative testing method has gained traction, fuzzing. Fuzzing is different from traditional testing methods due to the mindset, instead of searching through the application, and making sure of correct behaviour and functionality, we search for bugs and unexpected behaviour. In the correct context, fuzzing is extremely effective tool, as an example Google has been able to uncovered more than 25,000 defects in their products and more than 22,000 problems in open-source applications[15], and B. Miller shows that utilising the same fuzzing tools, as was first introduced in the 1980's, today is still just as effective on command-line tools in popular unix-based distribution only, where B. Miller found that the failure-rate was still significant with 12% on Linux, 16% on MacOS and 19% on FreeBSD[40].

Recent years the concept of IoT has increased in popularity. Systems of small devices are used to monitor and control smart-home systems as well as industry. Google introduced the Thread protocol in 2017 [2] to bridge this gap between smart-home networks and industry, by utilising already known protocols such as IEEE 802.15.4, 6LoWPAN, and making it all addressable via IPv6 addresses [47]. For Thread, just like all other IoT protocols, an important aspect of the protocol is security, resulting in the need for a fuzzer that can fuzztest a Thread implementation, and hold it up against the specification.

In this report we present a behavioural-analysing mutation-based blackbox fuzzer for the networking protocol Thread, Thread-Fuzz. Thread-Fuzz is the proof of concept of a new generation of fuzzers that search for unexpected behaviours other than just crashes and memory corruption, namee behavioural fuzzers. Thread-Fuzz introduce the idea by implementing a state machine, which checks if the server state of the target satisfy the specification, if not it gets logged with an indication level matching that of a crash. Thread-Fuzz utilises the mutator of a fork of the well established tool AFL, namely AFLNet, contrary to AFL, AFLNet focuses on network protocols. In order to understand the idea behind Thread-Fuzz we introduce the protocol stack of Thread and a more detailed description of MLE, which is the novel part of Thread. We also introduce the terms black-, grey- and whitebox fuzzers as well as mutation and generation based fuzzers, in order to better place Thread-Fuzz.

As we discovered, AFLNet required that applications was instrumented in order to to utilise the state-aware fuzzing built into AFLNet, and due to the fact that we were unable to compile OpenThread with the custom GCC compiler that AFLNet provides, we were unable to run AFLNet with its full set of functionalities. Instead we introduced our own more intelligent StateMachine within a router between AFLNet and OpenThread called the Monitor-component, the Monitor-component has a build in parser for Threads messaging protocol MLE, and a build in StateMachine with knowledge about the sequence of messages that is correct according to the Thread specification. With this extension of AFLNet, we were able to find a buffer overflow vulnerability in OpenThread. Discovering this buffer overflow has the potential to be a significant error, even if it is not possible to exploit this bug for code execution, it could still be utilised for making entire Thread networks go down without much effort. In addition, Thread-Fuzz was put through a series of tests, to assess how well it would discover introduced bugs in Thread, and it was able to find six out of 10 of the introduced errors.

Lastly, we conclude that in spite of it being rough around the edges, and needing significant work to be able to compete with other protocol fuzzers, the concept of behavioural fuzzing is novel, and we demonstrate with Thread-Fuzz a proof-of-concept that not only is it feasible, but it also assisted us in discovering the buffer overflow error.´

Thread-Fuzz: Mutation-based Behavioural Blackbox Fuzzing of Thread

Lars Bo P. Frydenskov Asger G. Weirsøe

Cassiopeia, Department of Computer Science, Aalborg University, Denmark
{lfryde17, aweirs15}@student.aau.dk

Abstract The modern world is increasingly reliant on software for not only everyday use cases but also critical systems such as healthcare and banking, and for these systems is it extremely important there are not any bugs or unexpected behaviours present. A method to avoid this is testing, up until now testing has mostly been unittesting, QA-testing and slow release strategies, but recently an alternative method for testing has gained more popularity, fuzzing. Fuzzing is different from traditional testing methods in regards to the mindset, instead of testing for correct behaviour and functionality, the focus is on trying to find bugs and unexpected behaviour. In the right cases can fuzzing be extremely effective Google has, as an example, uncovered more than 25,000 defects in their products and more than 22,000 problems in open-source projects[15] and B. Miller utilised fuzzing against command line tools in popular distributions in 2020 where failure rates of 12% on Linux, 16% on MacOS and 19% on FreeBSD was found[40]. In this report we present a mutation-based blackbox fuzzer for the networking protocol Thread, Thread-Fuzz. Thread-Fuzz is the proof of concept of a new generation of fuzzers that search for unexpected behaviours other than just crashes and memory corruption, behavioural fuzzers. Thread-Fuzz introduce the idea by implementing a state machine, which checks if the server state of a target satisfy the specification. Thread-Fuzz is tested against OpenThread, the open source implementation of Thread. During our testing a buffer overflow is found which potentially can lead to unauthenticated remote code execution. Thread-Fuzz is also tested against ten introduced bugs in OpenThread where it found six of them. We conclude that Thread-Fuzz is unpolished but functional.

Contents

0 Introduction

Today the world is heavily influenced by automatic systems and computers, and critical systems such as healthcare and banking are relaying on these systems. It is therefore immensely important that these systems behave correctly, and in order to ensure features and correct behaviour testing have been big part of developing new software. While the traditional method of testing, such as user test, system test and unit testing is widely used, it is clearly not effective enough, since each year, even each month we see examples of faulty or insecure behaviour in software, which malicious agents utilise for gaining scrutiny over systems. Recently it was a sudo exploit (CVE-2021-3156) [16] that made headlines, where a simple off by one error results in a heap-based buffer overflow, which can be used by to get the highest privileged access. Another example is the Heartbleed bug [17] found in the SSL implementation OpenSSL, where the a client implicitly trust a request for a user-defined size of a buffer, and thus a malicious user were able to dump the working memory and therefor read sensitive data. A method that can assist in finding these small but critical bugs is fuzzing
. The idea of fuzzing was first officially exercised in 1988 by B. Miller [37], but have in the last few years risen in popularity. The primary difference compared to other test methods are the approach. Where traditional testing is about testing features and correct behaviour, fuzzing is the opposite. Fuzzing is the search for unexpected behaviour, memory corruption and crashes. Recently it has evolved into three branches, they are are named after how they approach the target application, blackbox, greybox and whitebox fuzzing. The first branch is blackbox fuzzing, which is the same kind of fuzz method that B. Miller did in 1988, simply by generating random inputs to a target application and then observing the output in the search of crashes or unexpected behaviour. While this is a rather naive way of testing, it can be sufficient in the search for crashes, in fact did B. Miller et al. find in 2020 failure rates of 12% on Linux, 16% on MacOS and 19% on FreeBSD of 75t̃ested applications on each platform. Furthermore due to its nature, blackbox fuzzig is widely adopted in the cyber security community, as a automated tool to find obvious attack vectors in applications, as tools such as Burp Suite[10], SQLMap[53] and WFuzz[57] is used in red teaming and penetration testing. Greybox and whitebox fuzzing are closer related, since they both implement knowledge about the implementation in order to do more effective and efficient fuzzing, instead of treating the application naively as in blackbox fuzzing.

Greybox fuzzing is a smarter fuzzing method, by utilising instrumentation on the targeted application, the fuzzer is able to keep track of branches within the application, that has been explored. This though, requires direct access to the source-code, as it needs to be compiled in a way, where the fuzzer knows which state the application is in, in any given point of time. Greybox fuzzing requires more of a setup, than blackbox fuzzing, but can none the less be an effective tool in finding bugs, in fact Google uncovered more than 25,000 defects in their products and more than 22,000 problems in open-source projects [15]. The greybox fuzzing methodology is heavily influenced by the tool American Fuzzing Lop (AFL) [3] which was release in 2013. The success of AFL relies in usability, which allows user to relatively quick run a fuzz test against any C or C++-based implementation with little knowledge about the target, but access to the source code. AFL provides a custom GCC and G++ compiler, which is used to compile the target application and injecting instrumentation after compilation. The instrumentation takes the form by placing logging statements in each block, it is able to differentiate execution paths for each input, and rank inputs in order to mutate inputs-of-interest to cover a bigger part of the code. AFL is no longer maintained but author is refers to the community maintained project American Fuzzing Lop++ (AFL++)[55, 19]. Although AFL has reached its end-of-life regarding development it is still considered the "state of the art" greybox fuzzer and is still used as the base for or compared up against in new fuzzers, some examples of different branches of AFL is; American Fuzzing Lop Net (AFL-Net) [49], with a focus on fuzzing network protocols. Win AFL [24], with a focus on fuzzing windows binaries. AFLGo [7], with a focus on fuzzing applications written in the GoLang language. LibAFL [1], is a library that provides mutators from AFL and other fuzzers. New forks of AFL is still being created with the most recent is SnapFuzz[5], which was presented this year and is a network protocol fuzzer.

The last fuzzing approach is whitebox fuzzing, which utilises the symbolic execution and constraint solving. While symbolic execution is not a new method, in fact it was presented around 1976 in [33] by James C. King, it is major part of whitebox fuzzing. Many whitebox fuzzers present symbolic executa-

tion, some examples of this is; SAGE [23], BuzzFuzz [21], Driller [54] and [22]. The idea behind whitebox fuzzing is to analyse the whole source code of the target application by the use of symbolic execution, and then create a path condition for each path and then by constraint solving finding which inputs leads to what execution paths. While this makes the fuzzing more precise, it is of the cost of time analysing the source code, this is mainly due to the problem of path explosion [34] and the satisfiability modulo theories problem [6].

All three fuzzing approaches are still influenced by the approach of finding crashes, memory corruption or related bugs. A natural step of fuzzing is to look upon other kind of behaviours, such as; *does the implementation satisfy the specification.* The area of verifying the correctness of applications is a popular field within computer science. An acknowledged methodology is model checking [13], which is the method of constructing a model of a system and then checking if the properties of certain queries hold for the model. One can imagine a fuzzing tool which sends generated inputs to an target application, then monitoring the states of the target and checking whether the states achieved satisfies the specified queries.

As mentioned as automatic systems increase in popularity, so does the field of Internet of Things (IoT). A recent IoT protocol release by Google Inc's Nest Labs back in July 2014 is Thread [51]. Given that it was developed by one of the major tech firms, is IPv6 based, and has been used in the Google Nest Wifi router devices from 2019, the Thread protocol may become the new de facto standard of IoT protocols. [47]. Though as the protocol is young, there are as of now only two implementations that we know of, the closed-sourced Thread implementation developed internally by ThreadGroup, which also authors the specification, and the open-source implementation OpenThread.

This leads to the contribution of of report, the mutation-based behavioural blackbox fuzzer for Thread, Thread-Fuzz. Thread-Fuzz is tested against the open source implementation of Thread, OpenThread (OT) and was able to find introduced bugs aswell as one non-introduced potential bug.

Paper outline We will start by presenting Thread and highlight key features of Thread in Section 1. Then in Section 2 a more detailed introduction to fuzzing, where the different approaches and methods of fuzzing are presented and the tool AFLNet. In Section 3, we present the tool Thread-Fuzz, the workings of the tool, some of the challenges met and design decisions. Then in Section 4 are we testing Thread-Fuzz, we describe the testsetup on different tests, as well as the results of these tests. In Section 5 we will discuss the design of Thread-Fuzz fuzz and also the general fuzzing approach with a critical look. Lastly in Section 6 are related work presented, in Section 7 we present the conclusion of this report and in Section 8 is the future work discussed.

1 Thread

Thread is an IoT protocol backed by Google, which utilises well-known protocols, such as Internet Protocol, Version 6 (IPv6), User Datagram Protocol (UDP) and many more. In combination with the development of new methods and protocols, Thread is able to maintain connectivity, security, and availability throughout a Thread network. Thread is designed with compatibility, security, low power consumption, scalability, with future technology in mind and as it is based on the IPv6 protocol it enables Thread devices to interact with other IPv6 networks. [2, 45] In this section we will walk through some of the major parts that Thread provides, such as the roles, the protocol stack and MLE. We will also give an example for attaching-to-parent, and lastly we will talk about the open source implementation of Thread, called OT. Much of this section is based on the unpublished article "*Towards Modelling and Verification of the Thread Protocol*" by C. S. Andersen et al. [4]

1.1 Goals of thread

To assist in the understanding of the design choices made with the Thread specification, in this section we are going to cover Thread's goals. The primary purpose of the Thread protocol is to standardise how smart devices are integrated into intelligent households and industrial environments, where networks can seamlessly adopt new devices, handle device failures, and manage traffic securely.

The main goals of Thread are listed below [45] [2, p. 1-3]:

1. **Simplicity**; Threads should be simple to configure, install, and maintain.
2. **Reliability**; A Thread network must operate without a single point of failure, continuously provision new devices, troubleshoot the network, and consolidate partitions.
3. **Efficiency**; A Thread network must be efficient, both in networks with few and many nodes.
4. **Security**; A Thread network must perform operations securely within the network, allowing nodes in the network have a secure communication, and without the risk of malicious devices to join the network.
5. **Scalability**; Thread should be able to scale up to hundreds of nodes, without suffering from performance issues.

Although the goals are broad and difficult to measure, they are covered in part by the protocols that make up Thread, which are already well known and widely adopted. We will briefly cover some of the most relevant functions, roles or general information in the coming sections, where a more detailed description and clarification of these topics can be found in the unpublished article "*Towards Modelling and Verification of the Thread Protocol*" by C. S. Andersen et al. [4]

1.2 Thread Roles

Thread introduces different roles to accomplish its goals, which include eliminating single points of failure and reducing the power consumption of battery-powered devices, as stated in Section 1.1. Thread Routers are comprised of three major groupings; Border, Leader, and Router Eligible End Devices (REEDs), as opposed to more common host devices; End Devices (EDs), which may be more connected in the case of Full End Devices (FEDs) and less linked in the case of Sleepy End Devices (SEDs).

Every Thread Network supports a maximum of one Leader, 32 Routers, and 511 EDs per Router, for a total of 16351 devices per network. If the number of Routers falls below 16, a REED will be promoted if possible.

In this section, we describe the various device types and roles in Thread and highlight the primary function of each role. Each role is listed in Table 1b with associations to its legend in later figures and device type; Full Thread Device (FTD) and Minimal Thread Device (MTD).

1.2.1 Forwarding Roles

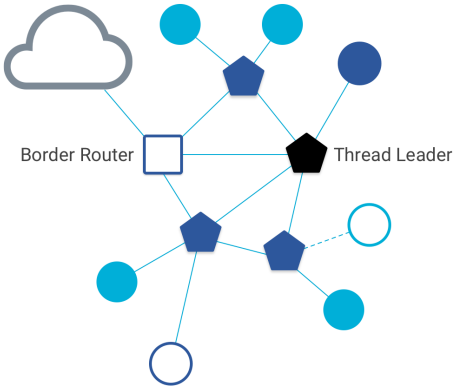
Nodes in a Thread Network are defined by two forwarding roles: Routers and EDs. When new devices desire to join a Thread Network, routers forward packets and provide commissioning services. An ED transmits packets to a single Router and can disable its transceiver to conserve energy. Figure 1a depicts EDs as blue circles and their corresponding perimeters, and Routers as pentagons. As it can be seen, Routers and EDs have Parent-Child relationships, and an ED can only have one Router.

1.2.2 Full Thread Device

A FTDs transceiver is constantly turned on and it subscribes to the all-router multicast addresses, where all link setup and advertisement messages are sent, as detailed in Section 1.5, and it maintains mappings between Thread RLOCs and IPv6 addresses, as detailed in Section 1.3.3. Possible FTDs functions include Router, REED, and FED. The distinction between a REED and a FED is that a REED can, if necessary, be promoted to Router. Though, this does not apply to a FED. [2, 45]

In the following instances, upgrading and downgrading of devices and Routers is possible:

- When a device wants to join the network but there are no routers within range, a REED requests that the Leader promote itself to a Router.
- A REED is upgraded when the minimum number of Routers in the network is not satisfied.
- A REED is degraded when the maximum number of routers in the network is reached.



(a) A Thread Network showing every type of device [46]. Legend described in table 1b. Figure from [45].

Role	Device
Thread Leader (Black pentagon)	FTD
Border Router (Dark blue square)	FTD
REED (Dark blue circle)	FTD
FED (Dark blue perimeter)	FTD
MED (Light blue circle)	MTD
SED (Light blue perimeter)	MTD

(b) Roles Thread devices can have in a Thread Network. Rows indicates roles and columns device types.

Figure 1: An example of a Thread Network with its associated device types.

- A Router is demoted to a REED if it has no children and the minimum number of Routers has been reached.

This demonstrates Thread's ability to dynamically balance the number of Routers within a Thread Network.

1.2.3 Minimum Thread Device

A MTD does not subscribe to the multicast address for all routers and instead sends all packets to its parent. A MTD has two possible roles: SED and Minimal End Device (MED). MED and SED are EDs and cannot become Routers. A MED's receiver and transceiver are always active so that it may connect with its parent Router. In order to preserve power, a SED's receiver and transceiver are typically disabled. It regularly awakens to poll its parent Router for messages, which necessitates a synchronisation step to create this form of communication.

1.2.4 Thread Leader

The Thread Leader is a Router that handles the configuration state of the Thread Network. The Thread Leader is dynamically self-selected for fault tolerance and continuously distributes configuration information to all other network devices, including active Border Routers, valid prefixes, and commissioning information. A Leader also appoints active Commissioners who are in charge of joining devices.

1.2.5 Border Router Device

A Border Router's function is to provide an interface between a Thread Network and a non-Thread Network. Any FTD can operate as a Border Router in the network. A Border Router performs standard IP packet routing between the IEEE 802.15.4 link-layer interface and typically either Wi-Fi or Ethernet IP link-layer interface, as depicted in Figure 1, where the Border Router, denoted by the square, interfaces with the Thread Network and a non-Thread Network, denoted by the cloud. This route is transparent for IP communication end-to-end. Although a Border Router does not handle commissioning, it does provide a Commissioning Protocol Border Agent, which simplifies the connection of new devices to the Thread Network. This Thread role is comparable to the Gateway device type in other Internet of Things protocols.

A border router must offer the following capabilities:

- IP communication in both directions between the Thread and Wi-Fi/Ethernet networks
- Service discovery using bidirectional mDNS and SRP

- Thread-over-infrastructure that combines IP-based Links with Thread partition
- External Thread Commissioning to authenticate Thread devices and allow them to join Thread Networks

1.3 The Protocol Stack of Thread

The Thread Protocol is based on already publicly available protocols, these protocols are the basis for Thread, and it is thus referred to as the Protocol Stack of Thread. Figure 2 shows the protocol stack of Thread, where IEEE 802.15.4 provides the base for the Thread protocol [9, 25], and is also responsible for the Physical- and MAC layer of the protocol stack. Thread then expands on the IEEE 802.15.4 standard, with IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) and IPv6 for the network layer, and UDP for the transport layer. In the Figure 2 the sections with blue background are the part of the protocol stack, that Thread provides and can be referred to as the Thread protocol.

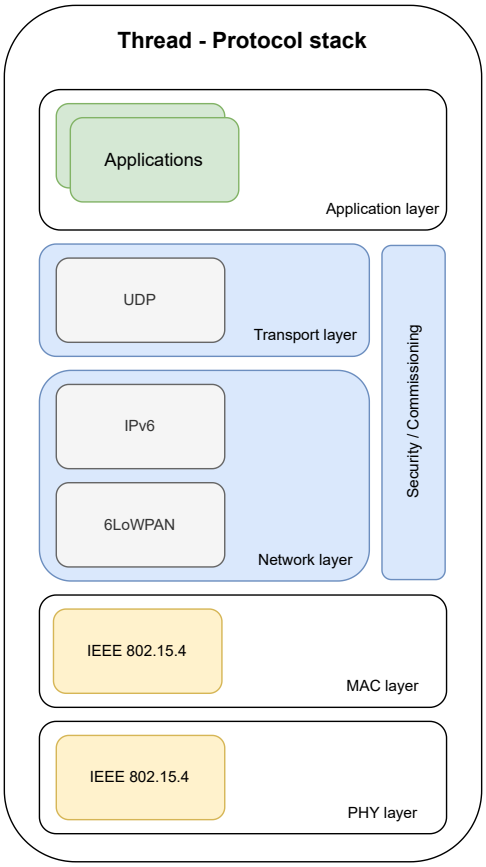


Figure 2: The Thread Protocol Stack, which Stack is composed of the following: For the physical- and MAC Layer; IEEE 802.15.4. For the Network Layer; 6LoWPAN and IPv6. For the Transport Layer; UDP, and we have the application layer. Everything that is marked with a blue background is the Thread Layers.

As our focus of this article is to methodise fuzzing of application layer of OT, most of the protocols are surpassed, thus we will only describe the different protocols briefly in this section, a more detailed description is available in the unpublished article "*Towards Modelling and Verification of the Thread Protocol*" by C. S. Andersen et al. [4]

1.3.1 IEEE 802.15.4

IEEE 802.15.4 is a wireless-network standard, and is also known as the mesh-standard as it provides the basis for a mesh networks lower layers in the OSI reference model. [9] The main requirement for IEEE

802.15.4 is that it should provide these layers for a low cost, low power, and low speed Wireless Personal Area Network (WPAN), meaning that it is designed for battery-powered communication devices. The physical layer of the IEEE 802.15.4 provides a link to the physical medium, such as the radio etc. The MAC layer of IEEE 802.15.4 provides the basic functionality of the MAC layer, such as frame transmission, frame reception, and frame filtering. [9] The Thread specification relies on IEEE 802.15.4 to provide reliable end-to-end communication. [25] In addition to Thread, IEEE 802.15.4 lays the ground for other wireless communication protocols such as ZigBee, 6LoWPAN and wirelessHART, which are all an extended IEEE 802.15.4 by definition of higher network layers. [27, 4]

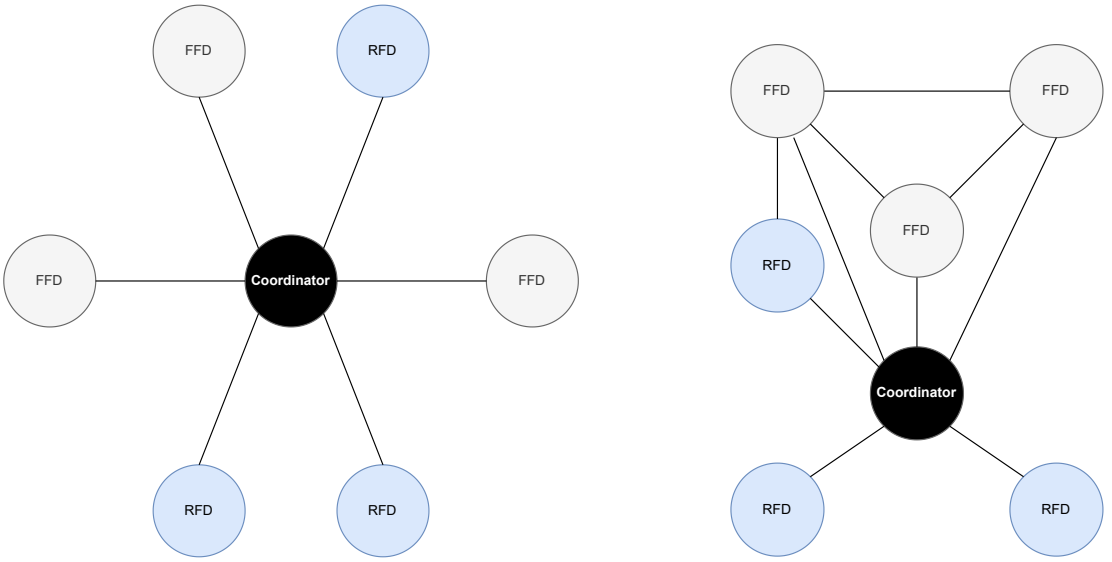
1.3.1.1 Device Types

IEEE 802.15.4 defines two groups of devices, Full Functioning Device (FFD) and Reduced Function Device (RFD). A FFD is a fully functional node, which means that an FFD node can send, receive, and forward data. Furthermore, a FFD node is also capable of acting as a coordinator, which is a special node in WPAN responsible for network configuration and online network management. [9]

RFD is a node with reduced function level. RFD is a kind of simple device with low power consumption and limited resources, usually they act as a end-device or leaf-node of the topology. Since RFDs are simple device types, they can only communicate with FFD types of devices. Furthermore, they are incapable of relaying messages or acting as dispatchers. [9] Note how these device types closely correspond to Thread device types, as described in Section 1.2; FFD is similar to FTD, RFD is similar to MED/SED. [9, 2]

1.3.1.2 Network Topologies

IEEE 802.15.4 defines two types of network topologies, star network and peer-to-peer network. A peer-to-peer network is a network topology where nodes are connected to each other in a mesh network, while a star network is a network topology where nodes are connected to each other in a star network with the Coordinator in the middle. [9]



(a) IEEE 802.15.4 star topology with a central coordinator, which all notes communicate with/through.

(b) IEEE 802.15.4 peer-to-peer topology where nodes communicate directly or through the coordinator

Figure 3: IEEE 802.15.4 topology where grey nodes are of type FFD, blue nodes of type RFD and the coordinator is black. Thread utilises the peer-to-peer network topology provided by IEEE 802.15.4.

1.3.2 6LoWPAN

As per Section 1.1, the Thread group picked IPv6 for scalability, interoperability, and connection. As IEEE 802.15.4 defines a maximum packet size of 127 bytes, IPv6 is not directly compatible with IEEE 802.15.4, thus Thread overcomes this mismatch using 6LoWPAN. [26]

6LoWPAN is a protocol specification for low-powered wireless communication like IoT devices. [42] 6LoWPAN was meant to deliver IP packets over an abstract network, but it defines how to send them through IEEE 802.15.4 lines. [42, 26] Furthermore, 6LoWPAN handles this by fragmenting IPv6 packets, which are reassembled by the recipient. [26] 6LoWPAN compresses IPv6's 40-byte headers to save transmission overhead and energy [26, 42]. Header compression allows sending the smallest IPv6 packet in 4 bytes [42].

1.3.3 IPv6

IPv6 is the top layer of Thread's network layer in the protocol stack. IPv6 is a sophisticated networking protocol that adds capabilities to Thread such as end-to-end routing compatibility, internet-wide addressability, and a 128-bit address space. In this part, we examine how Thread utilises IPv6 addressing and IPv6 scopes, as well as a brief overview of the important capabilities IPv6 provides in comparison to its predecessor IPv4.

1.3.4 UDP

UDP is the making out the Transport layer of Thread's protocol stack, it is used as an abstraction between the network layer and the application layer. UDP is a protocol that provides a simple, reliable, and efficient way to transfer data between applications. [14]

1.4 Thread Network Data

In a Thread Network Partition (TNP), data is stored and distributed throughout the network; this is essential for the Thread Network to operate correctly and uniformly. This data comprises information regarding how devices communicate within and outside the TNP, as well as the status of commissioning between the Commissioner and a Joiner, as specified in the unpublished article [4]. The Thread Network Data encapsulates the network setup and the network's current active status. The Thread Network Data is transmitted throughout the network using a Trickle-like method, whereby all Routers broadcast periodic Mesh Link Establishment (MLE) Advertisements with their highest known Thread Network Data Version number. Whenever a Router receives an MLE Advertisement from a neighbour whose Thread Network Data Version number is greater than its own, it will request a copy of that neighbour's dataset. By employing a Trickle-like strategy, Thread can finally guarantee consensus in a solid and scalable manner. [36][2, p. 127, 344].
All the different datasets can be seen in Appendix B, and a more detailed explanation of these can be found in the unpublished article [4].

1.5 Mesh Link Establishment

The Thread specification presents the MLE[1] protocol. MLE messages are used for establishing and configuring secure radio connections, detecting neighbouring devices, and maintaining routing costs between devices. In addition, MLE messages are used to transfer network settings across devices and to prioritise resources based on the identification of trustworthy links. [25]

MLE messages are also used to identify link quality between neighbours, as this quality might be asymmetrical, in order to minimise wasted effort in the construction of bad links. In reality, devices utilise MLE to transmit link-local multicasts including quality assessments of their local connection. In other words, a device sends MLE messages to all reachable devices in a single radio broadcast. [2, p. 59] Additionally, MLE messages are utilised by devices without network configuration values to request required network configuration values. This is accomplished by the device sending a unicast request to a neighbour with the desired values. In exchange, the neighbour transmits the requested values. [2, p. 59]

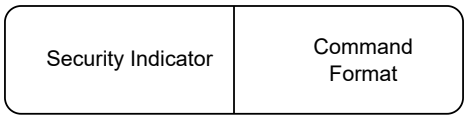
[1] MLE was initially introduced by R.K. Kelsey in the draft available in [32]; the Thread specification enhances this protocol.

For Media Access Control (MAC) layer setup (layer 2), MLE messages are transmitted through UDP, which was chosen to facilitate interoperability with existing systems.

1.5.1 Message Format

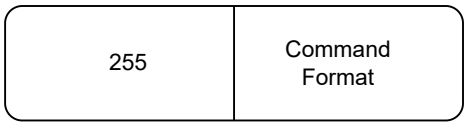
As previously stated, MLE communications have a broad variety of applications; thus, the message structure must enable several alternatives. Figure 4a is a schematic representation of the message format. As seen, the first byte of communications shows the security suite employed by the message. If the initial byte of a message contains the value 0, the message is encrypted. This is illustrated in Figure 4b, which depicts the inclusion of an auxiliary security header and a message integrity code in addition to a command — the command format is detailed in further detail below. Figure 4c depicts a message in which the first byte contains the value 255. This implies that no security is utilised, and the message contains no security data as a result. [2, p. 59]

Both of these messages contains additionally some value in a Type-Length-Value (TLV) format, which is detailed below.



(a) The general MLE message format. A MLE message begins with a byte that specifies the message's security. The second section is the command type.

(b) A MLE message that begin with a zero-valued byte. This indicates that the communication is secure; thus, an auxiliary security header and a Message Integrity Code (MIC) is supplied.



(c) A MLE message that begin with a 255-valued byte. This indicates that no security is used for the message, and thus no additional security header is provided, the Thread Specification only allows a MLE message to have a security indicator of 255 when its either sending a discovery request or discovery response.

Figure 4: The General structure of a MLE message is indicated in Figure 4a, and a secured and an unsecured are shown in Figure 4b and Figure 4c respectively.

Value	Command Type	Definition
0	Link Request	A request to establish a link to a neighbour
1	Link Accept	Accept a requested link
2	Link Accept and Request	Accept a requested link and request a link with the sender of the original request
3	Link Reject	Reject a link request
4	Advertisement	Inform neighbours of network information and a device's link state
7	Data Request	A request, typically containing a TLV Request TLV that indicates which TLVs are being requested
8	Data Response	A response to a request, containing whatever TLVs were requested
9	Parent Request	A multicast request used to find neighbouring devices that can act as a Parent
10	Parent Response	Response to a Parent Request, identifying a potential Parent
11	Child ID Request	Request for a Child ID sent by a device to a Router or REED.
12	Child ID Response	Response from a Router to a device assigning it a 16-bit network ID
13	Child Update Request	Response from Parent on Child request to update parameters
14	Child Update Response	Response from Parent on Child Request to update parameters
15	Announce	A multicast message used to notify neighbouring devices of the Thread Network's current Channel, PAN ID, and Active Timestamp
16	Discovery Request	A multicast message used to discover networks
17	Discovery Response	Response from a device on the Thread Network to a discovery request

Table 1: MLE message command types with definitions [2, p. 60-61]

In a MLE message, independent of whether it is secure or not, the command format contains a set amount of bytes. The first byte of the command format contains the command type. The command types can be seen in Table 1, and the definitions of each command type can be found in the Thread Specification [2, p. 60-61]. The command type defines also which TLVs are included in the message, an illustration of how the command with the TLVs can be seen in Figure 5.

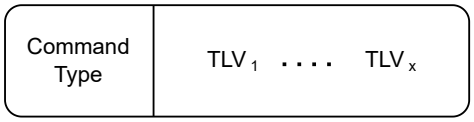


Figure 5: This block is the illustration of a Command Format from Figure 4a. The first byte specifies the command type, and it is drectly followed by a list of TLVs. [2]

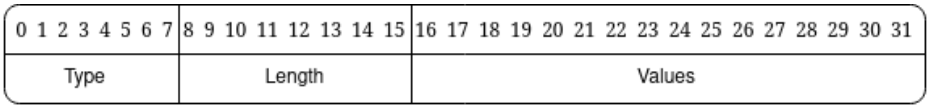


Figure 6: The structure of a TLV. This example is 32 bits long, but the length of an TLV can be at max 32 bytes long, and at minimum 3 bytes. The first byte is the TLV type, the second byte is the length of the TLV, and it determines how much data is to be read from the values part of the TLV before the next start, or the message has ended. [2]

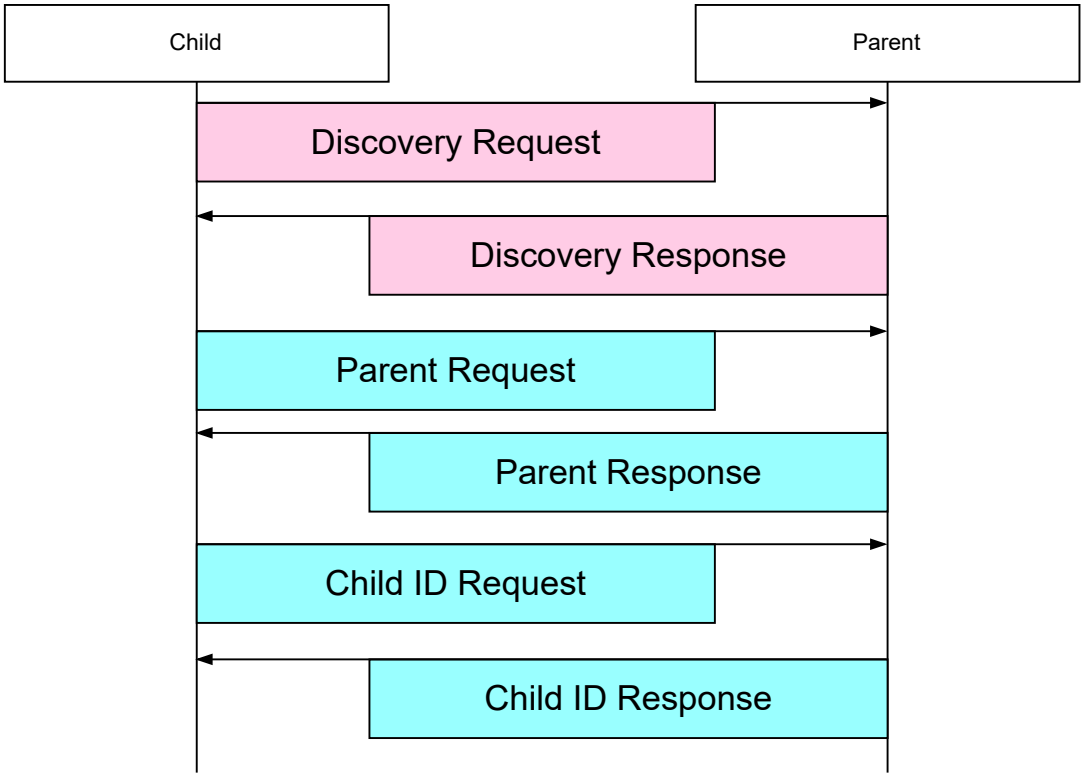


Figure 7: The sequence diagram illustrating the MLE messages send between an attaching child and a parent. The red messages are representing the discovery sequence, and the blue messages the attaching-to-parent sequence.

1.6 Attaching-to-Parent

In this section we will describe the attaching-to-parent sequence and discovery. Both attaching-to-parent and discovery happens using MLE and uses the TLV format. We will refrain from describing the bit format of each TLV, if it is of interest we refer to the specification [2] or the unpublished article [4]. The attaching-to-parent sequence is how new devices join a Thread Network. In order to imitate attaching-to-parent, the new device needs to now the credentials of the Thread Network, as well as the PAN ID. The protocol itself is a four-message exchange, between the attaching device and the parent. Before the attaching-to-parent sequence can begin, does the attaching device need to be aware of the parent, which is done by sending a Discovery Request on Link-local multicast address. Since the discovery sequence is a natural part of the attaching-to-parent sequence, we have chosen to include it in this section as well. The process from discovery to attached to a parent is illustrated on Figure 7.

1.6.1 Discovery

Discovery is a two message exchange between a sender and a receiver. The messages sent is the Discovery Request from the sender and Discovery Response as the response. Both messages includes the Thread Discovery TLV, which can contain different subtlvs. The Discovery Response only contain the subtlv Discovery Request TLV is of length 32 bit. For the Thread 1.1.1 Specification [2], is all the bits constant, except the joiner flag, which describe the senders intention of doing commissioning. Commissioning is the process authenticating new devices, which is out of scope in this report, but explained in the unpublished article by C. S. Andersen et al. [4]. The Thread Discovery TLV in the Discovery Response, contains multiple subtlvs. These contain the extended PAN ID, the network name and if the joiner flag is set, then also the information about how to do proper commissioning.

We have chosen to include Discovery in the scope, since it is the only MLE messages, which is allowed without encryption. While there is no secrets in the messages itself, it can be performed by

unauthenticated entities.

1.6.2 Parent Request

The Parent Request is sent by the attaching device, and in order to prove it is authenticated, the message itself is encrypted with the networks MLE key. The PAN ID gained from Discovery is used to locate the address needed to do the Parent Request, which is the link-local all-routers multicast address. Messages send to the link-local all routers multicast address is received by all routers within one hop. The Parent Request must then contain the following TLVs:

- Mode TLV
- Challenge TLV
- Scan Mask TLV
- Version TLV

The Mode TLV contain information of how the attaching device, wants to be treated. This include indication of the role of the attaching device such as MTD or SED, and if it is router eligible. The Challenge TLV is used as replay protection, and is simply a random bitstring, which is sent back in the response in the Response TLV. The Scan Mast TLV contain the information about what devices should respond to the Parent Request. The distinguishing is of routers and REEDs. Lastly is the Version TLV, which contains the version number of the implemented Thread protocol, and is used to check if the attaching device is compatible with the version used by the parent.

1.6.3 Parent Response

The Parent Response is the answer to the attaching device after the Parent Request, which is send as a unicast message. The Parent Response is only sent if the to be parent does not meet any the following criteria:

- It has no child capacity
- It is disconnected to its partition
- It has infinite routing cost to the Thread Leader

The Parent Response it used by the attaching device to find the best parent to connect to. The Parent Response contains nine TLVs:

- Source Address TLV
- Leader Data TLV
- Link-layer Frame Counter TLV
- Response TLV
- Challenge TLV
- Link Margin TLV
- Connectivity TLV
- Version TLV

The Source Address TLV contains the address of the sender. The Leader Data TLV contains the Leader Data which is information about the Partition ID, Partition weight, and which data set version is currently active. Link-layer Frame Counter TLV is a 32 bit value representing the current frame counter. The Response TLV and Challenge TLV are as earlier explained used as replay protection. The Link Margin TLV is the link margin between the sender and receiver, and is connected to how well the two devices can communicate without noise. The Connectivity TLV contains information of the link quality to the receiver, the cost to the leader from the sender, the Parent Priority, which is used to choose the best parent and if the receiver is a SED device, how bit the senders SED buffer is.

1.6.4 Child ID Request

The Child ID Request is unicasted to the chosen parent, the parent chosen is based on the link quality as well as the parent priority. The Child ID Request is the request of getting a child ID and a part of the parents router table. The Child ID Request contains nine TLVs:

- Response TLV
- Link-layer Frame Counter TLV
- Mode TLV
- Timeout TLV
- Version TLV
- Address Registration TLV
- TLV Request TLV
- Address16 TLV
- Active Timestamp TLV

The Response TLV is as earlier described as well as Mode, Link-layer Frame Counter and Version TLV. The Timeout TLV is a 32 bit integer value, which represent the maximum number of seconds from the sender before it expects a timeout. The Address Registration TLV contains none to multiple addresses which are used by the sender. When the Address Registration TLV is read by the receiver it adds the addresses to the router table. The TLV Request TLV is a sequence of TLV id, which is needed by the sender in order to finish the attaching-to-parent. The address16 TLV contains the sender current MAC address. The Active Timestamp TLV is a timestamp using the unix time.

1.6.5 Child ID Response

The Child ID Response is the last message in the attaching-to-parent sequence, and contains up to eight TLVs, which is needed by the attaching device in order to function as a ordinary Thread device.

- Source Address TLV
- Leader Data TLV
- Address16 TLV
- Network Data TLV
- Route64 TLV
- Address Registration TLV
- Active Operational Dataset TLV
- Pending Operational Dataset TLV

Most of the TLVs are already discussed. The Network Data and Operational Dataset is contained in the given TLVs, and is important part of how the Thread network share and save information about current state of the network.

1.7 The Thread Implementation: OpenThread

OpenThread (OT) is the open source implementation of the Thread specification, it is published by the google under the BSD 3-Clause License[2] in 2017[3] shortly after the first specification of the Thread specification also was released. [2, 45]

OT provides a library to be used in source-codes for the actual devices. Some examples for different chip-sets are also provided, making it almost a plug and play solution for devices using those. In addition to this, a simulation component is also provided, which compiles to an executable that can simulate a Thread Node, with a console that accepts commands to test everything from discovery and attaching-to-parent to commissioning and partitioning. The OpenThread organization also provides a network simulator[4], the network simulator is a playground for OT allowing users to observe visually as a web-application, how the Thread Network behaves in a network, attaching a detaching depending on whether the Nodes moves closer or further apart. [45, 2]

One command-line parameter is required for the execution of the Simulation of OT node. The `id` parameter, this parameter is used for OT to find which UDP port it should subscribe to, and figure out if a message is directed at it in the link-local multi-cast address.
In order to fuzz-test OT, some modifications was made to it such that we could abstract all of the lower layers away, these modifications is explained later in the Section 3.4.

2 Fuzzing

In the previous section we gave an overview of Thread, with focus on MLE and the attaching-to-parent sequence where new devices joins a Thread network. In this section, we will elaborate on the concept of fuzzing, by presenting the different categories of fuzzing, how it is different to fuzz a network protocol implementation and, lastly, AFLNet, an extension of AFL specialised in network protocols.

2.1 Fuzzing methods

The concept of fuzzing as a technique was first introduced by B. Miller in 1988[37], where it was used to validate the robustness of UNIX terminal applications by randomly inserting inputs and checking if they crashed. B. Miller et al. found that up to 33%[39] of all tested applications had crashing problems on certain inputs, showing that fuzzing can indeed be a useful tool for finding critical bugs. Though fuzzing did not get much traction, until many years later. In the years after his initial find, B. Miller conducted similar tests on UNIX systems and found that the fuzz testing methods, applied back in 1988, were still applicable even now, 30+ years after [38]. The field of fuzzing has grown after B. Miller first showed that fuzzing is a useful tool for automatic testing of implementations of applications. It is now used not only as a testing tool but also as a security assessment tool.

It is common terminology when describing fuzzing tools to use the term black-, grey-, and white box fuzzers. The three colours describe how the target is perceived by the fuzzer and how much knowledge the fuzzer has about the implementation. In the following three sections, we will present the different methods, as well as the difference between mutation and generation-based fuzzers.

2.1.1 Blackbox

Blackbox fuzzing is when the fuzzing target is perceived as a black box, just like B. Miller et al. it is usually used for as a method of just giving random inputs to the target in the search for crashes or other unexpected behaviours. This method is effective when finding obvious mistakes such as buffer overflows or lacking sanity checks and is easy to implement. [8] While the simplicity in blackbox fuzzing makes it attractive, it often fails at being precise. An example can be viewed just below:

Example 2.1. Consider this small example in the language C, of the function `target()` which takes a single input of type `int`.

[2] Very permissive license allowing for distribution and modification, both in commercial- and private use
[3] Provided through the repository; `https://github.com/openthread/openthread`
[4] Provided in the repository; `https://github.com/openthread/ot-ns`

```
int target(int x){ // x is the input
  int y = x + 10;
  if(y == 19) abort(); // Error
  return 0;
}
```

In this case, there is a 1 to 2^{32} chance of satisfying the conditional state and thereby provoking an error. This is for a single statement and it should be obvious that as the complexity and size of the target rises, so does the chance to find such errors. △

That said, blackbox fuzzing is still widely used, especially in web security assessments. In this case, the fuzzing inputs are rarely fully random, but instead already known lists of malicious inputs. While one can argue it is less fuzzing and closer to unit testing, it is still very effective in finding incorrect user-input sanitation bugs. Examples of blackbox fuzzing tools are Gobuster [44], Wfuzz [57], which are used to fuzz HTTP requests, and the more commercialised Burp Suite [10].

2.1.2 Whitebox

In contrast to the blackbox fuzzing method, the whitebox fuzzing method was presented and has gained increasing popularity. The whitebox fuzzer is defined by how it perceives its target. The whitebox fuzzer is able to know the inner workings of the targets, which means it is knowledgeable of the source code of the target. This allows the fuzzer to instrument the target's source-code and, thereby, perform dynamic analysis as well as static analysis. With this more complicated tool for fuzz testing, the scope is also expanded to include bugs that do not cause a crash, like buffer overflow and memory corruption. [19] The most widely used approach to whitebox fuzzing is symbolic execution in combination with constraint solving. An example of such is SAGE [23] and the model-based whitebox fuzzing approach [56]. To get a better understanding of how whitebox fuzzing works, we will give a short introduction to symbolic execution and constraint solving. Lastly in this section, we will present some of the challenges whitebox fuzzers face when the targets code base grows.

2.1.2.1 Symbolic Execution

In this section, we present the main idea behind symbolic execution and the theory presented here is based on [33]. Symbolic execution is a method to test and analyse programs. It is used to find what inputs lead to different paths in code execution, representing all discovered cases as an execution tree. The method is the use of symbols as input to a program, instead of concrete values. The program can then be executed as normally but with symbolic formulas instead. This is particularly interesting when having conditional branch types of statements, since with the use of symbols we can discover all branches that are reachable. Programming languages have an execution semantics which describes the data objects, how they are manipulated, interpreted and how control flows through the statements of such a program. Through defining a symbolic execution semantics, we also define the execution of a symbolic program. This includes extending operators to accept symbolic inputs and produce symbolic outputs. Normally, when executing a program, the state of the program is preserved in the variables and the statement counter, which denotes the statement currently being executed. As earlier mentioned, the conditional statements are of interest in symbolic execution, and for an *if*-statement the notion of path condition is added to the state. The path condition is a boolean expression over the symbolic inputs. This means that no program variables are included in the path condition. The path condition consists of a series of expressions of the form $Rleq0$, $neg(Rleq0)$, $R = R$, or $Rnot = R$, where R is a polynomial over the symbolic inputs. The initial value of the path condition is always *true*. A symbolic run of an `if`-statement is in a similar fashion to a normal program execution, where the condition is evaluated by replacing the variables with its values. A simple evaluation of a run can be viewed in Example **??**

Example 2.2. Consider the small program we presented at Example 2.1 in this case we can replace `x` with the symbolic input λ, then replace `y` with λ `+ 10` when evaluating the conditional statement: `y == 19`. It will lead two paths, where as $\lambda + 10 = 19$ is the path condition for the *then* path. Since the *then*

path leads to an *abort*(), the rest of the program will not be executed, but symbolically both paths can be run independently. The other path, `return`-path, we have the path condition: $\lambda + 10 \neq 19$. △

The result of a symbolic execution is the execution tree, which illustrates all the paths followed. A node in the tree is associated with each statement, and the forking happens for every conditional statement, whereas the `if`-statement forks in two directions, one for *true* and $false$.

2.1.2.2 Constraint Solving

In this section we introduce the main idea behind constraint solving, and we do so based on [20]. The problem of solving constraints is used together with symbolic execution since solving a path condition and solving constraints are the same problem. Constraints are relations that must be satisfied. The method of constraint solving consists of finding the variables that satisfy a set of constraints. The definition of a constraint system is as follows:

Definition 2.1. A constraint system is the 2-tuple, (C, V) consisting of constraints, C, and a set of variables, V. A constraint, C, is an n-ary relation between subsets of V, $v_1 \times v_2 \times \cdots \times v_n$ where $v_1, v_2, \ldots, v_n \in V$. Each constraint contains a set of methods, where any of those can be executed to satisfy the constraint. A method, $M : v \to v'$ where $v, v' \in V$ and $v \cap v' = \emptyset$, uses some of the variables of the constraint and the remainder as output. A method may can only be executed if all of its inputs and none of the outputs have been determined by other constraints. △

Constraints can be contradictory, and therefore sometimes it can be desirable to relax the constraints or prioritise some constraints over others. This can be expressed by labelling each constraint with a strength. While the strengths can be varying, it must hold for a constraint system, the strongest label only labels one constraint. This is due to the case of two contradictory constraints sharing the strongest label, in which it is not able to make a logic decision.

Definition 2.2. Consider the constraint system (C, V). Then we define the constraint hierarchy to such that strongest constraint to be $C_0 \in C$ and then $C_1 \in C$ to be second strongest constraint, which goes on until $C_n \in C$ which is the weakest constraint. △

Constraints can be contradictory, and therefore sometimes it can be desired to relax the constraints or prioritise a constraint over others. This can be expressed by labelling each constraint with a strength. While the strengths can be varying it holds that for all constrains that are labelled with the strongest strength, called required, must be satisfied. We write $C_0 \in C$ for the set of required strengths for the constraint system (C, V), $C_1 \in C$ for the set of strongest preferred constraints and $C_2 \in C$ is the set of second strongest preferred constraints, until $C_n \in C$ for the set of weakest preferred constraints.

With the notion of the constraint system and constraint hierarchy, we are now able to solve such a system. A solution to a constraint system is a mapping from variables to values, and we say, when all the required constraints are satisfied, that the solution is admissible. It is possible that there exist multiple admissible solutions for a hierarchy, but we also want to consider preferred constraints as well, with respect to the strength hierarchy. It can be formulated such that the best solution exists when all the required constraints are satisfied and the preferred constraints are satisfied such that no better solution exists. We will start by defining the set of admissible solutions:

Definition 2.3. Consider the constraint system (C, V), then we define the set of admissible solution, S_0 to be:

$$S_0 = \{x \mid \forall c \in C_0 \text{ where } x \text{ satisfies } c\}$$

△

With the definition of admissible solution we are now able to express the set of best solutions.

Definition 2.4. Consider the constraint system (C, V), then we define the set of best solutions, S to be:

$$S = \{x \mid x \in S_0 \text{ and } \forall y \in S_0 \text{ where } \neg better(y, x, C)\}$$

where the function *better* returns $True$ if the first argument is better than the second according to the hierarchy given as third argument, otherwise $False$. △

The comparison used by *better*, can vary if the measure is using error metrics or simply boolean predicates. The comparator used can also differ from constraint-by-constraint to aggregate measures.

In order to set constraint solving in context, we are now able to find the solution to the constraints, presented in the conditional statements for calling the `abort()`-function in Example 2.2. The solution is presented in Example 2.3

Example 2.3. Recall Example 2.2, in order to call the `abort()`-function, `y` have to satisfy the condition `y == 19`. But as we showed by substituting, we got the constraint:

$$\lambda + 10 == 19$$

We can define this to be our required constraint and, hereby, the solution is s:

$$s(\lambda) = 9$$

It is clear that there only exists one solution to the given constraint. △

2.1.2.3 The Path Explosion Problem and SMT

Whitebox fuzzing face two major scalability problems, which are inherited from symbolic execution and constraint solving, the path explosion problem and the satisfiability modulo theories. The path explosion problem is met in symbolic execution, when exploring nested calls, loops and conditions. The number of paths are growing exponential to the number of such structures. [34] It is not unrealistic to imagine hundreds of nested calls, loops or conditions, which result in enormous execution trees. Exploring all paths can be time consuming. A solution to this is directing the fuzzer only exploring interesting paths. Constraint solving have the same problem as presented by the satisfiability modulo theories, which is the problem of determining whether a mathematical formula is satisfiable. It is often compared to the simpler problem of boolean satisfiablility, SAT, which is NP-complete, but since it is harder it is typically NP-hard or undecidable, depending on the theory. [6] Both problem have been tackled, but still proves the biggest challenges of whitebox fuzzing.

2.1.3 Greybox

Greybox fuzzer is the term used for fuzzers, which uses more than just the output as feedback or measurement. The method was, as earlier illustrated, presented due to the unreliable blackbox fuzzing, and is lighter than whitebox fuzzing. Greybox fuzzers utilise the lighter instrumentation, compared to the computational heavy symbolic execution and constraint solving, and/or more complex mutation techniques than blackbox fuzzing in order to provoke unexpected behaviour. The efficiency of symbolic-execution whitebox fuzzers versus the greybox fuzzing method have been discussed, since it is showed that most vulnerabilities is found using lightweight fuzzers, that uses little to no analysis. [11]

When discussing greybox fuzzers, it is often split into two categories, directed and coverage-based. The main difference is simply that the directed fuzzers tries to reach a specific program branch or state, and the coverage-based tries to cover as much of the branches as possible. Both utilise instrumentation as their main tool to navigate the code. In the following two sections we want to give a brief introduction to both methods.

2.1.3.1 Coverage-based Greybox Fuzzing

AFL [3] is one of the most well-known and branched coverage-based greybox fuzzers. Due to the popularity of AFL, it made the baseline for coverage-based greybox fuzzing, which is the reason for adopting the method used in AFL, in order to explain coverage-based greybox fuzzing. AFL uses instrumentation to identify each basic block and while running the hit count on each block. AFL uses this information to decide what generated input is used to mutate further on and what the next block is. The creator of AFL argues that by using hit count buckets, it is able to tackle the path explosion problem. The instrumentation is done by compiling the source-code with the custom compiler provided by AFL. The custom compiler inserts instrumentation into the compiled code, by intercepting the assembler and changing it to log the basic blocks. The logging is done by inserting functions in the assembly, which is executed on run-time. [19]

2.1.3.2 Directed Greybox Fuzzing

Directed greybox fuzzers is much like the coverage-based using instrumentation to navigate, as earlier mentioned. Instead of aiming to reach most possible code, it aims to execute a sole block. Just as coverage-based it is possible to use the instrumentation feedback to choose which output leads to what block. The distance between the reached block and the target one is evaluated to select if the input brings the fuzzer closer to exploring the target block or further. [7]

2.1.4 Mutation and Generation

As previously established, fuzzing is just giving data as input to a target in order to find unexpected behaviours such as crashes. An important aspect of the fuzzing process is the inputs, and we distinguish between two methods for creating these inputs, mutation and generation based. The mutation based method is when there have been a collection of correct data, which is then modified during fuzzing. This can both be at random, such as flipping random bytes, but also with some kind of heuristic approach, as rearranging strings or replacing short strings with longer strings and so forth. The other method is called generation based, which is when the input is generated. The generator is built around the relevant specification and is then able to generate testcases as inputs. To make efficient fuzzing, the generator have to make testcases, which is predominantly malformed in order to provoke crashes. [41]

Mutation based fuzzing is easier to implement since, a fuzzer using the mutation based method, just needs correct examples in order to start fuzzing. This is, of course, the opposite for generation based fuzzing, as the implementation of the generator have to be rather specific to each case. That said, the inputs generated from the generation based method can be more directed against inputs, which in turn could promote finding an input that can provoke crashes or any unexpected behaviour faster, where as the mutation based method is more naïve. This is also expressed in their aliases where generation based fuzzers are called intelligent and mutation based, dumb.

2.1.5 AFLNet

In this section we want to give an overview of how AFLNet works. AFLNet[49] is a stateful greybox network protocol fuzzing tool, which is build upon the well known greybox fuzzer AFL. AFLNet extends AFL with a more effective method for fuzzing, in the context of network protocols, by implementing the notion of a stateful and message driven server. Network protocols often implement a server and a client where the server is message driven obtaining certain states as a result of certain message sequences. By replaying such message sequences, it is possible to direct the fuzzer to a specific server state before fuzzing. AFLNet implements these message sequences by parsing *.pcap* files, while using those as templates for fuzzing. Furthermore, the AFLNet adds the infrastructure needed in order to fuzz network protocols by directing the generated inputs to sockets.

AFLNet uses a mutation based fuzzing approach, where messages are mutated in order to generate new message sequences. This have multiple advantages compared to the generation based approach. The obvious advantage is that little to no code is needed in order to fuzz the target, and by mutating on valid traces of traffic when generating a new message sequences, then the new sequences is more likely to be valid too. Compared to the generation based approach where the a detailed specification of the protocol and message templates are needed, in order to do same kind of directed fuzzing. Another advantage of the mutation based approach, is that interesting messages can be favoured, when generating new messages, in order to discover new states, transitions, or program branches.

In order to do a coverage based search, AFLNet makes use of a custom compiler, which is needed to compile any binary that is fuzzed by AFLNet. The custom compiler inserts a logging function in each basic block, which then is used by AFLNet to identify blocks executed when fuzzed. AFLNet offers custom compilers for the known compilers `gcc`, `g++` and `clang`. It also offers a LLVM mode, which increases performance significantly by assigning ids to each block at run time inline, as well as writing to shared memory inline.

2.1.5.1 The Components of AFLNet

The architecture of AFLNet is consisting of five main components:

- Request Sequence Parser
- State Machine Learner
- Target State Selector
- Sequence Selector
- Sequence Mutator

The **Request Sequence Parser** parses the *.pcap* files according to the specific protocol, in order to extract individual messages in the correct sequences. The **State Machine Learner** read the server responses from the server under test and determines when new states are achieve, based on the status codes in the responses. The **Target State Selector** chooses which state that AFLNet focuses on next, based on the data from the state machine learner. These states are rated and chosen based on how well the previous states had contributed to increased code coverage and state coverage. At the start of the run, the states are randomly selected, since there are no knowledge about the states at that point. The **Sequence Selector** selects a message sequence that can reach the target state when the target state is selected. The possible message sequences are contained in the *seed corpus*, which contains message sequences as queue entries, which also contains relevant data about the entry. The seed corpus also contains a hashmap, that maps a state identifier to queue entries. The **Sequence Mutator** provide the fuzzing operation used to do the protocol aware mutation. The architecture of AFLNet and the interaction between components is illustrated on Figure 8.

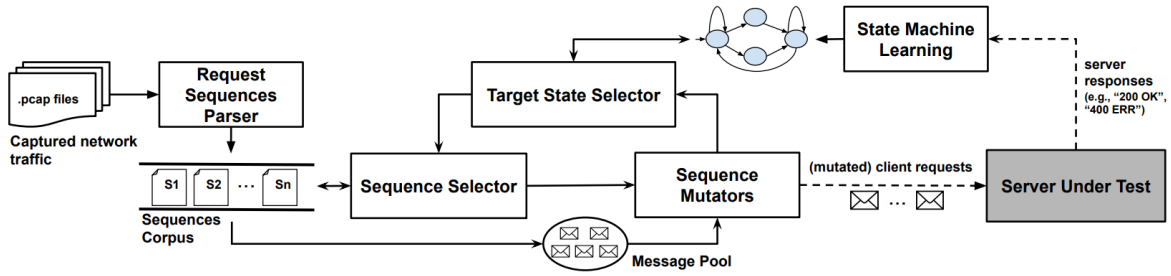


Figure 8: The architecture of the Stateful Greybox Fuzzing into AFLNet. [3]

2.1.5.2 Fuzzing operations and mutation

As previously described, AFLNet uses a mutation based approach in order to generate new messages. The definition of a mutated message sequences in AFLNet is as follows:

Definition 2.5. Given a target state s and a message sequence M, which can be split in thee parts:

1. The prefix M_1 is required to reach s;
2. the candidate subsequence M_2, which contains all message that can be send after M_1, while still being in the state s; and
3. the suffix M_3 which is the left-over such that $\langle M_1, M_2, M_3 \rangle = M$.

AFLNet generated a new message sequence M' by mutation such that $M' = \langle M_1, mutate(M_2), M_3 \rangle$. $\triangle$

This way, it is ensured that the generated message sequence always reach the targeted state. In order to modify the candidate subsequence, AFLNet offers protocol-aware mutation operators. From the seed corpus, a message pool is generated, which is a collection of the real traces and messages and the generated, that can be substituted or added in the valid message sequences. AFLNet supports the protocol aware mutation operations replacement, insertion, duplication and deletion of messages

when mutating candidate subsequences. Furthermore, AFLNet also supports the more common fuzzing operators such as bit flipping and substitution as well as insertion and deletion of bytes. The byte-level mutation and protocol aware mutation are both applied when generating the mutated candidate subsequence.

3 Implementation

In this section we want give a detailed description of Thread-Fuzz. We will refrain from describe the whole code base, and only present code if it is essential for Thread-Fuzz. A big part of the particular design used in Thread-Fuzz is due to some of the challenges met, when implementing a complicated protocol, as Thread, in AFLNet. These challenges will be presented in the following sections as well as the solutions or workarounds. First we present an overview of the architecture, where we introduce each component as well as their main purpose. Then we describe each component, more in depth, and this includes the changes made in OT and the added support for OT in AFLNet[5].

3.1 Architecture

We want to build an fuzzing tool for MLE which is the new protocol used with Thread, more specifically, the attaching-to-parent sequence. The attaching-to-parent is an important message sequence between a joining device and the Thread network, as the nature of Thread is based on trust once in the network. If a malicious device gets attached it is easy to disrupt the network, and thus it is not interesting to search for faults when the device is already in the network.

The architecture of Thread-Fuzz is consisting of two components, which in turn contains in total five components; The Monitor-component, that contains the StateMachine and and the fuzzing target, in our case an OT server. The second component is AFLNet, that is used solely for its mutator algorithm, AFLNet, uses a dummy binary, CCookie which can be controlled to let AFLNet known when to send messages. An illustration of the architecture can be viewed at Figure 9. The Monitor-component is written in Python, this goes for the StateMachine as well. Both OT and AFLNet is written i C or/and C++.

[5]All of the code explained about in this section can be found in their respective repositories on `https://github.com/Speciale-Projekt/`

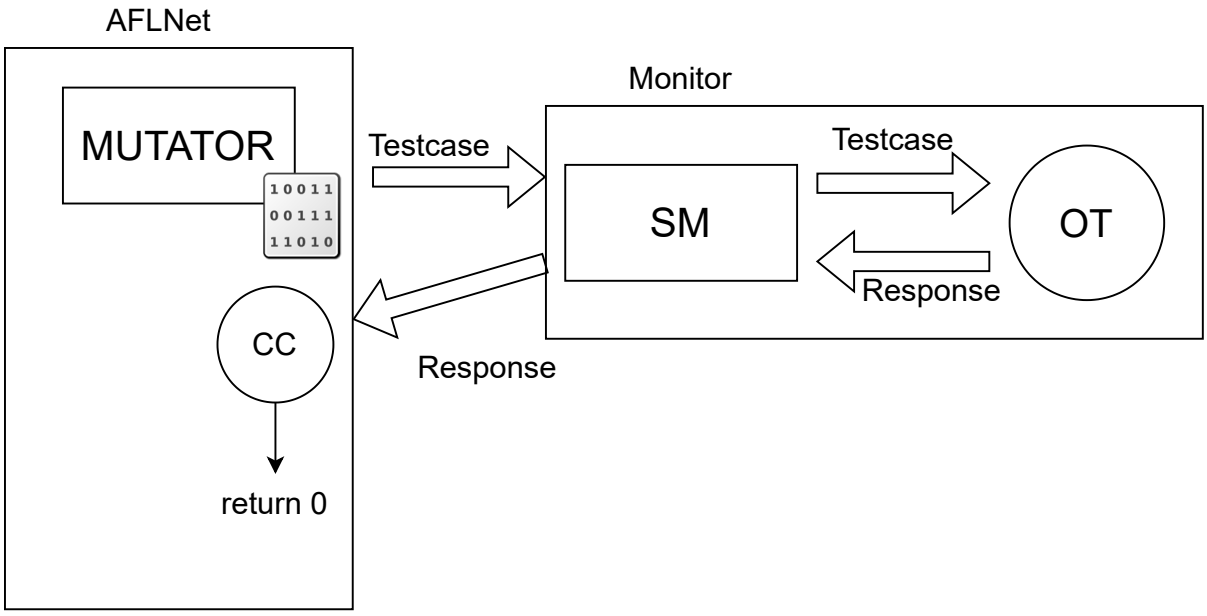


Figure 9: The overall architecture used to fuzz OT. The left is the fuzzer which have an instance of OT running inside of it as well as the StateMachine. To the right is AFLNet, which is sending the mutated testcases through UDP to Monitor, which is then send to the OT-instance in Monitor.

The initial idea behind Thread-Fuzz is to extend AFLNet with support for Thread, and then add the statemachine as either a part of AFLNet or as an proxy in the between OT and AFLNet. Both is feasible solutions, but we observe a strange behaviour when compiling the OT source with the custom compiler of AFLNet. When the custom-AFLNet compiler, which is described in the Section 2.1.5 and Section 2.1.3.1, is used for compiling OT it prevents Nodes from communicating, and thus unable to do the processes' discovery and attaching-to-parent described Section 1.6. When using this very crude method for instrumenting, as AFLNet uses, a lot of cleanup has to be done to preserve the original control-flow [50], and thus it is not surprising that some errors occur during the instrumentation. Alas, the fact that AFL (which uses the same compiler as AFLNet) is considered the state-of-the-art fuzzing tool, an in-depth exploration as to the difference of the resulting binary from using standard `gcc` and `afl-gcc` would be interesting project and is further discussed in the future works Section 8.

As we could observe a significant difference during the run-time of the instrumented and un-instrumented version of OT, it has resulted in the exclusion of instrumentation during our compilation of OT. Leading to the our current state where AFLNet is only utilised for its mutator algorithm.

The communication between AFLNet and Monitor-component is UDP through sockets, which also hold for the communication between Monitor-component and CCookie. The communication between OT is done from the statemachine to the OT instance as UDP and from OT to the statemachine through a file.

3.2 The Monitor-component

The main function of the Monitor-component[6] is to Monitor-component the state of the OT-instance. The Monitor-component maintains three threads, one for monitoring and running the OT-instance, one to handle messages from OT, and lastly one to handle messages from AFLNet. The communication from OT to the Monitor-component is done through file read and writes. This is not the preferred solution, but since OT states in their style guide [48] "*The use of the C++ Standard Library shall be avoided.*", it is challenging to make a simple UDP-connection. The implementation of a file write is relatively simple, and does not need any dependencies except standard library of C. Testcases are generated by AFLNet and received by the Monitor, which works as a proxy, sending them to the OT-instance. The Monitor-component parses the testcase before sending it to the OT-instance and the responses outputted from

[6]The monitor-component can be found in `https://github.com/Speciale-Projekt/Monitor-component`

OT. The parsed messages are given to the statemachine which then makes assumption about the states of the OT-instance. The overview of the Monitor-component is viewed on Figure 10.

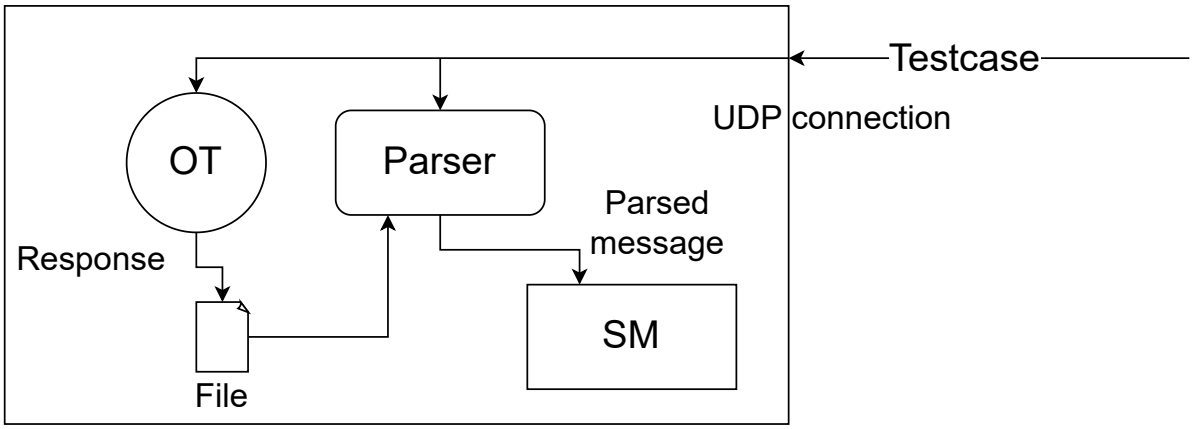


Figure 10: The Monitor-component consist of two subcomponents, the parser and the statemachine, as well as the running the OT-instance.

Another important responsibility of the Monitor-component is that of monitoring the OT-instance. Every crash is logged to a local file, with the given message that resulted in the crash and the return code of the OT-instance. When the OT-instance crashes, it simply creates a new instance with the same dataset. Due to the interaction with AFLNet, no testcases are dropped, since the CCookie is waiting, until a response is received. This is further discussed in Section 3.3.1. The case of an crash in OT is illustrated on Figure 11

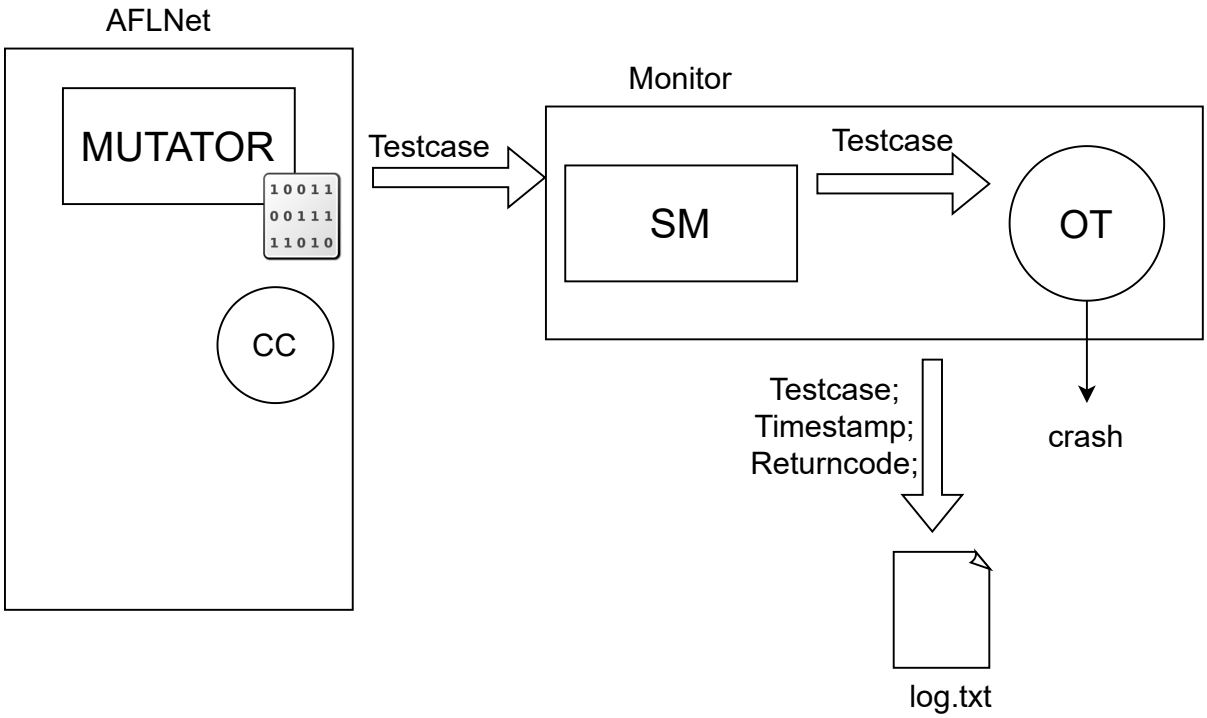


Figure 11: When the currently running OT-instance crashes, then the Monitor-component will log the relevant information, and restart the instance. AFLNet will simply wait for the OT-instance to run since the CCookie does not get any messages.

3.2.1 Parser

The Parser is a small module used by the Monitor-component in order to convert the bit-strings sent by AFLNet (testcases), and responses from OT to human readable strings. This have two purposes, where the obvious one is to make it possible to make a meaningful log entry, whenever we detect a crash or unexpected behaviour. The second purpose is of the statemachine which in order to make assumption about the server state must know which command type a MLE message if any and what TLVs is included. The parsing follows how the specification [2] describe it and is described in Section 1.5.

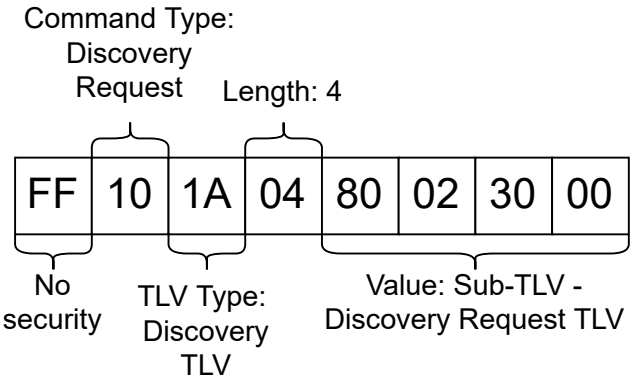


Figure 12: An hexadecimal representation of the Discovery Request. It holds for the Discovery Request that is not encrypted hereby the security byte is `FF`. The Command Type is of Discovery Request and contains only one TLV. The Discovery TLV with a length of four bytes.

An representation of an not encrypted message is illustrated on Figure 12. As we can see, is the first byte always the security byte, which tells us if the message is encrypted or not. Then followed by the Command Type if the message is not encrypted, and then the TLV format. We know when each TLV start, since all the TLVs are sequential and the length is defined in every TLV.

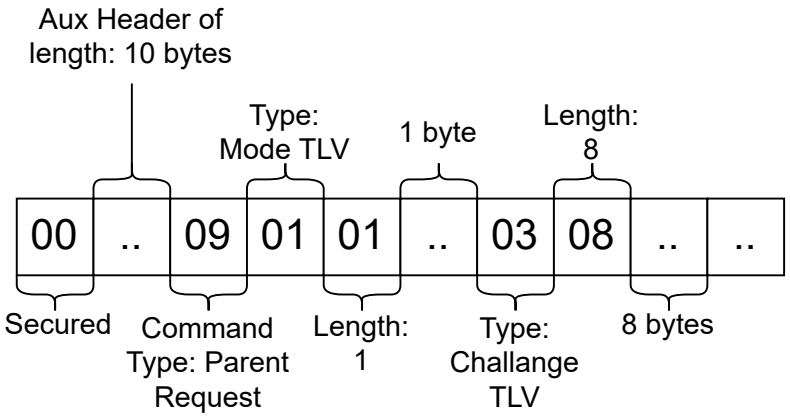


Figure 13: An decrypted hexadecimal representation of a Parent Request. As Parent Request messages must be encrypted, which is also represented by the first byte which are `00`. The difference from a not encrypted messages is the Auxiliary Security Header which contain information of the encryption used. The Auxiliary Security Header is 10 bytes long. Then it follow the TLV format as described in Section 1.5.

Encrypted MLE messages follows same format, but instead of the 2nd byte being the Command Type it is the 11th byte. This is due the Auxiliary Security Header, which carry information of how to decrypt the message, but since we omit encryption in OT we are not interested in this particular

information. Then the MLE messages follows the same structure, and we can simply go through all TLVs until reaching the end.

3.2.2 StateMachine

The StateMachine is the part of the Monitor, which keeps track of the OT server states. Since we do not have any other feedback than the response from the OT-instance, we make assumption sole on those as well as testcases. The principle behind the StateMachine is exploring the states and transitions of OT server and check if the state of the OT server does something unexpected. The simple flow of a testcase, which provoke unexpected behaviour found by the StateMachine is illustrated at Figure 14.

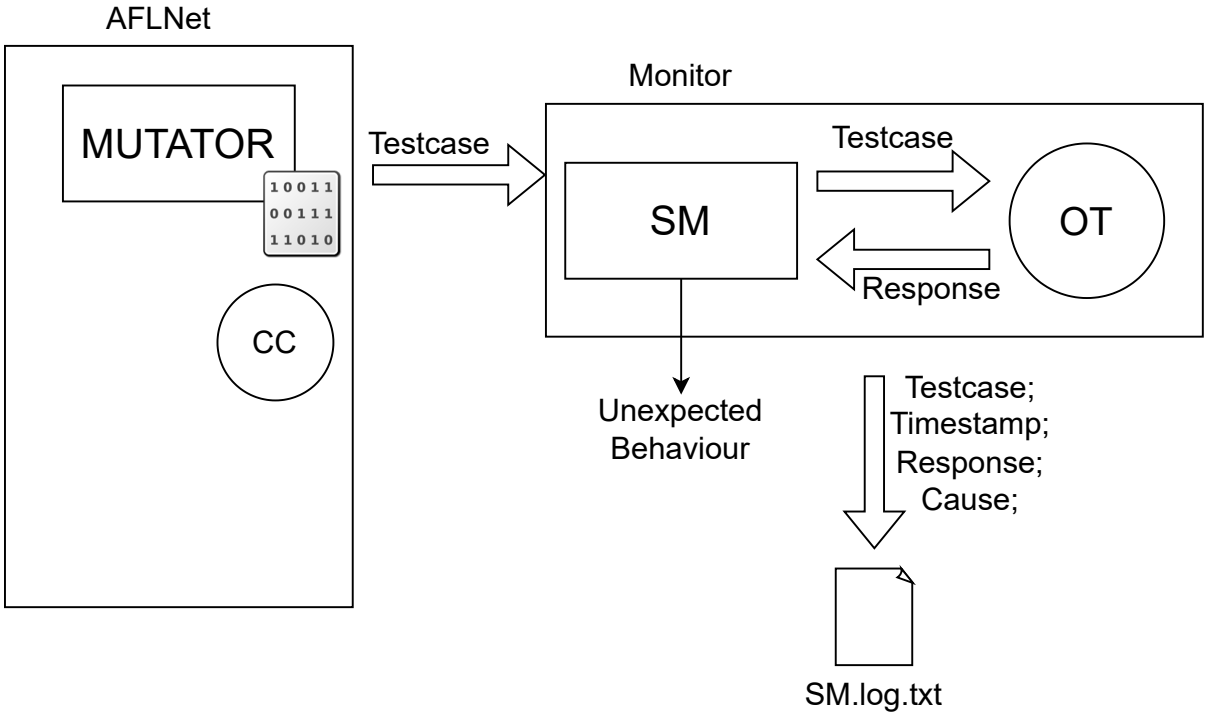


Figure 14: The flow of a testcase, which provoke unexpected behaviour at the target. In the case of unexpired behaviour found by the StateMachine, the testcase is logged in is own log file, different from crashes.

As earlier mentioned the scope of this report is just the subset of Thread Discovery and attaching-to-parent of the MLE protocol. Because we know the messages sent between the attaching device and the parent is ordered and must contain the TLVs dictated by the Thread specification [2], we are able to design the StateMachine such that every correct message to the OT-instance change the state. When a response is received from OT we can assume the instance was in a given state based on the response. Say the fuzzer send a correct Discovery Request, we can assume that the instance is processing the request until we get a Discovery Response and then we know for sure, that the Discovery Request was processed. Furthermore, we are able to look on the TLVs in the response and check whether they match with the specification or not. The StateMachine checks for this as well, we know exactly each TLV and the sequence of messages as described in Section 1.6, and the check can be done rather trivially. Three examples of how the StateMachine handles testcases and responses are shown on Figure 15.

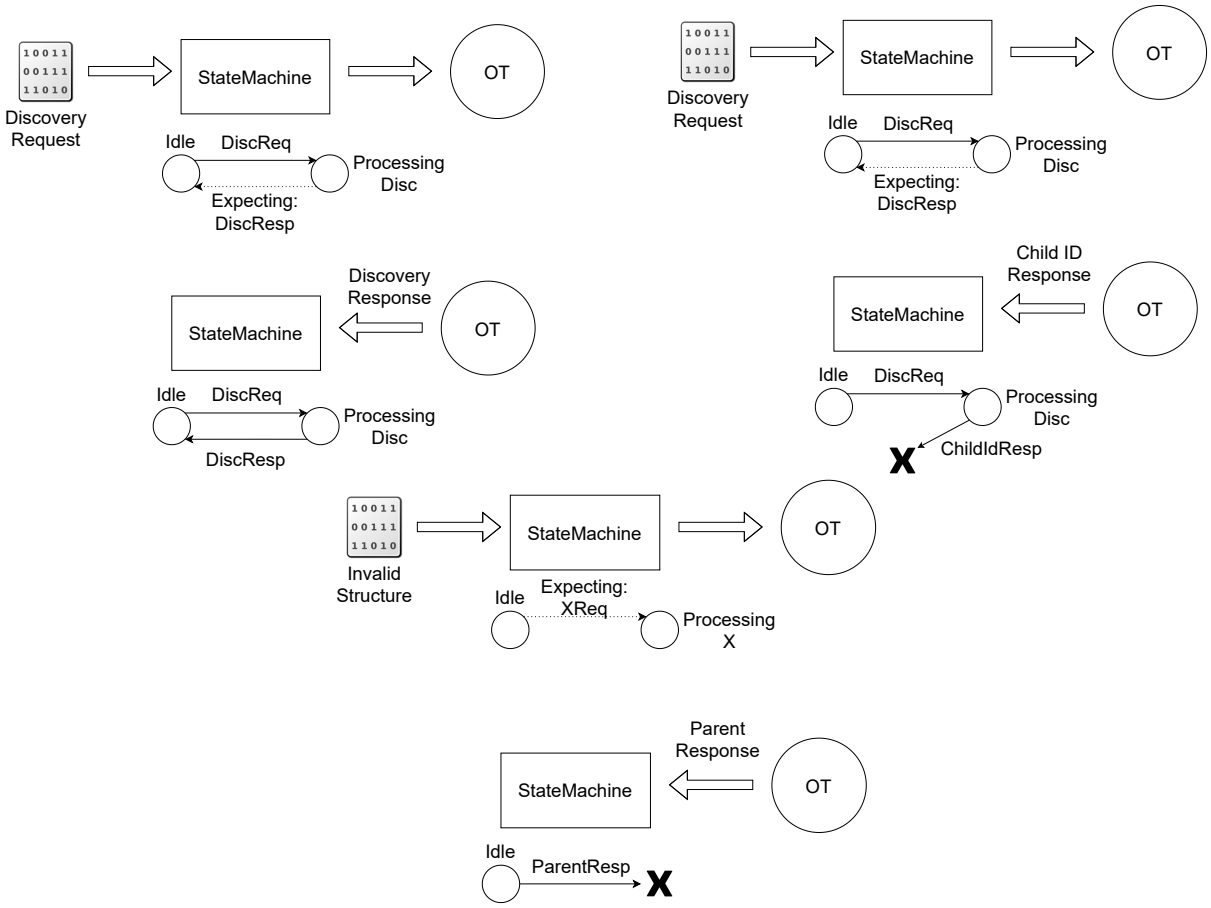


Figure 15: Three cases of the StateMachine, the first case (upper left corner) is a correct execution of OT with a well formed testcase. The second case (upper right corner) is incorrect run, where a well formed testcase is sent and received by OT but respond with a unexpected message. Last case (bottom) is the case of a invalid structure of a testcase and then OT respond with a well formed message. The two last cases is unexpected behaviour and is logged by the StateMachine.

A well formed or correct message is checked by the StateMachine by parsing each message using the Parser, and then depending of the Command Type of the message the corresponding TLVs are checked as well. Due to this simple check we found an interesting case in the implementation of OT, which is further described in Section 4.

3.3 Changes To AFLNet

In order to make AFLNet recognise and mutate messages correctly, we need to implement two new functions[7]. The two functions added is the `extract_requests_OT` and `extract_responses_OT` which are used to parse the input from .pcap files, and the output sent from Thread to AFLNet in order to do stateaware fuzzing. Since response and request follows the same format, MLE, is it relatively trivial to implement. Furthermore, does the implementation of `extract_requests_OT` and `extract_responses_OT` follows same format as the parser made in the Monitor-component, which is described in Section 3.2.1.

While the changes with AFLNet was trivial, we were unable to utilise all of AFLNet's features, and thus had to create our own components in order to fuzz OT. This lead to the solution of instead using

[7]The changes to AFLnet and the implementation of OT packages, can be found in `https://github.com/Speciale-Projekt/aflnet/tree/testing/ot`

a small binary as the handle to AFLNet, CCookie.

3.3.1 CCookie

The CCookie[8] is a small binary, with the sole purpose to interact with AFLNet. The CCookie is written in C and is compiled with the custom compiler of AFLNet. The CCookie is listening on an UDP socket and receiving messages from the Monitor-component, in order to signal AFLNet to send a new input. The CCookie is an important part in controlling the flow of inputs, since AFLNet is efficient in generating testcases, in fact is AFLNet able to, if not for the CCookie able to send hundreds of request per second. We experienced that a single OT instance is not able to handle the tidalwave of messages in correct sequence, and therefore we introduced the logic to time gate AFLNet. Since AFLNet is build upon AFL we are able to signal a correct run by stopping the CCookie with a return code of 0. First when this is registered by AFLNet it is allowed to send a new input. The other case, when OT does not respond, we have sat a static timeout by using a input parameter of AFLNet, which is a millisecond. This result in worstcase runtime per testcase is one millisecond. This is not achieved in the testsetup, probably due to physical constraints, which is explained in Section 4.2.

3.4 Changes To OpenThread

As the primary goal of Thread-Fuzz, is to fuzz the novel of the Thread protocol, MLE, and not the already known protocols in the Thread protocol stack, as IPv6 or IEEE 802.15.4, thus instead of making the Monitor-component able to communicate with 6LoWPAN, we modified the OT codebase, to allow direct UDP messages via UDP socket, and directly into the MLE message handling, and had OT write its responses to a file in addition to sending the message via 6LoWPAN, see Figure 10. In addition to this we also removed the encryption and decryption operation in OT, without changing how OT perceives a message to be encrypted, and thus we can still send messages in the encrypted format, where OT still acts as though the message is encrypted, up until the decryption point which is skipped, and OT can handle the message as if it was decrypted.

In this section we will go more into detail as to what changes has been made, and why it will not influence the quality of the results we present later on[9].

3.4.1 Direct connection to OT through UDP

When the simulation of OT is started, it starts by monitoring the multicast address `224.0.0.116`, the simulation of OT is using this address for all communication between nodes. Next its going to start a Command-Line interface (CLI) interface, where the user can start OT, with some Thread specific commands. As what the Thread radio expects to receive on the interface `224.0.0.116` is packages utilising the 6LoWPAN protocol, we chose to avoid communicating with OT through that interface, since it will require to wrap all messages as 6LoWPAN. Instead, we have abstracted the lower layers of the protocol stack away to only focus on the transport layer and the application layer.

The modification to OT looks substantial, though in essence it does not change much of the code which is needed for the transport layer and application layer to work as what is expected, first of a new thread is created when initialising OT, this Thread listens on the UDP port 5000 + `id of node` the id of the node is determined through the Command-line parameter talked about in Section 1.7, the job that the new thread introduces can be viewed in fully in Appendix C.1.

Now we just need to punch a hole through from the simulation into the `Mle::HandleUdpReceive`, we do this by introducing a function `handleUDP` in the API of OT, see code snippet below, which expose the function `Mle::HandleUdpReceive`. After compilation, it is then possible for `main.c` to call the C++ code located in the API of OT.

[8]The CCookie can be found in `https://github.com/Speciale-Projekt/CCookie`

[9]The current state of the modified version of OT can be found in our Github repository `https://github.com/Speciale-Projekt/openthread/tree/testing/aflnet`

```
void handleUDP(otInstance *aInstance, otMessage * aMessage, const otMessageInfo *aMessageInfo ){
    AsCoreType(aInstance).Get<Mle::Mle>().HandleUdpReceive(AsCoreType(aMessage),
    ↪ AsCoreType(aMessageInfo));
}
```

3.4.2 Print to file when sending message

To utilise a state-machine to verify the correct sequence of messages, the response package must also be received from the OT instance. OT is set up to send packages using the 6LoWPAN, and thus instead we changed OT to also write the same message to a file, that the Thread-Fuzz can read. The changes can be summarised to removing encryption, and in addition to sending the usual way via the simulated radio, it also writes to a file. The entire change in this regard can be found in the Appendix C.3.

3.4.3 Command line dataset and automatic instantiation

As we cannot interact with the interactive CLI-environment with AFLNet we need to instantiate the OT instance on startup. Usually this requires manual intervention from a user of the CLI-environment, and does not allow for automatic start of the OT instance. Thus, a more substantive change to the OT-simulation implementation, was the need to modify command-line parameters. We added a parameter to specify the dataset with `--dataset {"Network_Key": "cf70867da8d41fbdb614aa9677addf9e", "PAN_ID": "0x7063"}`, in addition we introduced a command `--master`, as two simulated OT instance cannot communicate if both of them has used the command `dataset init new`. The change to the code for adding this automated instantiation of the simulated OT instance, can be found in the Appendix B, where after initialising the OT instance, we call a series of commands based on the command parameter. Now the command to execute the simulated OT instance is; `./ot-cli-ftd 1 --master --dataset "{\"Network_Key\": \"cf70867da8d41fbdb614aa9677addf9e\", \"PAN_ID\": \"0x7063\"}"`, where the components of this command is; red: the program, brown: the `id` of the instance, violet: indicating that a new dataset should be created when the node is initialising, blue: the network key and pan id used with the simulated OT instance.

4 Evaluation

We ran in total 10 test with Thread-Fuzz, one is the base case where we have not introduced any errors, and then nine with introduced errors. Thread-Fuzz is able to detect five of which is to emulate a developer error resulting in a crash, demonstrating the fuzzing aspect of Thread-Fuzz, and five of which is errors in the sequence of message sending, demonstrating the statemachine aspect of Thread-Fuzz.

In this section we present the setup for running our test, and evaluate on the results obtained from these tests. We will also discuss how the tests represents actual errors that developers could have introduced in a real life program, and why these are good examples of failures, Thread-Fuzz can find in a Thread implementation such as OT.

4.1 Setup

The longer you are able to run a fuzz-test, the more combinations of different bytes we can test, and thus we can either find more flaws or be more certain that no errors exists. We ran all of our tests on four machines, all of which running some version of GNU/Linux OS, each of these split up into one laptop running Ubuntu with a 2 cored (4 threads) i7-5500U CPU 2.40GHz, and 8 gigabyte ram, and the rest being 3 desktops running Arch with a 4 cored (4 threads) i5-3470 CPU 3.20GHz, and 8 gigabyte ram. All computers were set up with a swap partition which was relevant for some tests, these swap partitions were all set up to be the same size as the ram.

Some of the tests only took a few hours before finding the expected bug, and we could then start the next test and therefore we could run several tests on the same machine in relatively quick concession, but others took much longer or did not find any errors, and thus ran for a longer time period. What tests ran on which machine can be seen in the Table 2. An image of the setup can be seen in the Appendix D

Table 2: Testcases ran on the computer setup

Comp_1	**Comp_2**	**Comp_3**	**Laptop**
T0	T1	Base (C0)	Base (C0)
T3	C1		C2
T4	C3		
C4			

The Base Case test, where no errors was introduced ran continuously for more than 600 hours on the labtop, before the introduction of the StateMachine, and once the StateMachine was introduced, we moved the base case test on to Comp_3, and the base case test was run for the remainder of the test period to avoid much downtime in our fuzz-testing.

The process of developing tests was we created a branch in our fork of OT, with the name of each test and commit it the relevant bug for the test. Then on the machine where the test was supposed to run, we cloned the testing repository `complete-scripts`[10], that contains git-submodules pointing to the master/main branch of our fork of OT, AFLNet and our components CCookie, Monitor-component and in addition some scripts that was relevant for running the tests, such as a `build.sh` script that makes sure that all components are build in the correct order, and a `run.sh` script that runs the applications. Then to run a specific test, we had to go into the git-submodule for OT and change branch to the relevant tests.

4.2 Experiments

In this section we describe each of the 10 tests as well as what we expect too see and if we have introduced a error, why we have chosen to introduce it. We have categorised the 10 tests after which behaviour we expect to see. The main purpose of each test is to test if Thread-Fuzz works as intended. It is a *C* test if we expect to see a crash, and a *T* test if we expect the StateMachine to capture unexpected behaviour. While the tests are categorised, we have to make it clear that the tests are categorised after what we expect, and due to the connection between unexpected behaviour and memory corruption, it is highly possible that some test results in both being logged in the StateMachine logs as well as the crash log.

4.2.1 C - For Crash Test

The following paragraphs present each of the test, which we expect to result in a crash or memory corruption.

C0 - The Base Case The *C0* test is running our setup with a fresh compiled OT-instance, and no errors has been deliberately introduced the only changes to the source code is introduced in Section 3.4. This is the most exciting test, since we can not say for sure if Thread-Fuzz will find any unexpected behaviour. If we see a mistake it is probably a simple developer mistake, and that is why we categorised this test to be a *C* test.

C1 - Off by One We have chosen to introduce a off by one mistake. This is meant to represent a common simple developer mistake. The mistake relies in the function `HandleUdpRecieve`, which is called whenever a MLE message is received. Whenever a MLE message is received by an OT-instance it checks whether if the message is encrypted or not, if it is encrypted it must be decrypted before checking the contents. Even though we removed the de-/encrypting function it still loops through the message as if the de-/encryption operation was applied. Since it is only the content of the MLE message that is encrypted, the decryption function is not applied to the header. We have inserted the off by one when the offset is moved to after the header, before doing the loop where the decryption should have happened. The small change:

[10] `https://github.com/Speciale-Projekt/complete-script`

```
aMessage.MoveOffset(header.GetLength() - 1); // before
aMessage.MoveOffset(header.GetLength());     // after
```

C2 - Remove Assertion in `WriteBytes` As a result of running *C0* did we become aware of a assertion statement which is called whenever bytes are written to a MLE message. This is mainly called when OT build a message for sending. The removed line is:

```
OT_ASSERT(aOffset + aLength <= GetLength());
```

It is a simple sanity check, where it is checked that the offset added with the length of the buffer is not longer than the message it self. Removing this line makes it possible to write out of the message objects bounds. Due to the discovery of the assert being called in specific cases, explained in Section 4.3, we can expect it OT to write somewhere unexpected provoking a crash or some kind of memory corruption.

C3 - Removed Valid Header Check Another sanity check is removed in this test, we have chosen to remove the check for valid headers in `HandleUdpReceive`. Whenever a MLE message is received the validity of the header is checked, and if it is not a valid MLE header the message is dropped. This check is trivial but important. The removed line is:

```
VerifyOrExit(header.IsValid() && header.GetLength() <= length, error = kErrorParse);
```

We can not say for sure what it results in, but after this check, it is assumed that the header is valid and checks on security/no-security or Command Type is done on the header.

C4 - Removed Valid Discovery Request Check This test is similar to `C3`, but instead of removing the check of the header, we removed the check for valid Discovery Request in `handleDiscoveryRequest`. As earlier described in Section 1.6, the Discovery Request is the only MLE message which is not encrypted and can be sent by unauthenticated devices. Which makes them of interest as being the only way to interact with an OT-instance without knowing the MLE key. The removed check is the deletion of the conditional statement:

```
if (discoveryRequest.IsValid())
{
    ...
}
```

The deletion of the statement is resulting in any message with the Command Type of 16, which is the Discovery Request, is treated like a Discovery Request. This mean even if the packet is malformed it will be treated as a Discovery Request if first byte is `0xFF` for no security and second byte is `0X10` for integer value 16.

4.2.2 T - For Unexpected Behaviour

The T-tests are tests targeting the behaviour and states of the OT-instance.

T0 - No Version TLV The Version TLV is used in the attaching-to-parent sequence to check of the two devices run a compatible Thread version. We have removed the check for the Version TLV in OT when handling Parent Requests, which results in Parent Requests without the Version TLV is accepted as if they had it. The removed lines are:

```
SuccessOrExit(error = Tlv::Find<VersionTlv>(aMessage, version));
VerifyOrExit(version >= OT_THREAD_VERSION_1_1, error = kErrorParse);
```

The Thread specification dictate that there must be a Version TLV in a Parent Request, and if that is not the case the MLE message should be dropped. But in the modified OT it treat is like a valid Parent Request and respond with a Parent Response. This is unexpected behaviour and should be logged by the StateMachine.

T1 - No check for Discovery TLV In this test we removed the check for Discovery TLV which is the only TLV included in the Discovery Request and Discovery Response. This mean that the OT-instance will send a Discovery Response to any Discovery Request, even if it does not include the Discovery TLV. This should be caught by the StateMachine, since it is unexpected behaviour of a correct Thread implementation. The removed line is:

```
VerifyOrExit(Tlv::FindTlvOffset(aMessage, Tlv::kDiscovery, offset) == kErrorNone, error = kErrorParse);
```

T2 - Handle all messages as Parent Request In this test we modified OT to handle all MLE messages as Parent Requests. It should be obvious that this should course problems. The StateMachine should log when a valid test is send that either no response or a Parent Response is sent back. The moved line in OT is:

```
    Get<MleRouter>().HandleParentRequest(aMessage, aMessageInfo);
```

From the code sample it is not clear what is changed, but the `HandleParentRequest` call seen above is moved out before the check on Command Type.

T3 - Handle all messages as Child ID Responses This is test is of same nature as *T2*. We have modified OT such that every MLE messages is handled as a Child ID Response. This is incorrect in multiple ways and obvious is of course that messages which are not Child ID Responses are still handled as one. The other incorrect assumptions here is the server should never handle a Child Response since it is only send by a attaching node. We expect to see nothing in this test, since the OT instance should not answer any message of this response. The modified code is:

```
    HandleChildIdResponse(aMessage, aMessageInfo, neighbor);
```

Which simply is moved out of the check on Command Type in `HandleUdpReceive`.

T4 - Missing Connectivity TLV In this test we removed the Connectivity TLV in OT when the Parent Response is build and send. This means that the OT-instance is sending a invalid Parent Response every time it gets a valid Parent Request, this is of course incorrect behaviour since the Connectivity TLV is an important part when the attaching device is choosing a parent. This should be logged by the StateMachine. The removed line is:

```
    SuccessOrExit(error = AppendConnectivity(*message));
```

4.3 Results

This section will cover the results of the tests, all of the results can be found in our github repository `https://github.com/Speciale-Projekt/results`.

4.3.1 C0 - Base case

Although this is not an unmodified OT-instance, as we did do the necessary modification discussed in Section 3.4, no error were deliberately introduced for this test, and thus it is essentially just the Thread-Fuzz fuzzer fuzzing the application layer of the Thread implementation OT. As seen in Table 2 this test ran twice, first on the Laptop and last on Comp_3. During the initial run of this test we found that during the right condition and messages the OT application would hard crash with a return code of 134[11], this is a `SIGABRT` and means that at some point in the code there is an assertion that is not satisfied, and this problem is not handled in the source code of OT, and as such crashing application.

When OT crashes it gives a line and file of where the crash occurred, and hereby are we able to find the exact location of the assertion. Namely in the function `WriteBytes`, the assertion is a sanity check for whether the length of the message, actually is the size that is indicated by the header of the message, the assertion is highlighted in Figure 16. We found that `WriteBytes` is called indirectly through our modification of OT, through the call to `otMessageWrite` see Appendix C.1. Therefore without looking further into the problem, we interpreted this as an error introduced by our code changes and not an error introduced by OT, so we disregarded it. After having the Base Case test run on the Laptop for approximately 3 weeks without the StateMachine, it accumulated this error in total 22 times, but no other faults was detected. We killed the process on the Laptop in favour of running the same test on Comp_3, but this time with a functioning StateMachine, and not long after the first instance of a crash with the same error code occurred, yet this time we had a functioning StateMachine. While looking at the log files produced by running the tests we found some examples of what we considered malformed `Discover Requests` that resulted in a `Discovery Response` from OT. We investigated this further, and found that it is possible to send a `Discover Request` with an arbitrary amount of TLVs, and the TLVs did not have to follow the Thread specification. In the Thread specification it is defined that a `Thread Discovery` TLV must be as shown in Figure 12 [2, p. 290], but we can send `Discovery Requests` with TLVs length of value longer and shorter than what we indicated in the second byte (length), and OT would respond with a valid `Discover Response`. This resulted in us doing manual testing of the OT application, with different set of messages, and even lead us back to our original find during the initial run of this test, we tried to throw messages that resembled the messages in the log, modifying it a bit, and sending it again.

This led to our discovery of a stack overflow error of OT on the unsecured MLE message, `Discovery Request`.

The stack overflow error that we found was based on that we can write more into the buffer than the size allocated to it, if we look at the code for allocating a received message see Figure 17.

[11] In python this is return code -6, and as we developed the StateMachine, the way we wrote the return codes into the logs changed to use the Python format rather than the C format. But we will keep referring to return codes as those of a C application to avoid confusion.

```
614 void Message::WriteBytes(uint16_t aOffset, const void *aBuf, uint16_t aLength)
615 {
616     const uint8_t *bufPtr = reinterpret_cast<const uint8_t *>(aBuf);
617     MutableChunk   chunk;
618
619     OT_ASSERT(aOffset + aLength <= GetLength());
620
621     GetFirstChunk(aOffset, aLength, chunk);
622
623     while (chunk.GetLength() > 0)
624     {
625         memmove(chunk.GetBytes(), bufPtr, chunk.GetLength());
626         bufPtr += chunk.GetLength();
627         GetNextChunk(aLength, chunk);
628     }
629 }
```

Figure 16: The assert that was not satisfied

It can be seen on line 2771, that only 64 bytes are allocated the buffer, and when a MLE message that is larger than these 64 bytes is send, then it will overwrite other objects on the stack with the rest of the message, such as `length` and `command`, if long enough the return pointer.

By sending the message:

```
"FF 10 1a ff 00 00 00 00" + "00" * 500
```

we even got an `SEGFAULT` exception from OT, meaning that we was indeed beginning to overwrite stuff in the stack that we should not be allowed to do, probably the return pointer.

Currently our assumptions leads us to believe that we write into the next declared variable `length`, which is later used in the assertion displayed in Figure 16 and thus the assertion is not satisfied. But further research is required to know this for certain. The implications of having a stack overflow is a very real possibility of arbitrary code execution, and as we can trigger this from outside a network, possibly affecting many devices at the same time, this could lead to a very severe security issue. Imagine being able to utilise some of already available reverse shell binaries from [52] to have a single message that we could broadcast in the middle of Thread network, and suddenly gain access to all devices within radio distance.

How to further research this vulnerability is discussed in the Future Works Section 8.

To fix this issue is simple enough. All that is required for OT is to insert a check validating the length of the received MLE message to not exceed the allocated size of the buffer. In regards to the fact that we can send non-Thread specification-compliant `Discovery Requests` and received a `Discovery Response` is just as easy. Thread requires a certain `Thread Discovery` TLV, and OT needs to check if the TLV is compliant with the specification.

```
2759 void Mle::HandleUdpReceive(Message &aMessage, const Ip6::MessageInfo &aMessageInfo)
2760 {
2761     Error              error = kErrorNone;
2762     Header             header;
2763     uint32_t           keySequence;
2764     const KeyMaterial *mleKey;
2765     uint32_t           frameCounter;
2766     uint8_t            messageTag[kMleSecurityTagSize];
2767     uint8_t            nonce[Crypto::AesCcm::kNonceSize];
2768     Mac::ExtAddress    extAddr;
2769     Crypto::AesCcm     aesCcm;
2770     uint16_t           mleOffset;
2771     uint8_t            buf[64];
2772     uint16_t           length;
2773     uint8_t            tag[kMleSecurityTagSize];
2774     uint8_t            command;
2775     Neighbor           *neighbor;
2776     bool               skipLoggingError = false;
```

Figure 17: Allocations of variables used within the function HandleUdpReceive

4.3.2 Rest of the tests

During our verification process of the error messages from Thread-Fuzz, we found an issue in the logging of the Monitor-component, due to the fact that we still utilise AFLNet for its mutation algorithm, the Monitor-component experiences some concurrency problems in regards to logging the messages that caused the problem. As whenever the CCookie receives a message it returns out, and AFLNet sends a new message to the Monitor-component, would happen at the same time as when OT would respond to the previous message, and thus when whenever a problem occurred, the Monitor-component would sometimes get a new message before logging, and write that message down instead. Resulting in that often the messaged logged, is the message *after* the problem occurred, and not the message that actually was at fault.

Thus a lot of investigation was required to determine which of messages was the cause of the problems introduced with the test. In the Table 3, the rows marked with a (?) we can see that we found the exception through the logs, but we are uncertain what message actually caused this problem. How to fix this issue, is discussed in the Future Works Section 8.

In addition to this, we found that errors accumulated over sequential runs, most likely due to the problem described in Section 4.3.1, and as thus our logging tool would report this error even on valid packages. A possible solution to this issue is also discussed in the Future Works Section 8, but it is in turn easy to disregard these messages as a valid message is trivial to distinguish from a invalid one.

4.3.2.1 The cases we did not find an error in

Out of the ten cases we ran, only four resulted in us not finding an error, or only finding the same error as described in the Base Case. We will go through the reason for us not finding these error in this subsection.

C1 - Off by One This test was created with the mindset of inconsiderate developer were to forget the difference between index and size of a list. Removing the -1 from a lookup in the code base would under normal circumstances result in unexpected behaviour as its addressing something outside of the allocated size of the array. But as we designed this test with an unmodified OT instance in mind, we did not account for the fact that we removed decryption of the message, which is where this error would have been thrown. Thus, in our modified version of OT no exception is raised, and Thread-Fuzz did therefor not find any error, besides the one described in the Base Case.

Table 3: Results from running our tests

Test name	Did the fuzzer find an error (Yes/No)	SM / Crash / Both
C0 - The Base Case	Yes	Both
C1 - Off by One	No	
C2 - Remove Assertion in `WriteBytes`	Yes(?)	Crash
C3 - Valid Header Check	No	
C4 - Valid Discovery Request	No	
T0 - No version TLV	Yes(?)	SM
T1 - No check for Discovery TLV	Yes	SM
T2 - Handle all messages as ChildIDResponses	No	
T3 - Handle all messages as ParrentRequest	Yes	SM
T4 -Missing Connectivity TlV	Yes	SM

C3 - Valid Header Check For this case we removed a check for whether the header of the message was correct, as per the Thread Specification. The assumption being that it would propagate to a larger error when Thread-Fuzz got to fuzz on the messages and in turn corrupt the header of the message. But alas OT had a more robust set-up in regards to the header than they do in almost all other aspects of the MLE message, and thus no additional errors was found for this test.

C4 - Valid Discovery Request We did not find any error through the Thread-Fuzz, although the following message would result in a StateMachine error;

```
"FF 10 1a"
```

This is because `discoveryRequest.IsValid()` essentially checks whether the `Thread Discover` TLV is valid, and the only thing that OT checks on in regards to this is whether that the TLV is at least 2 bytes long (Type and Length).

An error in the Monitor-component results in that it is unable to handle a TLV of only one byte, and is not sent to OT, and for this reason (and quite possibly because AFLNet favours appending bytes rather than sending small packages) this exception is never caught.

T3 - Handle all messages as ChildIDResponses As expected OT did not respond to this response. Future implementation of a more in-depth StateMachine might use this test to walk through the entire process of becoming a Node within the OT network.

5 Discussion

In this section we want to take a step back, and take a look upon the method used when developing Thread-Fuzz and discuss the promises of fuzzing compared to our experience. First we want to discuss the categorisation of Thread-Fuzz and why it is hard to compare the idea presented with other fuzzing approaches. Then we will discuss the limitation of AFLNet and the compatibility between AFLNet and real implementations of networks protocols, with the context of OT. Lastly we will discuss our approach to building Thread-Fuzz and what changes to our approach in hindsight.

5.1 Categorisation of Thread-Fuzz

We can easily categorise Thread-Fuzz to be a mutation-based blackbox fuzzer, due to the mutation algorithm used via AFLNet and due to the only feedback from the target is the output. That said Thread-Fuzz introduce a unique perspective on fuzzing, by assuming states of the application and hereby check for unexpected behaviour which is not crashes or memory corruption. Thread-Fuzz is basically searching for a input to Thread implementation which leads to behaviours which is not described in the Thread specification. Due to not being able to classify and categorise Thread-Fuzz in the known terms, are we suggesting a new term, which is *behavioural* fuzzing. Behavioural fuzzers is the set of all fuzzers which search for unexpected behaviour, this mean that the traditional fuzzers are included in behavioural fuzzers as well. With the new term of behavioural fuzzing, we can categorise Thread-Fuzz to be a mutation-based behavioural blackbox fuzzer.

5.2 Usability of AFLNet and The Promises of Fuzzing

While AFL is widely used and popular, its functionality is still limited to a small subset of possible fuzz targets. However it is modular of design and hereby easy extendable. This is one of the major points of using the successor AFLNet. AFLNet implements the notion of states and message sequences, in order to get a better coverage. While it is relatively easy to implement the support for Thread in AFLNet, we met major issues with the instrumentation and the amount of dummy code, code needed in order to do fuzzing. We experienced that the semantics of OT changed when compiling the source code with custom compiler of AFL. As an attempt to solve this problem, we directed the custom compiler, but due to the size of OT code base, it was not trivial. Linking the custom compiled code with the rest was cumbersome, since OT uses autogenerated makefiles and we had to solve the linking by hand. This did not solve the problems with the custom compiler, which led us to the current solution of not using instrumentation of OT at all, but instead using a dummy program, CCookie, to control the flow of AFLNet.

In order to do efficient fuzzing it is possible to change the source code of the target. We choose to remove encryption operation and expose the function that handles MLE messages, while we had experience with Thread, we have limited experience with the code base of OT. This led to a throughout investigation of the relatively large codebase of OT. While the examples of AFLNet was on small protocols, it have become clear that it is not suited for real implementations of network protocols such as OT. To back the statement up is the successor of AFLNet Snapfuzz[5], which was presented in the same time period as the report was written. All in all the most appealing features of greybox fuzzing is the limited code analysis and the off-to-go solution presented by AFLNet, but in our experience the usability of AFLNet is not there quite yet.

5.3 Thread-Fuzz in Hindsight

While Thread-Fuzz proved to working, somehow, as intended it is still unpolished. We saw in Section 3 that Thread-Fuzz was able to find unexpected behaviour, and even one suspected flaw in the current implementation of OT. We believe alot of the complexity of making the tool was getting AFLNet to work properly, and in hindsight, we believe that Thread-Fuzz would be more mature if it had is own mutation algorithm and did not depend on AFLNet. Furthermore, are the key-features of AFLNet not used by Thread-Fuzz and therefore it is hard to argue that we keep the connection to AFLNet. An alternative to use AFLNet as mutatator would be writing a Thread specialised mutator, which iw aware of the TLV structure by using fuzzing libraries such as LibAFL[1].

Another change which we believe, in hindsight, could have improved Thread-Fuzz immensely is the use of an integrated tool using the theory of model checking, such as UPPAAL TRON[35]. Instead of defining simple static checks such as the chronological order of messages as in the StateMachine currently, we might be able utilise UPPAAL TRON to make and check more complicated queries without too much effort.

6 Related Work

Although fuzzers has been around since B. Miller's first introduction to the topic in the 1980's, recently the topic has gained a lot of traction, and a lot of new and exiting fuzzing tools has been developed just within the last few years. Not to forget that many new conferences focused exclusively on this subject, some not much more than one year old [31, 29, 30, 28].

AFL++ is the successor to the greybox fuzzer AFL, expanding on the ground that AFL already layed out, it incorporate even more mutators, and better support for compilers such as LLVM. *AFL++* promises to close the gap between industrial and research, by building on top of AFL and still provide an easy-to-use experience for the user. [18] As the successor to the no longer maintained AFL [58], one could speculate that in terms of non-protocol fuzzers, *AFL++* is going to win even more ground, until a better fuzzer comes a long.

SnapFuzz is just like Thread-Fuzz build on top of AFLNet utilising the mutation-algorithm for fuzzing network protocols, by creating a fork-server that can take the output of AFLNet and awaiting when the target is ready to receive a new package. Doing it this way they overcome the delays that is caused by using user specified delays in AFLNet, and thus is able to archive a much higher throughput of the fuzzer. One of the major issues of running a fuzzer with a network protocol as a target, is to make sure that each iteration of the fuzzing happens on a "clean slate", such that a faulty state from a previous run, does not interfere with the next run. *SnapFuzz* overcomes this obstacle by using a temporary file-system (`tmpfs`), and thus the clean-up phase between runs is trivialised. [5]

Though contrary to Thread-Fuzz, *SnapFuzz*'s primary focus in on increasing the throughput of AFLNet, and reducing the *fuzzing harness* (the code that the user needs to write before being able to fuzz a network protocol). During the implementation of Thread-Fuzz we had many struggles with this *fuzzing harness* and as a result, we introduce the notion of moving away from AFLNet in our discussion. Though *SnapFuzz* overcomes this obstacle by removing the need for manually introduce timed delays and removing the need for creating a clean-up script for in between runs. [5]

For OT we did not have to introduce neither a hard-coded timed delay, nor a clean-up script, and therefor *SnapFuzz*'s changes would thus not leverage much of the obstacles that we experienced during our development with AFLNet on OT, besides this, even if we did gain something by introducing these functionalities of *SnapFuzz*, they have unfortunately decided to keep their source-code closed-sourced. Though in regards to increasing the throughput of AFLNet, the removal of having to create manually timed delays and instead introduce a forkserver that would inform AFLNet when the target was ready to receive a new package, they successfully increased the speed at which they were able to find the same errors without their implementation. The speed-up ranged from 8.6 times for the protocol `Dcmqrscp` to 62.8 times for the protocol `LightFTP`. [5]

Driller is a whitebox fuzzer, utilising concolic execution (also known as dynamic symbolic execution) to step through difficult conditions that are almost impossible to fuzz the result of. Contrary to Thread-Fuzz, *Driller* requires a direct access to the source code of the application that it is fuzzing so that it can calculate the required parameters for difficult conditions. Besides that *Driller* is not build for network protocols, and thus leverages AFL for the mutational algorithm, instead of AFLNet. [54]

Muzz is a greybox fuzzer, with a increased focus on concurrency problems, which AFL and its derivations has problems with [43]. By creating a thread-aware feed-back for *Muzz* they are able to adapt their mutation algorithm to promote seeds that explore multiple branches of the multi-threaded application, while demoting seeds that only explores the same branches in what they describe as the "Matthew Effect" [12] (the better the seeds the more they will be used for generation, vice versa with bad seeds). They conclude that *Muzz* is significantly better at finding concurrency issues than AFL. [12]

7 Conclusion

We gave a overview of the Thread protocol, with focus on MLE, the novel part of Thread. We then presented the different fuzzing approaches in order to establish terminology for Thread-Fuzz. We presented architecture, design choices of Thread-Fuzz as well as how the different components interact each other in order to fuzz OT. We tested the implementation of Thread-Fuzz by running ten test where of Thread-Fuzz found five of the introduces bugs and one in the current implementation of OT. The impact of the found bug needs more research and we can only speculate, but it could worst case lead to unauthenticated remote code execution. Lastly we discussed the suggestion of the term *behavioural* fuzzers, which contain a bigger set than the traditional fuzzers, that only searched for crashes and memory corruption, as well as the flaws of AFLNet and the developing of Thread-Fuzz in hindsight.

We can conclude that it is possible to incorporate the assumption of states in a blackbox fuzzer, and use these assumption of states to search for wrong handling of inputs. The proof-of-concept implementation of a mutation-based behavioural blackbox fuzzer for the network protocol Thread is working and able to discover incorrect behaviour in the implementation compared to the specification. While Thread-Fuzz is unpolished it still proves, the idea of finding other incorrect behaviours than crashes and memory corruption, can be done with fuzzing. We can conclude that Thread-Fuzz could benefit of some major changes, such as implementing its own mutator and using third party tools such as UPPAAL TRON to handle the logic of states.

8 Future Work

It is clear that idea of checking behaviour based on target program states is possible, and can be expanded. Thread-Fuzz only covers a small subset of the Thread protocol and it can easily be imagined that the idea can be expanded to the rest of Thread as well as other protocols. In this section we want to present the immediate next step for Thread-Fuzz and the future work for some of the open ended question, that results of this report.

Repair Logging in Thread-Fuzz The most obvious features which must be fixed before considering Thread-Fuzz as functional fuzzing tool is the logging function. Without correct logging the message that causes unexpected behaviour, it is hard to prove that a given behaviour exists and furthermore it is near impossible to research and find a possible fix. This is a critical feature and due to the triviality of the problem in Thread-Fuzz it is hard to argument, that this is not the immediate next step for Thread-Fuzz. As we earlier discussed does the current problem rely in a concurrency problem, and we have discussed two possible solutions to the problem. The first solution is to decouple from AFLNet and then keeping testcases in memory, while it is obvious how this solves the problem, it will makes us able to control the flow of data much more accurate. Hereby making the concurrency problem obsolete, by simply not sending any messages before we get a response or after a specified timeout. The second solution is to save all messages in the Monitor, hereby able to see the full history of messages and responses up until the crash. This feature is also a great tool investigate crashes provoked by an accumulation of messages. Otherwise it would also make sense to have an option where the OT instance is reset between every message sequence in order to have a clean run every time.

Removing dependencies to AFLNet As earlier described, are we of that belief that Thread-Fuzz can benefit of cutting it ties to AFLNet. This will result in Thread-Fuzz need another mutator, which can be implemented using LibAFL, the fuzzing library based on AFL. Furthermore is it of interest make the mutator aware of the TLV structure. This can be in form of mutation operations such as adding and removing TLVs. Removing the coupling to AFLNet and making the mutator a part of the Monitor-component would also result in a more efficient run. Due to the communication between AFLNet and the Monitor-component is done via sockets, it is possible to increase the time per execution, by instead using shared memory. Furthermore is trivial to fix the logging problem mentioned earlier.

Adding Support for More MLE Thread-Fuzz cover a small subset of MLE and it can easily be extended to support for other important parts, such as Commissioning, Network Data sharing between

nodes or Partitioning.

Integration of UPPAAL TRON In Section 5 did we mention the use of an third party tool to track changes of the target, and check if it satisfy the Thread specification, Uppaal TRON. Uppaal TRON[35] is a blackbox conformance testing tool, which utilise the Uppaal model checker engine in order to get efficient model exploration. Interaction with Uppaal TRON is much like the currently StateMachine just input and output channels. It is of big interest to implement Uppaal TRON as replacement of the StateMachine.

Further Investigation of The Bug in OT As explained did we observe a potential stack overflow in OT, and as is are we not able to determine whether of this is a real problem, minor or not existing. We believe it is real due to its nature, the bug it self allows to make writes to the stack of any length. This can potentially be exploited to get remote code execution, but we can not say for sure without further investigation. Best case, does there exist a check on the length on a lower level in the protocol stack, and hereby does are MLE messages that are longer than specified in dropped.

Further Investigation of The Instrumentation Technique Presented by AFL The instrumentation technique used by AFLNet is inherited by the predecessor AFL, and is used in the many inheritances of AFL. As earlier discussed, did we experience a change in the semantics of the program after we compiled the source code of OT with the custom compiler. There might be an unexpected side effect of inserting logging functions in the assembly, but we are not able to tell for sure without deeper research of the instrumentation. We already know that the custom compiler used to compile OT is the least recommended one, but due to OT is written in C99 and C11 code, we are not able to use the `afl-clang-fast` which is the LLVM solution and recommended.

Acknowledgements

We would like to thank our supervisors René Rydhof Hansen and Danny Bøgsted Poulsen for their guidance and feedback.

A Thread MLE TLVs

Name	Type	Length in bit	Type Description	Note
Source Address	0	16	A senders' 16-bit MAC address	Must be included whenever the sender has a valid 16-bit MAC address
Mode	1	8	A byte string representing the link mode used by the source of a message	Format defined in Section **??**
Timeout	2	32	A 32-but unsigned integer, which is used as the expected maximum interval between transmissions in seconds	explained in more detail in Section **??**
Challenge	3	max 64	A randomly-chosen byte string, used to determine the freshness of replies to a message	Must be at least 4 bytes a new value must be choosenfor each Challenge
Response	4	8	A byte string copied from a Challenge TLV	
Link-layer Frame Counter	5	32	The senders' current outgoing link-layer Frame Counter	Encoded as an 32-bit unsigned integer
MLE Frame Counter	8	32	The senders' current outoing MLE Frame Counter	Encoded as a 32-bit unsigned integer
Route64	9	32	Used for distribution of of active Router IDs and routing information	See chapter 5 in [2] for more details
Address16	10	16	A 16-bit MAC address sent by a Parent to a new Child, for the Child to be assigned this address	Also used to reconfirm a Router neighbor's 16-bit MAC address
Leader Data	11	32	The Network Leader Data	Format defined in Section **??** and Figure 4-6 in [2]
Network Data	12	32	The sender's current Network Data	TLV encoding is described in [2] section 5.18
TLV Request	13	8	A list of TLV codes that indicates requested TLVs by the sender	Format defined in [2] Figure 4-7

Table 4: TLVs types with descriptions and references to format definitions when necessary. The table is continued in Table 5

Scan Mask	14	8	Flags that indicates which devices types that should responds to a multicast request	Format defined in Section ??
Connectivity	15	32	Shows how connected the sender is to other devices	Format defined in [2] Figure 4-9
Link Margin	16	8	Sender's calculated link margin in dB for the destintation	Format defined in Section ??
Status	17	8	Status response to a request	Format defined in [2] Figure 4-11
Version	18	16	Version number of the Thread Protocol implemented by the sender	Format defined in [2] Figure 4-12
Address Registration	19	32	Zero or more addresses that have been configure by the source of the message	Format defined in Section ??
Channel	20	24	The channel page and channel of adjacent Network partitions operation on a different Active Operational Dataset	Format defined in [2] Figure 4-14
PAN ID	21	16	PAN ID of an adjacent Network Parition operation on a different Active Operational Dataset	Format defined in [2] Figure 4-15
Active Timestamp	22	32	An Active Timestamp	Format defined in [2] Figure 4-16
Pending Timestamp	23	32	A Pending Timestamp	Format identical to TLV type 22
Active Operational Dataset	24	32	Sender's Active Operational Dataset encoded as a series of Network Management TLVs	For more details see Chapter 8 in [2]
Pending Operational Dataset	25	32	Senders' Pending Operational Dataset encoded as a series of Network Management TLVs	For more details see Chapter 8 in [2]
Thread Discovery	26	Any	A series of Mesh Commissioning Protocol TLVs used for network discovery on IEEE 802.15.4 interfaces	For more details see section 8.4.4.1.1 in [2]

Table 5: Continuation of table 4. TLVs with type description and references to format definitions when necessary

B Description of Variables Used in the Thread Network Data sets

The following appendix is an taken word by word from the unpublished article [4].

B.1 Valid Prefix Set

- `P_prefix` which is the IPv6 prefix for the Thread Network. The prefix and what it is used for is explained in 1.3.3.
- `P_domain_id` which identifies the Provisioning Domain.
- `P_border_router_16` which is the Routing Locator (RLOC) for the Border Router providing this prefix.
- `P_stable` determines if this prefix is considered stable.
- `P_on_mesh` will show whether the `P_prefix` is On-mesh.
- `P_preferred` identifies whether the address which has been auto-configured using the `P_prefix` is preferred.

- P_slaac determines whether the nodes in the network is allowed to create an auto-configured address based on the P_prefix.
- P_dhcp determines whether the address located in P_border_router_16 is a DHCPv6 Agent, that manages the address configuration for the P_prefix.
- P_configure determines whether the address located in P_border_router_16 is a DHCPv6 Agent, that supplies configuration data.
- P_default determines if P_border_router_16 offers a default route, for messages that uses the P_prefix.
- P_preference will show the preference of the route offered in P_border_router_16 if it offers a default route, determined by P_default.
- P_nd_dns describes whether the Border Router located at P_border_router_16 is able to supply DNS information.

B.2 External Router Set

- R_domain_id identifies the Provisioning Domain, that this data set is associated with.
- R_border_router_16 is the RLOC of the Border Router.
- R_prefix is the IPv6 prefix for the route.
- R_stable determines if this data set is considered stable.
- R_preference is the preference of the external router this data set is associated with.

B.3 6LoWPAN Context ID Set

- CID_id is the Context ID for the 6LoWPAN
- CID_prefix is the IPv6 prefix in which the 6LoWPAN encodes.
- CID_stable determines if this context id is considered stable.

B.4 Server Set

- S_enterprise_number is the number assigned to the device by the Internet Assigned Numbers Authority, to the vendor that designed the server in question. If no such assignments has been made the default value in thread is 44970.
- S_service_date is any information relevant to servicing of the server, currently this is unspecified exactly how to use it in the Thread Specification.
- S_server_16 describes the RLOC of the server.
- S_server_data is the data containing server-specific information. This has not been clearly defined as to how Thread Servers should behave yet.
- S_stable determines if this Server data set is considered stable.
- S_id is the ID of the server assigned by the Leader of the TNP. This value is between 1 and 15. If two servers share the same S_enterprise_number and the same S_service_data then they also share the same S_id.

C Code changes in OpenThread

This appendix contains the code changes that was made to OT, such that a more streamlined Fuzz testing of the application layer could be done.

C.1 Direct connection to OpenThread with UDP

```
void *udpSocketListener(void *instance) {
    struct sockaddr_in serverAddr, clientAddr;
    int                listenAddr   = 5000 + id;
    char              *listenDomain = "127.0.0.1";
    int                sockfd       = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        exit(1);
    }
    bzero(&serverAddr, sizeof(serverAddr));
    serverAddr.sin_family      = AF_INET;
    serverAddr.sin_port        = htons(listenAddr);
    serverAddr.sin_addr.s_addr = inet_addr(listenDomain);
    // Bind socket
    if (bind(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0) {
        exit(1);
    }
    otMessageSettings *settings    = malloc(sizeof(otMessageSettings));
    settings->mLinkSecurityEnabled = 0;
    settings->mPriority            = 1;
    otMessage *aMessage;

    otMessageInfo *b = malloc(sizeof(otMessageInfo));
    b->mLinkInfo     = malloc(2);
    b->mHopLimit     = 255;

    while (1) {
        // Get message
        char buffer[1024];
        socklen_t clilen = sizeof(clientAddr);
        ssize_t n = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *) &clientAddr, &clilen);
        aMessage = otUdpNewMessage(instance, settings);
        if (n > 0) {
            if (otMessageSetLength(aMessage, sizeof(buffer)) != OT_ERROR_NONE) {
                perror("message write");
            };
            if (0 > otMessageWrite(aMessage, 0, buffer, sizeof(buffer))) {
                perror("message write");
            };
            handleUDP(instance, aMessage, b);
        }
        otMessageFree(aMessage);
    }
}
```

C.2 Instantiation of a Thread node without CLI-interactions

```
4907 //[...]
4908 if (networkKey != NULL && panId != NULL)
4909 {
4910     if (useAsMaster) {
4911 
4912         sprintf(command, "dataset init new");
4913         Interpreter::sInterpreter->ProcessLine(command);
4914     }
4915     sprintf(command, "dataset networkkey %s", networkKey);
4916     Interpreter::sInterpreter->ProcessLine(command);
4917     sprintf(command, "dataset panid %s", panId);
4918     Interpreter::sInterpreter->ProcessLine(command);
4919     sprintf(command, "dataset commit active");
4920     Interpreter::sInterpreter->ProcessLine(command);
4921     sprintf(command, "ifconfig up");
4922     Interpreter::sInterpreter->ProcessLine(command);
4923     sprintf(command, "thread start");
4924     Interpreter::sInterpreter->ProcessLine(command);
4925     free(command);
4926 }
4927 //[...]
```

C.3 Write to file when sending UDP message

```
Error Mle::SendMessage(Message &aMessage, const Ip6::Address &aDestination)
{
    //[...] The declaration of code
    FILE            *fp = fopen("child.bin", "w+"); // New

    IgnoreError(aMessage.Read(0, header));
    uint16_t offset = aMessage.GetOffset();

    if (header.GetSecuritySuite() == Header::k154Security)
    {
       // [...] Reads from header, used in encryption
        while (aMessage.GetOffset() < aMessage.GetLength()) //New
        {
            length = aMessage.ReadBytes(aMessage.GetOffset(), buf, sizeof(buf));
            fwrite(buf, 1, length, fp);
            aMessage.MoveOffset(length);
        }

        aMessage.SetOffset(header.GetLength() - 1);

        // Usual send message, encryption is removed
        while (aMessage.GetOffset() < aMessage.GetLength())
        {
            length = aMessage.ReadBytes(aMessage.GetOffset(), buf, sizeof(buf));
            // aesCcm.Payload(buf, buf, length, Crypto::AesCcm::kEncrypt);
            aMessage.WriteBytes(aMessage.GetOffset(), buf, length);
            aMessage.MoveOffset(length);
        }

        // aesCcm.Finalize(tag);
        SuccessOrExit(error = aMessage.AppendBytes(tag, sizeof(tag)));

        Get<KeyManager>().IncrementMleFrameCounter();
    } else { // New
        aMessage.SetOffset(offset);

        while (aMessage.GetOffset() < aMessage.GetLength())
        {
            length = aMessage.ReadBytes(aMessage.GetOffset(), buf, sizeof(buf));
            fwrite(buf, 1, length, fp);
            aMessage.MoveOffset(length);
        }
    }
    fflush(fp); // New
    fclose(fp); // New
    // [...] sets message info based on destination
    SuccessOrExit(error = mSocket.SendTo(aMessage, messageInfo));

exit:
    return error;
}
```

C.4 Remove decryption from HandleUDPReceive

```
//[...]
 aMessage.MoveOffset(length);
}

// aesCcm.Finalize(tag);
/*#ifndef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION
    if (memcmp(messageTag, tag, sizeof(tag)) != 0)
    {
        // We skip logging security check failures for broadcast MLE
        // messages since it can be common to receive such messages
        // from adjacent Thread networks.
        skipLoggingError =
            (aMessageInfo.GetSockAddr().IsMulticast() &&
↪  aMessageInfo.GetThreadLinkInfo()->IsDstPanIdBroadcast());
        ExitNow(error = kErrorSecurity);
    }
#endif*/

if (keySequence > Get<KeyManager>().GetCurrentKeySequence())
//[...]
```

D An image of the testing setup

Figure 18: Laptop on top of Comp_1 through Comp_3. The Computer in the background is not used due to a hardware error.

References

[1] *AFLplusplus/LibAFL: Advanced Fuzzing Library - Slot your Fuzzer together in Rust! Scales across cores and machines. For Windows, Android, MacOS, Linux, no_std, ...* https://github.com/AFLplusplus/LibAFL. (Accessed on 06/13/2022).

[2] Robert Alexander et al. *Thread Specification*. ThreadGroup, 2017.

[3] *american fuzzy lop*. https://lcamtuf.coredump.cx/afl/. (Accessed on 05/30/2022).

[4] Christoffer Sand Andersen et al. "Towards Modelling and Verification of the Thread Protocol". In: (2022). URL: https://projekter.aau.dk/projekter/da/studentthesis/towards-modelling-and-verification-of-the-thread-protocol(70b7397d-1e31-4dc6-9f66-72bb80b9bb63).html.

[5] Anastasios Andronidis and Cristian Cadar. "SnapFuzz: An Efficient Fuzzing Framework for Network Applications". In: (Jan. 2022).

[6] Clark Barrett et al. "Satisfiability modulo theories". eng. In: *Frontiers in Artificial Intelligence and Applications*. Vol. 185. 1. 2009, pp. 825–885. ISBN: 1586039296.

[7] Marcel Böhme et al. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2329–2344.

[8] Ella Bounimova, Patrice Godefroid and David Molnar. "Billions and billions of constraints: Whitebox fuzz testing in production". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 122–131. DOI: 10.1109/ICSE.2013.6606558.

[9] Chiara Buratti, Roberto Verdone, Gianluigi Ferrari et al. *Sensor networks with IEEE 802.15. 4 systems: distributed processing, MAC, and connectivity*. Springer Science & Business Media, 2011.

[10] *Burp Suite - Application Security Testing Software - PortSwigger*. https://portswigger.net/burp. (Accessed on 06/02/2022).

[11] Marcel Böhme, Van-Thuan Pham and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 489–506. DOI: 10.1109/TSE.2017.2785841.

[12] Hongxu Chen et al. "{MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2325–2342.

[13] Edmund M Clarke. "Model checking". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1997, pp. 54–56.

[14] Cloudflair. *What is User Datagram Protocol (UDP/IP)*. URL: \url{https://www.cloudflare.com/learning/ddos/glossary/user-datagram-protocol-udp/}.

[15] *ClusterFuzz - ClusterFuzz*. (Accessed on 06/14/2022).

[16] *CVE - CVE-2021-3156*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156. (Accessed on 06/13/2022).

[17] Zakir Durumeric et al. "The Matter of Heartbleed". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, 475–488. ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. URL: https://doi.org/10.1145/2663716.2663755.

[18] Andrea Fioraldi et al. "{AFL++}: Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020.

[19] Andrea Fioralldi et al. "Registered report: Dissecting american fuzzy lop - A fuzzbench evaluation". In: *FUZZING 2022, 1st International Fuzzing Workshop, 24 April 2022, San Diego, CA, USA / Co-located with NDSS 2022*. San Diego, 2022.

[20] Bjorn N Freeman-Benson, John Maloney and Alan Borning. "An incremental constraint solver". In: *Communications of the ACM* 33.1 (1990), pp. 54–63.

[21] Vijay Ganesh, Tim Leek and Martin Rinard. "Taint-based directed whitebox fuzzing". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[22] Patrice Godefroid, Adam Kiezun and Michael Y. Levin. "Grammar-Based Whitebox Fuzzing". In: New York, NY, USA: Association for Computing Machinery, 2008, 206–215. ISBN: 9781595938602. DOI: 10.1145/1375581.1375607. URL: https://doi.org/10.1145/1375581.1375607.

[23] Patrice Godefroid, Michael Y Levin and David Molnar. "SAGE: whitebox fuzzing for security testing". In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[24] *googleprojectzero/winafl: A fork of AFL for fuzzing Windows binaries*. https://github.com/googleprojectzero/winafl. (Accessed on 06/13/2022).

[25] Thread Group. *Thread Network Fundamentals*. English. 2020. URL: https://www.threadgroup.org/Portals/0/documents/support/ThreadNetworkFundamentals_v3.pdf.

[26] Thread Group. *Thread Usage of 6LoWPAN*. English. 2015. URL: https://www.threadgroup.org/Portals/0/documents/support/6LoWPANUsage_632_2.pdf.

[27] David Hanes et al. *IoT fundamentals: Networking technologies, protocols, and use cases for the internet of things*. Cisco Press, 2017.

[28] *IEEE International Conference on Fuzzy Systems*. 2021. URL: https://attend.ieee.org/fuzzieee-2021/ (visited on 14/06/2022).

[29] *International Conference on Applications of Neural Networks and Fuzzy Logic in Electrical Engineering ICANNFLEE in June 2023 in Barcelona*. URL: https://waset.org/applications-of-neural-networks-and-fuzzy-logic-in-electrical-engineering-conference-in-june-2023-in-barcelona (visited on 14/06/2022).

[30] *International Conference on Fuzzy Logic Applications in Electrical Engineering ICFLAEE in June 2023 in Barcelona*. URL: https://waset.org/fuzzy-logic-applications-in-electrical-engineering-conference-in-june-2023-in-barcelona (visited on 14/06/2022).

[31] *International Conference on Fuzzy Logic Systems ICFLS in February 2023 in Barcelona*. URL: https://waset.org/fuzzy-logic-systems-conference-in-february-2023-in-barcelona (visited on 14/06/2022).

[32] Richard Kelsey. *Mesh Link Establishment*. Internet-Draft draft-kelsey-intarea-mesh-link-establishment-06. Work in Progress. Internet Engineering Task Force, May 2014. 19 pp. URL: https://datatracker.ietf.org/doc/html/draft-kelsey-intarea-mesh-link-establishment-06.

[33] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[34] Saparya Krishnamoorthy, Michael S Hsiao and Loganathan Lingappan. "Tackling the path explosion problem in symbolic execution-driven test generation for programs". In: *2010 19th IEEE Asian Test Symposium*. IEEE. 2010, pp. 59–64.

[35] Kim G. Larsen, Marius Mikucionis and Brian Nielsen. *UPPAAL TRON User Manual*. 2007.

[36] Phil Lewis et al. "RFC6206: The Trickle Algorithm". In: *Internet Engineering Task Force* (Mar. 2011). ISSN: 2070-1721.

[37] Bart Miller. "COMPUTER SCIENCES DEPARTMENT UNIVERSITY OF WISCONSIN-MADISON". In: 736.CS (1988), 1–3. URL: https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf.

[38] Barton Miller, Mengxiao Zhang and Elisa Heymann. "The relevance of classic Fuzz Testing: Have we solved this one?" In: *IEEE Transactions on Software Engineering* (2020), 1–1. DOI: 10.1109/tse.2020.3047766.

[39] Barton P. Miller, Louis Fredriksen and Bryan So. "An empirical study of the reliability of Unix Utilities". In: *Communications of the ACM* 33.12 (1990), 32–44. DOI: 10.1145/96267.96279. URL: https://dl.acm.org/doi/pdf/10.1145/96267.96279.

[40] Barton P. Miller, Mengxiao Zhang and Elisa Heymann. "The Relevance of Classic Fuzz Testing: Have We Solved This One?" In: *ArXiv* abs/2008.06537 (2020).

[41] Charlie Miller, Zachary NJ Peterson et al. "Analysis of mutation and generation-based fuzzing". In: *Independent Security Evaluators, Tech. Rep* 4 (2007).

[42] Geoff Mulligan. "The 6LoWPAN Architecture". In: *Proceedings of the 4th Workshop on Embedded Networked Sensors*. EmNets '07. New York, NY, USA: Association for Computing Machinery, 2007, 78–82. ISBN: 9781595936943. DOI: 10.1145/1278972.1278992. URL: https://doi-org.zorac.aub.aau.dk/10.1145/1278972.1278992.

[43] *Multithreaded applications · Issue #2 · AFLplusplus/AFL-Snapshot-LKM · GitHub*. https://github.com/AFLplusplus/AFL-Snapshot-LKM/issues/2. (Accessed on 06/15/2022).

[44] *OJ/gobuster: Directory/File, DNS and VHost busting tool written in Go*. https://github.com/OJ/gobuster. (Accessed on 05/30/2022).

[45] *OpenThread Dokumentation*. https://openthread.io/guides. Accessed: 2022-05-27.

[46] *OpenThread: Node Roles and Types*. https://openthread.io/guides/thread-primer/node-roles-and-types. Accessed: 2022-05-21.

[47] *OpenThread: Thread Benefits*. https://www.threadgroup.org/What-is-Thread/Thread-Benefits. Accessed: 2022-05-16.

[48] *openthread/STYLE_GUIDE.md at main · openthread/openthread*. https://github.com/openthread/openthread/blob/main/STYLE_GUIDE.md. (Accessed on 06/16/2022).

[49] Van-Thuan Pham, Marcel Böhme and Abhik Roychoudhury. "AFLNet: A Greybox Fuzzer for Network Protocols". In: *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*. 2020.

[50] S. Priyadarshan. "A study of Binary Instrumentation techniques". In: 2019.

[51] Noel Randewich. *Reuters: Google's Nest launches network technology for connected home*. https://www.reuters.com/article/us-google-nest-idUSKBN0FK0JX20140715. Accessed: 2022-05-16.

[52] Jonathan Salwan. *shell-storm — Shellcodes Database*. 2008. URL: https://shell-storm.org/shellcode/ (visited on 14/06/2022).

[53] *sqlmapproject/sqlmap: Automatic SQL injection and database takeover tool*. https://github.com/sqlmapproject/sqlmap. (Accessed on 05/30/2022).

[54] Nick Stephens et al. "Driller: Augmenting fuzzing through selective symbolic execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[55] *The AFL++ fuzzing framework — AFLplusplus*. https://aflplus.plus/. (Accessed on 05/30/2022).

[56] Petar Tsankov, Mohammad Torabi Dashti and David Basin. "SECFUZZ: Fuzz-testing security protocols". In: *2012 7th International Workshop on Automation of Software Test (AST)*. 2012, pp. 1–7. DOI: 10.1109/IWAST.2012.6228985.

[57] *Wfuzz: The Web fuzzer — Wfuzz 2.1.4 documentation*. https://wfuzz.readthedocs.io/en/latest/. (Accessed on 05/30/2022).

[58] Michal Zalewski. *american fuzzy lop*. https://lcamtuf.coredump.cx/afl/. (Accessed on 06/15/2022).

Acronyms

6LoWPAN IPv6 over Low-Power Wireless Personal Area Networks.

AFL American Fuzzing Lop.

AFL++ American Fuzzing Lop++.

AFLNet American Fuzzing Lop Net.

ALOC Anycast Locator.

CLI Command-Line interface.

DNS Domain Name Service.

ED End Device.

EID Endpoint Identifier.

FED Full End Device.

FFD Full Functioning Device.

FTD Full Thread Device.

GDB GNU Debugger.

IID Interface Identifier.

IoT Internet of Things.

IPv6 Internet Protocol, Version 6.

LLA Link-Local Address.

MAC Media Access Control.

MED Minimal End Device.

MeshCoP Mesh Commissioning Protocol.

MIB Management Information Base.

MIC Message Integrity Code.

MLE Mesh Link Establishment.

MTD Minimal Thread Device.

OT OpenThread.

PAN ID Personal Area Network ID.

REED Router Eligible End Device.

RFD Reduced Function Device.

RLOC Routing Locator.

ROLC Routing Location.

SED Sleepy End Device.

SLAAC Stateless Address Autoconfiguration.

TLV Type-Length-Value.

TNP Thread Network Partition.

UDP User Datagram Protocol.

URL Uniform Resource Locator.

WPAN Wireless Personal Area Network.

XPAN ID Extended Personal Area Network ID.

Glossary

American Fuzzing Lop American Fuzzing Lop, is a free opensource fuzzer that using its generic algorithm has assisted in detecting significant security issues in major free software projects, such as x.Org server, Firefox, PHP OpenSSL and many more..

American Fuzzing Lop Net A branch of AFL which focus on network protocols, by implementing the notion of a server and supports fuzzing over sockets..

American Fuzzing Lop++ The community driven successor of AFL, which is refereed by the author of AFL.

Anycast Locator The Anycast Location is an IPv6 address that specifies the location of one or several Thread nodes in a Thread Network, as it can be assigned to several interfaces. The ALOC is used when the RLOC is unknown at the time of sending.

Domain Name Service A service to look up addresses in a domain name register.

End Device A device in a Thread Network, that usually only talks with a single router. Can turn off its transiver to save power.

Endpoint Identifier A catagory of IPv6 unicast identifiers. Is distinct from RLOC.

Extended Personal Area Network ID 8-byte Extended Personal Area Network ID.

Full End Device A device in a Thread Network not eligible for a promotion to a router.

Full Functioning Device An IEEE 802.15.4 node type with full functionality in a WPAN.

Full Thread Device A device in a Thread Network, can either be a Router, REEDs or FEDs.

GNU Debugger A tool for Debugging C, C++ and other compiled languages..

Interface Identifier Is the last 64 bits of the IPv6.

Link-Local Address An EID that identifies a Thread interface reachable by a single radio transmission.

Management Information Base A database used for managing nodes within a network..

Media Access Control A transfer policy that determines how data should be transmitted between two devices.

Mesh Commissioning Protocol Is the protocol Thread uses for securely authenticating, commissioning, and joining new untrusted nodes to a Thread Network.

Minimal End Device A MED does not subscribe to the all-routers multi-cast events. Forwards all packages to its Parent.

Minimal Thread Device An Thread Device that can turn off its radio to save power. It cannot become a REED.

On-mesh This describes whether the whole path from sender to receiver is within the Thread Network. All packets that are not "on-mesh" are forwarded to the Border Router.

OpenThread The open source implementation of Thread.

Personal Area Network ID 2-byte Personal Area Network ID.

Reduced Function Device An IEEE 802.15.4 node type with reduced functionality in a WPAN.

Router Eligible End Device A device in a Thread Network eligible for promotion to a router.

Routing Location Is a category of IPv6 unicast identifiers. In the Interface Identifiers (IIDs), it is the last part. Is always calculated by appending the Router ID and the Child ID.

Routing Locator The Routing Location is an IPv6 address that specifies a location of a given Thread node within a Thread Network.

Sleepy End Device A device in a Thread Network, which is in a sleep state, that is occasionally polling messages from its parent.

Stateless Address Autoconfiguration Is also called IPv6 Stateless DHCP, it is used for auto configuration of IPv6 interfaces or hosts..

Thread Network Partition The Thread Network consists of one or more Partitions. Each Partition can act as its own Thread Network, with its own Commissioner, Leader, Thread Servers and border routers. A Thread Network aims to only have one Thread Network Partition, but certain situations gives way for even up to a large number of Partitions to be within the same Thread Network.

Uniform Resource Locator A common way of reference web-pages.