UPPSALA UNIVERSITET

UPTEC IT 23027

Examensarbete 30 hp

June 2023

# Using Requirement-Driven Symbolic Execution to Test Implementations of the CoAP and EDHOC Network Protocols

Sabor Amini

Civilingenjörsprogrammet i informationsteknologi

UPPSALA
UNIVERSITET

# Using Requirement-Driven Symbolic Execution to Test Implementations of the CoAP and EDHOC Network Protocols

Sabor Amini

## Abstract

As the number of Internet of Things devices is increasing rapidly, it is of utmost significance that the implementations of protocols for constrained devices are bug-free. In general implementations of network protocols are error-prone due to their complex nature and ambiguities in the protocol specification. Implementations of network protocols often contain critical errors which could be exploited. To avoid bugs and vulnerabilities, the implementation of network protocols has to adhere to their specifications. The objective of this thesis is to use symbolic execution to test one implementation of the Ephemeral Diffie-Hellman Over COSE (EDHOC) protocol and one implementation of the Constrained Application Protocol (CoAP) against their specifications. The goal is to identify bugs such as crashes, non-conformances, memory errors, and security vulnerabilities that may occur if the implementations are not adhering to their specifications. The methodology to do this consists of three steps: 1) extracting requirements from the protocols Request For Comments and expressing them as formulas, 2) preparing the system under test for symbolic execution and applying the formulas during symbolic execution to detect any paths that violate a requirement, 3) for every path which violates a requirement, the concrete value that the symbolic execution engine provided is used in the unmodified implementation to validate the bug. In total seven non-conformances were found which have been reported to developers. One non-conformance was found in the EDHOC implementation and six were found in the CoAP implementation

**Teknisk-naturvetenskapliga fakulteten**

**Uppsala universitet, Utgivningsort Uppsala/Visby**

Handledare: Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson Ämnesgranskare: Konstantinos Sagonas

Examinator: Lars-Åke Nordén

# Sammanfattning

Eftersom antalet Internet of Things enheter ökar snabbt är det av yttersta vikt att implementeringarna av nätverksprotokoll för Internet of Things enheter är korrekta. Generellt sett är implementeringar av nätverksprotokoll felbenägna på grund av deras komplexa natur och oklarheter i protokollspecifikationen. Implementeringar av nätverksprotokoll innehåller ofta kritiska buggar som kan utnyttjas. För att undvika buggar och sårbarheter måste implementeringar av nätverksprotokoll följa sina specifikationer. Målet med detta examensarbete är att använda symbolisk exekvering för att testa en implementation av protokollet Ephemeral Diffie-Hellman Over COSE (Ephemeral Diffie-Hellman Over COSE (EDHOC)) och en implementation av protokollet Constrained Application Protocol (Constrained Application Protocol (CoAP)) mot deras specifikationer. Syftet är att identifiera buggar såsom krascher, icke-konformiteter, minnesfel och säkerhetssårbarheter som kan uppstå om implementeringarna inte följer sina specifikationer. Metodiken för att uppnå detta består av tre steg: *1)* extrahera krav från protokollens specifikationer och uttrycka dem som formler, *2)* förbereda systemet som ska testas för symbolisk exekvering och tillämpa formlerna under symbolisk exekvering för att upptäcka eventuella vägar som bryter mot ett krav, *3)* för varje väg som bryter mot ett krav används det konkreta värde som den symboliska exekveringsmotorn tillhandahåller i den oförändrade implementationen för att validera buggen. Totalt sett hittades sju icke-konformiteter. En icke-konformitet hittades i EDHOC implementeringen och sex hittades i CoAP implementeringen.

# Acknowledgements

First and foremost, I want to express my appreciation to Bengt Jonsson, my supervisor, for granting me the opportunity to undertake this thesis and for providing feedback on the report. I would also like to extend my gratitude to Hooman Asadian and Paul Fiterau Brostean from the Department of Information Technology at Uppsala University for their guidance and for assisting with technical issues throughout this project. Lastly, I am thankful to Konstantinos Sagonas, my reviewer, for providing feedback on the report.

# Table of Contents

# List of Tables

# List of Figures

# List of Acronyms

| | |
|---|---|
| **IoT** | Internet of Things |
| **RFC** | Request For Comments |
| **EDHOC** | Ephemeral Diffie-Hellman Over COSE |
| **CoAP** | Constrained Application Protocol |
| **SMT** | Satisfiability Modulo Theories |
| **CBOR** | Concise Binary Object Representation |
| **PDU** | Protocol Data Unit |
| **UDP** | User Datagram Protocol |
| **SUT** | System under test |
| **ECDH** | Elliptic-curve Diffie–Hellman |
| **HTTP** | Hypertext Transfer Protocol |
| **TCP** | Transmission Control Protocol |
| **EAD** | External Authorization Data |
| **DTLS** | Datagram Transport Layer Security |
| **TLS** | Transport Layer Security |
| **MQTT** | Message Queuing Telemetry Transport |

# 1 Introduction

According to IoT Analytics, the number of connected Internet of Things (IoT) devices is estimated to be 27 billion in 2025 [1]. The Research and Development community has predicted that the impact of IoT will be more than the internet, leading to improvement in the well-being of society and industries [2]. As the number of IoT devices is increasing rapidly, it is of utmost significance that the implementations of network protocols for constrained devices are bug-free.

Implementations of network protocols are error-prone due to their complex nature and ambiguities in the protocol specifications [3], [4]. Implementations of network protocols often contain critical errors which could be exploited, IoT devices are resource constrained and have limited protection against exploitation, therefore it is important that errors are detected before deployment [3].

To prevent bugs and vulnerabilities, it is crucial for the implementation of network protocols to adhere to their specifications. Even minor deviations from the specifications can have serious consequences [4]. A prominent illustration is the Heartbleed vulnerability in OpenSSL, where a memory block was mistakenly returned based on the stated payload size rather than the actual size, resulting in unauthorized access to unrelated data [5]. Numerous other examples highlight how implementation errors have led to security vulnerabilities, including the TLS POODLE downgrade vulnerability [6] and logical vulnerabilities in the Wi-Fi handshake [7]. These incidents underscore the importance of rigorous adherence to protocol specifications to ensure robustness and security.

Additionally, testing network protocols is a difficult task because network protocols are stateful, which means that they can be in different states depending on the sequence of packets received. Consequently, in order to effectively test a specific requirement, first a specific sequence of packets has to be sent enabling it to reach the apporiate state before the requirement can be tested [4].

Security protocols are a key building block in IoT systems. One quite new lightweight security protocol for IoT devices is Ephemeral Diffie-Hellman Over COSE (EDHOC). EDHOC is a key exchange protocol that is used between two endpoints to compute symmetric keys to protect application data. EDHOC is at the time of writing an Internet Draft and its current version is 19. EDHOC can use the Constrained Application Protocol (CoAP) for the transport of the messages in an EDHOC session [8].

CoAP is a web transfer protocol similar to the Hypertext Transfer Protocol (HTTP), but designed for constrained devices and networks. CoAP is based on a client-server architecture developed for machine-to-machine applications. The CoAP protocol is an Internet Standard and is referred to as RFC 7252 [9].

The objective of this thesis is to test one implementation of the EDHOC protocol and one implementation of the CoAP protocol against their specifications. The goal is to identify bugs such as crashes, non-conformances, memory errors, and security vulnerabilities that may occur if the implementations are not adhering to their specifications.

The methodology to do so will be that in the paper by Asadian et al. [4]. This methodology involves several steps. Firstly, requirements are extracted from the protocols Request For Comments (RFC)s. These requirements are then incorporated as assertions and assumptions within the source code. Symbolic execution is then utilized to analyse the code and identify packet sequences and paths that violate these requirements. Any packet sequences or paths that violate a requirement will be validated by using the corresponding packets that caused the violation in the unmodified implementation.

## Research Question

The methodology presented in the paper by Asadian et al. [4] has only been used to test implementations of the Datagram Transport Layer Security (DTLS) protocol. The question this thesis aims to answer is:

- Is the methodology as efficient in testing other network protocols in the IoT domain?

## Outline

This thesis comprises a total of eight chapters following this chapter. Chapter 2 provides a theoretical background on symbolic execution, EDHOC, and CoAP. In Chapter 3, the methodology employed in this study is described. Chapter 4 presents the extracted requirements in the form of formulas. Chapter 5 provides a detailed description of the implementation and experiments. The findings are presented in Chapter 6. In Chapter 7 related works are discussed. In Chapter 8 the results and methodology are discussed. In the final chapter, conclusions and future work are presented.

# 2 Background

This section presents background information on EDHOC, CoAP, and symbolic execution. We begin by illustrating the process of symbolic execution through an example. Next, we delve into the description of the different messages sent in an EDHOC session. Finally, we present the message format of the CoAP protocol.

## 2.1 Symbolic Execution

Symbolic execution is a program analysis technique introduced in the mid-70s. It was the first time presented in the paper by King [10]. In symbolic execution, symbolic values are used instead of concrete values as input to a program. The program is executed with the symbolic inputs and when the symbolic execution engine encounters a conditional branch the execution is forked and the conditions in the conditional statement are stored and referred to as path constraints.
A Satisfiability Modulo Theories (SMT) solver is eventually used to evaluate the path constraints for every explored path. SMT solver is a tool used to determine whether a mathematical formula is satisfiable. The SMT solver will check if the path constraint for each explored path can be satisfied by assigning some concrete values to the program's symbolic input. These concrete values form a test case that follows the exact path [11].
To understand how symbolic execution works, an example is presented. In Figure 2.1, the goal is to find all the inputs that lead to the failure of the assertion.

```
void foobar(int a, int b){
    int x = 1, y = 0;
    if(a != 0){
        y = 3 + x;
        if(b == 0){
            x = 2 * (a + b);
        }
    }
    assert(x - y != 0);
}
```

Figure 2.1: Function foobar [11]

In order to facilitate comprehension of the symbolic execution, a symbolic execution tree is employed. In Figure 2.2 the symbolic execution tree of the code snippet in Figure 2.1 is presented.

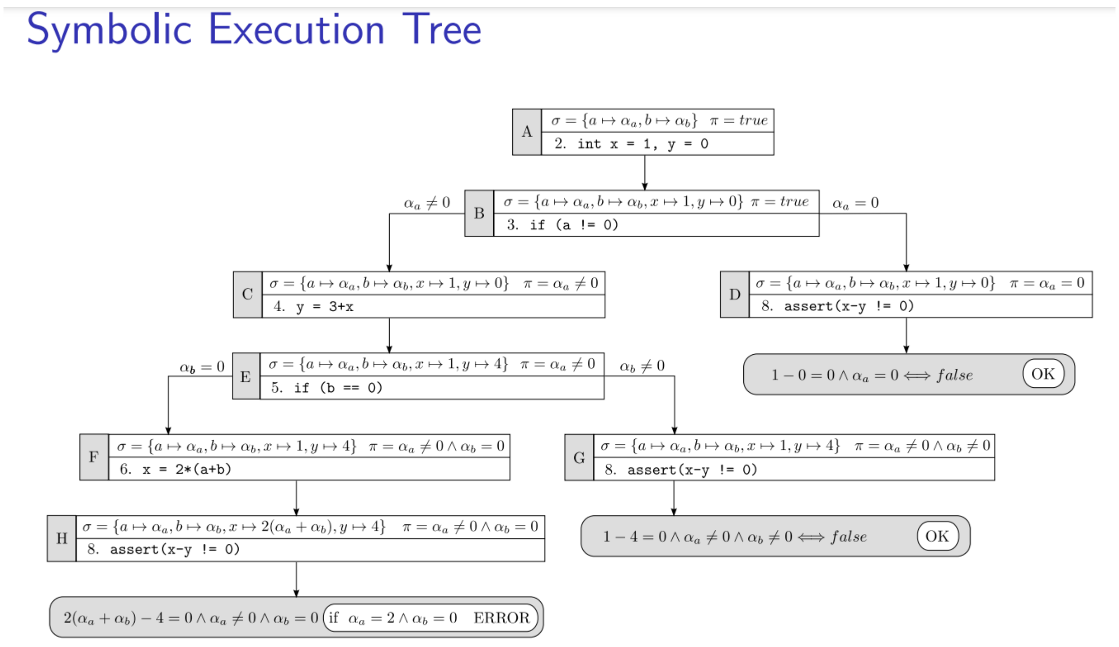
Symbolic Execution Tree
A
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi = true$
2. int x = 1, y = 0
$\alpha_a \neq 0$
B
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = true$
3. if (a != 0)
$\alpha_a = 0$
C
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a \neq 0$
4. y = 3+x
D
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a = 0$
8. assert(x-y != 0)
$1 - 0 = 0 \wedge \alpha_a = 0 \iff false$ OK
$\alpha_b = 0$
E
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0$
5. if (b == 0)
$\alpha_b \neq 0$
F
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
6. x = 2*(a+b)
G
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b \neq 0$
8. assert(x-y != 0)
H
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
8. assert(x-y != 0)
$1 - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b \neq 0 \iff false$ OK
$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$ if $\alpha_a = 2 \wedge \alpha_b = 0$ ERROR

Figure 2.2: Symbolic execution tree of function foobar [11]

The symbolic execution engine maintains a state for a program executing symbolically. The execution state (stmt, $\sigma$, $\pi$) is defined by the three variables stmt, $\sigma$ and $\pi$. Stmt is the next statement to execute, $\sigma$ is the symbolic store that maps program variables to expressions over concrete values or symbolic values and $\pi$ is representing the path constraints. The path constraint is a formula that is based on the conditional branches to reach a specific path. Initially $\pi$ is true.
In Figure 2.2 we can see that in state A, arguments a and b are associated with symbolic values, and the path constraint is initially set to true. In line two in Figure 2.1, the symbolic store is updated by associating the variables x and y to the concrete values 0 and 1 respectively and the program moves to execution state B. In line three which is a conditional branch, the execution is forked and two execution states are created. Depending on the branch, different assumptions are made on the symbolic variable $\alpha$. Arithmetic expressions in the program manipulate the symbolic values. When every execution state has reached the assert statement in line eight, it is possible to see that only state H can make the assertion fail. The path constraint in state H is defining the set of inputs that can lead the assertion to fail. A SMT solver can be used to solve the path constraint and find concrete values for the symbolic arguments a and b. In this example, the values a=2 and b=0 would make the assertion fail.

## 2.2 EDHOC

EDHOC is a compact, lightweight, and authenticated key exchange protocol which is intended for usage in constrained devices. In EDHOC there are three mandatory messages and an optional fourth message that are sent between an initiator and a responder to derive a shared secret session key. The shared secret session key is then used to derive application keys to protect application data. EDHOC incorporates an error message mechanism to handle potential errors that may arise during an EDHOC session. All the messages in EDHOC are deterministically encoded in Concise Binary Object Representation (CBOR) sequences [8].
In Figure 2.3, an EDHOC session which includes the optional fourth message is presented.

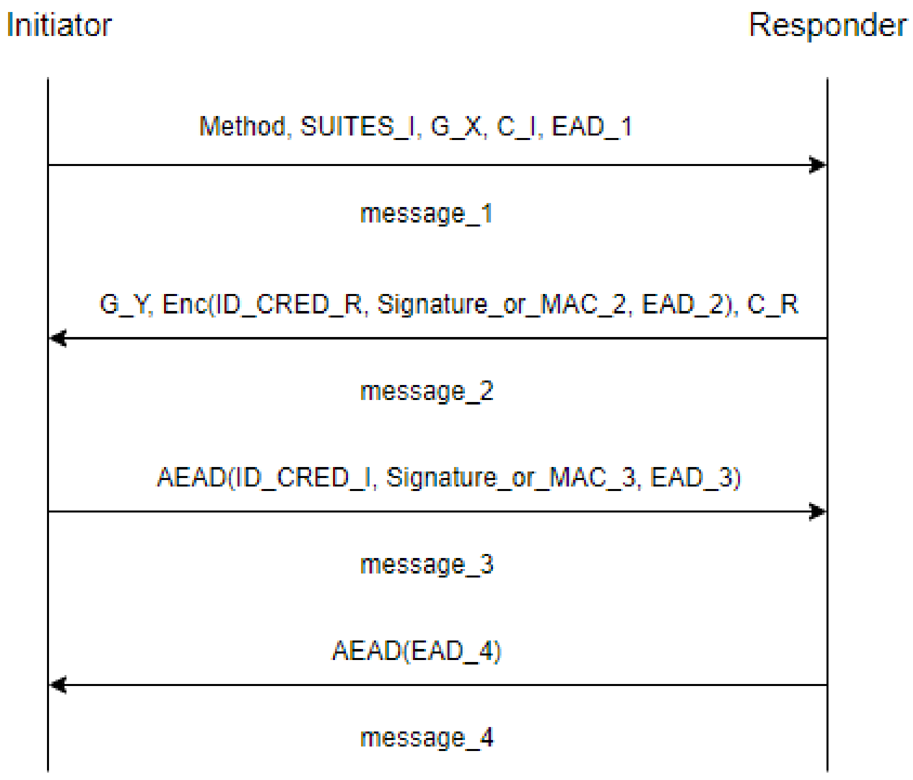
Initiator
Responder
Method, SUITES_I, G_X, C_I, EAD_1
message_1
G_Y, Enc(ID_CRED_R, Signature_or_MAC_2, EAD_2), C_R
message_2
AEAD(ID_CRED_I, Signature_or_MAC_3, EAD_3)
message_3
AEAD(EAD_4)
message_4

Figure 2.3: EDHOC handshake

### EDHOC Message_1

In the first message which is sent by the initiator, the Method field is an integer that specifies the authentication method. The responder and initiator can choose to use either a signature key or a static Diffie-Hellman key to authenticate themselves, so the Method field can have four different values such as:

- 0: Both initiator and responder authenticate with signature keys.
- 1: Initiator authenticates with a signature key and responder authenticates with a Static DH key.
- 2: Initiator authenticates with a static DH key and responder authenticates with a signature key.
- 3: Both initiator and responder authenticate with static DH keys.

The SUITES_I field is a list of cipher suites supported by the initiator. The cipher suites in the list are encoded as integers and this field is used in the cipher suite negotiation. The cipher suites supported by the initiator are listed in the order of preference, where the first cipher suite in the list is the most preferred one and the last suite in the list is the selected one. All cipher suites that are more preferred by the initiator over the selected one must be included in the list. When SUITES_I contains only one cipher suite, it is encoded as an integer and not a list.
G_X is the ephemeral public key of the initiator, which is used to derive the Elliptic-curve Diffie–Hellman (ECDH) shared secret.
The connection identifier C_I is a byte string used for distinguishing between different connections. It can also be used to correlate messages and keep track of the protocol state during an EDHOC session. The connection identifiers C_I and C_R are chosen by the responder and initiator respectively.
EAD_1, which stands for external authorization data can be used by external security applications to transport external authorization data to reduce round trips and numbers of messages sent [8]. The EAD may comprise multiple ead items. An ead item is represented as a CBOR sequence consisting of an ead_label and an optional ead_value. An ead item is formatted as in Figure 2.4 where ? indicates that the field is optional.

```
ead = (
ead_label : int
? ead_value: bstr ,
)
```

Figure 2.4: EAD item

An ead item can be critical or non-critical. A critical ead item has a negative ead_label. If a critical ead item is not recognized or cannot be processed by the receiver then the receiver must discontinue the protocol and send an error message. A non-critical item has a non-negative ead_label and can be ignored when it cannot be processed. The security application which registers a new ead item has to specify how the ead item should be processed and under what conditions the ead item is critical or non-critical.

#### EDHOC Message_2

In message 2, the ID_CRED_R field contains information that is necessary to obtain the authentication credentials of the responder. The authentication credentials of the responder CRED_R contain the public authentication key of the responder. In EDHOC, the authentication credentials CRED_R and CRED_I of the responder and initiator respectively serve two purposes. The authentication key within the credentials is utilized in conjunction with the Signature_or_Mac field to verify the proof of possession of the private key. Additionally, it is employed as input for integrity verification through the MAC fields.

#### EDHOC Message_3

The third message sent by the initiator contains the EAD_3 and Signature_or_MAC_3 that has to be verified by the initiator using either a signature key or a static DH key. If a signature key is used as the authentication method, then the Signature_or_MAC_3 field contains a signature generated by the private key of the authenticating party. The signature is computed over the concatenated values of the preceding messages. The recipient can verify the signature using the public key associated with the authenticating party to ensure the integrity and authenticity of the message.

#### EDHOC Message_4

The optional fourth message is used to give key confirmation to the initiator in cases where no protected application data is sent from the responder to the initiator.

### 2.2.1 Error Handling & Error Messages

In EDHOC an error message can be sent by the initiator or responder in case an error occurs. An error could for instance occur during the processing of a message. An error message is fatal, which means that the sender must discontinue the session after sending an error message. An error message is a CBOR sequence formatted in the following way:

```
error = (
ERR_CODE : int,
ERR_INFO : any,
)
```

Figure 2.5: Error message

In Figure 2.5 ERR_CODE is an integer and ERR_INFO is information about the error. The currently defined error codes are:

- 0: Indicates Success, this error code can be used internally in an application.
- 1: Indicates an unspecified error, this code is used for errors that do not have an error code defined.
- 2: Indicates wrong selected cipher suite, this code can only be used when replying to message_1 in case the selected cipher suite by the initiator is not supported by the responder or if the responder supports a cipher suite more preferred by the initiator than the selected cipher suite.

## 2.3 CoAP

CoAP is a web transfer protocol for constrained devices and networks. It is mainly designed for machine-to-machine applications. Similarly to HTTP, CoAP is based on a client-server architecture, where the client sends a request and the server replies with a response. Unlike HTTP, CoAP messages are transported over User Datagram Protocol (UDP) and not the Transmission Control Protocol (TCP), moreover the messages are encoded in a simple binary format [9].

### 2.3.1 CoAP Protocol Data Unit

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+-------+---------------+-------------------------------+
|Ver| T |  TKL  |      Code     |          Message ID           |
+---+---+-------+---------------+-------------------------------+
| Token (if any, TKL bytes) ...                                 |
+---------------------------------------------------------------+
| Options (if any) ...                                          |
+---------------+-----------------------------------------------+
| 11111111      |  Payload (if any) ...                         |
+---------------+-----------------------------------------------+
```

Figure 2.6: CoAP message format

The CoAP Protocol Data Unit (PDU) consists of a fixed 4-byte header, followed by the token, options, and payload [9].

#### Header

The first two bits in the header are the version, currently CoAP only supports one version, so this field must be set to 1. The subsequent two bits represent the message type, the message types are used in different scenarios and for different purposes. A message in CoAP can be of four types:

- Confirmable: A message of type Confirmable requires an Acknowledgment from the receiver. This is a reliable way of sending a message.
- Non-confirmable: A message of type Non-confirmable requires no Acknowledgement from the receiver thus this is an unreliable way of sending a message.
- Acknowledgment: Used to indicate to the sender that a specific message was received.
- Reset: Sent as a response when a specific message has been received but some context is missing to properly process it.

The remaining four bits in the first byte are the Token Length field which indicates the length of the token. The second byte in the header is the Code field. It can be set to a method code or a response code. The code field is an 8-bit unsigned integer that is split into two parts. The three most significant bits are called the class and the five least significant bits are the detail. The class can have four different values:

- 0: indicates a request.
- 2: indicates success response.
- 4: indicates client error response.
- 5: indicates server error response.

The remaining class values are reserved. The detail is providing additional information to the class. For instance, the response code 2.01 means "Created" successfully.
The last field in the header is the Message_ID which is a 16-bit unsigned integer. The message_id is used to match messages of types confirmable and non-confirmable to messages of types reset and acknowledgment. Following the header, the subsequent field is the Token which is used to match a request to a response. After the token is the Options that can be used in a request or response [9]. For instance, the following options can be used to specify a target resource for a request to a server:

- Uri-Host
- Uri-port
- Uri-path
- Uri-Query

The last field in the PDU is the optional payload. If the payload is of non-zero length then it is prefixed by a fixed, one-byte Payload Marker (0xFF) which indicates the end of options and the start of the payload. The payload data extends from after the marker to the end of the UDP datagram.

# 3 Methodology

The methodology is divided into three separate parts, *1)* requirements are extracted from the protocols RFCs and translated into constraint formulas, *2)* in the next step, the system under test is being prepared for symbolic execution and the formulas are incorporated as assumptions and assertions in the source code. The system under test is then executed symbolically to identify potential bugs, *3)* in the final step, test cases for the bugs are created and validated in the unmodified implementation.
The primary focus of this thesis is on testing the server side, where symbolic inputs are exclusively sent to the server. The symbolic execution engine that is used in this project is called KLEE, KLEE was developed by a team of researchers at Stanford University. It is described in the paper by Cadar et al. [12]. KLEE is open-source and is currently maintained by the Software Reliability Group in London.
The EDHOC implementation to be tested is uOSCORE-uEDHOC, version v2.1.3. The CoAP implementation to be tested is cantcoap, commit cce97e5. This particular implementation is utilized as an external library within the EDHOC implementation. The last commit in the CoAP implementation was in December 2021.

## 3.1 Extracting Requirements

The first step is to extract requirements from the protocols RFCs. Two types of requirements are of interest, input validity requirements and input-output requirements. Input validity requirements refer to requirements that check the validity of the inputs, whereas input-output requirements refer to requirements that check the validity of the outputs in response to a valid input. During the process of extracting requirements from the RFCs keywords such as "MUST", "SHOULD", "MUST NOT" and "SHALL NOT" is used. These keywords are used in most of the RFCs and indicate how important a certain requirement is [4]. The keyword "MUST" indicates that the requirement is very important to follow in the protocol whereas the keyword "SHOULD" indicates that it is not an absolute necessity to follow the requirement. For instance a requirement from the CoAP RFC can be regarding the version number indicating the version of CoAP. In the RFC ([8], p. 16) it is stated:

> *Implementations of this specification MUST set this field to 1 (01 binary). Other values are reserved for future versions. Messages with unknown version numbers MUST be silently ignored.*

For a set of messages M during an EDHOC session, it is possible to represent this requirement as a formula. The formula for this requirement can be expressed as:

$$\forall m \in M : m.version = 1$$

## 3.2 Preparing the System Under Test & Inserting Instrumentation

To prepare the system under test for symbolic execution, it is necessary to do modifications to the implementation. The implementation has to be de-randomized so it responds in the same way when running symbolically as during the time messages are recorded. Encryption and decryption of the messages and function call to external libraries has to be replaced, this is done to avoid complicated code used for encryption and decryption when running the code symbolically. Cryptographic primitives such as encryption and decryption generate complex symbolic expressions that are difficult for the SMT solver to handle [3].
When the System under test (SUT) is executing deterministically and encryption and decryption are disabled, the messages of an EDHOC session are recorded and stored in separate files to be used during the symbolic execution of the program.
The next step is to insert instrumentation for each requirement into the source code. The overall strategy consists of two steps. In the first step, for a given requirement, input constraints for which a requirement can be violated are inserted in the source code as assumptions. In the second step, it is checked if the program would perform an action that violates a requirement by using an assert statement. The assert statement is used at the place where the program is about to finish the session, this is a way of indicating that the system under test is completing the session with an invalid packet.
In the case of an input validity requirement, the negation of the formula is used in a assume statement, and an assert statement is used to check that invalid packets are not handled in a forbidden way.
In the case of an input-out requirement, an assume statement is used first to ensure the validity of the input, and the formula is used in an assert statement to check the correctness of the output.
When instrumentation in the source code is done the system under test is executed symbolically to find paths that satisfy the assumptions and trigger an assertion violation, crash, etc.

## 3.3 Validating the Non-Conformances

The last step is to construct test cases and use them to validate the bugs in the unmodified implementation. When the system under test is executed symbolically the symbolic execution engine will for every explored path generate corresponding values for the symbolic inputs. For every path that triggers an assertion or crash, the concrete value for the symbolic input is then used in the original implementation.
If the program crashes, it is considered a bug, while the triggering of an assertion is interpreted as a non-conformance.

## 3.4 Example

An example of the process is described below. In this example, we are testing an input-output requirement regarding the message_id. In the RFC ([8], p. 16) of CoAP it is specified that:

*The Message ID MUST be echoed in the Acknowledgment or Reset message by the recipient.*

This means that the recipient must use the same message_id without modifying it when replying to a confirmable message with an acknowledgment or reset message.
Considering a set of messages M used in an EDHOC session, then it is possible to formulate constraints over the fields in the messages. In this case, the field is message_id and if resp(m,i) represent the i-th output generated in response to input message m, this requirement can be formulated as:

$$\forall m \in M : resp(m,1).\text{type} \in \{\text{Reset, Acknowledgment}\} \Rightarrow \\ resp(m,1).\text{message\_id} = m.\text{message\_id}$$

In this formula, if the type of the response message is reset or acknowledgment then the response must have the same message_id as the received message.
Instrumentations are inserted into the source code based on the logical formula. First, the specific fields relevant to the requirement are made symbolic, in this case, the message_id. In the case of an input-out requirement, the input must be valid therefore an assume statement is used to ensure the validity of the input, and an assertion is used on the output to check if it violates the requirement. In the case of the message_id, an assert statement will be as follows:

```
klee_assert(txPDU->getMessageID() == symbolic_message_id);
```

In this example, if the message_id of the acknowledgment or reset message is not the same as the message_id of the request then this assertion will fail, indicating there exists at least one message_id value for which the response doesn't have the same message_id as the request.
If the assertion is triggered, then the symbolic execution engine will provide a concrete value for the symbolic message_id. The concrete value will then be used in the unmodified implementation to check whether the system behaves in the same way. For instance, if the symbolic execution engine provides the value zero as the message_id, then in the unmodified implementation the message_id in the appropriate input message will be set to zero to check if the output will have a different value.

# 4 Requirements

The requirements that were found from the EDHOC and CoAP RFCs are presented in this chapter.

## 4.1 EDHOC

From the EDHOC RFC two input validity requirements were extracted.

Considering the four messages message_1, message_2, message_3, and the optional message_4 in an EDHOC session then it is possible to formulate constraints over the fields in the messages.

*1.) Method:* The method field in the first message is an integer that determines what kind of keys are used for authentication. In the RFC ([8], p. 14) it is stated:

> *The authentication key (i.e., the public key used for authentication) MUST be a signature key or static Diffie-Hellman key.*

The formula for this requirement can be expressed as:

$$message\_1.method \in \{0,1,2,3\}$$

*2.) Supported cipher suite:* In the RFC ([8], p31) it is described how the responder should process message one:

> *Process message_1, in particular, verify that the selected cipher suite is supported and that no prior cipher suite in SUITES_I is supported.*

It is also stated in the RFC:

> *If any processing step fails, then the Responder MUST send an EDHOC error message back as defined in Section 6, and the protocol MUST be discontinued.*

This means if the selected cipher suite by the initiator is not supported by the responder, then the responder must send an error message and discontinue the protocol.
The field SUITES_I in message_1 contains the supported and selected cipher suite by the initiator. SUITES_R contains the cipher suites supported by the responder. The formula for this requirement can be written as:

$$message\_1.SUITES\_I.selected \in SUITES\_R$$

## 4.2 CoAP

From the CoAP RFC eleven requirements were extracted, nine input validity requirements, and two input-output requirements.
To express the requirements as formulas we consider that the function num_bytes() returns the number of bytes of its argument, resp(m) represents the output generated in response to the input message m, and resp(m,i) represents the i-th output generated in response to input message m. Considering a set of messages M used in a CoAP session and using and m and $m'$ to iterate over individual messages it is possible to formulate constraints over the fields in the messages.

*1.) Version:* The RFC specifies one version for CoAP, in the RFC ([8], p. 16) it is specified:

*Version (Ver): 2-bit unsigned integer. Indicates the CoAP version number. Implementations of this specification MUST set this field to 1 (01 binary). Other values are reserved for future versions. Messages with unknown version numbers MUST be silently ignored.*

The formula for this requirement can be formulated as follow:

$$\forall m \in M : m.version = 1$$

*2.) Type:* CoAP supports four type of messages, in the RFC ([8], p. 16) it is specified:

*Type (T): 2-bit unsigned integer. Indicates if this message is of type Confirmable (0), Non-confirmable (1), Acknowledgement (2), or Reset (3).*

It is also described how the different types are used. Regarding the reset message it is specified RFC ([9], p. 8):

*A Reset message indicates that a specific message (Confirmable or Non-confirmable) was received, but some context is missing to properly process it.*

It is also specified ([9], p. 21):

*The Reset message MUST echo the Message ID of the Confirmable message and MUST be Empty.*

Regarding the messages of type non-confirmable it is stated in the RFC ([9], p. 23) :

*A Non-confirmable message always carries either a request or response and MUST NOT be Empty.*

Regarding messages of type acknowledgment it is stated in the RFC ([9], p. 21) :

*The Acknowledgement message MUST echo the Message ID of the Confirmable message and MUST carry a response or be Empty (see Sections 5.2.1 and 5.2.2).*

To summarize, different message types are used for different purposes. Messages of type confirmable can contain a request, response or be empty. Messages of type non-confirmable can contain a request or a response. Messages of type acknowledgment can contain a response or be empty. Messages of type reset must always be empty. This requirement can be formulated using the code field as follow:

$$\begin{aligned}\forall m \in M : &(m.type = Confirmable \Rightarrow m.code \in \{0.00-0.31,\ 2.00-5.31\}) \quad \wedge \\ &(m.type = Non\_confirmable \Rightarrow m.code \in \{0.01-0.31,\ 2.00-5.31\}) \quad \wedge \\ &(m.type = Acknowledgment \Rightarrow m.code \in \{0.00,\ 2.00-5.31\}) \quad \wedge \\ &(m.type = Reset \Rightarrow m.code = 0.00)\end{aligned}$$

*3.) Message ID:* An input-out requirement is regarding the message_id, in the RFC ([8], p. 24) it is specified:

> *The Message ID MUST be echoed in the Acknowledgement or Reset message by the recipient.*

It means that the message_id in the acknowledgment or reset message must be the same as the message_id in the request without any modifications. This requirement can be captured with the help of the message type in the following way:

$$\begin{aligned}\forall m \in M : &resp(m,1).\text{type} \in \{\text{Reset, Acknowledgment}\} \Rightarrow \\ &resp(m,1).\text{message\_id} = m.\text{message\_id}\end{aligned}$$

*4. Message ID unique:* Another requirement regarding the message_id is its uniqueness. In the RFC ([8], p. 24) it is specified:

> *The same Message ID MUST NOT be reused (in communicating with the same endpoint) within the EXCHANGE_LIFETIME (Section 4.8.2).*

This means that during a session every request must have a unique id and using the code field the formula for this requirement can be expressed as:

$$\begin{aligned}\forall m, m' \in M : &(m.code = request) \wedge (m'.code = request) \wedge (m \neq m') \Rightarrow \\ &m_i.message\_id \neq m'.message\_id\end{aligned}$$

This formula indicates that two requests cannot have the same message_id during the exchange lifetime.

*5. Token:* It is specified in the RFC ([8], p. 34) that:

> *Every request carries a client-generated token that the server MUST echo (without modification) in any resulting response.*

This means that the token value in the response should be the same as the token value in the request. This requirement can be formulated in the following way:

$$\forall m \in M : resp(m).token = m.token$$

This formula indicates that the response to message m which has a payload must have the same token value.

*6. Token unique:* Another requirement regarding the token is its uniqueness.
In the RFC ([8], p. 35) it is specified that:

> *The client SHOULD generate tokens in such a way that tokens currently in use for a given source/destination endpoint pair are unique.*

This means that the token values should be different for every request sent to the same endpoint during an CoAP session. This requirement can be captured with the following formula:

$$\forall m, m' \in M : (m.code = request) \wedge (m'.code = request) \wedge (m \neq m') \Rightarrow \\ m.token \neq m'.token$$

*7. Token length:* In the RFC ([8], p. 16) it is specified that:

> *Token Length (TKL): 4-bit unsigned integer. Indicates the length of the variable-length Token field (0-8 bytes). Lengths 9-15 are reserved, MUST NOT be sent and MUST be processed as a message format error.*

Regarding the token length field, two requirements were extracted, the first requirement is about valid values for the token length field:

$$\forall m \in M : m.token\_length \in \{0,1,2,3,4,5,6,7,8\}$$

*8. Token length correlation:*
The second requirement checks if the token length field represents the actual token length:

$$\forall m \in M : m.token\_length = num\_bytes(m.token)$$

*9. Payload marker:* Another requirement is regarding the payload marker. In the RFC [8], p. 17 it is stated:

> *The absence of the Payload Marker denotes a zero-length payload. The presence of a marker followed by a zero-length payload MUST be processed as a message format error.*

This means that if there is a payload marker then there should be a payload and vice versa, otherwise, the message must be processed as a message format error. The value of the payload marker must be 255.

$$\begin{aligned}&\forall m \in M :\\ &\quad (m.payload\_marker = 255 \Rightarrow num\_bytes(m.payload) \neq 0) \quad \land\\ &\quad (m.payload\_marker \neq 255 \Rightarrow num\_bytes(m.payload) = 0)\end{aligned}$$

*10. Empty message:* In the RFC ([9], p. 21) it is specified that:

> *An Empty message has the Code field set to 0.00. The Token Length field MUST be set to 0 and bytes of data MUST NOT be present after the Message ID field. If there are any bytes, they MUST be processed as a message format error.*

This means that an empty message must only contain the 4-byte header. This requirement can be formulated as follow:

$$\forall m \in M : m.code = empty \Rightarrow num\_bytes(m) = 4$$

*11. Code:* In the RFC ([9], p. 16) it is stated:

> *A recipient MUST either (a) acknowledge a Confirmable message with an Acknowledgement message or (b) reject the message if the recipient lacks context to process the message properly, including situations where the message is Empty, uses a code with a reserved class (1, 6, or 7), or has a message format error. Rejecting a Confirmable message is effected by sending a matching Reset message and otherwise ignoring it.*

The code is as mentioned earlier split into two parts, the three most significant bits are called class for which only values 0, 2, 4, and 5 are valid, and all other values are reserved. In case a message is received with a reserved class value, the recipient must ignore it and send a matching reset message. The formula for this requirement can be expressed as follow:

$$\forall m \in M : m.code\_class \in \{0,2,4,5\}$$

# 5 Implementation & Experimentation

In this chapter, the implementation is described, divided into three parts. It includes **Preparing the System Under Test for Symbolic Execution** , **Instrumentation & Symbolic Execution**, and **Test Case Construction and Validation**.

## 5.1 Preparing the System Under Test for Symbolic Execution

As mentioned earlier the SUT had to be de-randomized to execute deterministically so it responds in the same way when capturing the messages as during executing the system under test symbolically. To de-randomize the SUT for symbolic execution the function **sign** which is responsible for the signature had to be modified. It was modified to return the same signature every time.
To avoid cryptographic operations encryption and decryption of the second and third messages had to be disabled. To disable encryption and decryption of the second and third message the functions **xor_arrays** and **aead** had to be modified, **xor_arrays** was encrypting and decrypting message_2 and **aead** was encrypting and decrypting message_3. The functions were modified to do nothing except for copying the content of the plaintext buffer into the ciphertext buffer and vice versa.
The functions **shared_secret_derive**, **hash**, **hkdf_extract**, **verify** and **cert_x509_verify** belonged to external libraries and were related to cryptographic and authentication operations. They had to be modified similarly.
When the SUT was executing deterministically and cryptographic operations and function calls to external libraries were modified it was time to run a session of the EDHOC handshake and record the messages in separate files. Functions were developed that loaded the messages from the files and stored them in the correct EDHOC structure to be further processed by the system under test. For instance, for the first CoAP message this function was created:

```
void read_first_coap_message_from_file(char *buffer);
```

## 5.2 Instrumentation & Symbolic Execution

To run the system under test symbolically and test the requirements a specific function was developed for each requirement. Every function made a specific field symbolic depending on the requirement being tested. The functions were also responsible for

assuming the negation of the input validity constraints in the case of an input validity requirement and ensuring the validity of the input in the case of an input-output requirement.
For instance to test the requirement regarding the code field. The first step was to make the code field, or more specifically the three most significant bits of the code field which represent the class, symbolic. A function has been developed, which takes as input a reference to the original code field and returns a symbolic code field. The signature of the function is:

```
uint8_t klee_make_code_class_symbolic(uint8_t *code);
```

To create a symbolic variable with the same size and type as the code field the following function was used:

```
klee_make_symbolic(&symbolic_code, sizeof(symbolic_code),
"symbolic_code");
```

In the case of the code field, only the three most significant bits have to be made symbolic, to only make these bits symbolic and not the whole code field symbolic it is necessary to perform bit masking in the following way:

```
klee_assume((symbolic_code & 0b00011111) == (*code & 0b00011111));
```

The result of the function above is a byte where only the three most significant bits are symbolic and the rest of the bits keep their original values.
To assume negotiation of the input validity constraint formulated for the code field this function is used:

```
klee_assume((((symbolic_code & 0b11100000) >> 5) != 0) &
(((symbolic_code & 0b11100000) >> 5) != 2) &
(((symbolic_code & 0b11100000) >> 5) != 4) &
(((symbolic_code & 0b11100000) >> 5) != 5));
```

The result of this function is a code field that has a symbolic class that cannot have a valid value for the three most significant bits, which in this case are 0, 2, 4, and 5.
To check if the implementation will proceed with an invalid argument an assertion is being used at the end of the main function. If this assertion is triggered, it indicates that the implementation has finished the EDHOC session with invalid input.

When dealing with an input-output requirement, the implemented function initially verifies the validity of the input before assessing the validity of the output. As an example, to evaluate requirement three concerning the message_id, where the message_id of the acknowledgment or reset message should match the request, a function with the following signature was developed:

```
uint16_t klee_make_message_id_symbolic(uint16_t message_id);
```

This function receives the message_id as input and checks the validity of the input using an assume statement in the following way :

```
klee_assume((0<=message_id) & (message_id <= 65536));
```

Secondly, the message_id is made symbolic using this function:

```
klee_make_symbolic(&symbolic_message_id,
sizeof(symbolic_message_id), "symbolic_message_id");
```

Finally, an assertion is used as shown below:

```
assert(txPDU->getMessageID() == symbolic_message_id);
```

This assertion is being used when the output (txPDU) has been created. The assertion checks if the output can have a message_id that differs from the input message_id. If this assertion is triggered, it indicates a violation of requirement three concerning the message_id

## 5.3 Test Case Construction and Validation

The final part of the implementation is to validate possible bugs in the unmodified implementation. To do this, test cases have to be created for every input that triggers some assertion violation. When the program is executed symbolically the symbolic execution engine will for every explored path generate corresponding values for the symbolic variables. For every path that triggered the assertion, the concrete value provided by the symbolic execution engine was used in the unmodified implementation to validate if the same behavior could be observed. The message in the original implementation, which includes the specific field, is modified by assigning it the concrete value provided by the symbolic execution engine. The implementation is then executed with this value and it is observed if the original implementation violates a requirement.

# 6 Results

This chapter presents the non-conformances that were found in the implementations. First a description of what non-conformances were found in the EDHOC implementation followed by the non-conformances found in the CoAP implementation.

## 6.1 Non-Conformances in EDHOC

Table 6.1: Results testing EDHOC

| Requirement | Bug | Paths | Time (s) | Status |
|---|---|---|---|---|
| 1. Method | X | 1 | 3 | Fixed |
| 2. Cipher suite | | 3 | 1 | |

Table 6.1 provides the results of testing the EDHOC implementation, where each row corresponds to a specific requirement being examined and the X symbol indicates the presence of a non-conformance. Additionally, the table includes information regarding the number of executed paths, the corresponding time duration, and the current status of each bug, indicating whether it has been reported, confirmed, or fixed by the developers. The table in the next section is structured in the same way.

In EDHOC one non-conformance was found concerning the method field. In the EDHOC implementation, it was possible to set the method field to an invalid value, for instance, 8 and the handshake was still finishing successfully. The non-conformance issue regarding the method field in the EDHOC implementation stemmed from the absence of adequate checks. When the responder in EDHOC received message_1, the method field was utilized in a switch statement to determine the authentication process for both the responder and initiator. However, in the event of an invalid method field, the program defaulted to the switch's default case, which merely exited the switch function. As a result, the program variables static_dh_i and static_dh_r were not updated according to the method field. These variables are used to indicate how the responder and initiator will authenticate themselves.
In the program, the variable static_dh_i is initialized to zero and the variable static_dh_r is only declared. The value assigned to static_dh_r during program execution varied.

In some cases, the compiler assigned it a value of zero, and the program proceeded as expected since the sample program was configured to use signature keys for authentication. However, when the compiler assigned a value of one to static_dh_r, the server attempted to authenticate itself using a static-DH key. This caused the program to terminate with an error code, because the sample program was designed to utilize signature keys for authentication.

## 6.2 Non-Conformances in CoAP

Table 6.2: Results testing CoAP

| Requirement | Bug | Paths | Time (s) | Status |
|---|---|---|---|---|
| 1. Version | | 1 | 1 | |
| 2. Type | X | 1 | 17 | Reported |
| 3. Message ID (echo) | | 1 | 5 | |
| 4. Message ID (uniqueness) | X | 1 | 6 | Reported |
| 5. Code | X | 2 | 6 | Reported |
| 6. Token (echo) | | 1 | 5 | |
| 7. Token (uniqueness) | X | 1 | 6 | Reported |
| 8. Token length | | 2 | 0 | |
| 9. Token length (correlation) | X | 3 | 14434 | Reported |
| 10. Payload marker | | 38 | 1203 | |
| 11. Empty message | X | 2 | 6 | Reported |

Table 6.2 provides the results of testing the CoAP implementation. In the context of CoAP, eleven requirements were extracted. Two input-output requirements (3 and 6) and eight input validity requirements. Six instances of non-conformance were identified:

- 4. Message ID (uniqueness)- The same message_id could be used for multiple requests.
- 7. Token (uniqueness)- A similar instance of non-conformance was also identified with the token field. It was possible to use the same token value for multiple requests.
- 5. Code- Reserved code classes such as one and three could be used.
- 2. Type- The message type field could be used incorrectly. For instance, a message of type reset could contain a request.
- 11. Empty message- An empty message with code 0.00 could contain data.
- 9. Token length field (correlation)- The token length field did not accurately represent the length of the token. For example, an actual token length of four bytes could be represented with a token length field value of two.

The non-conformances related to the message_id (4) and token (7) fields occurred due to the sender's inability to adequately track and manage these identifiers. As a result, the same message_id and token could be reused for multiple requests, leading to potential issues. This flaw can result in incorrect responses being sent by the CoAP implementation in response to received requests. Moreover, if the token values are used in the EDHOC implementation to correlate EDHOC messages, it can lead to incorrect correlation of these messages.

The non-conformance related to the token length field (9), where it does not accurately represent the actual token length, is a significant issue. In this scenario, the client sends a request with a token to match it with the corresponding response. The server, upon receiving the message, echoes the token from the request. However, it is possible to manipulate the token length field by setting it to the maximum value of eight while setting the subsequent options and payload fields to zero. As a result, the server will respond with an eight-byte token, even if the actual token is only one byte. This means that seven unrelated bytes can be retrieved.
In the implementation of EDHOC, where CoAP is used as an external library, exploiting this non-conformance would cause the implementation to terminate. The program terminates because of the tag length of the cipher suite. The tag length is eight bytes and the payload has to be bigger which means that the payload has to be at least nine bytes, and therefore it is not possible to retain something else except for the EDHOC message. However, it is important to note that this non-conformance can potentially be exploited in other settings.

The non-conformance issue in the CoAP implementation related to the code field (5) occurred due to a lack of proper checks. It was possible to use invalid code values that were reserved. Although a check was in place to ensure that the code class falls within the acceptable range of zero to five, it failed to prevent the use of reserved classes one and three.

The non-conformance concerning empty messages (11) emerged because of a lack of verification that a message with code 0.00 should strictly consist of four bytes. The client had the ability to send a message with code 0.00, indicating an empty message, but the actual message could still contain data. The implementation only checked the validity of the code field, neglecting to verify its correlation with the rest of the protocol data unit.

# 7 Literature Review

There exist several techniques to test network protocols. One technique is fuzzing, there are different variations of fuzz testing. For instance, Fiterau-Brostean et al. [13] have used protocol state fuzzing to test different implementations of the DTLS protocol. Somorovsky [14] used a two-stage fuzzing approach to evaluate Transport Layer Security (TLS) server behavior.
Sagonas and Typaldos [15] created a protocol state fuzzer, EDHOC-Fuzzer, to test implementations of the EDHOC protocol. This tool uses model learning to create a state machine model of the implementation. The model can be used to find non-conformances, bugs, and security vulnerabilities. They used this tool on three EDHOC implementations, uOSCORE-uEDHOC, RISE, and EDHOC-RS. The uOSCORE-uEDHOC (commit fbaa96c) implementation is the same as in this project. In this particular implementation, it was identified that the client is not waiting for a response to message_3 but terminates immediately after sending message_3, which is non-conforming to the EDHOC protocol specification.
Fuzzing in general is a testing technique that involves sending random or invalid data to the protocol implementation to test its ability to handle unexpected inputs. This can help identify bugs and vulnerabilities that may not be apparent during normal testing. The process of fuzzing starts with choosing some inputs. These inputs are then repeatedly mutated taking into consideration the observations of the program, the mutated input is evaluated by sending it to the program, and if any "interesting" observation is observed the mutated input and the observation are stored. An "interesting" observation is defined differently for different fuzzers. In a black-box fuzzer, the only observation which is interesting is input that leads a program to a crash. In grey-box testing, observations can also be information about the execution such as which branches were executed, etc. The output from a fuzzer is some concrete input and configuration information such as the inputs to start with and the duration of execution. This information can be used to reproduce the observation [16].

Another technique is model-based testing, which has for instance been used to test implementations of Message Queuing Telemetry Transport (MQTT) servers [17]. In model-based testing, a formal model of the system is created from requirements specifications. A model is created to represent the behavior, structure, or functionality of the system. From the model test cases are generated automatically by applying specific algorithms or techniques to explore the different paths or states in the model. When test cases have been derived, the input value of the test case is provided to the system under test and the output from the system is compared to the model's output. Model-based testing can help ensure that the protocol implementation adheres to the protocol specification [18].

There has also been an extensive amount of research involving testing protocols with symbolic execution [3], [4], [19]–[24].

The overall methodology in this project has been following the methodology in the paper by Asadian et al. [4]. Asadian et al. [4] tested four different implementations of the DTLS protocol. The implementations were OpenSSL, Mbed TLS, and two variants of TinyDTLS which is a lightweight DTLS implementation targeting IoT devices. Numerous bugs and vulnerabilities were detected. The methodology described in this paper consisted of three steps, first requirements were extracted from the protocols RFC and turned into input validity formulas. In the next step, the formulas were inserted as assumptions and assertions in the source code, and the code was executed symbolically to find paths that violate a requirement. Finally, test cases were constructed and validated in the unmodified implementation.

Song et al. [20] used symbolic execution in a rule-based approach for the verification of network protocols. The methodology of Song et al. [20] consisted of four steps. The first step was to formulate rules from the protocol specification, secondly, the code was executed symbolically to collect a set of test inputs that gives a high code coverage. In the third step, the test packets were replayed on the original implementation, and the output packets from the daemon were stored together with the input. In the final step, the stored input and output packets are checked against the rules that were formulated in step one. This is done by translating the rules into a set of non-deterministic finite automata (NFAs) and then an analyzer matches all captured replay packets against each NFA to detect rule violations. Implementations of the protocols Zeroconf (Avahi, Bonjour, JmDNS) and DHCP (isc-dhcp, udhcp) was tested. Bugs were detected in all the implementations.

Tempel et al. [3] presented a specification-based symbolic execution approach for stateful network protocol implementations. SYMEX-VP which is an open-source symbolic execution framework for RISC-V embedded software was used. In this paper, the aim was to reach high code coverage in stateful network protocols and to test sequences of packets instead of only a single packet. The approach was to increase the number of packets incrementally, so first the system under test is tested up to a certain depth, and software execution was suspended and the system under test was explored in

its breadth first and then the packet sequence length was incremented to explore its depth again. By executing the system under test symbolically to a certain depth, the initial state space was reduced and the complexity of the generated SMT queries was simpler. Four benchmark applications were tested. From the RIOT operating system, the DHCPv6 implementation and two MQTT-SN implementations (emcute and asymcute) were tested. From Zephyr, the DHCPv4 implementation was tested. The code coverage for the MQTT-SN implementations emcute and asymcute was increased by 33% and 19% respectively. For RIOT's DHCPv6 the code coverage was increased by 25% and for Zephyr's DHCPv4 the code coverage was increased by 7%. Bugs were detected in the DHCPv6 and asymcute implementations.
Song et al. [25] used symbolic execution to enhance conformance testing. The symbolic execution engine KLEE was used. A conformance testing suite consists of a set of packet sequences, including inputs and expected outputs. Every sequence of packets corresponds to a specific path in the program. Inputs were from an existing conformance test suite and the selected inputs were made symbolic and the system under test was executed symbolically. This way high quality test input packets were generated and recorded and used as inputs to the original SUT. Running the system under tests with the high quality inputs, the outputs were recorded. Finally, the high quality inputs with the corresponding outputs were validated against the conformance testing rules. The methodology was evaluated on two protocols, the Kerberos Telnet protocol (telnetd) and the DHCP protocol (udhcp). The results were 83,2% code coverage and detecting two critical security flaws in telnetd and five previously detected memory bugs with 76,3% source code coverage for udhcp.

# 8 Discussion & Analysis

In this section, symbolic execution as a testing technique is discussed. Also, the problems that were encountered during this project are presented.

## 8.1 Symbolic Execution

One challenge with symbolic execution is path explosion, for every conditional branch the execution is forked for each possible and yet realizable path [11]. To avoid time-consuming symbolic execution, in this project only a specific field that was relevant to a specific requirement was made symbolic, instead of making the whole message symbolic. With the exception of the token length requirement, which involved testing the correlation between the token length field and the actual token length, and the experiment regarding the payload marker, the majority of the experiments took a few seconds.
When testing a requirement related to, for example, the message_id, it is unlikely that other fields such as token length and code will impact the processing of the message_id. Therefore, these fields kept their concrete values. It is also more difficult for a constraint solver to solve a constraint involving two symbolic variables at the same time rather than one [11]. The disadvantage of this approach is that it can miss inputs and possible bugs that can only be detected when multiple fields are made symbolic simultaneously.
Another challenge with symbolic execution is the interaction with the environment such as external function calls and cryptographic primitives. Executing cryptographic primitives symbolically leads to complex symbolic expressions that are difficult for the SMT solver to handle [7]. To avoid cryptographic primitives, modifications were made to the system under test where the cryptographic primitives did nothing except for copying the content of the plaintext buffer into the ciphertext buffer and vice versa. The downside of this approach is that the unexecuted parts are not being tested.
Another challenge with stateful protocols lies in testing the deeper parts of the implementation. Sending just a single packet is inadequate when dealing with stateful protocols, as reaching the deeper layers of the implementation typically requires the transmission of multiple packets [4]. The approach employed in this project successfully addresses the testing of stateful protocols. Despite using a single message for each requirement, the recorded messages correspond to different protocol states. As a result, the testing process encompasses the examination of the implementation's deeper layers.

## 8.2 Problems Encountered

In EDHOC only the method field and the cipher suite field were tested. One reason was that the implementation of EDHOC was incomplete, for instance, cipher negotiation and error messages were not implemented and the EAD field was not used. As a result, these aspects could not be tested.
Moreover, EDHOC incorporates cryptographic primitives and fields within the messages that are not well-suited for testing through symbolic execution. For example, the ephemeral public keys and authentication parameters ID_CRED_I, CRED_I, and Signature_or_MAC are all associated with cryptographic and authentication operations. Additionally, message correlation, fragmentation, duplication, reordering, and flow control are the responsibility of the transport layer and not EDHOC [27]. Consequently, potential requirements pertaining to these properties were not applicable in this context.

# 9 Conclusion & Future Work

This study utilizes symbolic execution to test one implementation of the EDHOC protocol and one implementation of the CoAP protocol. The methodology involved extracting requirements from the RFCs and translating them into formulas. These formulas were then applied during symbolic execution to detect any paths that violate a requirement. This project investigated if the methodology presented in the paper by Asadian et al. [4] could be used to test other protocols. This approach uncovered non-conformances in both implementations. However, symbolic execution is not ideal for testing protocols with numerous cryptographic primitives and parameters as it results in complicated symbolic expressions that the SMT solver cannot manage. Modifications are required to enable symbolic execution to effectively test protocols that have numerous cryptographic primitives. The methodology outlined in this thesis is well-suited for testing the CoAP protocol. Unlike EDHOC, CoAP does not involve cryptographic primitives and encompasses several fields within the messages that can be effectively tested using the approach described in this thesis.

## 9.1 Future Work

Due to time constraints, only one implementation of each protocol was tested in this project. However, future work could involve testing additional implementations of the CoAP protocol, which would provide valuable insights. Certain aspects of the work conducted in this project could be leveraged for such testing. Notably, this project primarily focused on testing the server side, making it worthwhile to explore testing the client side as well.

# Literature

[1] M. Hasan, *State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally, IOT ANALYTICS*, https://iot-analytics.com/number-connected-iot-devices/, [Accessed: May 4, 2023].

[2] S. N. Swamy and S. R. Kota, "An Empirical Study on System Level Aspects of Internet of Things (IoT)," *IEEE Access*, vol. 8, pp. 188 082–188 134, 2020.

[3] Tempel, Sören and Herdt, Vladimir and Drechsler, Rolf, "Specification-based Symbolic Execution for Stateful Network Protocol Implementations in the IoT," *IEEE Internet of Things Journal*, pp. 1–1, 2023.

[4] H. Asadian, P. Fiterau-Brostean, B. Jonsson, and K. Sagonas, "Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 70–81.

[5] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 63–67, 2014.

[6] B. Möller, T. Duong, and K. Kotowicz, "This POODLE Bites: Exploiting The SSL 3.0 Fallback," *Security Advisory*, vol. 21, pp. 34–58, 2014.

[7] M. Vanhoef, "WiFuzz: detecting and exploiting logical flaws in the Wi-Fi cryptographic handshake," *Blackhat*, 2017.

[8] G. Selander, J. P. Mattsson, and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)," Internet Engineering Task Force, Internet-Draft draft-ietf-lake-edhoc-19, February 2023, Work in Progress, 108 pp.

[9] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (CoAP)*, RFC 7252, Jun. 2014.

[10] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[11] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[12] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, 2008, pp. 209–224.

[13] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS Implementations Using Protocol State Fuzzing," in *USENIX Security Symposium*, 2020.

[14] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1492–1504.

[15] K. Sagonas and T. Typaldos, "EDHOC-Fuzzer: An EDHOC Protocol State Fuzzer," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '23, New York, NY, USA: ACM, Jul. 2023.

[16] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138.

[17] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-Based Testing IoT Communication via Active Automata Learning," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 276–287.

[18] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, *Model-Based Testing of Reactive Systems*. Springer Science & Business Media, 2005, vol. 3472.

[19] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 331–340.

[20] J. Song, C. Cadar, and P. Pietzuch, "SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 695–709, 2014.

[21] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Computational verification of C protocol implementations by symbolic execution," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 712–723.

[22] D. Brand and W. H. Joyner Jr, "Verification of protocols using symbolic execution," *Computer Networks (1976)*, vol. 2, no. 4-5, pp. 351–360, 1978.

[23] R. Corin and F. A. Manzano, "Efficient Symbolic Execution for Analysing Cryptographic Protocol Implementations," in *Engineering Secure Software and*

*Systems: Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings 3*, Springer, 2011, pp. 58–72.

[24] J. Song, T. Ma, C. Cadar, and P. Pietzuch, "Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution," in *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, IEEE, 2011, pp. 1–8.

[25] J. Song, H. Kim, and S. Park, "Enhancing Conformance Testing Using Symbolic Execution for Network Protocols," *IEEE Transactions on Reliability*, vol. 64, no. 3, pp. 1024–1037, 2015.