Formal verification of the implementation of the MQTT protocol in IoT devices

Research Project 2
Master of System and Network Engineering
University of Amsterdam

Kristiyan Mladenov
`kristiyan.mladenov@os3.nl`

Supervisors:
Stijn van Winsen
Chris Mavrakis
KPMG Cyber

15 July 2017

Abstract

Message Queue Telemetry Transport (MQTT) is a protocol suitable for application in Internet of Things (IoT) devices. It is designed around requirements for low bandwidth and small code footprint. The count of the embedded devices that make use of it is constantly increasing. Therefore, a mistake in its implementation would be critical from both operational and security perspective.

The following research is aimed at finding a formal technique to verify whether the MQTT implementations are adhering to the standard. After discussing several possible methods, the Test and Test Control Notation version 3 (TTCN-3) language is selected. It is used to define different tests, based on the normative requirements in the standard. Executing those tests showed discrepancies between the definition of the protocol and the three different open source implementations that were selected for verification. As a side effect, the option to fingerprint those implementations based on the selected tests is also discussed.

Contents

1. Introduction

IoT devices were the main building block in recent record breaking Distributed Denial of Service (DDoS) attacks, like the ones performed by the Mirai botnet [1]. Most of the attacks rely either on unchanged default configurations of the devices or outdated software with unpatched vulnerabilities. While the first one is mainly caused by human error and/or ignorance, the later can be mitigated by regulations which are already being proposed by the security community [19]. Different vendors in the security evaluation and compliance field are also offering testing methodologies specifically focused on IoT devices [7]. It would be a good idea to extend those evaluations in order to include methods for formally verifying that the protocols implemented in the embedded devices are compliant with the standards they claim support for. This research is aimed at finding and putting into practice methods for formally assessing to which extent a given product adheres to the protocol specification.

A widespread standard in the IoT world, whose implementation verification will be studied further, is the MQTT protocol. Although its first version dates back to 1999, its current specification has become an official International Organisation for Standardisation (ISO) standard in 2016. However, it is already widely deployed for different IoT applications, including the backend of The Things Network (TTN) - a global IoT data network utilising the LoRaWAN specification. Therefore, a method to formally test its implementation could prevent future large-scale compromises.

Research question

A serious security compromise may be caused by weak protocol implementation in any given device. Therefore, the main research question is defined as follows:

Can the MQTT protocol implementation in IoT devices be verified formally?

In order to find an answer to this problem, a resolution to the following sub-question should be provided:

(1) What methods can be used to formally assess the implementation of a communication protocol?

A certain level of assurance that a given device does not incorporate vulnerabilities based on its protocol compliance can be reached through formal testing.

Once formal testing methods are identified, the next step in the research would be to test the implementation of the MQTT protocol as a relevant protocol in the IoT realm. This would lead to an answer of the second research sub-question:

(2) Using the chosen formal testing methods, does the MQTT implementation in certain selected IoT devices adhere to the standard?

Structure

The report is structured as follows. Chapter 2 on the following page summarises the relevant related research. What follows in chapter 3 on page 5 is the description of how the questions stated above were approached. The results based on this approach are described in chapter 4 on page 14 and discussed in chapter 5 on page 15. Finally, a conclusion is drawn in chapter 6 on page 16 and relevant future work is proposed.

2. Related work

The idea to formally verify protocol implementations, also known as conformance testing, is not new. The organisations involved in the standards development process are interested to propose frameworks facilitating the verification of a given implementation. The current report will investigate the implementation of the MQTT protocol. MQTT was recently standardised by ISO, so it is reasonable to expect that either that organisation or another organisation focused on communication standards would be proposing methods for conformance testing. Without aiming to be an encyclopedic summary, section 2.1 gives an overview of some of the relevant testing methodologies proposed by certain standardisation organisations and previous cases in which they were successfully applied. However, not every formal method is backed up by such body. Therefore, the next section 2.2 on the following page summarises formal protocol and systems verification techniques for which no proof was found, that they were endorsed or specified by any standardisation organisation. Finally, the relevant work dealing with the MQTT protocol is described in section 2.3 on page 4

2.1 Verification techniques used by standardisation bodies

To start with the larger standardisation bodies, ISO has developed the *Conformance testing methodology and framework* formally documented under the ISO/IEC 9646 standard. It is a family of several documents ranging from general concepts to test specifics and implementation details. Unfortunately, it is not an open standard, as each of the sections should be purchased separately for a certain fee and they are protected by copyrights [8]. However, Jan Tretmans has published an overview of the standard, which was used as a starting point [22]. One notable part of the *Conformance testing methodology and framework* includes the TTCN language originally known as Tree and Tabular Combined Notation. Its third iteration redefines it as TTCN-3 and is also freely accessible as a European Telecommunications Standards Institute (ETSI) standard. It was used to deploy a testing framework for the Session Initiation Protocol (SIP) widely deployed as a Voice over IP (VoIP) technology, which proves its ability to test communication protocols.

Despite the fact that it is part of the non-open ISO Conformance testing methodology and framework, TTCN-3 is listed by the International Telecommunication Union (ITU) and its telecommunications governing division ITU-T as a part of the Z recommendation series. Those series are describing languages and general software aspects for telecommunication systems. More specifically, the Z.100-Z.199 subdivision of recommendations, under which TTCN-3 is featured, has the name Formal Description Techniques (FDTs). Both the older Tree and Tabular Combined Notation and the newer Test and Test Control Notation version 3 have recommendation numbers within that range, more specifically Z.140 - Z.149 for the former and Z.160-Z.179 for the later. To conclude the FDT list in the Z recommendation series, other methods include the usage of the Specification and Description Language (SDL), Message Sequence Chart (MSC) and User Requirements Notation (URN). However, most of the studies listed herein, dealing with conformance testing, make use of TTCN-3, because of its ability not only to describe a system, but to execute the relevant tests as well.

To further verify the formal approach provided by TTCN, a work by Ruibing Hao and Jianping Wu was identified. It is named *Toward Formal TTCN-based Test Execution* and there they represent a way to describe the execution process of a TTCN-based test into the terms of an Input-Output Transition System [6]. According to them, the Input-Output Transition System can be treated as a form of a Labelled Transition System. Another example of scientific work proposing formal approach to conformance testing was performed by Jan Tretmans in his PhD thesis [21]. There he describes algorithms for deriving complete and correct tests based on specifications presented as a Labelled Transition System.

A study by Thomas Deiss proposes different techniques for writing effective TTCN-3 test cases for large systems [4]. He also describes several areas of application, where in order for the language to be effective, a future extension is required. Broadcasting messages and handling unspecified data types are some of the problems described. None of them would negatively affect its ability to be applied for the MQTT verification. Despite of the highlighted shortcomings, the same FDT was used to verify web applications [17], Intelligent Network designs [10] and probably many more.

Shifting the focus to the Industrial IoT (IIoT), the implementation of the IEC 61850 standard (Intelligent Electronic Devices for electrical substation automation systems), was studied by Georg Panholzer et al. in their research [16]. The group of researchers utilises TTCN-3 as a test specification and execution language and is successful in executing their conformance tests. They are also highlighting the TTCN-3‘s ability to use templates, parse Abstract Syntax Notation 1 (ASN.1) and Extensible Markup Language (XML) schema types and interface definitions done in the Interface Description Language (IDL). Another important benefit of the testing method the researchers make use of is its ability to run sets of distributed tests.

2.2 Other verification techniques

In order to make the transition to other testing methods, a paper dealing with the implementation of another IEC protocol was studied. Namely, this is research done by Max Kerkers assessing the security of the IEC 60870-5-104 protocol implementation in Industrial Control Systems [9]. The formalism used there is known as Mealy Machines, which are a form of Finite State Machines. The described approach treats the devices subjected to the test as black boxes and their reactions to externally generated messages are studied. Based on the responses received, a state machine is built and compared with the one derived by the definition of the standard. That allowed for identifying differences between the specification and the implementation. If this study is considered to be focused on the same class of IIoT protocols as the last one in section 2.1 on the preceding page, then an assumption can be made that both TTCN-3 and Mealy Machines can be used for comparable sets of problems.

The use of Input-Output Control theory is shown to be useful in component based testing in the research done by Machiel van der Bijl et al. [23]. They are even going further, showing that in certain specific cases when components are tested separately and integrated afterwards, the resultant system will also be conforming to the integrated specification.

An application of the χ specification language and the TorX testing tool was found in a case study of the ASML laser subsystem. During the study, discrepancies between the specification and the implementation were found [3]. There Niels Braspenning et al. argue that the most time-consuming task during the formal definition of a system is not writing down the specification, but coming up with a way to derive an unambiguous one.

Another interesting method for the formal verification of concurrent systems is proposed by the Temporal Logic of Actions plus (TLA+) formal specification language, developed by Leslie Lamport. It is based on set theory, first-order logic and temporal logic of actions [11]. Engineers at Amazon have already successfully used it in order to identify design issues with services they provide, like DynamoDB, Elastic Block Storage (EBS) and Simple Storage Service (S3) [15]. They are not the only ones that have documented their findings [25]. Unfortunately, no publication dealing with the applicability of the language for protocol compliance testing was identified. However, previous research papers focused on the protocol design verification using TLA+ were studied.

An example of relevant scientific work dealing with protocol checking using TLA+ is the work performed by Narayana, et al. which is describing a way for automatic vulnerability checking of the IEEE 802.16 Wimax protocol [14]. By modelling the protocol using the TLA+ language, they are aiming to identify Denial of Service (DoS) vulnerabilities in the specification. While that approach allowed them to highlight some deficiencies which could lead to a DoS condition during the Initial Ranging process, they could not do the same in the Authentication process with the model that they built.

To continue the topic of protocol compliance testing, Kerberos was also taken as an example in the paper by Lian Wan et al. where they study TLA as a tool for protocol checking and verification [24]. Based on the model that they built, a sequence of steps which might help an attacker impersonate an entity in the authentication process is shown.

2.3 MQTT

As the research is going to focus on the MQTT protocol and its latest version 3.1.1 [2], it is valuable to highlight what has already been done in this area. During the 24th DEFCON conference held in 2016, Lucas Lundgren presented critical remarks about the widespread use of the MQTT protocol [12]. It was found to be incorporated in medical equipment, fitness bands, messaging apps, ATM devices, and many others. Even the Deutsche Bahn have deployed it in different areas of their railway system [18].

Some testing of the protocol implementation can already be done using modules for the Scapy packet manipulation tool, as a MQTT module is available in its github repository [20]. The `nmap` scanning software also has a script in its scripting engine, which is designed for MQTT interaction. However, they do not fall under any of the formal categories described so far.

Summary

Without pretending to be a comprehensive list of all the relevant work performed in the past years, this section provided a general overview of some of the methods available. The level of diversity between the formal methods is high, ranging between labelled transition systems, finite state automation, set theory and first order logic.

Given the widespread adoption of the MQTT protocols and the availability of tools like Scapy and `nmap` which can be used to abuse the standard, a formal evaluation of the implementations would be beneficial.

3. Approach

Based on research studied thus far and summarised in chapter 2 on page 2, section 3.1 describes the tools selected to complete the implementation verification. Section 3.2 describes the environment setup and the implementations used to run the experiments. Then section 3.3 on the following page gives a brief overview of the MQTT protocol iteslf. The next section 3.4 on page 8 explains the approach used for deriving the tests which determine the protocol compliance levels. Section 3.5 on page 11 lists the normative requirements according to which the test cases were created. Finally, section 3.6 on page 12 describes a way to improve the test derivation process.

3.1 Tool selection

A formal description technique needs to be selected before choosing a tool for testing. The research, discussed previously, highlights some of the advantages of TTCN-3. It can be used not only for describing how a given implementation should behave, but also for verifying if the System Under Test (SUT) adheres to those definitions. It has the added benefit of being created by a large standardisation body like ETSI and it is backed up by other organisations, such as ISO, and ITU. Furthermore, it has hundreds of pages of documentation dealing with the numerous features of the language and examples involving various use cases. Because of that, TTCN-3 was selected as the formal language used for doing the testing.

Given that TTCN-3 has been originally defined in the 2000s, the industry has had considerable amount of time to mature and offer a variety of tools for describing and executing the tests. The official TTCN-3 web page[1] lists only two of the tools as being open source, namely the *Titan TTCN-3 Toolset* and *TRex - The TTCN-3 Refractoring and Metrics Tool*. Both projects are backed up by the Eclipse Foundation, with the first one being developed by the Swedish networking and telecommunications equipment provider Ericsson, and the second one by the University of Göttingen. The better support of languages and notations like ASN.1, XML and JavaScript Object Notation (JSON) made *Eclipse Titan* the preferred choice, which will be used during the project.

Another project dealing with TTCN-3, which might be useful in future studies, is the Eclipse IoT Testware[2]. Its production ready release is scheduled for the third quarter of 2018. It has planned support for protocols applicable in the IoT world like MQTT, discussed within this paper, and Constrained Application Protocol (CoAP). Ericsson is involved in the development of both this project and the Eclipse Titan project. Therefore, some of the TTCN-3 modules dealing with MQTT are already available in the Titan repository [5]. This includes definitions for the basic message types involved in the MQTT message exchange.

3.2 Testing environment setup

A list of the last four Eclipse Titan versions is given in table 3.1[3]. This research started with version 6.1.0. In the course of the experiments development, the work was moved to version 6.2.0. Initially, the release candidate was utilised, but by the end of the testing phase of the project, the production version was finally released and used for executing the experiments. Version 6.2.0 adds support for dynamic erroneous attributes definition. Their usage allows the message templates to be modified with values not specified by the standard. This allows for better ground for studying the SUT behaviour in those undefined cases.

Titan version	Release date
6.2.0	2017-06-21
6.1.0	2017-03-01
5.5.0	2016-06-15
5.4.0	2015-12-16

Table 3.1: Latest Titan versions

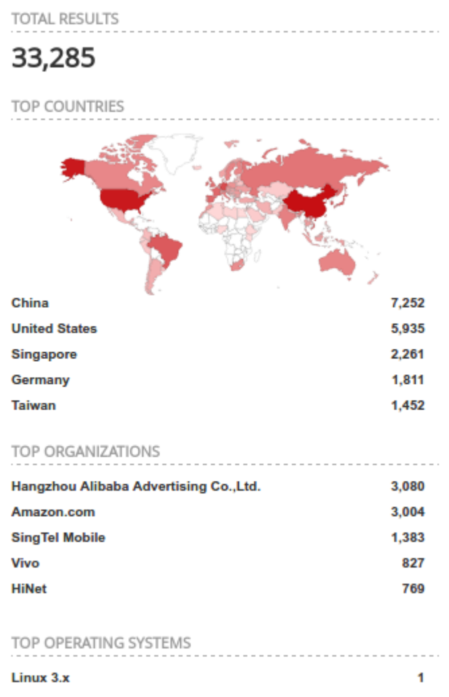
[1]Source: `http://www.ttcn-3.org/index.php/tools`
[2]Source: `https://projects.eclipse.org/proposals/eclipse-iot-testware`
[3]Source: `https://projects.eclipse.org/projects/tools.titan/governance`

The Eclipse Titan project offers both command line utilities and plugins for the Eclipse Integrated Development Environment (IDE). They were used interchangeably during the project.

In order to better understand the implementations to be subjected to testing, the MQTT standard was studied [2]. Section 7 of the standard is named *Conformance* and defines normative requirements which both the client and the server should meet. A closer look in the document shows that the amount of requirements for the server behaviour outnumbers those for the client. For this reason the focus of the research was shifted towards testing the server implementation rather than the client one. The MQTT protocol will be discussed in greater detail in section 3.3.

Some additional research was needed in order to identify if there is a predominant flavour of MQTT server implementation reachable over the Internet. Other services reachable over the Internet use banners to identify the running daemon. The banner is usually a brief message providing more information about a service listening on a given port. However, banner grabbing is not applicable to MQTT, as the protocol does not support the notion of banners.

As a next step, the Shodan[4] search engine was referred to. Shodan aims at scanning connected devices on the Internet. Apart from the information gathered through service banners, it also keeps track of which open ports each device exposes to the Internet. Searching for the default MQTT port 1883 did return a list of over 30,000 devices, but did not provide any further details about which exact implementation is bound to the port. An example of the search result is provided in figure 3.1.



TOTAL RESULTS

33,285

TOP COUNTRIES

China	7,252
United States	5,935
Singapore	2,261
Germany	1,811
Taiwan	1,452

TOP ORGANIZATIONS

Hangzhou Alibaba Advertising Co.,Ltd.	3,080
Amazon.com	3,004
SingTel Mobile	1,383
Vivo	827
HiNet	769

TOP OPERATING SYSTEMS

Linux 3.x	1

Figure 3.1: Shodan search for port 1883

To finally select the SUTs the information in the github MQTT wiki[5] was used. It contains details about the currently available implementations as well as a list of the publicly accessible brokers suitable for service testing. The implementations that were selected to continue the research with had to meet two criteria - to be open source and to have a publicly available broker, showing its applicability in large-scale deployment scenarios. None of the public brokers were subjected to testing during the course of this project due to ethical considerations. However, as long as they run the same software as the implementations selected for this project, the results should be applicable to them as well. Table 3.2 lists the selected software that was studied further.

The MQTT brokers which were subjected to the test, were set up on fully up-to-date virtualised servers running Ubuntu 17.04. With the exception of RabbitMQ, the implementations were running their default configurations. The configuration of RabbitMQ was altered, so that the MQTT plugin was enabled and anonymous access to the broker was configured.

Implementation	Version	Language
Mosquitto	1.4.10	C
Emqttd	2.2-rc.1	Erlang/OTP
RabbitMQ	3.6.10	Erlang

Table 3.2: Tested implementations

3.3 MQTT overview

The MQTT protocol utilises a simple Publish/Subscribe mechanism for message exchange. The message exchange is organised in topics, similar to the channels offered by the Internet Relay Chat (IRC) protocol. This research is focused on testing the implementations using the Transmission Control Protocol (TCP) as means of transport, but the standard also defines the use of WebSockets as admissible.

[4]Source: https://www.shodan.io/

[5]Source: https://github.com/mqtt/mqtt.github.io/wiki/

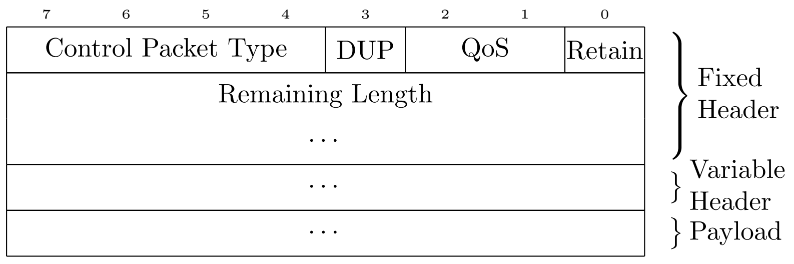


Figure 3.2: MQTT Packet structure

Figure 3.2 displays the general structure of each MQTT message. The bit order is specified in the same way as the standard. The only mandatory part of the message that should be always included is the Fixed header. There are 14 distinct Control Packet Types. Table 3.3 lists them together with the direction in which they are usually exchanged (Client → Server, Server → Client or bidirectional) and the supported values for the DUP (Duplicate), Quality of Service (QoS) and Retain flags. With the exception of the PUBLISH message, all other messages have either flags set to zero, or in the case of PUBREL, SUBSCRIBE and UNSUBSCRIBE, the first QoS bit set to one.

Name	Value	Direction	Description	Flags
Reserved	0	Forbidden		
CONNECT	1	C → S	Client connection request	0000
CONNACK	2	C ← S	Server response to CONNECT	0000
PUBLISH	3	C ↔ S	Message publishing	Varying
PUBACK	4	C ↔ S	PUBLISH acknowledgement	0000
PUBREC	5	C ↔ S	PUBLISH received	0000
PUBREL	6	C ↔ S	PUBLISH release	0010
PUBCOMP	7	C ↔ S	PUBLISH complete	0000
SUBSCRIBE	8	C → S	Client subscription to topic	0010
SUBACK	9	C ← S	Server response to SUBSCRIBE	0000
UNSUBSCRIBE	10	C → S	Client unsubscription to a topic	0010
UNSUBACK	11	C ← S	Server response to UNSUBSCRIBE	0000
PINGREQ	12	C → S	Keepalive PING request	0000
PINGRESP	13	C ← S	Keepalive PING response	0000
DISCONNECT	14	C → S	Client is disconnecting	0000
Reserved	15	Forbidden		

Table 3.3: MQTT packet types and their supported flags

The QoS field is two bits long with three supported values. A proper understanding of the values is essential in order to correctly interpret the tests described further in this paper.

- 0 (At most once delivery) - This delivery method sends the PUBLISH messages without expecting any type of acknowledgement. A level of assurance might be sought in the underlying TCP protocol and its acknowledgement mechanism, but except for the successful delivery of the message, no guarantees for the subsequent treatment of the data by the server/client are given.
- 1 (At least once delivery) - A PUBLISH message delivered with a QoS level of 1 requires that it is acknowledged with a PUBACK message. This delivery method assures the delivery of at least one copy of the message.
- 2 (Exactly once delivery) - A PUBLISH message with QoS level of 2 triggers the exchange of three additional messages - the PUBREC, PUBREL and PUBCOMP, making sure that exactly one copy of the message is getting delivered.

The fourth value is reserved and must not be used.

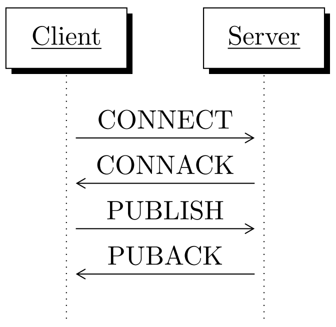


Figure 3.3: MQTT message exchange during publishing

Figure 3.3 shows the messages exchanged when a client publishes a message with QoS level of 1 (At least once delivery). After receiving the PUBACK message, the client may stay connected and send periodic keepalives using the PINGREQ message, or disconnect using the DISCONNECT message. The choice of behaviour should be made by whoever makes use of the protocol as means of message exchange and the specification allows both behaviours.

3.4 Deriving the tests

The initial course of action was to check if the ASN.1 language could be used to define the MQTT message structure. The creators of certain protocols, such as the Simple Network Management Protocol version 2 (SNMPv2)[6] have used ASN.1 as a FDT to define the Structure of the Management Information (SMI). It was also used for the definition of the format of the certificates in the X.509 cryptographic standard. Although ASN.1 proves to be a powerful language in those cases, it cannot be used to define correlation between the values it describes. For example the MQTT CONNECT message contains the two headers and the payload as defined by figure 3.2 on the previous page. On the other hand, the PINGREQ message contains only the Fixed Header. Based on their Fixed Header, it is possible to distinguish between the two by looking at the Control Packet Type field, but ASN.1 does not support syntax for that. Therefore, a separate ASN.1 definition would have been required for each message type.

Upon closer inspection of the TTCN-3 module available in the Eclipse Titan git repository[7] a relevant structure defining each message type was already available. Each possible MQTT message falls within the scope of the TTCN-3 unions defined in listing 3.1. More specifically, the message can either be treated as an octet (byte) string, or as a structure, which on its own is one of the 14 types of valid MQTT messages.

```
type union MQTT_v3_1_1_Message
{
  MQTT_v3_1_1_ReqResp            msg,
  octetstring                    raw_message
}

type union MQTT_v3_1_1_ReqResp
{
  MQTT_v3_1_1_Connect            connect_msg,
  MQTT_v3_1_1_Connack            connack,
  MQTT_v3_1_1_Publish            publish,
  MQTT_v3_1_1_Identifier         puback,
  MQTT_v3_1_1_Identifier         pubrec,
  MQTT_v3_1_1_Identifier         pubrel,
  MQTT_v3_1_1_Identifier         pubcomp,
  MQTT_v3_1_1_Subscribe          subscribe,
  MQTT_v3_1_1_Suback             suback,
  MQTT_v3_1_1_Unsubscribe        unsubscribe,
  MQTT_v3_1_1_Identifier         unsuback,
  MQTT_v3_1_1_Empty              pingreq,
  MQTT_v3_1_1_Empty              pingresp,
  MQTT_v3_1_1_Empty              disconnect_msg
}
```

Listing 3.1: A TTCN type defining union between all possible MQTT messages

Although the description of each different type of message is skipped in the paper, a simple example of a TTCN-3 template is displayed in listing 3.2 on the next page. It gives values for all

[6]Source: https://tools.ietf.org/html/rfc2578
[7]Source: http://git.eclipse.org/c/titan/titan.ProtocolModules.MQTT.git

the fields present in the CONNECT message. This specific template was found as an example in the Eclipse Titan Community Forums[8]. Based on it, templates for additional types of messages were defined during the research.

```
template MQTT_v3_1_1_FTypes.MQTT_v3_1_1_Message t_F_connect(integer p_id) := {
  msg := {
    connect_msg := {
      header := { packetType:='0001'B , flags := '0000'B, remLength:='00000000'O },
      nameLength:=0,
      name := "MQTT",
      protocol_level := 4,
      flags := {
        user_name_flag := '0'B,
        password_flag := '0'B,
        will_retain := '0'B,
        will_qos := AT_MOST_ONCE_DELIVERY,
        will_flag := '0'B,
        clean_session := '1'B,
        reserved := '0'B
      },
      keep_alive := 20480, // Due to endiandness, 20480 represents 10
      payload := {
        client_identifier := {stringLength:=0, stringItem:="TTCN_FUN"&int2str(p_id)
    },
        will_topic := omit,
        will_message := omit,
        user_name := omit,
        password := omit
      }
    }
  }
}
```

Listing 3.2: A TTCN template for CONNECT messages

The use of templates does not only provide for easy definition of the messages that need to be sent, but also for recognition of the messages that are originated by the SUT and received in response to the test packets. Before proceeding to the description of how the tests were derived, one additional TTCN-3 construct needs to be discussed. This is the `alt` construct, similar to the `switch/case` construct in some programming languages. In the case of TTCN-3, it waits for one of the specified events to happen - either a response is received, the session is disconnected or a timer expires. As this construct is defined within a test case, each individual alternative defines the result of the test by using the `setverdict` clause - it is either a `pass` or a `fail`.

```
  //Sends a message based on the t_data_connect function, using the above template
  p.send(t_data_connect(v_cid));

  //Starts a one second timer
  t.start(1.0)
  alt {

    // For simplicity, the code assigning the v_retCode variable to the
    // value of the Return Code in the CONNACK message is skipped.
    [] p.receive(ASP_RecvFrom:?) -> value v_ASP_RecvFrom {
      ...
      if (v_retCode == 0)
      {
        log("Connection request accepted");
        setverdict(fail)
      }
      else
      {
        log("Connection request failed");
        setverdict(fail)
      }
    };

    // The case when another TCP event like TCP reset is sent
    [] p.receive(ASP_Event:?) -> value v_ASP_Event {
      log("Unexpected ASPEvent!!!" ,v_ASP_Event );
```

[8]Source: https://www.eclipse.org/forums/index.php?t=msg&th=1085301&goto=1765338

```
54        setverdict(pass)
55      };
56
57      // The case of a timeout
58      [] t.timeout {
59        log("No answer received. Setting verdict to fail.");
60        setverdict(fail)
61      }
62    }
```

Listing 3.3: An example of a TTCN `alt` statement

Having defined the most used TTCN-3 constructs, the next thing to be discussed is how the tests were derived. The *Conformance* section of the MQTT standard defines the normative requirements for both the server and the client implementations. In summary, all the statements specified with the word MUST as per RFC 2119[9] have to be satisfied. Those requirements are further aggregated in Appendix B of the standard [2]. The tests derived herein were based on those specific statements. Figure 3.4 shows a simplified visualisation of the test execution process followed during the research.

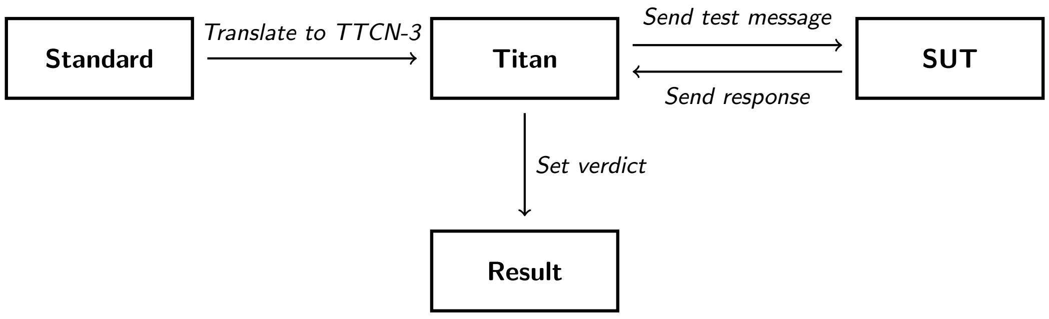


Figure 3.4: Simplified model of the approach

The first example uses the normative requirements described in table 3.4 and they are listed using their original definitions. Statement [MQTT-2.2.2-1] refers to Table 2.2, but this table number has been defined in the context of the original standard. In the current paper, the values of the Flag Bits, referred to in that table, are listed under the Flags column in table 3.3 on page 7.

Statement Number	Normative Statement
[MQTT-2.2.2-1]	Where a flag bit is marked as “Reserved” in Table 2.2 - Flag Bits, it is reserved for future use and MUST be set to the value listed in that table.
[MQTT-2.2.2-2]	If invalid flags are received, the receiver MUST close the Network Connection.

Table 3.4: Normative requirements for Section 2.2.2 of the standard

The CONNECT message is an example of a message which should have its 4-bit flags section in the Fixed Header set to `0`. Those bits are properly defined in the template described in listing 3.2 on the previous page. However, with the ability to dynamically add erroneous attributes to a template, available in Eclipse Titan version 6.2.0, they can easily be updated to a different value, for example `0010`. Listing 3.4 suggests two ways to achieve this - either through substituting the entire Fixed header in its octal form, or by substituting the flags section in the header.

If a message updated with the above values is sent to the SUT, according to [MQTT-2.2.2-2], the network connection should be closed in response. In that context, listing 3.3 on the previous page makes more sense, as if in response to a CONNECT message with Fixed Header flags not set to `0` the servers sends back a CONNACK message with Return Code `0`, that means that the server has accepted the connection. However, according to the standard, the client should have been disconnected. This makes the alt statement fall into the condition described on line 52 in listing 3.3 on the preceding page and set the overall verdict of the test to `pass`.

```
  // Either substitute the entire Fixed header:
  @update(t_F_connect) with {
```

[9]Source: https://tools.ietf.org/html/rfc2119

```
    erroneous (msg.connect_msg.header) "value := '1300000000'O "
  }

  // Or update only the flags field.
  @update(t_F_connect) with {
    erroneous (msg.connect_msg.header.flags) "value := '0010'B "
  }
```

Listing 3.4: Update the CONNECT template with erroneous values

Some of the tested systems responded to the erroneous message as specified by the standard, while others were not conforming and accepted the malformed connect message. Test cases were also defined for several other normative requirements listed in section 3.5.

3.5 Tested normative requirements

The MQTT standard sets more than 140 different normative requirements. Some of them are aimed at the server implementations, others are designed to specify the client behaviour. When given functionality is not specific only for the server or the client, the requirement should be met by both. While some normative statements govern the message exchange, others define the expected message storage capabilities of the MQTT-compliant devices. This research is focused mainly on the proper message exchange. However, TTCN-3 provides means to define more sophisticated test cases capable of verifying the message retention as well.

Using the TTCN-3 language a total of 13 tests were defined. Their execution was automatic, requiring the change only of the IP address of the SUT. The normative statements, for which the tests were defined, can be found in table 3.5 on the following page. They were selected in an attempt to cover as many different aspects of the MQTT message exchange defined in sections 2 and 3 of the standard, as possible. Therefore the list of the normative requirements tested, aims to be a search in breadth, more than in depth.

In some cases the requirements are clear about what the outcome of a certain test should be - for example when closing the network connection is specified. In other cases, like in [MQTT-3.12.4-1], the requirements are not clear about what should happen if a message different than a PINGRESP is sent to acknowledge a PINGREQ. The lack of precision does not prevent the derivation of a test case which expects a PINGRESP as the only positive outcome. The client behaviour in case of the opposite depends entirely on how does the implementer of the protocol understand the requirement.

Statement Number	Normative Statement
[MQTT-2.2.2-1]	Where a flag bit is marked as “Reserved” in Table 2.2 - Flag Bits, it is reserved for future use and MUST be set to the value listed in that table.
[MQTT-2.2.2-2]	If invalid flags are received, the receiver MUST close the Network Connection.
[MQTT-2.3.1-1]	SUBSCRIBE, UNSUBSCRIBE, and PUBLISH (in cases where QoS $>$ 0) Control Packets MUST contain a non-zero 16-bit Packet Identifier.
[MQTT-3.1.0-1]	After a Network Connection is established by a Client to a Server, the first Packet sent from the Client to the Server MUST be a CONNECT Packet.
[MQTT-3.1.0-2]	The Server MUST process a second CONNECT Packet sent from a Client as a protocol violation and disconnect the Client.
[MQTT-3.1.2-2]	The Server MUST respond to the CONNECT Packet with a CONNACK return code 0x01 (unacceptable protocol level) and then disconnect the Client if the Protocol Level is not supported by the Server.
[MQTT-3.1.2-24]	If the Keep Alive value is non-zero and the Server does not receive a Control Packet from the Client within one and a half times the Keep Alive time period, it MUST disconnect the Network Connection to the Client as if the network had failed.
[MQTT-3.1.3-8]	If the Client supplies a zero-byte ClientId with CleanSession set to 0, the Server MUST respond to the CONNECT Packet with a CONNACK return code 0x02 (Identifier rejected) and then close the Network Connection.
[MQTT-3.3.1-4]	A PUBLISH Packet MUST NOT have both QoS bits set to 1. If a Server or Client receives a PUBLISH Packet which has both QoS bits set to 1 it MUST close the Network Connection.
[MQTT-3.6.1-1]	Bits 3,2,1 and 0 of the fixed header in the PUBREL Control Packet are reserved and MUST be set to 0,0,1 and 0 respectively. The Server MUST treat any other value as malformed and close the Network Connection.
[MQTT-3.8.1-1]	Bits 3,2,1 and 0 of the fixed header of the SUBSCRIBE Control Packet are reserved and MUST be set to 0,0,1 and 0 respectively. The Server MUST treat any other value as malformed and close the Network Connection.
[MQTT-3-8.3-4]	The Server MUST treat a SUBSCRIBE packet as malformed and close the Network Connection if any of Reserved bits in the payload are non-zero, or QoS is not 0,1 or 2.
[MQTT-3.12.4-1]	The Server MUST send a PINGRESP Packet in response to a PINGREQ packet.

Table 3.5: Normative requirements tested during the research

3.6 The possible role of TLA+

The test derivation process described thus far makes use of the normative requirements listed in the MQTT standard. The values for the test cases were picked manually and at random, while still making sure that they should not be acceptable according to the standard. This technique is similar to the fuzzing method, which is designed to throw invalid, malformed or random data at a certain implementation and study what happens when it misbehaves. However, when building a test case, its author should be able to specify what the expected result should be. The verdict of the test case depends on the interpretation its author gave it. This is what makes it hard for the tester to assign correct verdicts in the case when the tests are done with random or scrambled variables.

At this point a change in the approach was attempted. As previously described in section 2.2 on page 3, other authors have used the TLA+ language for protocol design verification. Despite of the fact that it cannot be used to directly interact with the SUT, given that a proper formal definition of a protocol exists, the TLA+ Model Checker can produce execution paths leading to a certain behaviour. For example if it is tasked to search for a message that should be rejected, the model can produce an execution path that leads to that behaviour.

A proper TLA+ definition of the MQTT protocol should be able to describe the behaviour of both the client and the server. As the Model Checker has the ability to pick values at random from a set of predefined ones, it can explore numerous cases and patterns of message exchange. For example if the QoS value is defined as $QoS \in [0;2]$ a random value in that range can be selected and tested. The Client and Server processes‘ behaviour can be modelled depending on the values that they receive. As the QoS field is 2 bits long, its values can actually be in the range $QoS \in [0;3]$ with 3 being a Reserved value which should not be accepted and treated as an error.

```
EXTENDS $Naturals, TLC$

VARIABLES $srvMsg, clMsg, pc$

$vars \triangleq \langle srvMsg, clMsg, pc \rangle$

$Init \triangleq$ $\land\ srvMsg \in 0 \,..\, 15$
         $\land\ clMsg \in 0 \,..\, 15$
         $\land\ pc =$ "Initial"

$Initial \triangleq$ $\land\ pc =$ "Initial"
             $\land\ clMsg' = 0$
             $\land\ srvMsg' = 0$
             $\land\ pc' =$ "Sendping"

$Sendping \triangleq$ $\land\ pc =$ "Sendping"
               $\land\ clMsg' = 12$
               $\land\ srvMsg' = 0$
               $\land\ pc' =$ "Sendresp"

$Sendresp \triangleq$ $\land\ pc =$ "Sendresp"
               $\land$ IF $clMsg = 12$
                     THEN $\land\ srvMsg' = 13$
                     ELSE $\land\ srvMsg' = 0$
               $\land\ clMsg' = 0$
               $\land\ pc' =$ "Done"

$Next \triangleq Initial \lor Sendping \lor Sendresp$
          $\lor\ (pc =$ "Done" $\land$ UNCHANGED $vars)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Termination \triangleq \Diamond(pc =$ "Done"$)$
```

Figure 3.5: An example of the MQTT keepalive message exchange

While modelling the behaviour of the Client and the Server, one can choose whether he or she wants to include the cases when erroneous values are exchanged. By doing so, execution traces that lead to message rejection by the modelled processes can be observed. When those execution paths are translated to the TTCN-3 language, new test cases with a clear verdict to expect can be defined. The TLA+ Model Checker can be configured to look either for messages that should not be accepted, or for those that are conforming to the standard. The model checking is performed in a breadth first manner until the predefined conditions are met. Once an execution path is pinpointed, the model can be relaxed to not include the same result in consequent tests. The downside of this approach is that it may suffer from the so called state explosion, due to the nature of the breadth first search.

The MQTT Keepalive exchange involving the PINGREQ and PINGRESP message types was modelled, by only specifying what Control Packet Type value in the Fixed Header is exchanged. The definition is shown in figure 3.5. The Model Checker can be used to verify which types of messages lead to successfully reaching the *Done* value of the `pc` variable.

If the keepalive is the only message exchange defined by the protocol, then this model would have been enough to derive all possible execution paths. However, according to the standard, the keepalive messages are only exchanged in the absence of other messages. They are not expected as the first messages exchanged after the TCP connection is established. For this reason this simplified TLA+ model could not be used to derive a TTCN-3 test. Given that the sample model has no definition and check for the flag fields, more work needs to be done in order to successfully derive TTCN-3 tests based on a TLA+ model. There is no guarantee that such an approach would be successful, but there is also no indication that it would not be.

4. Results

The conformance to the requirements in table 3.5 on page 12 of the three selected implementations was tested and the results are listed in table 4.1.

	Normative Requirements												
	2.2.2	2.3.1-1	3.1.0-1a	3.1.0-1b	3.1.0-2	3.1.2-2	3.1.2-24	3.1.3-8	3.3.1-4	3.6.1-1	3.8.1-1	3.8.3-4	3.12.4-1
Mosquitto	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Emqtt	×	×	✓	✓	✓	×	×	✓	✓	×	×	✓	✓
RabbitMQ	×	×	✓	✓	✓	×	×	✓	✓	×	✓	✓	✓

Table 4.1: Results of the testing against the normative requirements

While it was not surprising for the implementations to be able to pass a test, it was not expected that all of them would be failing in their conformance to the same normative requirement. This has happened with the test designed around requirement [MQTT-2.3.1-1]. It states that the Packet ID should be greater than zero when certain types of messages with QoS level greater than 0 are exchanged. As an example, the Packet ID in the PUBLISH message is used to confirm the successful delivery of the message. In the case of QoS level of 1, a PUBACK message is used. QoS level of 2 triggers the exchange of PUBREC, PUBREL and PUBCOMP messages and all of them carry the same Packet ID. In this case, the test was built using QoS level of 1. Regardless of the fact that none of the implementations is compliant to that requirement, no way was identified in which such a misbehaviour might lead to a compromise. At least not with any higher chance than a message with a different Packet Identifier. Furthermore, this is a requirement for which the standard does not specify what reaction should be triggered. Therefore, the assumption in the test was that at least no PUBACK message acknowledging the delivery should be expected. The results in table 4.1 show that all the tested implementations sent a PUBACK in response to a PUBLISH message with QoS set to 1 and a Packet Identifier set to 0.

Another common reason for the failure of the test cases is the improper checking if the Flags in the Fixed Header are set to their reserved values as specified in table 3.3 on page 7. It is worth mentioning that RabbitMQ does not claim support for MQTT 3.1.1 messages with QoS level 2[1]. However, this explains the failure of only the [MQTT-3.6.1-1] test case.

Only two of the specified tests are enough to distinguish between the three different implementations. For example, the results from testing the [MQTT-3.1.2-2] and [MQTT-3.8.1-1] normative requirements may be interpreted the following way:

- The Mosquitto implementation provides a positive outcome for both test cases.
- The Emqtt implementation fails both test cases.
- RabbitMQ fails the first test case, but passes the second one.

No Common Vulnerability Exposures (CVE)[2] with publicly known security vulnerabilities were found directly targeting the interactions of the daemons over the network. However, there are some vulnerabilities already available for other parts of the implementation. If an attacker manages to fingerprint the running service, he might be able to utilise a publicly known or not yet disclosed, present or future vulnerability associated with it. The consequences might be various, as previous research shows that there are already plenty of devices communicating through MQTT.

[1]Source: https://www.rabbitmq.com/mqtt.html
[2]Source: https://cve.mitre.org/

5. Discussion

The present research was focused on testing the implementation of the MQTT protocol. From a security perspective, one of the main problems of this study is that it focuses on the implementation verification without knowing if the standard, whose implementation is considered, is secure itself. Taking this into account, it is possible to find an implementation that is perfectly compliant to a faulty standard. The standards usually originate from large organisations and standardisation bodies having abundance of experienced people contributing to them. In addition to that, public discussions about the protocol definitions also take place. This makes the mistakes in the protocol design less likely to happen. On the other hand, the development process in smaller companies is supervised by fewer people, which makes the mistakes more likely. Therefore, when that company makes a decision to support a given protocol, the chance for internal error rises. This makes the testing of the software more likely to reveal problems than the testing of the protocol itself.

Not adhering to a standard does not automatically render a given implementation unsafe. Albeit unlikely, it might be the case that a certain set of developers have isolated a problem with the standard itself and chose not to adhere to it. None of the discrepancies in the tested implementations, listed in section 4 on the previous page, lead to a direct vector for exploitation. However, the different compliance levels can be used to perform fingerprinting and immediately identify the running implementation. This could be the first step in an attack, helping to pick the proper attack vectors and tools, known to work against a specific application.

RabbitMQ, one of the tested implementations, does not claim support for QoS level 2 message exchange. However, only one of the test cases is based on such exchange. This means that it should implement properly the rest of the normative requirements.

The testing during this research involves sending messages to the SUT and studying their responses. This can also be considered as black-box testing, which involves no prior knowledge of the tested system. All the tested implementations have their source code fully or partially available in public git repositories, but it was not used during the development of the test cases. The reason why the code review was avoided was to make the approach applicable to implementations with closed source such as HiveMQ[1] and the others like it.

The MQTT protocol provides the notion of user authentication. It also supports message exchange over an encrypted TLS channel. Despite of these facts, the servers found to be running by both the Shodan search in 3.2 on page 5 and Lucas Lundgren [12] do not seem to employ any of those.

The interpretations of the requirements done herein are based on a natural human language. Sometimes this mean of expression lacks expressive capabilities and leaves certain details under-specified. An example for this is the behaviour that should be expected in the [MQTT-2.3.1-1] test case which failed for all the studied implementations. If the protocol is described in a formal language, certain or probably even all of the ambiguities would be avoided. This idea is not new, as it was already proposed by some authors whose work was studied in section 2 on page 2. There is a way to make use of the formal definition in the testing phase as well. If, for example, the formal definition was done using TLA+, it might have been possible to derive values for the tests by running the TLA+ Model Checker. It might be even possible to distinguish behaviours supported by the protocol from those that should be rejected. However, the study of the possible usage of this specific formal method is suggested as future work.

[1]Source: http://www.hivemq.com/

6. Conclusion

The current research has identified several possible formal methods which were shown to be suitable for testing of communication protocols. They involve techniques from the Set theory, First-order logic, Finite-state machines, Labelled transition systems and others. Previous research making use of all of them was also studied and summarised herein. A decision to use TTCN-3 was made, as it is endorsed by three large standardisation organisations - ETSI, ISO and ITU. It has the ability not only to describe the expected behaviour of a given protocol, but also to execute the defined tests.

As a second step, a subset of the conformance requirements specified in the MQTT version 3.1.1 standard were interpreted into the TTCN-3 language [13]. None of the three tested implementations proved to be fully compliant to the standard, according to those tests. Furthermore, each specific implementation failed a different subset of the tests. It was shown that only two of those tests could be sufficient to distinguish between the three tested implementations. This form of fingerprinting can be used to additionally plot attack vectors against the services. This would not have been possible if the tested systems met the conformance requirements specified in the standard.

Future work

The following ideas are suggested as future work:

- The MQTT protocol can be formally defined using the TLA+ language. This would benefit in two ways - it would give ground for testing whether the protocol definition itself is correct and it would also provide new input for the tests.
- The execution of the TLA+ model might be able to provide basis for automated test derivation. For example, the modelled message exchange can be mapped into a TTCN-3 test case and fired against a real system. Such approach might be possible not only for MQTT, but for any other protocol governing data exchange.

Glossary

ASN.1 Abstract Syntax Notation 1. 3, 5, 8

CoAP Constrained Application Protocol. 5

DDoS Distributed Denial of Service. 1

DoS Denial of Service. 3

ETSI European Telecommunications Standards Institute. 2, 5, 16

FDT Formal Description Technique. 2, 3, 8

IDE Integrated Development Environment. 6

IDL Interface Description Language. 3

IoT Internet of Things. 1, 3, 5

IRC Internet Relay Chat. 6

ISO International Organisation for Standardisation. 1, 2, 5, 16

ITU International Telecommunication Union. 2, 5, 16

JSON JavaScript Object Notation. 5

MQTT Message Queue Telemetry Transport. 1–8, 10–16

MSC Message Sequence Chart. 2

QoS Quality of Service. 7, 8, 13, 14

SDL Specification and Description Language. 2

SIP Session Initiation Protocol. 2

SUT System Under Test. 5, 6, 9–12, 15

TCP Transmission Control Protocol. 6

TLA+ Temporal Logic of Actions plus. 3, 12, 13, 15, 16

TTCN-3 Test and Test Control Notation version 3. 1–3, 5, 8–11, 13, 16

TTN The Things Network. 1

URN User Requirements Notation. 2

XML Extensible Markup Language. 3, 5

Bibliography

[1] K. Angrishi. Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): IoT Botnets. *arXiv preprint arXiv:1702.03681*, 2017.

[2] A. Banks and R. Gupta. MQTT version 3.1.1. *OASIS standard*, 2014. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[3] N. Braspenning, A. van de Mortel-Fronczak, and K. Rooda. Model-based testing with χ and TORX. *XOOTIC Magazine*, 2005.

[4] T. Deiß. TTCN-3 for large systems. In *Systems Validation workshop. Paris*, 2004.

[5] Ericsson. Eclipse Titan MQTT protocol module. http://git.eclipse.org/c/titan/titan.ProtocolModules.MQTT.git/tree/, 2017.

[6] R. Hao and J. Wu. Toward formal ttcn-based test execution. In *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, volume 1, pages 230–235. IEEE, 1997.

[7] D. Heiland. IoT security testing methodology. https://community.rapid7.com/community/infosec/blog/2017/05/10/iot-testing-methodology. Accessed: 2017-06-06.

[8] ISO/IEC. Copyright, standards and the internet. https://www.iso.org/files/live/sites/isoorg/files/archive/pdf/en/copyright_information_brochure.pdf. Accessed: 2017-06-07.

[9] M. Kerkers. Assessing the Security of IEC 60870-5-104 Implementations using Automata Learning. Master's thesis, University of Twente, 2017.

[10] S. Kim, H. Bae, and K. Jun. A formal TTCN-based protocol testing for intelligent network. In *Computers and Communications, 1998. ISCC'98. Proceedings. Third IEEE Symposium on*, pages 205–209. IEEE, 1998.

[11] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] L. Lundgren. Light Weight Protocol! Serious Equipment! Critical Implications! https://media.defcon.org/DEFCON24/DEFCON24presentations/DEFCON-24-Lucas-Lundgren-Light-WeightProtocol-Critical-Implications.pdf, 2016.

[13] K. Mladenov. TTCN-3 code developed for MQTT compliance testing during the research. https://github.com/rockedscientist/ttcn-mqtt, 2017.

[14] P. Narayana, R. Chen, Y. Zhao, Y. Chen, Z. Fu, and H. Zhou. Automatic vulnerability checking of IEEE 802.16 WiMAX protocols through TLA+. In *Secure Network Protocols, 2006. 2nd IEEE Workshop on*, pages 44–49. IEEE, 2006.

[15] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[16] G. Pahnholzer, C. Brandauer, S. Pietsch, and J. Resch. On Investigating the Benefits of TTCN-3-Based Testing in the Context of IEC 61850. In *Internationl Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2015.

[17] R. L. Probert, B. Stepien, and P. Xiong. Formal testing of web content using TTCN-3. In *TTCN-3 User Conference*, volume 2005, 2005.

[18] Deutshce Bahn. Deploying the Internet of Things on Germany's DB Railway System. https://iot.eclipse.org/resources/case-studies/EclipseIoTSuccessStory-DB.pdf, 2016.

[19] B. Schneier. Regulation of the Internet of Things. https://www.schneier.com/blog/archives/2016/11/regulation_of_t.html. Accessed: 2017-06-06.

[20] SECDEV. Scapy. https://github.com/secdev/scapy, 2017.

[21] G. J. Tretmans. *A formal approach to conformance testing.* PhD thesis, Twente University Press, 1992.

[22] J. Tretmans. An overview of OSI conformance testing. *Formal Methods Tools group University of Twente*, 2001.

[23] M. van der Bijl, A. Rensink, and J. Tretmans. Component based testing with ioco. *FATES 2003-Formal Approaches to Testing of Software*, 2931, 2003.

[24] L. Wan and W. Shi. Specifying and checking network protocol based on TLA. In *Anti-Counterfeiting, Security and Identification (ASID), 2012 International Conference on*, pages 1–4. IEEE, 2012.

[25] H. Wayne. Formal Methods in Practice: Using TLA+ at eSpark Learning. https://medium.com/espark-engineering-blog/formal-methods-in-practice-8f20d72bce4f. Accessed: 2017-06-07.