A Formal Model and Analysis of an IoT Protocol

Benjamin Aziz

School of Computing
University of Portsmouth
Portsmouth PO1 3HE
United Kingdom

Abstract

We present a formal model of the MQ Telemetry Transport version 3.1 protocol based on a timed message-passing process algebra. We explain the modelling choices that we made, including pointing out ambiguities in the original protocol specification, and we carry out a static analysis of the formal protocol model, which is based on an approximation of a name-substitution semantics for algebra. The analysis reveals that the protocol behaves correctly as specified against the first two quality of service modes of operation providing at most once and at least once delivery semantics to the subscribers. However, we find that the third and highest quality of service semantics is prone to error and at best ambiguous in certain aspects of its specification. Finally, we suggest an enhancement of this level of QoS for the protocol.

Keywords: Formal Verification, IoT, MQTT

1. Introduction

The Internet of Things (IoT) [1] is a new paradigm with the aim of creating connectivity for "everything" that can carry a minimum of storage and computational power, such that these connected things can collaborate anytime, anywhere and in any form, within applications in various domains such as per-

[1] benjamin.aziz@port.ac.uk
Tel:+442392842265
Fax:+442392842525

sonal and social, transportation, enterprise businesses and service and utility monitoring [2, 3]. Some recent estimates suggest that the number of IoT devices exceeds 30 billion with more than 200 billion intermittent connections [4] generating over 700B Euros in revenue by 2020 [5].

This connectivity of IoT devices has been boosted in recent times with the increasing popularity of mobile communications, such as wireless sensor networks and radio frequency identification technologies, and the proliferation of small hardware with minimum computational and storage capabalities. These coupled with the standardisation efforts of Machine-to-Machine (M2M) communication protocols, such as MQTT [6], XMPP [7] and others, meant that the global vision of the IoT is well within reach of industries and their markets. However, such global applications may require, in contexts where criticality is an issue, a minimum degree of reliability in terms of the correctness of the specification of the system as well as its level of assurance with respect to non-functional properties such as security and privacy. Therefore, there is a need for adopting formal analysis techniques to ensure that specifications are as little ambiguous as possible leading to more reliable and robust applications built out of those specifications.

This paper presents a formal model of the MQTT protocol above based on a timed process algebra, called TPi, and then defines an abstract static analysis that approximates the behaviour of processes by limiting the number of copies of input variables and new names that can be captured in the analysis during communications. The analysis is applied manually to the protocol to attempt to understand how robust the behaviour of the protocol is in the different quality of service scenarios mentioned above. The main contribution of the paper therefore is to formalise the specification of the MQTT protocol and to analysis its semantics. Based on this analysis, the paper also makes recommendations for future improvements of the protocol in order to remove current ambiguity in its specification and enhance its dependability.

1.1. The MQTT Protocol

The MQ Telemetry Transport (MQTT) protocol - version 3.1 [6] is described as a lightweight broker-based publish/subscribe messaging protocol that was designed to allow devices with small processing power and storage, such as those which the IoT is composed of, to communicate over low-bandwidth and unreliable networks. The publish/subscribe message pattern [8], on which MQTT is based, provides for one-to-many message distribution with three variaties of delivery semantics, based on the level of quality of service expected from the protocol.

In the "at most once" case, messages are delivered with the best effort of the underlying communication infrastructure, which is usually IP-based, therefore there is no guarantee that the message will arrive. This protocol, termed the QoS = 0 protocol, is represented by the following flow of messages and actions:

$$\begin{aligned} \textit{Client} \rightarrow \textit{Server} &: \textbf{Publish} \\ \textit{Server Action} &: \textit{Publish message to subscribers} \end{aligned}$$

In the second case of "at least once" semantics, certain mechanisms are incorporated to allow for message duplication, and despite the guarantee of delivering the message, there is no guarantee that duplicates will be supressed. This case is represented by the following flow of messages and actions:

$$\begin{aligned} \textit{Client} \rightarrow \textit{Server} &: \textbf{Publish} \\ \textit{Client Action} &: \textit{Store Message} \\ \textit{Server Actions} &: \textit{Store Message}, \\ &\quad \textit{Publish message to subscribers}, \\ &\quad \textit{Delete Message} \\ \textit{Server} \rightarrow \textit{Client} &: \textbf{Puback} \\ \textit{Client Action} &: \textit{Discard Message} \end{aligned}$$

The second message **Puback** represents an acknowledgement of the receipt of the first message, and if **Puback** is lost, then the first message is retransmitted

by the client (hence the reason why the message is stored at the client). Once the protocol completes, the client discards the message. This protocol is also known as the QoS = 1 protocol.

Finally, for the last case of "exactly once" delivery semantics, also known as the QoS = 2 protocol, the published message is guaranteed to arrive only once at the subscribers. This is represented by the following flow of messages and actions:

Client $\rightarrow$ *Server*	:	**Publish**
Client Action	:	*Store Message*
Server Actions	:	*Store Message OR*
		Store Message ID,
		Publish message to subscribers
Server $\rightarrow$ *Client*	:	**Pubrec**
Client $\rightarrow$ *Server*	:	**Pubrel**
Server Actions	:	*Publish message to subscribers*,
		Delete Message OR
		Delete Message ID
Server $\rightarrow$ *Client*	:	**Pubcomp**
Client Action	:	*Discard Message*

In this protocol, **Pubrec** and **Pubcomp** represent acknowledgement messages from the server, whereas **Pubrel** is an acknowledgement message from the client. The loss of **Pubrec** causes the client to recommence the protocol from its beginning, whereas the loss of **Pubcomp** causes the client to retransmit only the second part of the protocol, which starts at the **Pubrel** message. This additional machinery presumably ensures a single delivery of the published message to the subscribers.

The protocol additionally defines the message structure needed in communications between *client*, i.e. end-devices responsible for generating data from

their domain (the data source) and *servers*, which are the system components responsible for collating source data from clients/end-devices and distributing these data to interested subscribers.

1.2. Paper Structure

The rest of the paper is organised as follows. In Section 2, we describe related work in current literature and in Section 3, we provide an overview of the TPi process algebra, a timed version of the π-calculus [9]. In Section 4, we develop a model of the MQTT protocol based on TPi, and explain the various modelling options that we adopted. In Section 5, we give an analysis of the protocol in the context of its three versions of the delivery semantics. In Section 6, we discuss the outcome of the analysis and make a recommendation towards the enhancement of the MQTT protocol in the case of QoS = 2. Finally, in Section 7, we conclude the paper and provide directions for future research.

2. Related Work

Publish/subscribe is increasingly becoming an important communication paradigm [16], in particular within the domain of sensor device networks and the Internet-of-Things where messages can be communicated with more effeciency and less consumption of the devices' limited computatational power. IBM's MQTT-S protocol [17] was one of the first industrially backed lightweight publish/subscribe protocols that was deployed for wireless sensor and actuator networks. This was followed in year 2010 by version 3.1 [6], which is currently undergoing standardisation by the OASIS community.

There has been very little effort in applying formal analysis tools to IoT communication protocols, mainly due to the novelty of such protocols and their very recent arrival at the scene of communication protocols. On the other hand, some work has been done in the area of publish-subscribe protocols in general. An early attempt in [18] was made to model formally publish/subscribe protocols to capture their essential properties such as minimality and completeness,

however, without any attempt to incorporate hostile environments within which these protocols may run. One aspect of their model is the use of an incrementing global clock $\mathcal{T}$, similar to our concept of the function $\eth(P)$, which is needed in order to model the passing of time.

In [19], the authors define a formal model of publish/subscribe protocols, within the domain of Grid computing, based on Petri-Nets. Their model offers a mechanism for the composition of existing publish/subscribe protocols with model, hence offering a friendly approach for the validation of such protocols. Nonetheless, the focus of their work is mostly on Grid computing scenarios. The work of [20] is an early attempt in discussing security properties and requirements desireable in publish/subscribe protocols, in particular within the domain of Internet-based peer-to-peer systems, where such protocols became popular in their early forms.

Several works, e.g. in [21, 22, 23, 24, 25], have adopted model checking as an automated technique for verifying properties of systems to verify properties related to reliability and correctness within the context of publish-subscribe systems, and various levels of effeciency. [21, 24] define a general frameworks for model-checking publish-subscribe systems, without focusing on specific systems or properties. Their approach is more general than ours, though our approach is more effecient since it handles only one class of pub-sub systems (MQTT) and targets one type of properties. The approach of [22] is closer to our approach in that they adopt a specific system, *thinkteam*, as the target for their analysis. [23] propose a dedicated model checking technique to verify properties of publish/subscribe-based Message Oriented Middleware (MOM) systems.

Another close work to ours is that of [25], where probabilistic model checking is used to capture uncertainties inherent in publish-subscribe systems. However, despite also being stochastic in nature and hence similar to the TPi language we adopt here, a major difference is that they use probabilities rather than time as a means of expressing that some communications *probably may not* take place. On the same note, probabilistic model checking is also used in [26] to analyse quality of predictions in service-oriented architectures.

Within the domain of sensor network protocols, there is more focus of effort on the formal analysis and verification of such protocols. For example, in [27], the authors apply model checking techniques in the verification of a medium access control protocol called LMAC. Similarly, in [28] propose a formal model of flooding and gossiping protocols for analysing their performance probabilisitic properties. More recently, [29] proposed a formal model and analysis of clock-synchronised protocols in sensor networks based on timed automata.

An earlier version of this paper appeared in [30], where a formal analysis of the delivery semantics of the MQTT protocol was presented. The analysis identified the anomalies in the case of QoS = 2, which showed that the "exactly once" delivery was not always guaranteed. The current paper extends the previous work by reporting on the updates the case of QoS = 2 resulted for the MQTT standard in its 3.1.1 version [31], where the QoS = 2 flow of messages was updated to reflect the results of this work and to remove the ambiguity of its QoS = 2 delivery semantics. The current paper, beside covering a more detailed literature review, also provides more detail on the analysis and suggests improvements to the protocol.

3. TPi: A timed Process Algebra

The model of MQTT that we introduce here is based on a process algebra called TPi, originally inspired by [10] and further developed in [11], which is a synchronous message-passing calculus capabale of expressing timed inputs.

3.1. Syntax and Structural Operational Semantics of TPi

The syntax of the language defines processes, $P, Q \in \mathcal{P}$, based on names $x, y \in \mathcal{N}$ as follows:

$$P, Q ::= \overline{x}\langle y\rangle.P \mid \texttt{timer}^t(x(y).P, Q) \mid !P \mid (\nu x)P \mid (P \mid Q) \mid (P + Q) \mid \mathbf{0} \mid A(x)$$

The syntax corresponds to that of the standard synchronous π-calculus [9] except for the fact that input actions are placed within a timer, $\texttt{timer}^t(x(y).P, Q)$, where $t \in \mathbb{N}$ represents a time bound. The input action, $x(y).P$, can synchronise with suitable output actions as long as $t > 0$. Otherwise, when $t = 0$, the timer behaves as Q. There is an assumption that t is decremented by the environment of the process and that t can be any time unit (e.g. tick, second etc.). Finally, we utilised *process definition calls* in the form of $A(x)$. This calls a process definition $A(y) \stackrel{\texttt{def}}{=} P$, and at the same time, passes it the value x to replace y. If the definition A does not accept any input parameters, then we simply omit the input parameter y and write $A() \stackrel{\texttt{def}}{=} P$.

The structural operational semantics of TPi are given in terms of the structural congruence, $\equiv$, and labelled transition, $\stackrel{\mu}{\longrightarrow}$, relations as shown in Figure 1, where $fn(P)$ represents the set of free names of P.

The definition of $\equiv$ is standard, except for rules (6), (7), (8) and (9), which deal with expired and infinite timers, and parameterised/non-parameterised process definition calls, respectively. The labels, $\mu \in \{\stackrel{\overline{x}\langle y\rangle}{\longrightarrow}, \stackrel{\overline{x}(y)}{\longrightarrow}, \stackrel{x(z)}{\longrightarrow}, \stackrel{\tau}{\longrightarrow}\}$, express free and bound outputs, inputs and silent actions. Again, most of the rules for $\stackrel{\mu}{\longrightarrow}$ are straightforward and their explanation can be found elsewhere (e.g. [12, §3.2.2] and [11]) except for rule (19), where a *time-stepping* function, $\eth : \mathcal{P} \to \mathcal{P}$, expresses the ticking of activated timers (i.e. timed inputs that are already at the head of the process waiting to accept a message) by the external environment of the process:

$$\eth(P) = \begin{cases} \texttt{timer}^t(x(y).Q, Q'), & \text{if } P = \texttt{timer}^{t+1}(x(y).Q, Q') \\ & \text{and } 0 < t+1 < \infty \\ \eth(Q) \mid \eth(R), & \text{if } P = Q \mid R \\ \eth(Q) + \eth(R), & \text{if } P = Q + R \\ (\nu x)\eth(Q), & \text{if } P = (\nu x)Q \\ \eth(Q[x/y]), & \text{if } P = A(x) \text{ and } A(y) \stackrel{\texttt{def}}{=} Q \\ \eth(Q), & \text{if } P = A() \text{ and } A() \stackrel{\texttt{def}}{=} Q \\ P, & \text{otherwise} \end{cases}$$

Rules of the $\equiv$ relation:

(1) $(\mathcal{P}/\equiv, |, \mathbf{0})$ is a commutative monoid

(2) $(\nu x)\mathbf{0} \equiv \mathbf{0}$

(3) $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$

(4) $!P \equiv P \mid !P$

(5) $(\nu x)(P \mid Q) \equiv (P \mid (\nu x)Q)$ if $x \notin fn(Q)$

(6) $\texttt{timer}^0(x(z).P, Q) \equiv Q$

(7) $\texttt{timer}^\infty(x(z).P, Q) \equiv x(z).P$

(8) $A(x) \equiv P[x/y]$, where $A(y) \stackrel{\texttt{def}}{=} P$

(9) $A() \equiv P$, where $A() \stackrel{\texttt{def}}{=} P$

Rules of the $\stackrel{\mu}{\longrightarrow}$ relation:

(10) $\overline{x}\langle y\rangle.P \stackrel{\overline{x}\langle y\rangle}{\longrightarrow} P$

(11) $\texttt{timer}^{t+1}(x(z).P, Q) \stackrel{x(z)}{\longrightarrow} P$

(12) $P \stackrel{\overline{x}\langle y\rangle}{\longrightarrow} Q \;\Rightarrow\; (\nu y)P \stackrel{\overline{x}(y)}{\longrightarrow} Q$ if $x \neq y$

(13) $P \stackrel{\overline{x}\langle y\rangle}{\longrightarrow} P', Q \stackrel{x(z)}{\longrightarrow} Q' \;\Rightarrow\; P \mid Q \stackrel{\tau}{\longrightarrow} P' \mid Q'[y/z]$

(14) $P \stackrel{\overline{x}(y)}{\longrightarrow} P', Q \stackrel{x(z)}{\longrightarrow} Q' \;\Rightarrow\; P \mid Q \stackrel{\tau}{\longrightarrow} (\nu y)(P' \mid Q'[y/z])$

(15) $P \stackrel{\mu}{\longrightarrow} Q \;\Rightarrow\; (\nu x)P \stackrel{\mu}{\longrightarrow} (\nu x)Q$ if $x \neq fn(\mu)$

(16) $P \stackrel{\mu}{\longrightarrow} P' \;\Rightarrow\; P \mid Q \stackrel{\mu}{\longrightarrow} P' \mid Q$

(17) $P \stackrel{\mu}{\longrightarrow} P' \;\Rightarrow\; P + Q \stackrel{\mu}{\longrightarrow} P'$

(18) $P \stackrel{\mu}{\longrightarrow} P' \;\Rightarrow\; Q + P \stackrel{\mu}{\longrightarrow} P'$

(19) $P \stackrel{\tau}{\longrightarrow} \eth(P)$

Figure 1: The structural operational semantics of *TPi* [11].

This function reduces the timer's value as long as that value is still a positive number, expressed as $t + 1$. The function distributes over parallel composition, non-deterministic choice, restrictions and process definition calls, but it has no effect over all other processes. This is interesting for the cases of output and replicated processes since these are considered to be non-active unless they are synchronised or replicated. We assume that rule (19) is applied once every unit of time and that its application (including the call to the $\eth$ function) is completed before a single unit of time elapses.

In [11], we defined a non-standard name-substitution semantics for TPi, which when abstracted using an approximation function, was capable of yielding an abstract environment $\phi : \mathcal{N}^\sharp \to \wp(\mathcal{N}^\sharp) \in D^\sharp_\perp$, where $\mathcal{N}^\sharp$ represents the set of abstract names. Unlike $\mathcal{N}$, $\mathcal{N}^\sharp$ is finite and as a result, $\wp(\mathcal{N}^\sharp)$ is also finite. The resulting domain, $D^\sharp_\perp$, guarantees termination for an abstract interpretation computed over it, such as that defined in [11]. Finally, $\perp_{D^\sharp} = \phi_0$ is the empty environment where $\forall x \in \mathcal{N} : \phi_0(x) = \{\}$. We also defined in [11] an abstract interpretation based on $D^\sharp_\perp$, which was shown to be safe with respect to the name-substitution semantics, in similar fashion to previous analyses we defined for different variations of the π-calculus [12, 13, 14]. Later in Section 5, we apply this abstract interpretation to the model of the MQTT protocol introduced in the next Section 4.

4. A Model of MQTTv3.1

We now define a model of the MQTT protocol in TPi as shown in Figure 2, which captures the client/server protocol messages. Although the protocol also describes messages between the server and the subscribers, we only focus on one aspect of these, which is the initial publish message from the server to the subscribers.

The model expresses three protocols, one for each of the three levels of the quality of service specified in [6].

QoS Level 0 Protocol:

$Client(Publish) \mid Server()$, where:

$Client(z) \stackrel{\texttt{def}}{=} \overline{c}\langle z\rangle$

$Server() \stackrel{\texttt{def}}{=} c(x).\overline{pub}\langle x\rangle$

QoS Level 1 Protocol:

$Client(Publish) \mid Server()$, where:

$Client(z) \stackrel{\texttt{def}}{=} \overline{c}\langle z\rangle.\texttt{timer}^{t}(c'(y), Client(Publish_{DUP}))$

$Server() \stackrel{\texttt{def}}{=} !c(x).\overline{pub}\langle x\rangle.\overline{c'}\langle Puback\rangle$

QoS Level 2 Protocol:

$Client(Publish) \mid Server()$, where:

$Client(z) \stackrel{\texttt{def}}{=} \overline{c}\langle z\rangle.\texttt{timer}^{t}(c(y).ClientCont(y), Client(Publish_{DUP}))$

$ClientCont(u) \stackrel{\texttt{def}}{=} \overline{c'}\langle Pubrel_u\rangle.\texttt{timer}^{t'}(c'(w), ClientCont(u))$

$Server() \stackrel{\texttt{def}}{=} !c(l).(ServerLate(l) \ + \ ServerEarly(l))$

$ServerLate(x) \stackrel{\texttt{def}}{=} (\overline{c}\langle Pubrec_x\rangle.c'(v).\overline{pub}\langle x\rangle.\overline{c'}\langle Pubcomp_v\rangle.$
$!(c'(v').\overline{c'}\langle Pubcomp_{v'}\rangle))$

$ServerEarly(x) \stackrel{\texttt{def}}{=} (\overline{pub}\langle x\rangle.\overline{c}\langle Pubrec_x\rangle.c'(q).\overline{c'}\langle Pubcomp_q\rangle.$
$!(c'(q').\overline{c'}\langle Pubcomp_{q'}\rangle))$

Figure 2: A model of MQTTv3.1 in TPi considering the three levels of QoS.

4.1. The Subscribers

Our model of the subscribers is minimal, since we only care about the first step in their behaviour, which is listening to the published messages announced by the server:

$$Subscriber() \stackrel{\texttt{def}}{=} \; !pub(x')$$

This definition does not care about what happens to the message after it has been read by the subscriber on the channel *pub*. The main reason for including the replication, !, is to allow for the possibility of accepting multiple messages from the server. This will allow us later in the analysis to validate the different delivery semantics associated with the MQTTv3.1 protocol. The definition can capture the number of times the subscriber will read a message within a single run of the protocol since each instance of x' spawned under the replication is renamed with the labeling system x'_1, x'_2, etc. (following [15]). The definition also assumes that the subscriber can wait ad infinitum for a message to be published by the server, and if no such message is published, it will do nothing. This is not realistic, but sufficient for our purpose here.

4.2. The Attacker

The attacker in our case has a very primitive role, which is to offer the possibility of consuming the exchanged messages in the protocol. In fact, the attacker is only interested in disrupting messages between the client and the server. Therefore, its definition is to listen continuously on the channels c and c' over which the client and the server communicate:

$$Attacker() \stackrel{\texttt{def}}{=} \; !(c(y') + c'(u'))$$

Similar to the case of the subscribers, the attacker is not in a rush to obtain an input from the protocol, therefore it can wait ad infinitum for a message to be received on its channels c or c'. It is also possible to run a finite attacker model as follows:

$$Attacker() \stackrel{\texttt{def}}{=} \; (c(y'_1) + c'(u'_1)) \mid \ldots \mid (c(y'_n) + c'(u'_n))$$

Where the operator ! is replaced by a finite number n of the input choices all composed in parallel. In this finite model, the attacker is capable of only consuming n messages.

5. Analysis of the Protocol

We now define formally the three message-delivery semantics associated with the MQTT protocol, *at most once*, *at least once* and *exactly once* delivery, and we discuss the results of analysing the protocol in light of these three semantics.

5.1. QoS = 0 Protocol

The model of QoS = 0 protocol is straightforward. The client process is called from the top level protocol with the *Publish* message. This process is then run in parallel with the server process, which upon receiving the *Publish* message, it publishes it on the *pub* channel where interested subscribers are listening. For simplicity, we assume that the message is published as is. However, a more refined (but not of interest to us) server process would be expected to extract the relevant payload from *Publish* before publishing the actual data. We formalise the semantics of the protocol for the case of QoS = 0 in terms of the following theorem.

Theorem 1 (Delivery Semantics For QoS = 0). *The MQTTv3.1 protocol for the case of QoS = 0 has a delivery semantics of the publish message to the subscribers of "at most once".*

Proof. Given the definition of the subscribers' process in the previous section, a run of this protocol would be equivalent to the following in the absence of any attackers: ($Client(Publish) \mid Server() \mid Subscriber()$). Analysing the process renders the following value of Φ:

$$\phi = \{x \mapsto \{Publish\}, x'_1 \mapsto \{Publish\}\}$$

From this, we can see that the message arrives at the subscriber. However, if we re-run the analysis with the attacker process activated: ($Client(Publish) \mid$

Server() | *Subscriber*() | *Attacker*()), we obtain the following interesting outcome:

$$\phi_{atk} = \{y'_1 \mapsto \{Publish\}\}$$

This case shows a run of the protocol, which leads to only y' being instantiated with *Publish*. There is no instantiation of the x or x' variables.

From these results, it is easy to see that there are two possible outcomes. The first value of ϕ represents a normal run where $x' \mapsto \{Publish\}$, whereas in the second value of ϕ_{atk}, we have that $x' \mapsto \{\}$ by the definition of the default state ϕ_0. Hence, it is straightforward to see that the protocol *may* deliver the published message to the subscribers, and therefore, it correctly exhibits the at most once delivery semantics. □ In this case, we do not gain more information about the protocol if we were analysing for the case of $k > 1$ since the definition of the protocol in Figure 2 does not contain replicatin, which prevents it from being able to interact with an attacker that generates more messages (i.e. a spammy rather than a lossy attacker). Another noteworthy point is related to modelling the attacker as one that would emit at least one message. In the case of QoS=0, this would have the same effect as the no-attacker case above.

5.2. QoS = 1 Protocol

The QoS = 1 protocol has a semantics of "at least once" delivery. We model this in Figure 2 as a client process, which starts by sending a *Publish* message to the server. The server is capable of inputting this message, publishing it to the subscribers and then replying back to the client with the *Puback* message. Again, for simplicity, we abstract away from the structure of both *Publish* and *Puback*, and point out here that a more refined treatment of these messages (i.e. extracting their payload) does not affect our analysis in the paper.

The next part is the main difference from the QoS = 0 case above. The client will wait for a finite amount of time, t, on its input channel c' for the *Puback* message from the server. If this message delays (for any communication failing reason), the client will choose to re-call its process with a new $Publish_{DUP}$

message. The difference between $Publish_{DUP}$ and $Publish$ is that the DUP bit is set in the former to indicate that it is a duplication of the latter. The server on its side is capable of receiving this new publish message since its behaviour is replicated, which means that it can restart its process any number of times required by the context.

The two channels, c and c', distinguish between the two parts of the protocol ($Publish$ and $Puback$ parts). This is not necessary in practice, however it renders our model much simpler by avoiding unnecessary interferences between these two parts. In practice, there would be some message validation mechanisms to prevent such interferences occuring.

We formalise the delivery semantics for this protocol in terms of the following theorem.

Theorem 2 (Delivery Semantics For QoS = 1). *The MQTTv3.1 protocol for the case of QoS = 1 has a delivery semantics of the publish message to the subscribers of "at least once".*

Proof. Again, we first analyse the protocol under no attackers. In this case, we find the following subset value for ϕ:

$$\phi = \{x_1 \mapsto \{Publish\}, y \mapsto \{Puback\}, x_1' \mapsto \{Publish\}\}$$

This implies normal behaviour, where the published message eventually arrives at the subscriber only once. We now re-run the analysis with the attacker activated, which produces the following subset value of ϕ_{atk}, where k is set to 3 to allow three runs of the protocol to take place:

$$\begin{aligned}
&\phi_{atk} = \{x_1 \mapsto \{Publish\}, u_1' \mapsto \{Puback\}, x_1' \mapsto \{Publish\},\\
&x_2 \mapsto \{Publish_{DUP}\}, u_2' \mapsto \{Puback\},\\
&x_2' \mapsto \{Publish_{DUP}\}, x_3 \mapsto \{Publish_{DUP}\},\\
&u_3' \mapsto \{Puback\}, x_3' \mapsto \{Publish_{DUP}\}\}
\end{aligned}$$

The first subset of name substitutions corresponds to the first run where the attacker interferes with the protocol by consuming the $Puback$ message. In the

next two subsets, the client will issue a duplicate $Publish_{DUP}$. In both of these subsets, the attacker continues to consume the acknowledgement message and the client will continue to restart the protocol. Examining these results, we can easily see that the subscribers' input x' has more than one instantiation of the message *Publish*, including when the DUP bit is set. This indicates that the message may arrive more than once at the subscriber. □

In the above case of QoS=1, it was suffecient to go up to 3 copies to prove that duplicates will happen. Setting $k > 3$ also generates more copies and therefore more duplicates. This can be shown as a result of the safety of the analysis, as was proven in [12, 13, 14, 15]. Furthermore, it is worth mentioning that for the case of QoS=1, we never obtain a result where the number of published messages is zero (under either condition of no attacker/attacker).

Again here, it is worth noting that modelling an attacker as one that would emit at least one message would always push the number of k by one (i.e. $k = 4$), since the effect of such a behaviour is equivalent to the no-attacker case, albeit that only one message is allowed through.

5.3. QoS = 2 Protocol

The last protocol represents the highest quality of service level, indicated by the QoS bit setting of 2. The model of Figure 2 contains again the definitions of the client and the publishing server. Similar to (and for the same reasons above) for the case of QoS = 1, we use two channels for the client: c for the first part ending with the sending of *Pubrec* and c' for the second part ending with the seding of *Pubcomp*.

The client process has two parts. The first could be re-iterated, which will result in the *Publish* message being re-sent with the DUP bit set in case the *Pubrec* message is not received from the server within a time bound of t units. Note here that the standard protocol of [6] is not clear regarding the resent message. There is no explicit mentioning that the resent publish message is considered different from the original one. The assumption we make is that since DUP is set, then the resent message is a "duplicate" of the original one

and therefore it is the same message.

The second part of the client process, *ClientCont*, is instantiated by the first part only if *Pubrec* is received from the server within the time bound t. In this case, it will send a *Pubrel* message to the server parameterised by the same message id as received in the previous message (hence we write $Pubrel_u$). After this, it waits for an amount of time t' for the last message from the server, *Pubcomp*, at which point it terminates once this message is received. If this last message does not arrive within the time bound t', it will re-call itself (i.e. the *ClientCont* part), which will result in the re-commencement of the protocol from the point of the sending of the $Pubrel_u$ message. We believe the above two timed input actions model adequately the requirement "*If a failure is detected, or after a defined time period, the protocol flow is retried from the last unacknowledged protocol message; either the PUBLISH or PUBREL.*" [6, pp. 38].

Finally, the last part of the protocol represents the server process. This process after receiving the initial publish message splits into a choice of two processes, *ServerEarly* and *ServerLate*. The main difference between these is whether the publish message is published to the subscribers before (i.e. early) or after (i.e. late) sending the second message of the protocol $Pubrec_x$, which is parameterised by the message id received in the first message from the client.

The standard provides two alternative options for this case [6, pp. 38]. The first follows the sequence of actions (store message, publish message and delete message), whereas the second follows the sequence of actions (store message id, publish message and delete message id). We term the former a *late publish semantics* and the latter an *early publish semantics*. The standard's document states that "The choice of semantic is implementation specific and does not affect the guarantees of a QoS level 2 flow" [6, pp. 38], however, we demonstrate next in terms of the output of our static analysis later that this is not generally true.

The whole server process is replicated in order to be able to receive a repeat publish message from the client in the event that $Pubrec_x$ is not received at the client within the time limit.

The server process, after sending $Pubrec_x$, goes into the second part of the

protocol. In this part, it listens on $c'(v)$ or $c'(q)$ for the incoming *Pubrel* message from the client. It then continues depending on the choice made earlier to either publish the message and send $Pubcomp_v$ or just send $Pubcomp_q$. In both cases, the *Pubcomp* message is parameterised by the message id from the received *Pubrel* message from the client.

The final part now commences, which is a replicated process that again listens for the *Pubrel* message from the client, and once this is received, it sends another *Pubcomp* message back to the client. This last part of the server process is similar in both sides of the choice and it will replicate itself until the client receives successfully the *Pubcomp* message, at which point the client will cease re-sending *Pubrel* messages.

It is worth noting here that our model above assumes that the implementation of the server will cater for a non-deterministic choice of both the early and late publish semantics. However, it is also possible, as we shall see in the next section, to model and analyse the server assuming only one of the two semantics of message publishing is implemented. This would be equivalent to modelling the server process as either $!c(l).ServerLate(l)$ or $!c(l).ServerEarly(l)$.

We now capture the delivery semantics for this protocol in terms of the following property.

Property 1 (Delivery Semantics For QoS = 2). *The MQTTv3.1 protocol for the case of QoS = 2 has a delivery semantics of the publish message to the subscribers of exactly once.* □

In the first analysis we run, the attacker is deactivated. We obtain the following subset value for Φ when $k = 1$:

$\phi = \{x_1 \mapsto \{Publish\}, y \mapsto \{Pubrec_x\}, v_1 \mapsto \{Pubrel_u\}, x_1' \mapsto \{Publish\}, w \mapsto \{Pubcomp_v\}, x_1 \mapsto \{Publish\}, x_1' \mapsto \{Publish\}, y \mapsto \{Pubrec_x\}, q_1 \mapsto \{Pubrel_u\}, w \mapsto \{Pubcomp_q\}, \ldots\}$

The substitutions correspond to normal runs of the protocol for the two choices

of the late and early publish semantics. Now, let's examine some of the results of the static analysis when the attacker is *activated*. In particular, we consider the case of the early publish semantics where we analyse in the context of the server $!c(l).\mathit{ServerEarly}(l)$ and the simple attacker model $(c(y') + c'(u'))$. We obtain the following interesting subset of the results, with $k = 2$:

$$\phi_{atk1} = \{x_1 \mapsto \{\mathit{Publish}\}, x'_1 \mapsto \{\mathit{Publish}\}, y'_1 \mapsto \{\mathit{Pubrec}_x\}, x_2 \mapsto \{\mathit{Publish}_{DUP}\},$$
$$x'_2 \mapsto \{\mathit{Publish}_{DUP}\}, y \mapsto \{\mathit{Pubrec}_x\}, q_1 \mapsto \{\mathit{Pubrel}_u\}, w \mapsto \{\mathit{Pubcomp}_q\}, \ldots\}$$

The result is interesting, as it represents a single interference case by the attacker (since $k = 2$). The attacker manages to consume the *Pubrec* message ($y'_1 \mapsto \{\mathit{Pubrec}_x\}$) before the client does so. As a result, the first part of the protocol is repeated and hence, in addition to the initial publish message ($x'_1 \mapsto \{\mathit{Publish}\}$), this leads to a second instance of this message to be annouced to the subscribers ($x'_2 \mapsto \{\mathit{Publish}_{DUP}$).

Next, we re-apply the analysis on the case of the full server model and the simple non-replicated attacker model, where again we set $k = 2$ for simplicity:

$$\phi_{atk2} = \{x_1 \mapsto \{\mathit{Publish}\}, x'_1 \mapsto \{\mathit{Publish}\}, y'_1 \mapsto \{\mathit{Pubrec}_x\}, x_2 \mapsto \{\mathit{Publish}_{DUP}\},$$
$$y \mapsto \{\mathit{Pubrec}_x\}, v_1 \mapsto \{\mathit{Pubrel}_u\}, x'_2 \mapsto \{\mathit{Publish}_{DUP}\}, w \mapsto \{\mathit{Pubcomp}_v\}, \ldots\}$$

This represents another case of the attacker interfering with the protocol, however unlike the case of the first attack, a different choice of the publish semantics is made here in terms of the re-transmission of first part of the protocol. Here, we find that the Pubrec_x acknowledgement message sent by the server is captured by the attacker after an early publish semantics choice is taken involving announcing the publish message to the subscribers (by means of $x'_1 \mapsto \{\mathit{Publish}\}$). This failure in delivering Pubrec_x to the client causes a restart of the protocol, however, in this case a different choice is made with the late publish semantics. Continuing with this run, the second part of the protocol causes the duplicated publish message $\mathit{Publish}_{DUP}$ to be announced again to the subscribers. Note

that this attack would not be possible if either *ServerLate* or *ServerEarly* process only is adopted, but not a choice of both.

6. Discussion

Considering the results of the above three analyses, we find that the protocol as described in [6] and specified in Section 4 is quite sound in the cases of QoS = 0 and QoS = 1, however in the case of QoS = 2, the protocol description in [6] is ambiguous in terms of how the server is supposed to deal with the publication of duplicated messages and whether it may or may not implement both choices of early and late publish semantics.

More specifically, considering the two results of ϕ_{atk1} and ϕ_{atk2} for the case of QoS = 2, we conclude that there is more than one scenario where the protocol fails in adhering to its "exactly once" delivery semantics and where $Publish/Publish_{DUP}$ are delivered to the subscribers more than once. As a result, Property 1 defined in the previous section does not hold. It is also noteworthy that this failure occurs only due to more-than-once delivery reasons, and there is no case where the failure occurs due to less-than-once (i.e. zero) delivery, unless we assume a powerful attacker with replicated inputs whose capable of continuously blocking the *Pubrec* message from being delivered to the client.

We propose here the following two enhancements to the MQTT protocol in the case of QoS = 2. First, separate the implementation of the early and late publish semantics. This is currently not explicit in the specification of the protocol in [6]. We suggest that this be made explicit, if this is the intention, so that implementations of the protocol are not confused with the choice.

The second enhancement is to introduce a conditional guard on the publishing action at the server side, for both choices of early and late publish semantics.

This can be described as the following udpated QoS = 2 protocol:

Client $\rightarrow$ *Server*	:	**Publish**
Client Action	:	*Store Message*
Server Actions	:	*Store Message OR*
		Store Message ID,
		([$\neg$*Published(Message ID)*]*Publish message to subscribers*)
Server $\rightarrow$ *Client*	:	**Pubrec**
Client $\rightarrow$ *Server*	:	**Pubrel**
Server Actions	:	([$\neg$*Published(Message)*]*Publish message to subscribers*),
		Delete Message
		OR Delete Message ID
Server $\rightarrow$ *Client*	:	**Pubcomp**
Client Action	:	*Discard Message*

Where *Published(Message)* is a predicate on the local state of the server in which the message or its id is stored. This predicate checks whether a message has already been published. We also overload the predicate in the case of *Published(Message ID)* to check whether a message corresponding to an ID has been already published or not. If this predicate is not true, the publish action is neglected and the next action in line is applied.

In order to incorporate the predicate *Published(Message)* in our TPi-based specification, we need to modify the syntax of the language to include a new term $P \triangleleft p \triangleright Q$, where p is any local predicate, which is True, the process will evaluate as P and if False, it will evaluate as Q. The structural operational semantics of this can more formally be expressed by adding two additional rules to the $\equiv$ relation in Figure 1 as follows:

(10) $P \triangleleft p \triangleright Q \equiv P$ *if* $p =$ *True*

(11) $P \triangleleft p \triangleright Q \equiv Q$ *if* $p =$ *False*

This will then lead to the redefinition of the early and late publish semantics specification to include the additional expected conditional predicate, $published(x)$, as shown in Figure 3.

$$
\begin{aligned}
&Client(z) \stackrel{\mathtt{def}}{=} \overline{c}\langle z\rangle.\mathtt{timer}^{t}(c(y).ClientCont(y), Client(Publish_{DUP}))\\
&ClientCont(u) \stackrel{\mathtt{def}}{=} \overline{c'}\langle Pubrel_u\rangle.\mathtt{timer}^{t'}(c'(w), ClientCont(u))\\
&Server() \stackrel{\mathtt{def}}{=} !c(l).(ServerLate(l) \ + \ ServerEarly(l))\\
&ServerLate(x) \stackrel{\mathtt{def}}{=} \overline{c}\langle Pubrec_x\rangle.c'(v).\\
&\qquad (\overline{pub}\langle x\rangle.\overline{c'}\langle Pubcomp_v\rangle.!(c'(v').\overline{c'}\langle Pubcomp_{v'}\rangle)) \lhd (\neg\ published(x)) \rhd\\
&\qquad (\overline{c'}\langle Pubcomp_v\rangle.!(c'(v').\overline{c'}\langle Pubcomp_{v'}\rangle))\\
&ServerEarly(x) \stackrel{\mathtt{def}}{=}\\
&\qquad \overline{pub}\langle x\rangle.\overline{c}\langle Pubrec_x\rangle.c'(q).\overline{c'}\langle Pubcomp_q\rangle.!(c'(q').\overline{c'}\langle Pubcomp_{q'}\rangle)\\
&\qquad \lhd(\neg\ published(x)) \rhd \overline{c}\langle Pubrec_x\rangle.c'(q).\overline{c'}\langle Pubcomp_q\rangle.\\
&\qquad !(c'(q').\overline{c'}\langle Pubcomp_{q'}\rangle)
\end{aligned}
$$

Figure 3: The modified model for the case of QoS = 2.

We assume that the predicate is able to consider the contents of the parameter x containing the message and its id. Repeating the analysis for the case of the presence of the attacker and for the early and late publish semantics renders the following two subsets of the final ϕ environment:

$\phi_{atk1} = \{x_1 \mapsto \{Publish\}, x'_1 \mapsto \{Publish\}, y'_1 \mapsto \{Pubrec_x\}, x_2 \mapsto \{Publish_{DUP}\},$
$y \mapsto \{Pubrec_x\}, q_1 \mapsto \{Pubrel_u\}, w \mapsto \{Pubcomp_q\}, \ldots\}$

$\phi_{atk2} = \{x_1 \mapsto \{Publish\}, x'_1 \mapsto \{Publish\}, y'_1 \mapsto \{Pubrec_x\}, x_2 \mapsto \{Publish_{DUP}\},$
$y \mapsto \{Pubrec_x\}, v_1 \mapsto \{Pubrel_u\}, w \mapsto \{Pubcomp_v\}, \ldots\}$

Both of which clearly show no instantiation of the second and further copies of the x' input variable for the subscriber process, in which case we assume that the modification of Figure 3 achieves its intended aim.

7. Conclusion and Future Work

We have modelled and analysed in this paper the MQ Telemetry Transport version 3.1 protocol, which is a lightweight broker-based publish subscribe protocol that is used in communications with small devices that exhibit limited computational and storage power.

We found that the first two QoS modes of operation in the protocol are clearly specified and their message delivery semantics to subscribers can be easily verified to hold. However, according to the results of the analysis, the last case of an "exactly once" delivery semantics has potential vulnerabilities where a simple attacker model that adheres to the specified threat model of the protocol can cause the semantics to be undermined. At best, this semantics is vaguely specified in the standard [6], particularly in relation to issues to do with the choice of server-side behaviour.

Future research will be focused on studying the properties of the protocol under more aggressive attacker models and we plan to propose refined versions of the protocol, including the use of lightweight cryptography in scenarios where authentication of the small devices is required. In addition, although we carried out a simple modification to the QoS = 2 case that removes the duplicated publish message vulnerability, we would like to further investigate in-depth additional mechanisms for improving further the protocol. This would call for more automated approaches, namely using any of a number of automated verification tools that exist in literature, e.g. [32, 33, 34, 35, 36, 37]

References

[1] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions, Future Gener. Comput. Syst. 29 (7) (2013) 1645–1660.

[2] L. Atzori, A. Iera, G. Morabito, The Internet of Things: A Survey, Comput. Netw. 54 (15) (2010) 2787–2805.

[3] D. Bandyopadhyay, J. Sen, Internet of Things: Applications and Challenges in Technology and Standardization, Wireless Personal Communications 58 (1) (2011) 49–69.

[4] O. Vermesan, P. Friess, Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems, The River Publishers, 2013.

[5] O. Mazhelis, H. Warma, S. Leminen, P. Ahokangas, P. Pussinen, M. Rajahonka, R. Siuruainen, H. Okkonen, A. Shveykovskiy, J. Myllykoski, Internet-of-Things Market, Value Networks, and Business Models: State of the Art Report, Tech. Rep. TR-39 (2013).

[6] D. Locke, MQ Telemetry Transport (MQTT) V3.1 Protocol Specification (2010).

[7] P. Saint-Andre, K. Smith, R. Tronon, XMPP: The Definitive Guide Building Real-Time Applications with Jabber Technologies, O'Reilly Media, Inc., 2009.

[8] K. Birman, T. Joseph, Exploiting Virtual Synchrony in Distributed Systems, SIGOPS Oper. Syst. Rev. 21 (5) (1987) 123–138.

[9] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes, Information and Computation 100(1) (1992) 1–77.

[10] M. Berger, K. Honda, The Two-Phase Commitment Protocol in an Extended Pi-Calculus, Electronic Notes in Theoretical Comp. Science 39 (1).

[11] B. Aziz and G. Hamilton, Detecting Man-in-the-Middle Attacks by Precise Timing, in: Proceedings of the 2009 Third International Conference on Emerging Security Information, Systems and Technologies, SECURWARE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 81–86.

[12] B. Aziz, A static analysis framework for security properties in mobile and cryptographic systems, Ph.D. thesis, School of Computing, Dublin City University, Dublin, Ireland (2003).

[13] B. Aziz, G. Hamilton, D. Gray, A static analysis of cryptographic processes: The denotational approach, Journal of Logic and Algebraic Programming 64(2) (2005) 285–320.

[14] B. Aziz, G. Hamilton, The modelling and analysis of pki-based systems using process calculi, International Journal of Foundations of Computer Science 18(3) (2007) 593–618.

[15] B. Aziz, G. Hamilton, A Privacy Analysis for the π-calculus: The Denotational Approach, in: Proceedings of the 2$^{\text{nd}}$ Workshop on the Specification, Analysis and Validation for Emerging Technologies, no. 94 in Datalogiske Skrifter, Roskilde University, Copenhagen, Denmark, 2002.

[16] A. J. Stanford-Clark, G. R. Wightwick, The Application of Publish/Subscribe Messaging to Environmental, Monitoring, and Control Systems, IBM J. Res. Dev. 54 (4) (2010) 396–402.

[17] U. Hunkeler, H. L. Truong, A. Stanford-Clark, MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks, in: Proceedings of the Third International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2008), IEEE, 2008, pp. 791–798.

[18] R. Baldoni, M. Contenti, S. T. Piergiovanni, A. Virgillito, Modelling Publish/Subscribe Communication Systems: Towards a Formal Approach, in: 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), IEEE Computer Society, 2003, pp. 304–311.

[19] L. Abidi, C. Cerin, S. Evangelista, A Petri-Net Model for the Publish-Subscribe Paradigm and Its Application for the Verification of the BonjourGrid Middleware, in: Proceedings of the 2011 IEEE International Conference on Services Computing, SCC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 496–503.

[20] C. Wang, A. Carzaniga, D. Evans, A. Wolf, Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems, in: Proceedings

of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9 - Volume 9, HICSS '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 303–.

[21] D. Garlan, S. Khersonsky, J. S. Kim, Model checking publish-subscribe systems, in: Proceedings of the 10th International Conference on Model Checking Software, SPIN'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 166–180.

[22] M. H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, M. Sebastianis, Model checking publish/subscribe notification for thinkteam ®, Electron. Notes Theor. Comput. Sci. 133 (2005) 275–294.

[23] Y. Jia, E. L. Bodanese, C. I. Phillips, J. Bigham, R. Tao, Improved reliability of large scale publish/subscribe based moms using model checking, in: 2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014, IEEE, 2014, pp. 1–8.

[24] L. Baresi, C. Ghezzi, L. Mottola, On accurate automatic verification of publish-subscribe architectures, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 199–208.

[25] F. He, L. Baresi, C. Ghezzi, P. Spoletini, Formal analysis of publish-subscribe systems by probabilistic timed automata, in: Formal Techniques for Networked and Distributed Systems - FORTE 2007, 27th IFIP WG 6.1 International Conference, Tallinn, Estonia, June 27-29, 2007, Proceedings, Vol. 4574, Springer, 2007, pp. 247–262.

[26] S. Gallotti, C. Ghezzi, R. Mirandola, G. Tamburrelli, Quality prediction of service compositions through probabilistic model checking, in: Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures, QoSA '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 119–134.

[27] A. Fehnker, L. V. Hoesel, A. Mader, Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks, in: Proceedings of the 6th International Conference on Integrated Formal Methods, IFM'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 253–272.

[28] A. Fehnker, P. Gao, Formal Verification and Simulation for Performance Analysis for Probabilistic Broadcast Protocols, in: Proceedings of the 5th International Conference on Ad-Hoc, Mobile, and Wireless Networks, ADHOC-NOW'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 128–141.

[29] F. Heidarian, J. Schmaltz, F. W. Vaandrager, Analysis of a clock synchronization protocol for wireless sensor networks, Theor. Comput. Sci. 413 (1) (2012) 87–105.

[30] B. Aziz, A formal model and analysis of the mq telemetry transport protocol, in: 9th International Conference on Availability, Reliability and Security (ARES 2014), Fribourg, Switzerland, IEEE, 2014.

[31] A. Banks, R. Gupta, MQ Telemetry Transport (MQTT) V3.1.1 Protocol Specification: Committee Specification Draft 02 / Public Review Draft 02 (2014).

[32] ProVerif: Cryptographic protocol verifier in the formal model, http://prosecco.gforge.inria.fr/personal/bblanche/proverif/, accessed: 24-09-2014.

[33] D. A. Basin, S. Mödersheim, L. Viganò, OFMC: A symbolic model checker for security protocols, Int. J. Inf. Sec. 4 (3) (2005) 181–208.

[34] The Tamarin prover for security protocol analysis, https://hackage.haskell.org/package/tamarin-prover, accessed: 24-09-2014.

[35] Maude-NPA, http://maude.cs.uiuc.edu/tools/Maude-NPA/, accessed: 24-09-2014.

[36] Casper: A Compiler for the Analysis of Security Protocols, http://www.cs.ox.ac.uk/gavin.lowe/Security/Casper/, accessed: 24-09-2014.

[37] The Scyther Tool, http://www.cs.ox.ac.uk/people/cas.cremers/scyther/index.html, accessed: 24-09-2014.