Rigorous Security Analysis of RabbitMQ Broker with Concurrent Stochastic Games

Abstract

Modern Internet of Things architectures encompass various computational logic and communication protocols. However, the security issues inherent in these systems pose significant risks, especially at the device edges. Addressing security threats on the deployed communication edges is imperative to ensure a robust and secure communication system. In this study, we propose an approach that utilizes the Concurrent Stochastic Game model (CSG) to specify the behavior of RabbitMQ Broker in the context of IoT systems precisely while considering potential data corruption attacks. For parametrizable evaluation, these attacks and their frequencies are learned. We implement the CSG model in PRISM games for automated analysis, leveraging reward Probabilistic Alternating Temporal Logic (rPATL) to model security requirements as game goals. Empirical validation of the work is presented via an industrial case study that shows how data corruption attacks can impact the sensed data in water dam infrastructure. This assessment gives valuable insights into the RabbitMQ deployment at the edge.

Keywords: Edge computing, Game Model, Security Threats, Formal methods.

Contents

1. Introduction

Edge computing is a paradigm that decentralizes computational power, bringing it closer to the point of data generation, such as IoT devices (sensors and actuators). Unlike relying on a centralized cloud infrastructure, edge computing focuses on processing and analyzing data at or near the network edge, where it originates [1]. However, security concerns have become significant in modern IoT systems [2]. Insecure network connections pose risks such as data interception, unauthorized access, and tampering [3]. Hence, it is of utmost importance to identify and address these security concerns during the early stages of development [4, 5]. This underscores establishing precise semantic definitions for threats and attacks at the design phase.

To develop new and innovative Edge Servers with specific functionalities, enterprise companies rely on (mainly open source) "middleware" software [6], which needs to be enhanced with custom software components to implement multiple functions, including security, that add value compared to other edge servers on the market. Notably, middlewares have been tailored to process signals efficiently and route them from IoT sensors to nodes employing application-level protocols like Hypertext Transfer Protocol Secure (HTTPS) [7]. Beyond that, the literature shows recent advancements in developing appropriate protocols for edge core-side infrastructures (e.g., AMQP[1]) facilitating device-edge communication (e.g., CoAP [8]).

Among the wide range of brokers available, RabbitMQ[2] stands out as a message queue middleware that facilitates asynchronous message communication. It leverages the capabilities of the Erlang language [9] to implement the Advanced Message Queuing Protocol (AMQP) adopted by reputable vendors such as JPMorgan, NASA, Red Hat, Google, IBM, VMware, Mozilla, and others. In this paper, we undertake a comprehensive analysis of the security of RabbitMQ architecture using formal methods. We employ the model checker PRISM-games to formalize and verify RabbitMQ as a Concurrent Stochastic Game (CSG)[10], leveraging reward Probabilistic Alternating Temporal Logic (rPATL) [11] to model security properties as game goals. To validate our work, we explore a set of potential attacks at two distinct levels of the protocol: *messages received* and *queues*.

1.1. Contributions

To the best of our knowledge, RabbitMQ has not been examined for potential attacks within the context of a CSG formalism. In the available literature, the research conducted by [12] primarily centers around examining functional properties employing UPPAAL [13]. Contrarily, studies conducted by [14, 15, 16, 17] primarily utilize simulation techniques to assess performance aspects pertaining to the deployed servers, encompassing scalability, memory usage, and throughput. Nevertheless, these studies do not incorporate the modeling of attacks within the system. The paper under consideration contributes significantly to the field by addressing this gap, with the main contributions summarized as follows:

1. Formalizing RabbitMQ to comprehensively understand its communication broker behavior using operational semantics rules.
2. Formalizing the CAPEC-384 attack scenario using operational semantics rules.
3. Developing an interpretation of the formal representation of RabbitMQ under attacks in the formalism of PRISM-games.
4. Examining attacks in a use case scenario that addresses the deployment of RabbitMQ on a communication gateway.

By employing formal methods for studying attacks within the system, we provide valuable insights that enhance the understanding of the system's security aspects. The paper uses a set of abbreviations and references in Table 1.

[1] https://www.amqp.org/
[2] https://www.rabbitmq.com/

ATL	Alternating Temporal Logic
ARP	Address Resolution Protocol
CAPEC	Common Attack Pattern Enumeration and Classification
DDoS	Distributed Denial of Service
CTL	Computation Tree Logic
CSG	Concurrent Stochastic Game
ILP	Integer Linear Programming
MC	Model Checking
MDP	Markov Decision Process
MITM	Man in the Middle
PCTL	Probabilistic Computation Tree Logic
rPATL	reward Probabilistic Alternating Temporal Logic
SMC	Statistical Model Checking
WL	Water Level
WV	Water Volume
RP	Rain Precipitation

Table 1: A List of Acronyms Used in the Article.

1.2. Outline

The subsequent sections of this paper are organized as follows: In Section 2, we provide preliminaries encompassing the Concurrent Stochastic Game (CSG) and PRISM-games language. Following that, in Section 3, we present the workflow of the study. Section 4 presents the flow for learning attack frequencies. In Section 5, we employ CSG to model RabbitMQ and address security concerns. In Section 6, we conduct an experiment within the context of IoT systems for the practical application of our approach. In Section 7, an extensive discussion is presented, covering potential enhancements in scalability and outlining strategies for mitigating attacks. Section 8 summarizes the related works, highlighting their contributions and limitations. Lastly, in Section 9, we conclude our work by outlining the key findings and proposing directions for future research.

2. Background

This section covers the essential concepts related to Concurrent Stochastic Multi-player Games and their implementation in PRISM games. Furthermore, We will revisit the AMQP protocol stack, which is essential for understanding message routing.

2.1. Concurrent Stochastic Games

Concurrent stochastic games (CSGs) [10, 18] consider that players make choices concurrently in each state and then transition simultaneously. In CSGs, players have control over one or more modules, and the actions associated with these modules can only be utilized by the respective player who owns them. The CSG is defined as an extension of Probabilistic Automata (PA) [19] in [10, 18] as follows:

Definition 1. A concurrent stochastic multi-player game (CSG) is a tuple $G = \langle s_0, S, N, A, \Delta, \delta, AP, L \rangle$:

- s_0 is an initial state, such that $s_0 \in S$,
- S is a set of states,
- $N = \{1, ..., n\}$ is a finite set of players,
- $A = \Sigma_1 \times \ldots \times \Sigma_n$ where Σ_i is a finite set of actions available to player $i \in N$,
- $\Delta : S \longrightarrow 2^{\cup_{i=0}^{n} \Sigma_i}$ is an action assignment function,

- $\delta : S \times A \longrightarrow Dist(S)$ is a probabilistic transition function assigning for each $s \in S$ and $(\alpha_0, \ldots, \alpha_n) \in \Sigma_i$ a probabilistic distribution $\mu \in Dist(S)$, and $L : S \longrightarrow 2^{AP}$ is a labeling function that assigns each state $s \in S$ to a set of atomic propositions taken from the set of atomic propositions (AP).

When in state s, each player $i \in N$ selects an action from its available actions $Action_i(s) \stackrel{\text{def}}{=} \Delta(s) \cap \Sigma_i$ if this set is non-empty. A path π of a CSG G [10, 18] is a sequence $\pi = s_0 \xrightarrow{\alpha_i} s_1$ where $s_i \in S$, $\alpha_i = (a_j^1, ..., a_j^n) \in A$, $\alpha_i^j \in Action_i(s_i)$ for $i \in N$ and $\delta(s_j, \alpha_j)(s_{j+1}) > 0$ for all $j > 0$.

CSGs are augmented with reward structures [10] as $r_A : S \times A \longrightarrow \mathbb{R}$ is an action reward function that assigns a real value to each pair of state and action tuples, which accumulates when the action tuple is selected in the corresponding state and $r_s : S \longrightarrow \mathbb{R}$ is a state reward function assigns a real value to each state, which accumulates when the state is reached.

The properties related to CSGs are expressed in the temporal logic rPATL [11] (reward Probabilistic Alternating Temporal Logic). The property grammar is based on CTL [20] extended with coalition operator $\langle\langle \mathtt{C} \rangle\rangle$ of ATL [21] and probabilistic operator $\mathtt{P}$ of PCTL [22]. For instance, for the following property expressed in natural language: "*Players 1 and 2 have a strategy to ensure that the probability of shutdown occurring within 100 steps is less than 0.001, regardless of the strategies of other players*" is expressed in rPATL as: $\langle\langle 1, 2 \rangle\rangle \mathtt{P}_{<0.001}[\mathtt{F}^{\leq 100} \mathtt{shutdown}]$. Here, " `shutdown` " is the label that refers to the system states. Concerning rewards structure, the property expressed in natural language: "*What is the maximum commutative reward r within 100 steps to reach* " `fail` " *for both Players 1 and 2 for a selected strategy ?*" is expressed in rPATL as $\langle\langle 1, 2 \rangle\rangle \mathtt{R}_{\mathtt{max}=?}[\mathtt{C}^{\leq 100}]$

Example 1. Consider the CSG shown in Figure 1, which corresponds to two players repeatedly performing a scheduled read and write operation. Transitions are labeled with actions where $A = (r_1 r_2), (w_1 w_2), (w_2 r_1), (r_2 w_2), (reset_1 reset_2)$. The CSG starts in state 's0', and states 's1', 's2', and 's3' are labeled with atomic propositions corresponding to a player winning. Each player communicates through writing and reading operations.

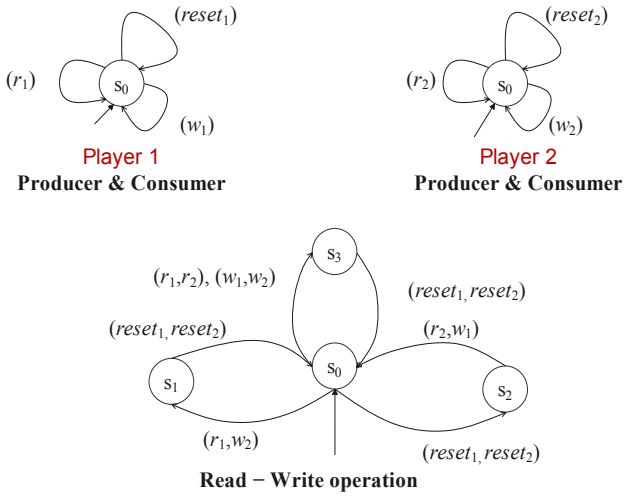

Figure 1: Read and Write Game Model in CSG.

In relation to the modeled system, if Player 1 commences the game and that player emerges victorious as it performs writing, the property is expressed as $\langle\langle 1, 2 \rangle\rangle \mathtt{P}_{>0.99} =?[\mathtt{F\ win} = 1]$. The model and properties associated with the example are available at [23].

2.2. The PRISM-games language

We rely on the CSG model to express the coalition game in PRISM language [24]. The PRISM model is composed of a set of modules that can synchronize. A set of variables and commands characterizes each module.

The variable's valuations represent the state of the module. A set of commands is used to describe the behavior of each module (i.e., transitions). A command takes the form: $[a_j, \ldots, a_m]g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$ or, $[a_j, \ldots, a_m]g \rightarrow u$, which means, for actions " a " if the guard " g " is true, then, an update " u_i " is enabled with a probability " λ_i ". A guard is a logical proposition consisting of variable evaluation and propositional logic operators. The update " u_i " is an evaluation of variables expressed as a conjunction of assignments: $v'_i = val_i + \ldots + v'_n = val_n$ where " v_i " are local variables and val_i are values evaluated via expressions denoted by " $eval$ " such that $eval : V \rightarrow \mathbb{D}$. Let $\mathbb{D}$ be a domain of variables such as $\mathbb{D} = \mathbb{N} \cup \{true, false\}$. We define valuations for variables as " θ " such that $\theta : V \rightarrow \mathbb{D}$ that associate each variable in V with a value in $\mathbb{D}$. We use $l_0, l_1, \ldots, l_n$ as new valuations of variables $v_0, v_1, \ldots, v_n$. Each PRISM command is encapsulated within a PRISM module [25] defined as follows:

Definition 2. (PRISM-Module). A PRISM module " $\mathscr{D}$ " is a tuple $\mathscr{D} = \langle \vartheta_L, Cm \rangle$, where:

- ϑ_L is a finite set of local variables associated with the module $\mathscr{D}$ initialized with $init$,
- Cm is a finite set of commands that defines the behavior of module $\mathscr{D}$. Formally, we consider the command of the form $[a_0, \ldots, a_m]g \rightarrow \lambda : u$ as $l_i \xrightarrow{g:\langle a_j,\ldots,a_m\rangle}_{\lambda} l'_i$ such that $l_i \models g$, $g \in Const(V)$ and $\theta' := \theta[v_i := eval(v_i)]$.

Example 2. In the PRISM code of Listing 1, a dedicated non-player module is used for orchestrating read and write operations. All commands are labeled with at least two ports, which correspond to the players responsible for triggering the internal write and read operations. The "win" variable defines the player's success in writing (taking values 1 or 2). The first commands shown in lines 5-6 represent unscheduled writing (i.e., reading) operations. As these operations are executed, a reset command is introduced in line 8 to indicate an idle state. Subsequently, the commands depicted in lines 9-10 enforce an order between writing and reading operations. The model and properties associated with the example are available at [23].

Listing 1: PRISM Code for Read/Write of Figure 1

```
1  module recorder // a non player model in charge of recording players
       actions
2   win : [0..2]init 0;
3   a : [0..2]init 0;
4
5   [w1,w2] s=0 -> (s'=1) & (win'=0);
6   [r1,r2] s=0 -> (s'=1) & (win'=0);
7
8   [reset1,reset2] s=1 | s=2 | s=3 -> (s'=0) & (win'=0);
9   [r1,w2] s=0 -> (s'=2) & (win'=2);
10  [w1,r2] s=0 -> (s'=3) & (win'=1);
11 endmodule
```

2.3. The RabbitMQ broker

RabbitMQ, a message broker, implements the Advanced Message Queuing Protocol (AMQP), standardized in ISO/IEC 19464:2014 [26]. AMQP defines the communication rules for messages where RabbitMQ serves as a concrete implementation. This paper utilizes the AMQP 0.9/1.0 implementation provided by RabbitMQ [27]. The reference communication stack, depicted in Figure 2, is adapted from the AMQP Architecture specification (AMQP Architecture: [27] and [28]). (1) Application Layer: This layer interacts with applications using client libraries and defines a message format and semantics (*see section 2.1 AMQ Model Architecture* in [27]). (2) Messaging Layer: This layer handles message routing, exchanges, queues, and bindings (*see section 3.1.1 Messages and Content* in [27]), with a focus on message routing and distribution. (3) Framing Layer: This layer takes messages

from the messaging layer and packages them into frames with specific headers and content data (*see section 2.3.5 Frame Details* in [27]), focusing on data structuring and encapsulation. (4) Transport Layer: This layer provides reliable and secure communication between peers (RabbitMQ clients/servers). It typically uses TLS/SSL for encryption and TCP for reliable delivery (*many features are discussed in section 2.3 AMQP Transport Architecture* in [27]). (5) Wire Level: This layer defines the byte format on the network for each frame (*see section 4.2 AMQP Wire-Level Format* in [27]), addressing low-level network communication details.

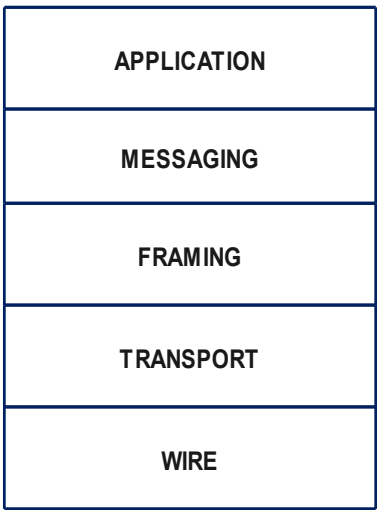

Figure 2: RabbitMQ Implementation of AMQP Architecture[28].

The main feature that can justify the use of the RabbitMQ broker for IoT implementing AMQP and in the case of IoT gateway [29, 30] is the implementation of a CoAP-to-RabbitMQ. It can bridge the communication gap between resource-constrained devices and robust messaging systems. This bridge translates CoAP messages into the richer format used by RabbitMQ. This approach fosters interoperability, allowing diverse device types to participate in a unified communication ecosystem supported by gateways in [29]. In addition, RabbitMQ broker implements clustering functions [31] that enhance the availability and reliability of the infrastructure.

3. Approach

This paper implements the workflow depicted in Figure 3 for modeling the RabbitMQ broker and conducting security analysis.

Frequencies learning. The frequencies of attacks, such as DoS, Mirai, and ARP spoofing, have been learned from a collected dataset using the algorithm described in Section 4. These attack frequencies are required for populating the PRISM model.

Modeling. The modeling phase entails translating protocol specifications, described in natural language [27], into a PRISM model consisting of modules that adhere to the PRISM language (as discussed in Section 2). This process results in a parametrizable PRISM model. Our work enhances this process by incorporating security considerations and formalizing specific attack scenarios within the CSG PRISM-supported language. The attack scenario examined in this research focuses on CAPEC-384 [32], which involves the "*Application API Message Manipulation via Man-in-the-Middle*". Therefore, the parametrizable PRISM model is populated with attack frequencies to capture the stochastic nature of the attacks within their cyber-physical environment. These frequencies represent the occurrence of attacks as probabilistic phenomena derived from an Open-Source dataset.

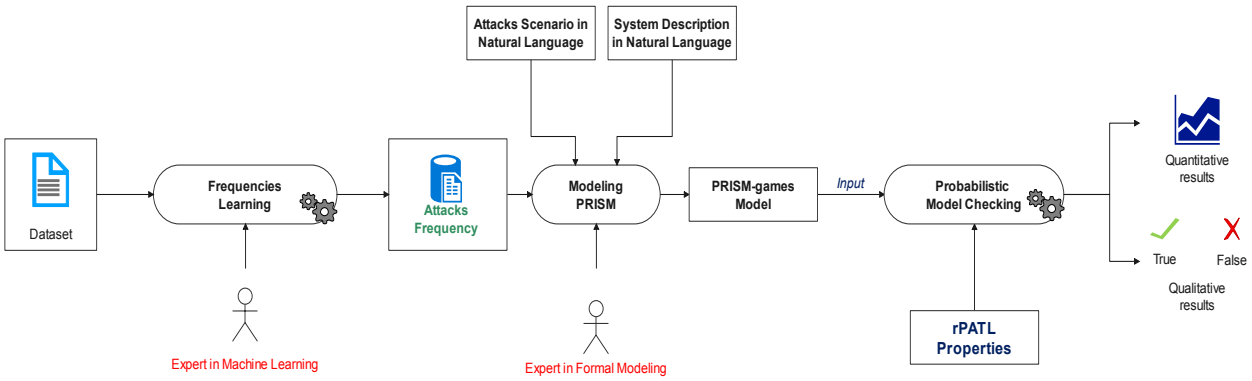

Figure 3: A Workflow for Learning, Modeling, and Verification of the RabbitMQ Broker

Probabilistic model checking. To assess the potential risks inherent in the RabbitMQ broker regarding attacks, we utilize rPATL properties. The risk involves assessing the likelihood of an attack occurring and its potential consequences on the communication protocol [33]. The rPATL properties provide an indication of the degree of data corruption in the form of game goals resulting from manipulated data within the broker using the PRISM-games model checker. By considering these properties, we gain insights into risks that may arise within the RabbitMQ broker.

4. Learning Attack Frequencies

This section delves into the process of learning attack frequencies from input dataset. We assume attacks occur within specific timeframes, as illustrated in Figure 4. The Mean Time Between Attacks (MTBA) is extracted from the dataset using established algorithms like the Broyden–Fletcher–Goldfarb–Shanno (L-BFGS-B) algorithm described in [34] and the Nelder-Mead algorithm [35]. Section 8 will discuss additional algorithms inspired by natural behaviors [36, 37, 38, 39, 40, 41] that can be employed for this purpose.

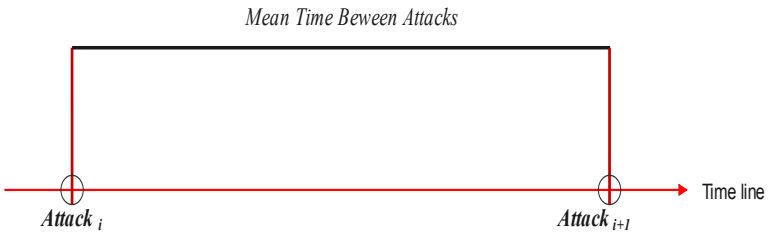

Figure 4: Mean Time Between Attacks

The classical approach for calculating the mean time involves considering the initial attack detected along with its successors, divided by the total number of attacks in the dataset. Formally, for a dataset consisting of n features: $f_1, \ldots, f_n \in F$ where each feature is associated with its domain $Dom : F \to \mathbb{D}$. $\mathbb{T}$ is the time domain for attack feature occurrence. We are able to implement the Algorithm 1. The algorithm is a simple exploration of the dataset where two attacks are collected a_1 and a_2, where a_2 is the successor of a_1. When an attack is detected, its successor is determined and the difference is stored in the stack. Subsequently, the mean time between attacks is calculated once the successor is empty. For handling large datasets, we utilize both the L-BFGS and Nelder-Mead algorithms. The Python source code for this implementation can be found in the Artefacts section 6.4.

Algorithm 1: Mean Time Between Attacks Computation.

Data: Feature f_i.
Result: p as attack frequency.

$Stack \leftarrow Empty$; /* A stack of meantime of attacks. */
$a_1 \in \mathbb{T}$; /* initial attacks. */
$a_2 \in \mathbb{T}$; /* sucessor of the attack. */
for *line* $\in$ *dataset* **do**
  if *attack* $\in$ *line* **then**
    $a_1 = extractTimefromfeature(f_i, attack)$; /* Extract the first attack. */
    $a_2 = extractNextAttackTimefromfeature(f_i, attack)$; /* Extract the successor of the first attack. */
    $Stack.append(|a_2 - a_1|)$; /* Store the time differences */
  end
  if a_2 *is* $Empty$ **then**
    $satisfy \leftarrow false$;
    $p \leftarrow meanTime(Stack)$; /* Store the calculated mean time as the probability of an attack occurring. */
  end
end

5. RabbitMQ architecture and threats modeling

This section presents the formalization of the RabbitMQ Architecture to make it compatible with the Concurrent Stochastic Game (CSG). We also provide a formal implementation of RabbitMQ in PRISM games. We formalize and model the CAPEC-384 [32] attack at various protocol levels, specifically focusing on data corruption, and establish semantic rules to capture their impact.

5.1. The RabbitMQ architecture

In Figure 5, we present the architectural depiction of RabbitMQ deployed on an IoT gateway. This deployment comprises a set of Producers, specifically three sensors: water level (WL), water volume (WV), and rain precipitation (RP), as well as two consumers, including one actuator and the Fog. Additionally, a connectivity interface is incorporated to convert signals into data, which is susceptible to attacks, and it serves both the consumers and producers. The exchange is designed to accept messages from the producers and direct them to message queues based on a designated **binding key**. For instance, the water level queue is associated with the binding key *binding key = wl_pl*. The exchange scrutinizes the message properties derived from the producers' messages in order to extract the **routing key**, which functions as a virtual address that determines the appropriate message queue for storage by matching the binding keys. Formally, a RabbitMQ system during execution is defined as follows:

Definition 3. (RabbitMQ). A RabbitMQ system " $\mathscr{RQ}$ " is a tuple $\mathscr{RQ} = \langle \mathscr{M}, RK, BK, E, Q \rangle$, where:

- $\mathscr{M}$ is the message structure alive in $\mathscr{RQ}$,
- $RK = \{rk_0, \ldots, rk_n\}$ is a set of routing keys,
- $BK = \{bk_0, \ldots, bk_n\}$ is a set of binding keys,
- E is the exchange node of the system, and
- $Q = \{q_0, \ldots, q_n\}$ is a set of queues supported by $\mathscr{RQ}$.

Throughout the paper, we consider a message composed of a *routing key and a payload*. Formally, we define a RabbitMQ message as:

Definition 4. (Message). A RabbitMQ message " $\mathscr{M}$ " is a tuple $\mathscr{M} = \langle rk, pld \rangle$, where:

- rk is the routing key of sensors message and
- pld is the payload of the sensed data

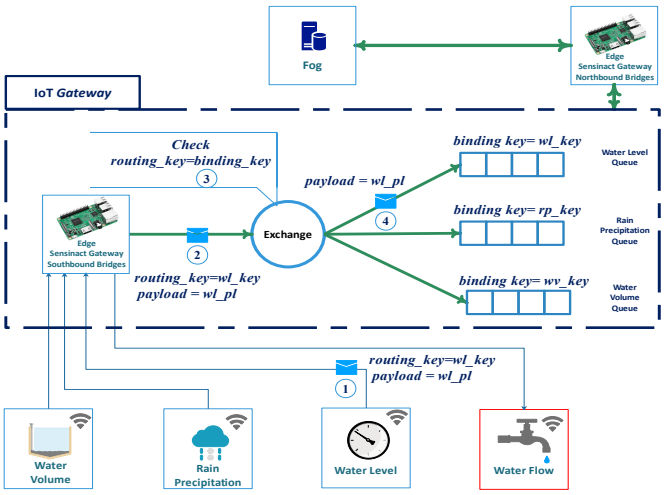

Figure 5: The RabbitMQ Architecture for IoT.

5.2. Communication formalism for RabbitMQ architecture in PRISM-games

In the PRISM-games formalism, the modeling of communication and resource access is facilitated through the use of player and non-player modules. Players are distinguished by unique action labels, while non-CSG players are identified by multiple action labels. For instance, each component of the modeled system in Figure 5 is considered as a PRISM module where reading and writing is enabled by access to local PRISM module variables.

The algebraic expression of the message m transmitted from module $\mathscr{D}_1$ to $\mathscr{D}_2$ is expressed through channeling [20] using send (!) and receive (?) symbols as : $[\![\mathscr{D}_s]\!] = l_1 \xrightarrow{\langle a!m \rangle} l_2$ saying that $\mathscr{D}_s$ transmit the message through the channel a (PRISM action) and $[\![\mathscr{D}_r]\!] = l_3 \xrightarrow{\langle a?x \rangle} l_4$ saying that $a?x$ receives a message via channel a and assign it to variable x .

However, building upon the player definition provided in the previous section, we introduce a two-player rule that involves a competition based on writing. $\mathscr{D}_3$ records the messages received from both players $\mathscr{D}_1$ and $\mathscr{D}_2$. The rule *Writing* is defined as follows:

$$\frac{[\![\mathscr{D}_1]\!] = l_1 \xrightarrow{a!m_1} l_1' \wedge [\![\mathscr{D}_2]\!] = l_2 \xrightarrow{b!m_2} l_2' \wedge [\![\mathscr{D}_3]\!] = l_3 \xrightarrow{a?k_1, b?k_2} l_3'}{\langle l_1, \ldots, l_2, \ldots l_3, \theta \rangle \xrightarrow{a,b} \langle l_1', \ldots, l_2', \ldots, l_3', \theta' \rangle} \quad (\textit{Writing})$$

where $\theta' := \theta[k_1 := m_1, k_2 := m_2]$ and $\mathscr{D}_1, \mathscr{D}_2$ are two CSG players.

5.3. Queues modeling

The RabbitMQ queues are responsible for storing the sensed data based on the routing key present in the transmitted messages. The Exchange module verifies the correspondence between the routing key and the binding key in order to carry out the operation. Firstly, the $Queue$ module is regarded as a player responsible for data storage. It is characterized by the queue variable and the identifier binding key, as depicted in lines 1-3 of the

code snippet of Listing 2. As the PRISM language lacks native support for the list data type, it is necessary to define the list cases and the index variable, as mentioned in lines 1-2. To traverse the queue, we require an item index that is initialized to 0 at line 5.

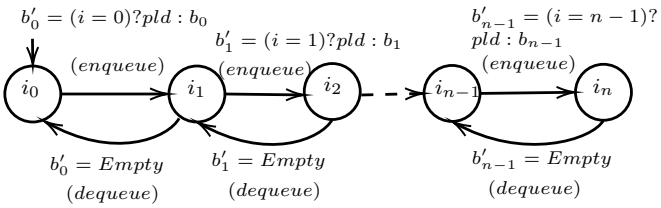

Figure 6: Queue Model.

To enhance clarity, the enqueue and dequeue operations are modeled as automata in Figure 6. At each state (corresponding to an index value), the *enqueue* operation is performed, and the variables $b_0, \ldots, b_1$ are assigned the value of the payload, denoted as pld following the rule *Writing*. The consumer (for example a Fog) executes the dequeue operation, while a non-player module stores the dequeued data that will be consumed. The command in line 6 executes the automata model depicted in Figure 6, where each state represents an index value. Additionally, we utilize a conditional structure in command line 6 to assign the value of pld to the corresponding variable b_i based on the index value. In the dequeue operation, the synchronization based on the dequeue channel is carried out with the command in line 9.

Listing 2: PRISM Code for the Queue Player

```
1  module Queue
2   b0: [INIT_VAL..MAX_VAL] init EMPTY;
3   b1: [INIT_VAL..MAX_VAL] init EMPTY;
4   bk : [KEY_0..KEY_0] init KEY_0;
5   i    :[0..QUEUE_MAX] init 0;
6   [enqueue] i<QUEUE_MAX -> (i'=mod(i+1,QUEUE_MAX))  &(b0'=(i=0)?pld3:b0)
        &(b1'=(i=1)?pld3:b1);
7   [dequeue] i>0 -> (i'=i-1)  &(b0'=(i=0)?EMPTY:b0) &(b1'=(i=1)?EMPTY:b1);
8  endmodule
```

The model, consisting of one producer (sensor), one queue, and one consumer (fog), can be accessed through the following link: [23] under reference M2. In the following section, we instantiate multiple queues and sensors to facilitate the exchange of messages through a use case.

5.4. Attacks modeling

In this section, we formalize the attack CAPEC-384 [32] that can impact the messages exchanged between communicating entities, thereby affecting both the payload and the routing key. For each message m we use the following notations: rk denotes the routing key of the message m, rk_x denotes the erroneous routing key of the message m, pld denotes the payload of the message m, and pld_x denotes the erroneous payload of the message m.

Tampering [42] leads to the alteration of messages transmitted by the sender component through the communication port. Within the context of $\mathscr{RL}$ system, pld is altered during the message transfer. Considering the sensor player $\mathscr{D}_1$ and attacker player $\mathscr{D}_2$ where the exchange node manages the interaction modeled as $\mathscr{D}_3$. When refining the rule (*Writing*), the outcomes of routing key tampering are expressed through rules *rk Success* and *rk Failure*. Specifically, rule *rk Success* represents the successful tampering of the routing key, while rule *rk Failure* represents the failure to tamper with the routing key.

$$\frac{[\![\mathscr{D}_1]\!] = l_1 \xrightarrow{a!\langle rk_1, pld_1\rangle} l_1' \wedge [\![\mathscr{D}_2]\!] = l_2 \xrightarrow{b!\langle rk_x, pld_2\rangle} l_2' \wedge [\![\mathscr{D}_3]\!] = l_3 \xrightarrow{a,b?\langle rk_3, pld_3\rangle}_{p} l_3'}{\langle l_1, \ldots, l_2, \ldots l_3, \theta\rangle \xrightarrow{a,b}_{p} \langle l_1', \ldots, l_2', \ldots, l_3', \theta'\rangle} \quad \textit{(rk Success)}$$

where $\theta' := \theta[rk_3 = rk_x, pld_3 = pld_2]$ and $\mathscr{D}_1$ is the sensor, $\mathscr{D}_2$ is the attacker. p is the success rate of the attacker.

$$\frac{[\![\mathscr{D}_1]\!] = l_1 \xrightarrow{a!\langle rk_1, pld_1\rangle} l_1' \wedge [\![\mathscr{D}_2]\!] = l_2 \xrightarrow{b!\langle rk_x, pld_2\rangle} l_2' \wedge [\![\mathscr{D}_3]\!] = l_3 \xrightarrow{a,b?\langle rk_3, pld_3\rangle}_{1-p} l_3'}{\langle l_1, \ldots, l_2, \ldots l_3, \theta\rangle \xrightarrow{a,b}_{1-p} \langle l_1', \ldots, l_2', \ldots, l_3', \theta'\rangle} \quad \textit{(rk Failure)}$$

where $\theta' := \theta[rk_3 = rk_1, pld_3 = pld_1]$ and $\mathscr{D}_1$ is the sensor, $\mathscr{D}_2$ is the attacker. $1-p$ is the failure rate of the attacker.

Since the model incorporates the stochastic behavior of the system, it accurately represents the success and failure of the attacker using a stochastic parameter p.

This parameter enables us to effectively model the message loss of the sensor at the exchange level as a result of the attack. While we address the issue of routing key tampering, it is also necessary to model payload tampering caused by similar attacks. Then, the success and failure rules are expressed by the rules *pld Success* and *pld Failure*.

$$\frac{[\![\mathscr{D}_1]\!] = l_1 \xrightarrow{a!\langle rk_1, pld_1\rangle} l_1' \wedge [\![\mathscr{D}_2]\!] = l_2 \xrightarrow{b!\langle rk_2, pld_x\rangle} l_2' \wedge [\![\mathscr{D}_3]\!] = l_3 \xrightarrow{a,b?\langle rk_3, pld_3\rangle}_{p} l_3'}{\langle l_1, \ldots, l_2, \ldots l_3, \theta\rangle \xrightarrow{a,b}_{p} \langle l_1', \ldots, l_2', \ldots, l_3', \theta'\rangle} \quad \textit{(pld Success)}$$

where $\theta' := \theta[rk_3 = rk_1, pld_3 = pld_x]$ and $\mathscr{D}_1$ is the sensor, $\mathscr{D}_2$ is the attacker. p is the success rate of the attacker.

$$\frac{[\![\mathscr{D}_1]\!] = l_1 \xrightarrow{a!\langle rk_1, pld_1\rangle} l_1' \wedge [\![\mathscr{D}_2]\!] = l_2 \xrightarrow{b!\langle rk_2, pld_x\rangle} l_2' \wedge [\![\mathscr{D}_3]\!] = l_3 \xrightarrow{a,b?\langle rk_3, pld_3\rangle}_{1-p} l_3'}{\langle l_1, \ldots, l_2, \ldots l_3, \theta\rangle \xrightarrow{a,b}_{1-p} \langle l_1', \ldots, l_2', \ldots, l_3', \theta'\rangle} \quad \textit{(pld Failure)}$$

where $\theta' := \theta[rk_3 = rk_1, pld_3 = pld_1]$ and $\mathscr{D}_1$ is the sensor, $\mathscr{D}_2$ is the attacker. $1-p$ is the failure rate of the attacker.

Now, with the formal specification of attacks at the exchange, we can proceed to provide a projection onto the PRISM code. Before modeling the exchange module of the interaction system, we present in Figure 7 the attacker and producer players model. They are characterized by two channels (we employ the terminology used for the channeling system [20]): write and wait. The write channel is responsible for transmitting the produced value from the producer (i.e., attacker) to the exchange node. The writing and waiting operations are performed each round r.

The exchange system captures the interactions between producers (sensors) and attackers in the Listing 3, specifically focusing on the payload transfer. This is achieved through the utilization of two channels, namely $write_1$ and $write_2$. The success and failure rules, denoted as *pld Success* and *pld Failure*, are implemented through the PRISM commands found in lines 4-5. Additionally, players have the ability to enter a waiting mode, which is modeled using two channels, $wait_1$ and $wait_2$, implemented as actions in PRISM. In the case of waiting modes, the exchange module resets both internal binding keys and payload in line 6. However, if one of the players is available to transfer its payload, a non-probabilistic command is implemented in lines 7-8.

Remark. The definition of binding key tampering follows the same principle as payload tampering.

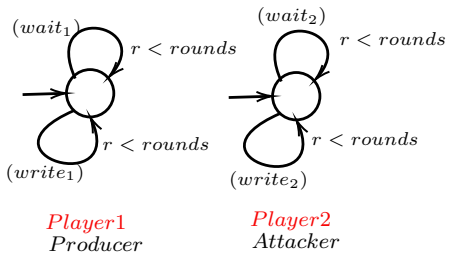

Figure 7: Attacker and Sensor Models.

Listing 3: PRISM code for Exchange Module

```
module Exchange
pld3 : [INIT_VAL..MAX_VAL] init EMPTY;
rk3 : [INIT_VAL..MAX_VAL] init EMPTY;
[write1, write2] true -> (1-p):(pld3'=pld1)&(rk3'=rk3) +p:(pld3'=pld2)&(rk3'=rk2);
[wait1 , wait2] true -> (pld3'=EMPTY) & (rk3'=EMPTY);
[write1, wait2 ] true -> (pld3'=pld1) & (rk3'=rk1 );
[wait1 , write2] true -> (pld3'=pldx) & (rk3'=rk2 );
endmodule
```

6. Experiments

In the context of our experiments, we refer to a use case scenario from the collaborative European research project [43] for smart water flow assessment. Three sensor nodes gather and transmit diverse measurements such as water level (WL), water volume (WV), and rain precipitation (RP) payload to the Fog system through the Sensinact gateway [29] to assess the water flow (WF). The internal digital architecture of the industrial use case is partially depicted in Figure 5. The deployment of RabbitMQ on the Sensinact gateway has not yet been completed. As a result, questions regarding the feasibility of this deployment are being addressed and explored in our study. We aim to gain a deeper understanding of the potential impact of an attack.

The requirements described below are a summary (with some revisited information) of the security properties of RabbitMQ. They are expressed as a set of properties in natural language to protect the corresponding assets.

- ***Property 1***: When the sensors transmit payload messages (wv, wl, and $rp \in [1 \dots 3]$) to the fog, we are interested in determining the probability of payload tampering at each round.
- ***Property 2***: When the sensors transmit payload (wv, wl, and $rp \in [1 \dots 3]$) messages to the fog, our objective is to determine the probability of message loss after each round.
- ***Property 3***: When the sensors transmit payload (wv, wl, and $rp \in [1 \dots 3]$) messages to the fog, Our objective is to determine the probability of message loss caused by the unrouted payload after each round (Tampering affecting only routing keys).

We intend to provide a formal specification of the functional and security requirements for message delivery that RabbitMQ must fulfill to ensure its effectiveness. We will consider DoS, MITM ARP spoofing, and Mirai attacks at southbound bridges (sensors) and queues.

6.1. Experimental setup

Our analyses rely on PRISM-games v4.7, where experiments are conducted on an Ubuntu-I7 system with 32GB RAM. The system model is accessible via the link provided in [23] and adheres to the *CSG* formalism

supported by PRISM-games. The engine is set to "Explicit"; other engines are supported by PRISM that offers performance regarding model structure (Please refer to documentation). Player definitions within the model are identified by "player tag" delimiters. For example, the player `p1` represents the attacker, as exemplified in: `player p1 Attacker endplayer`. Parameters associated with data manipulation attacks are derived from the dataset denoted as "`C1`" in [23]. The selection of attack frequency hinges on the chosen scenario, declared as a `const int` variable. For instance, scenario 1 corresponds to DDoS attacks, scenario 2 to ARP attacks, and scenario 3 to Mirai attacks. A dedicated Python code was developed to calculate the attack rate based on this dataset. Properties relative to our models are expressed in the Modeling section.

6.2. Modeling

The RabbitMQ models are partitioned into two distinct models to analyze the impacts of threats on different components of the architecture, specifically the southbound bridges ("M3") and queues ("M4"). This division is essential to address the challenges of the state space explosion.

We enhance the model by incorporating an integer constant as in [44] and a module (see Listing 4) to keep track of the number of rounds. In this case, as the commands are unaffected by the players' choices, they are considered unlabelled with empty action. Consequently, these commands are executed regardless of the actions taken by the players. Furthermore, the module is deterministic due to the disjoint guards present in both commands.

Listing 4: Rounds module to Express multiple Tentatives

```
const k; // number of rounds
module rounds // module to count the rounds
 rounds : [0..k+1];
 [] rounds<=k -> (rounds'=rounds+1);
 [] rounds=k+1 -> true;
endmodule
```

The properties are defined as a set of requirements in rPATL, mapped from the original natural language expressions in the use case.

Property 1

$$\langle\langle p1, p2\rangle\rangle \mathtt{Pmax} =?[!\text{“payload\_tamper”}\,\mathtt{U}\,\text{“payload\_tamper”}],\ \mathtt{rounds} = 1:30:1 \tag{1}$$

Property 2

$$\langle\langle p1\rangle\rangle \mathtt{R}\{\text{“damage”}\}\mathtt{max} =?[\mathtt{F}\ \mathtt{r} = \mathtt{rounds}]/ \langle\langle p2\rangle\rangle \mathtt{R}\{\text{“sent”}\}\mathtt{max} =?[\mathtt{F}\ \mathtt{r} = \mathtt{rounds}],\ \mathtt{rounds} = 1:30:1 \tag{2}$$

In the context of reward structure, "damage" refers to intercepted and corrupted messages, while total messages "sent" by sensors represent the overall number of messages transmitted.

Property 3

$$\langle\langle p1, \ldots, p8\rangle\rangle \mathtt{R}\{\text{“empty”}\}\mathtt{max} =?[\mathtt{F}\ \mathtt{r} = \mathtt{rounds}]/ \langle\langle p1, \ldots, p8\rangle\rangle \mathtt{R}\{\text{“sent”}\}\mathtt{max} =?[\mathtt{F}\ \mathtt{r} = \mathtt{rounds}],\ \mathtt{rounds} = 1:30:1 \tag{3}$$

By "empty" rewards, we refer to the state of queues that are not filled with the corresponding payload.

6.3. Experiments and analysis of the results

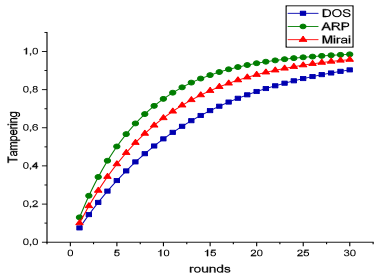

Figure 8: Verification of Property 1.

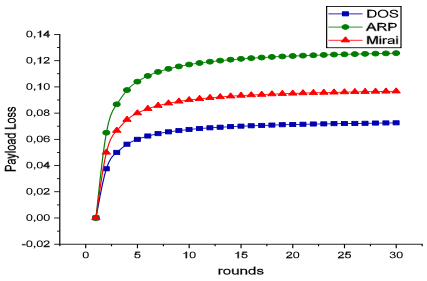

Figure 9: Verification of Property 2.

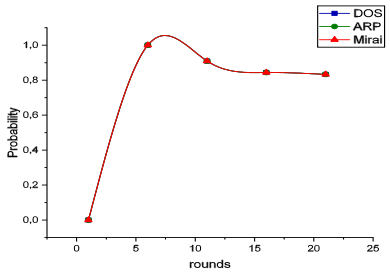

Figure 10: Verification of Property3.

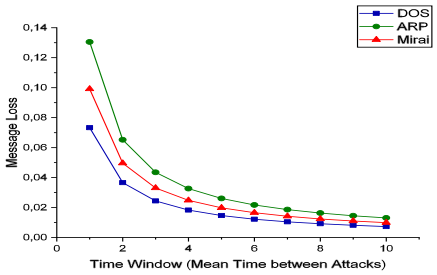

Figure 11: Verification of Property 2 for different time windows.

The model is then verified against the previous properties in the context of DoS, MITM ARP spoofing, and Mirai attacks. We collect the rate of each attack to populate our CSG model. In the first case, we focus on attacks targeting the southbound bridge ports, which have an impact on the sensed data and corrupt its values (***Property 1***, ***Property 2***). The second case investigates scenarios in which attacks have an impact on the routing keys (***Property 3***).

6.3.1. On southbound bridges attacks

For the first experiment, the label "payload_tamper" in ***Property 1*** is implemented as the payload received at the southbound bridge differs from the expected attacker value. The model is downloadable from [23] with ID "M3". The results are portrayed in Figure 8. The player p1 represents the attacker, while p2 represents the sensors responsible for payload transmission. We observe that the payload tampering increases at each round of the CSG game to reach 98% . However, the impacts of tampering differ from one attack to another, as it depends on the failure rate present in the dataset. During the experiment, it was observed that ARP spoofing had the highest impact on the model, followed by Mirai and DoS attacks.

The second experiment illustrates the damages incurred due to payload loss, as expressed in requirement ***Property 2*** using rPATL. The attacks do not exceed an acceptable increase of 14% with respect to the sensitive infrastructure. However, to ensure accurate learning, the data must be correct and avoid the need for preprocessing. For example, in the research performed in [45], a processing step involves removing outliers and erroneous data through mathematical operations, which is expensive regarding the large dataset. In Figure 9, The damage is primarily caused by ARP spoofing, followed by Mirai and DoS attacks, and remains stable at less than 14% .

6.3.2. On queues attacks

The third experiment aims to investigate how the queue remains empty despite the payload messages being communicated to the gateway using property expressed in ***Property 3***. The label "empty" assigns a value of 0.5 (since there are two items in the queue), indicating that the queue is considered empty. Upon interpreting Figure 10, we observe that the queue is 100% empty during the first five rounds, which is expected since the number of

messages sent matches the total number of empty queues. However, we observe that the loss of payload messages is close to 99% during the occurrence of tampering with the routing key. Other experiments could be conducted to investigate the impact of tampering with routing keys on the southbound bridges or the exchange module. However, since the results are expected to be similar to payload tampering, we have avoided performing these operations.

6.4. Artefacts

The experiments elucidated in this section are openly accessible and entirely replicable. The source code can be obtained from the public GitHub website and repository [23]. The website offers comprehensive instructions on replicating the experiments and employing PRISM-games with individual examples. The repository encompasses a Python code that extracts the mean time between attacks from a CSV file to populate the PRISM model. The input dataset employed for learning is sourced from the Canadian Institute for Cybersecurity.

7. Discussion

7.1. Beyond model checking

We have showcased the practicality of formal methods, particularly model checking of stochastic games with PRISM-games, in the modeling and verifying of the RabbitMQ communication broker for integration with the Sensinact gateway. We encoded the vital requirements of the RabbitMQ broker using rPATL and formally verified them. In the context of our experiment, these requirements encompassed the transmission of messages from sensors to the fog, ensuring that the messages are received even in the presence of DoS, Mirai, and ARP attacks that attempt to corrupt the payloads and the routing keys. In Section 5, we formalize rules about the occurrence of tampering during the transmission of payload messages and routing keys. Users may find implementing these rules in different formalisms beneficial instead of using the PRISM games language. However, it should be noted that the verification process becomes lengthy when models incorporate a significant number of variables, such as queues with item variables. In our experiments, verifying the property in ***Property 3*** took 3 seconds to construct the model and 18 seconds to perform the verification. In such cases, abstraction could be beneficial in reducing the computational overhead.

7.2. Risk mitigation

Thanks to the Python artifacts we developed, we are able to capture the initial window of attacks and attempt to mitigate it by increasing the mean time between attacks. The mitigation strategy suggested by RabbitMQ in their documentation when an attack is detected is to ban the attacker for a specific duration, thereby extending the time window available for verification. The results of such modification are portrayed in Figure 11 with PRISM code available on [23] with ID "`M5`". As the time window increases, the probability of message loss decreases, eventually reaching a low value of 2% . This mitigation strategy demonstrates significant utility in scenarios where the system is comprehended and all potential environmental vulnerabilities are duly acknowledged.

7.3. Network and latency

Networking using communication protocols. RabbitMQ offers a wide range of communication mechanisms based on Bindings. These mechanisms allow for routing payload messages to specific queues or multiple queues. An efficient message broadcasting feature is provided through Fanout Exchanges, where messages are routed to all bound queues. This makes them particularly suitable for scenarios where the same information needs to be delivered to multiple consumers simultaneously. Availability is enhanced through load balancing and message distribution by replicating data across queues. Additionally, RabbitMQ provides Remote Procedure Calls (RPC) [46] functionality. Implementing RPC over RabbitMQ is straightforward, as a client can send a request message and expect a response from the server. To ensure that the client receives the response, a 'callback' queue address must be included along with the request.

Network Latency. The AMQP model of RabbitMQ includes queues, which can cause latency as their size grows. Moreover, RabbitMQ supports clustering and distribution, as described in the documentation [31]. Clustering ensures high availability by distributing message workloads across multiple nodes, enhancing overall reliability and performance even under network issues. Studied critical scenarios for performance evaluation are performed and explained in [47]. We will delve deeper into the technical details of these clustering features in future work.

8. Related work

The literature review mentions a limited number of papers related to the formal verification of the RabbitMQ broker. However, it highlights the presence of research addressing validation through simulation. The RabbitMQ broker offers a range of robust features, including Authentication and Access Control [48]. This is achieved using JWT-encoded OAuth 2.0 access tokens [49] and certificate-based client authentication. Another critical feature is Clustering, which ensures high availability and fault tolerance [31]. In the event of a node failure, clients can reconnect to an alternate node, recover their topology, and resume their operations uninterrupted. A wealth of related features can be found on the RabbitMQ documentation website [50]. The RabbitMQ broker has undergone validation and verification in multiple studies, highlighting its distinguished features [12, 51, 14, 15, 16, 17].

The paper by Li et al. [12] presents a formal verification of the RabbitMQ broker using timed automata and the UPPAAL model checker[13]. Essential properties, including Reachability of Data and Message Acknowledgement, are successfully verified, confirming RabbitMQ's adherence to these properties. An extension of this work is presented in a subsequent article by Li et al. [51], which emphasizes the integration of the Kerberos network authentication protocol [52] to ensure secure communication (related to authentication). Using UPPAAL [13], the authors model and verify the enhanced protocol, providing evidence of RabbitMQ's ability to maintain secure communication.

In the paper by Ionescu [14], a performance comparison of RabbitMQ and ActiveMQ [53] brokers in message-oriented middleware applications is conducted, with a specific focus on message sending and receiving. The analysis reveals distinctions between the two brokers: ActiveMQ exhibits superior message reception speed, while RabbitMQ demonstrates qualitative message delivery to clients due to its implemented security functions during reception. Furthermore, Rostanski et al. [17] investigate design considerations for scalability and high availability in RabbitMQ. The study explores the usage of clustered RabbitMQ nodes and mirrored queues and presents simulation test results to evaluate performance.

During our review of the relevant literature, we came across the work conducted by Li et al. [12], which primarily focuses on the functional properties of RabbitMQ using UPPAAL. An extension of their work is presented in Li et al. [51], where they specifically address verification of unauthorized access to the communication channel. In contrast, studies such as [14, 15, 16, 17] primarily utilize simulation techniques to assess performance aspects of deployed servers, including scalability, memory usage, and throughput. However, these studies do not consider the security aspects related to unauthorized data modification given the complexity of the architecture. In contrast, our paper aims to bridge this gap by providing a comprehensive approach to modeling and analyzing unauthorized data modification attacks within the considered system architecture. We accomplish this by employing a game model that captures concurrent access through Concurrent Stochastic Games (CSGs) using the PRISM games.

We rely on a straightforward algorithm that utilizes Python libraries when extracting attack frequency and optimizing techniques. This algorithm allows us to calculate the Mean Time Between Attacks (MTBA) from the dataset using established algorithms from the literature, such as the Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS-B) [34] and the Nelder-Mead algorithm [35]. These algorithms have been widely used in the field and are known for their effectiveness in this context. Other nature-inspired metaheuristic algorithm, specifically focusing on optimization techniques that could be used in such optimization. In Hu et al. in [36] introduces a nature-inspired metaheuristic algorithm called GKS optimizer (GKSO), inspired by the behavior of the Genghis Khan shark. GKSO performs optimization tasks achieved by simulating hunting, movement, foraging, and self-protection mechanisms. The algorithm's viability and superiority are validated through qualitative and quantitative analyses, demonstrating its exploration and exploitation capabilities. Ezugwu et al. in [37] introduces

prairie dog optimization (PDO), a nature-inspired metaheuristic algorithm that mimics the behavior of prairie dogs in their natural habitat. PDO utilizes foraging and burrow-build activities for exploration while exploiting the prairie dogs' communication skills to converge toward promising locations. Agushaka et al. in [38] presents the dwarf mongoose optimization (DMO) algorithm inspired by the foraging behavior of dwarf mongooses. DMO utilizes three social groups (alpha group, babysitters, and scout group) to mimic the mongoose's foraging strategy. Agushaka et al. in [39] introduces the Gazelle Optimization Algorithm (GOA), inspired by the survival behavior of gazelles in predator-dominated environments. The algorithm incorporates two phases: exploitation, where gazelles graze peacefully, and exploration, evading predators.

Hu et al. in [41] presents DETDO, an adaptive hybrid dandelion optimizer based on the Dandelion Optimizer (DO). DETDO combines adaptive tent chaotic mapping, differential evolution strategy, and adaptive t-distribution perturbation to prevent local optima and improve convergence speed. Ghasemi et al. in [40] introduces Lung performance-based optimization (LPO) inspired by the performance of lungs in the human body. LPO draws inspiration from the adaptability and optimization of the respiratory system in solving complex optimization problems.

9. Conclusion

In this paper, we presented a formal specification and analysis of the RabbitMQ architecture from a security perspective in a Concurrent Stochastic Game, leveraging reward Probabilistic Alternating Temporal Logic (rPATL) to model security requirements as game goals. Based on this model, we have verified security properties concerning threats related to payloads and message routing keys. An evaluation of the proposed formalization is presented through a practical application to a use case in the IoT domain facing DoS, MITM ARP spoofing, and Mirai attacks at southbound bridges (sensors) and queues using PRISM-games model checker. Furthermore, we have highlighted the importance of attack mitigation mechanisms supported by the protocol.

We identified some limitations to consider. Firstly, the PRISM currently has limited language constructs. It only supports integer values within modules, which may restrict the expressiveness of our models. Secondly, the paper does not fully address the model's scalability with increasing module numbers. While including features like enqueuing and dequeuing, further investigation is needed to ensure the model remains efficient and performant as the system grows in complexity.

In our forthcoming research endeavors, we intend to explore the state space resulting from constructing the model in PRISM-games. This exploration will deepen our understanding of the system's security at different levels of refinements and abstractions. Further, we aim to investigate the integration of our formalization process with established Model-Based Development (MBD) methodologies and tools. Also, we plan to implement various optimization techniques, as identified in the related work, to calculate attack frequencies. Moreover, we intend to model the potential impacts of these attacks on the availability and reliability of the protocol.

Acknowledgement

The research leading to the presented results was conducted within the research profile of s**A**fety and se**C**ur**I**ty as**S**urrance for critical **IoT** systems(ACIS-IoT), supported by the Centre National de la Recherche Scientifique (CNRS). The authors express their gratitude to the individuals involved in the European projects CPS4EU and BRAIN-IoT projects, acknowledging their valuable contributions and feedback.

References

[1] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, A. Ahmed, Edge computing: A survey, Future Generation Computer Systems 97 (2019) 219–235. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18319903. doi:https://doi.org/10.1016/j.future.2019.02.050.

[2] A. Baouya, S. Ouchani, S. Bensalem, Formal modelling and security analysis of inter-operable systems, volume 13343, Springer International Publishing, 2022, pp. 555–567.

[3] H. Pourrahmani, A. Yavarinasab, A. M. H. Monazzah, J. Van herle, A review of the security vulnerabilities and countermeasures in the internet of things solutions: A bright future for the blockchain, Internet of Things 23 (2023) 100888. URL: https://www.sciencedirect.com/science/article/pii/S2542660523002111. doi:https://doi.org/10.1016/j.iot.2023.100888.

[4] V. Casola, A. De Benedictis, C. Mazzocca, V. Orbinato, Secure software development and testing: A model-based methodology, Computers & Security 137 (2024) 103639. URL: https://www.sciencedirect.com/science/article/pii/S0167404823005497. doi:https://doi.org/10.1016/j.cose.2023.103639.
[5] B. Hamid, D. Weber, Engineering secure systems: Models, patterns and empirical validation, Computers & Security 77 (2018) 315–348. URL: https://www.sciencedirect.com/science/article/pii/S0167404818303043. doi:https://doi.org/10.1016/j.cose.2018.03.016.
[6] J. Zhang, M. Ma, P. Wang, X. dong Sun, Middleware for the internet of things: A survey on requirements, enabling technologies, and solutions, Journal of Systems Architecture 117 (2021) 102098. URL: https://www.sciencedirect.com/science/article/pii/S1383762121000795. doi:https://doi.org/10.1016/j.sysarc.2021.102098.
[7] P. Maiti, H. K. Apat, B. Sahoo, A. K. Turuk, An effective approach of latency-aware fog smart gateways deployment for iot services, Internet of Things 8 (2019) 100091. doi:https://doi.org/10.1016/j.iot.2019.100091.
[8] Z. Shelby, K. Hartke, Coap: An application protocol for resource-constrained iot devices, in: Proceedings of the 8th International Conference on Embedded Networked Sensor Systems, ACM, 2010.
[9] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2013.
[10] M. Kwiatkowska, G. Norman, D. Parker, G. Santos, Automatic verification of concurrent stochastic systems 58 (2021).
[11] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, A. Simaitis, Automatic verification of competitive stochastic systems, Springer Berlin Heidelberg, 2012.
[12] R. Li, J. Yin, H. Zhu, Modeling and analysis of rabbitmq using uppaal, in: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2020, pp. 79–86. doi:10.1109/TrustCom50675.2020.00024.
[13] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, M. Hendriks, Uppaal 4.0 (2006).
[14] V. M. Ionescu, The analysis of the performance of rabbitmq and activemq, in: 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), 2015, pp. 132–137. doi:10.1109/RoEduNet.2015.7311982.
[15] X. J. Hong, H. Sik Yang, Y. H. Kim, Performance analysis of restful api and rabbitmq for microservice web application, in: 2018 International Conference on Information and Communication Technology Convergence (ICTC), 2018, pp. 257–259. doi:10.1109/ICTC.2018.8539409.
[16] A. E. Bagaskara, S. Setyorini, A. A. Wardana, Performance analysis of message broker for communication in fog computing, in: 2020 12th International Conference on Information Technology and Electrical Engineering (ICITEE), 2020.
[17] M. Rostanski, K. Grochla, A. Seman, Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq, in: 2014 Federated Conference on Computer Science and Information Systems, 2014, pp. 879–884. doi:10.15439/2014F48.
[18] M. Kwiatkowska, G. Norman, D. Parker, G. Santos, Correlated equilibria and fairness in concurrent stochastic games, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, volume 13244, Springer International Publishing, 2022, pp. 60–78. doi:10.1007/978-3-030-99527-0_4.
[19] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, Automated verification techniques for probabilistic systems, in: M. Bernardo, V. Issarny (Eds.), Formal Methods for Eternal Networked Software Systems (SFM'11), volume 6659 of *LNCS*, Springer, 2011, pp. 53–113.
[20] C. Baier, J.-P. Katoen, Principles of model checking, The MIT Press, 2008.
[21] R. Alur, T. A. Henzinger, O. Kupferman, Alternating-time temporal logic, J. ACM 49 (2002).
[22] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability 6 (1994) 512–535.
[23] J. Swnow, Paper Artefacts Sources, https://acis-iot.github.io/iot24.html, 2023.
[24] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: Proc. 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of *LNCS*, Springer, 2011, pp. 585–591.
[25] M. Kwiatkowska, G. Norman, D. Parker, Quantitative analysis with the probabilistic model checker prism, Electronic Notes in Theoretical Computer Science 153 (2006). Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005).
[26] Iso standard 64955, https://www.iso.org/standard/64955.html, Accessed 2024.
[27] RabbitMQ - amqp 0-9-1, 2019. URL: https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf.
[28] Amqp architecture, https://www.amqp.org/product/architecture, Accessed 2024.
[29] Kentyou, Sensinact delivrables. Brain-IoT D4.5: Final Deployment and Operation Enablers, 2023.
[30] A. Baouya, S. Chehida, S. Bensalem, L. Gürgen, R. Nicholson, M. Cantero, M. Diaznava, E. Ferrera, Deploying warehouse robots with confidence: the brain-iot framework's functional assurance, The Journal of Supercomputing 80 (2024) 1206–1237. URL: https://doi.org/10.1007/s11227-023-05483-x. doi:10.1007/s11227-023-05483-x.
[31] Rabbitmq clustering documentation, https://rabbitmq.com/clustering.html, Accessed 2024.
[32] T. C. R. Community, CAPEC - Attack Pattern ID: 384, https://capec.mitre.org/data/definitions/384.html, 2018. [Accessed: January-2024].
[33] G. Strupczewski, Defining cyber risk, Safety Science 135 (2021) 105143. URL: https://www.sciencedirect.com/science/article/pii/S0925753520305397. doi:https://doi.org/10.1016/j.ssci.2020.105143.
[34] D. C. Liu, J. Nocedal, Limited memory bfgs method for large scale optimization, Mathematical programming 45 (1989) 503–528.
[35] J. A. Nelder, R. Mead, A simplex method for function minimization, The Computer Journal 7 (1965) 308–313.
[36] G. Hu, Y. Guo, G. Wei, L. Abualigah, Genghis khan shark optimizer: A novel nature-inspired algorithm for engineering optimization, Advanced Engineering Informatics 58 (2023) 102210. doi:https://doi.org/10.1016/j.aei.2023.102210.
[37] A. E. Ezugwu, J. O. Agushaka, L. Abualigah, S. Mirjalili, A. H. Gandomi, Prairie dog optimization algorithm, Neural Computing and Applications 34 (2022) 20017–20065. URL: https://doi.org/10.1007/s00521-022-07530-9. doi:10.1007/s00521-022-07530-9.
[38] J. O. Agushaka, A. E. Ezugwu, L. Abualigah, Dwarf mongoose optimization algorithm, Computer Methods in Applied Mechanics and Engineering 391 (2022) 114570. URL: https://www.sciencedirect.com/science/article/pii/S0045782522000019. doi:https://doi.org/10.1016/j.cma.2022.114570.
[39] J. O. Agushaka, A. E. Ezugwu, L. Abualigah, Gazelle optimization algorithm: a novel nature-inspired metaheuristic optimizer, Neural Computing and Applications 35 (2023) 4099–4131. URL: https://doi.org/10.1007/s00521-022-07854-6. doi:10.1007/s00521-

022-07854-6.
[40] M. Ghasemi, M. Zare, A. Zahedi, P. Trojovský, L. Abualigah, E. Trojovská, Optimization based on performance of lungs in body: Lungs performance-based optimization (lpo), Computer Methods in Applied Mechanics and Engineering 419 (2024) 116582. URL: https://www.sciencedirect.com/science/article/pii/S0045782523007065. doi:https://doi.org/10.1016/j.cma.2023.116582.
[41] G. Hu, Y. Zheng, L. Abualigah, A. G. Hussien, Detdo: An adaptive hybrid dandelion optimizer for engineering optimization, Advanced Engineering Informatics 57 (2023) 102004. URL: https://www.sciencedirect.com/science/article/pii/S1474034623001325. doi:https://doi.org/10.1016/j.aei.2023.102004.
[42] A. Shostack, Threat Modeling: Designing for Security, Wiley, 2014.
[43] Brain-iot, https://www.brain-iot.eu/, 2020.
[44] M. Kwiatkowska, G. Norman, D. Parker, G. Santos, Equilibria-based probabilistic model checking for concurrent stochastic games, in: Proc. 23rd International Symposium on Formal Methods (FM'19), volume 11800 of *LNCS*, Springer, 2019, pp. 298–315.
[45] K. K. Al-jabery, T. Obafemi-Ajayi, G. R. Olbricht, D. C. Wunsch II, 2 - data preprocessing, in: K. K. Al-jabery, T. Obafemi-Ajayi, G. R. Olbricht, D. C. Wunsch II (Eds.), Computational Learning Approaches to Data Analytics in Biomedical Applications, Academic Press, 2020, pp. 7–27. URL: https://www.sciencedirect.com/science/article/pii/B9780128144824000024. doi:https://doi.org/10.1016/B978-0-12-814482-4.00002-4.
[46] Rabbitmq tutorial: Python (tutorial six), https://www.rabbitmq.com/tutorials/tutorial-six-python.html, Accessed 2024.
[47] Rabbitmq 3.10 performance improvements, https://blog.rabbitmq.com/posts/2022/05/rabbitmq-3.10-performance-improvements/, Accessed 2024.
[48] Rabbitmq access control documentation, https://rabbitmq.com/access-control.html, Accessed 2024.
[49] D. Hardt, The oauth 2.0 authorization framework, Internet Engineering Task Force, 2012. URL: https://tools.ietf.org/html/rfc6749.
[50] Rabbitmq documentation, https://rabbitmq.com/documentation.html, Accessed 2024.
[51] R. Li, J. Yin, H. Zhu, P. C. Vinh, Verification of rabbitmq with kerberos using timed automata, Mobile Networks and Applications 27 (2022) 2049–2067. URL: https://doi.org/10.1007/s11036-022-01986-8. doi:10.1007/s11036-022-01986-8.
[52] B. Neuman, T. Ts'o, Kerberos: an authentication service for computer networks, IEEE Communications Magazine 32 (1994) 33–38. doi:10.1109/35.312841.
[53] ActiveMQ, Apache activemq, https://activemq.apache.org/, Accessed 2024.