MQTTactic: Security Analysis and Verification for Logic Flaws in MQTT Implementations

Bin Yuan[*,†,1,3], Zhanxiang Song[*,†,1,3], Yan Jia[‡,2], Zhenyu Lu[*,3], Deqing Zou[*,2,3], Hai Jin[§,3], Luyi Xing[†,2]
[*]*School of Cyber Science and Engineering, Huazhong University of Science and Technology, China*
[†]*Indiana University Bloomington, USA*
[‡]*DISSec, College of Cyber Science, Nankai University, China*
[§]*School of Computer Science and Technology, Huazhong University of Science and Technology, China*

***Abstract*—IoT messaging protocols are critical to connecting users and IoT devices. Among all the protocols, the *Message Queuing and Telemetry Transport* (MQTT) is arguably the most widely used. Mainstream IoT platforms leverage MQTT brokers, server side implementation of MQTT, to enable and mediate user-device communication (e.g., the transmission of control commands). There are over 70 open-source MQTT brokers, which have been widely adopted in production. Any security defects in those open-source MQTT brokers easily get into many vendors' IoT deployments with amplified impacts, inevitably endangering the security of IoT applications and millions of users. We report the first systematic security analysis of open-source MQTT brokers in the wild. To enable the analysis, we designed and developed *MQTTactic*, a semi-automatic tool that can formally verify MQTT broker implementations based on generated security properties. *MQTTactic* is based on static code analysis, formal modeling, and automated model checking (with off-the-shelf model checker Spin). In designing *MQTTactic*, we characterize and address key technical challenges. *MQTTactic* currently focuses on authorization-related properties, and discovered 7 novel, zero-day flaws practically enabling serious, unauthorized access. We reported all flaws to related parties, who acknowledged the issues and have been taking actions to fix them. Our thorough evaluation shows that *MQTTactic* is effective and practical.**

***Index Terms*—IoT Security, MQTT, Logic Flaw**

1. Introduction

With the popularity of the *Internet of Things* (IoT) comes the demand for effectively connecting the IoT devices and users, which has been facilitated by the deployments of messaging protocols. Among all, the *Message Queuing and Telemetry Transport* (MQTT) protocol has become arguably the most widely adopted for IoT in the wild [1]. Almost all leading, commercial IoT cloud services and platforms (e.g., AWS [2], Google [3], IBM [4], Microsoft [5], Baidu [6], and Alibaba [7]) use the MQTT protocol. Moreover, there are many open-source MQTT implementations — more than 70 open-source MQTT brokers (see Appendix A.4) developed in various languages, e.g., C, Rust, and Go [8]. These open-source MQTT implementations have been widely adopted in production: e.g., HiveMQ announces that it is adopted by more than 120 companies [9], such as T-mobile, BMW, and Audi; according to ZoomEye [10], there have been over 318,000 deployments of Mosquitto. Consequently, any security defects in those open-source MQTT brokers easily get into many vendors' IoT deployments with amplified impacts, inevitably endangering the security of IoT applications and millions of real users. However, little has been done to systematically analyze security risks with open-source MQTT brokers, not to mention formally verify their security for elevated assurance, a critical effort for the IoT supply chain.

Challenges. We summarize challenges for systematic, formal security analysis on open-source MQTT brokers below.

• ***C1**: Implementation-specific protocol customization*. Despite the standard MQTT specification [11], the brokers are usually customized by developers. This is due to inadequate or intentionally open specification in standard protocols and project-specific optimizations (detailed in § 3.1). For example, where in the messaging/logic process one actually implements a permission check is often broker-specific, and we show that the customization can easily go wrong (i.e., lack of authorization at proper points, § 5.1). Hence, one cannot use a single unified model to abstract all MQTT implementations. To verify broker implementations, one has to consider their customized logic and flows, which is nontrivial. Actually, fully modeling the operations in MQTT implementations quickly become intractable [12], [13], in particular for complex projects with large code footprints (usually with more than 10,000 LOC), and complicated control flows that cannot completely align with the protocol.

• ***C2**: General model definition and implementation-specific model construction*. For practical formal analysis (e.g., to avoid state explosion [14]), one should abstract the logic and flows in the broker source code that are relevant to key broker states while dropping those non-related flows and implementation details. This requires identification and a general definition of key states in a broker's internal

[1] *Most of the work was done when Bin Yuan was at Indiana University Bloomington. Zhanxiang Song was an intern of Indiana University Bloomington at the time of the work.*
[2] *Corresponding authors: Luyi Xing, Yan Jia, Deqing Zou.*
[3] *B. Yuan, Z. Song, Z. Lu, D. Zou and H. Jin are also with Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab.*

operations. Then we need an approach to automatically extract model-relevant information from a specific broker's source code and generate the model implementation (i.e., representation of the broker-specific model using a modeling language, versus theoretical level model definition), so it can be verified by automatic model checkers. The approach should easily support many different broker implementations in the wild (see Appendix A.4).

Our goals and the gaps in prior works. In this work, we aim to develop formal methods and systematic techniques to analyze the security of open-source MQTT brokers, especially focusing on control and logic flows of MQTT brokers' operations (e.g., flawed control and logic flows lacking necessary permission checks allow unauthorized access). Although many works studied the security of IoT platforms [15]–[29], their approaches cannot be directly applied to formally model or analyze open-source MQTT brokers. A most related work is MPInspector [15], which assesses multiple IoT messaging protocols. Essentially, MPInspector does not analyze source code and requires deploying and running brokers for dynamic testing, inheriting limitations of dynamic analysis for coverage (depending on the test cases) and scalability. Hence, an analysis approach such as ours (see below) that leverages information in the source code, using static analysis and formal model checking, will be complementary to the prior approaches [15], [27]–[29]. § 6.1 details our comparison with MPInspector including effectiveness and manual efforts (e.g., we can find authorization flaws, while MPInspector cannot).

Moreover, prior approaches for model construction and analysis [15], [30]–[39] generally cannot be directly applied to generate useful models for MQTT broker implementations, because (1) they lack definitions for formal models that can describe MQTT states and semantics; (2) they are not suited to abstract key operations and logic from source code that are relevant to key broker states.

Our solution: *MQTTactic*. In this paper, we design and develop *MQTTactic*, a semi-automatic tool that can formally verify MQTT broker implementations (§ 4). *MQTTactic* is based on static code analysis, formal modeling and automated model checking (with an off-the-shelf model checker Spin [40]). In the design, *MQTTactic* first formally defines a state-machine model for MQTT brokers, including the definition of what constitute key states of an MQTT broker's operation, what actions and operations that lead to state transitions (§ 4.2). Following the general model definition, which is implementation-agnostic, *MQTTactic* extracts model-relevant information from a specific broker's source code (based on static analysis and symbol execution) and then automatically translates the model information into a `concrete model` of the broker. A `concrete model` essentially is a highly abstracted (or simplified) implementation of the broker in the modeling language (e.g., Promela [41] in our implementation), keeping only execution paths that operate on or impact key broker states of interest (defined by the general model definition). *MQTTactic* then adopts Spin [40] for model checking, which verifies the model against a set of security properties. Our current security properties are related to authorization, and the model checking exhaustively visits the broker states by traversing the execution paths in all possible orders, and reports any states where unauthorized access occurs (e.g., an unauthorized users' message publishing or delivery indeed goes through the course — indicating insufficient permission check along the execution path). We validated the issues reported by *MQTTactic* through proof-of-concept exploit experiments (see results below). We release all source code and generated models online [42].

Results. Empowered by *MQTTactic*, we verify 7 popular open-source MQTT brokers and find 11 authorization-related logic flaws, including 7 zero-day flaws that bear novel, subtle logic issues never reported or fully understood before. The flaws allowed practical, unauthorized access to IoT devices, potentially directly impacting real vendors that adopt the open-source brokers, with serious security implications (with vendor acknowledgment, see our responsible disclosure in § 5.2). Our findings suggest that fully securing the MQTT systems is nontrivial — in the absence of security guidance and with the complex or customized MQTT logic flows, it is quite difficult for the developers to achieve complete mediation in the source code, leaving doors for the attackers to gain unauthorized access to the IoT devices. Our thorough evaluation (§ 6), including a detailed comparison with prior works, indicates that *MQTTactic* is effective and practical (with small amounts of manual efforts in configuration).

Contribution. We summarize our contributions as follows:

- *New techniques*. We designed and developed *MQTTactic*, including a set of novel techniques and designs that can formally verify MQTT broker source code for logic flaws. *MQTTactic* is capable of modeling MQTT brokers' internal operation states and generates highly effective model representation from a large code base (e.g., more than 10,000 LOC), enabling efficient model checking.
- *New understandings of security risk in IoT broker supply chain*. We performed the first (up to our knowledge) systematic study on open-source MQTT brokers, characterized their essential customization of the protocol. We showed that the customization of messaging flow and logic easily went wrong, especially for the frequent lack of necessary security checks at proper logic process. The 7 zero-day logic flaws brought to light new types of security-critical flaws, and also indicated the seriousness of IoT supply risk that stakeholders (open-source community, downstream vendors, users, and regulators) should be aware of.

2. Background

2.1. The Basics of the MQTT Protocol

MQTT [43] is a popular publish-subscribe messaging protocol for IoT. For two MQTT clients (a subscriber and a publisher) to communicate with each other, the MQTT server (broker) defines a `topic` to represent the subscriber's interested message category. The subscribers express their interest by sending `SUBSCRIBE` request with the `topic`

to the broker for receiving messages; after the publisher sends the message containing the `topic` using a `PUBLISH` packet to the broker, the broker then transmits the message to all the subscribers of the `topic`.

Note that, as per MQTT specification v3.1.1, when a client publishes a large number of messages to the same `topic` in an MQTT connection, the `topic` name would be repeated in all the `PUBLISH` packets, which causes a waste of bandwidth resources. To reduce resource consumption, MQTT specification v5.0 introduces a new feature: the `Topic Alias`, which is a 2-byte integer encoded as an attribute field in the `PUBLISH` packet. The client and broker can first negotiate a mapping relationship between the `Topic Alias` and the `topic`, such that all the subsequent `PUBLISH` packets over the same connection can carry the 2-byte `Topic Alias` to replace the original `topic`. Upon receiving packets with `Topic Alias`, the broker would use the previously built mapping relationships to retrieve the `topic` for these `PUBLISH` packets [44].

2.2. The Messaging Flows of MQTT

The three key processes in MQTT. The recent MQTT version 5.0 [11] defines 15 types of control packets along with the flags to designate the messaging flows. We briefly introduce the *connection*, *publishing*, and *subscription*, the three key processes in MQTT that cover most packet types.

• *Connection*. Three control packet types are used to define the messaging flow of connections: the `CONNECT` packet indicates a connection request from the client to the broker; the `CONNACK` packet is used to acknowledge the `CONNECT` request; the `DISCONNECT` packet is used for disconnection notification. Note that, the `CONNECT` packet should include the client's identifier (called the `ClientID`), which is unique among all clients. After the `CONNECT` request is accepted and acknowledged by the broker, a session (identified by the `ClientID`) will be created for the client. Moreover, a `CONNECT` request with an already existing `ClientID` would result in the broker creating a new session to take over the existing session and closing the existing session.

Further, the `Will` flag in the `CONNECT` packet is used to designate the broker to store a `Will` message and deliver it to the appropriate subscribers when an abnormal disconnection happens (e.g., network outage). The `CleanStart` flag defined in the `CONNECT` packet determines how to set up the new session. In specific, a `CONNECT` packet with `CleanStart` set to `true` would lead the broker to create a new session; if a `CONNECT` packet with `CleanStart` set to `false` and there is an existing associated session (the session with the same `ClientID`), the broker will resume the communications with the client based on the state of the existing session. This flag is usually used to recover the session of the client that is temporarily offline.

• *Publishing*. The `PUBLISH` packet is used to publish messages. There are two important flags in the `PUBLISH` packet. The `Retained` flag, if set, instructs the broker to store the message first and later deliver it to new subscribers automatically. The `QoS` flag indicates the level of assurance for message delivery — `QoS 0` indicates at most once delivery; `QoS 1` indicates at least once delivery; `QoS 2` indicates exactly once delivery. To enforce `QoS 1` messaging, the MQTT specification defines the `PUBACK` packet as an acknowledgment of the `QoS 1` message. Moreover, the `PUBREC`, `PUBREL`, and `PUBCOMP` packets are defined to enforce the `QoS 2` messaging (see Figure 1 in § 3.2).

• *Subscription*. The `SUBSCRIBE` and `UNSUBSCRIBE` packets are used to add/remove the subscription of the client who sends the request. The `SUBACK`/`UNSUBACK` packet is used to acknowledge the subscription/unsubscribing.

The asynchronous paradigm implied in MQTT. Generally speaking, the messaging flows defined in the MQTT specification imply asynchronous IoT messaging paradigms, logically. Specifically, the publishing from the publisher to the broker and the message delivery from the broker to the subscriber are decoupled and asynchronous — the ownership of a message transfers to the broker after the broker accepts it; after that, the delivery of the message (from the broker to the subscribers) usually does not rely on the subsequent behavior(s) of the publisher.

2.3. Threat Model

We consider realistic attack and application scenarios. We consider IoT systems that use MQTT brokers (in the cloud/server) to manage and mediate communication between clients (users and IoT devices). The administrator(s) (e.g., device owners) can grant other users (e.g., an Airbnb guest or an employee) access rights for IoT devices. The users' access rights are subject to revocation and expiration. We consider the broker and the devices are benign, while the users can be malicious and may attempt to escalate their existing privilege to access other devices that they are not entitled for. The malicious users can collect and analyze network traffic between the broker and their own (MQTT) clients, but cannot eavesdrop on or interfere with the communication between other users and the broker.

3. Customization of MQTT Implementation and A Motivating Example

In general, an MQTT broker is supposed to follow the logic and messaging flows defined in the protocol specification. However, we find that an actual implementation is much more complicated than the abstracted specification, always presenting a unique customization of MQTT (§ 3.1). This section also provides a motivating example (a zero-day vulnerability we found) to show how the customization can easily go wrong (§ 3.2).

3.1. The Customization of MQTT Implementation

We summarized three sources of customization, often essential in the MQTT protocol implementation.

• *Missed authorization model*. The MQTT specification mainly defines the logic of message transmission (§ 2.2) and provides little or no guidance on essential security models, in particular: (1) where in the messaging/logic process one should implement a security/permission check; (2)

what are sufficient security checks for the messaging/logic flows of MQTT. Lacking essential security specification in the protocol, we find that developers often implanted ad-hoc (unsound) authorization mechanisms somewhere in the MQTT messaging logic or missed necessary security checks — we elaborate on a real example below (Flaw 1) as a motivating example and more real examples in § 5.1.

• *Customized messaging logic flows*. Developers sometimes implement MQTT with customized logic flows that are not defined in the MQTT specification either in consideration of resource constraints or to facilitate customized features. For example, to prevent the client from sending duplicate `PUBLISH` packets in the `QoS 2` messaging, MQTT specification requires the broker to send a `PUBREC` packet to the client upon receiving and storing the message. However, we find that Mosquitto intentionally defers the `PUBREC` packet if its message queue is full (a kind of resource constraint, see **Flaw 7** in § 5.1)

• *Ambiguous definition*. There are several ambiguous definitions of the control flow in the MQTT specification. For example, MQTT specification v5.0 states that, when processing a `QoS 2` message, "*The receiver does not need to complete the delivery of the Application Message before sending the `PUBREC` or `PUBCOMP`.*" Hence, the time to deliver the `QoS 2` message to the subscriber(s) is implementation-specific, and an unsound implementation could lead to security loopholes, such as **Flaw 1** (see Figure 1).

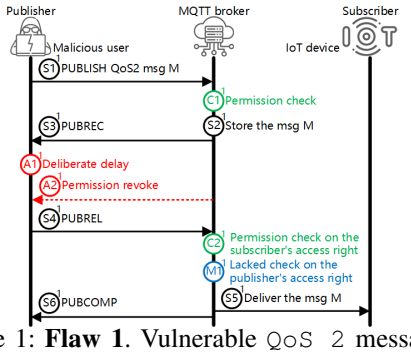


Figure 1: **Flaw 1**. Vulnerable `QoS 2` messaging

3.2. A Motivating Example

In the following, we first describe the `QoS 2` messaging flow defined in the MQTT specification, and then report a new security flaw we find. The flaw is due to the absence of necessary security checks in the `QoS 2` messaging flow, which unfortunately is not defined in the protocol and is thus difficult for developers to make right.

QoS 2 messaging flow. As shown in Figure 1, the MQTT specification defines the `QoS 2` messaging flow as follows.
Ⓢ1¹: The publisher sends a `QoS 2` message (`M`) to the broker included in a `PUBLISH` packet.
Ⓢ2¹: The broker stores the message `M` to process it later.
Ⓢ3¹: The broker confirms receiving the `QoS 2` message `M` by sending a `PUBREC` packet to the publisher.
Ⓢ4¹: Upon receiving the `PUBREC` packet from the broker, the publisher triggers the broker to deliver the message `M` to the subscriber with a `PUBREL` packet.
Ⓢ5¹: The broker delivers the message `M` to the subscriber.
Ⓢ6¹: The broker informs the publisher of the completion of message publication with a `PUBCOMP` packet.

Flaw 1: Timing manipulation for `QoS 2` delivery. To secure `QoS 2` messaging, Mosquitto first checks (step Ⓒ1¹) the publisher's permission of publishing the message `M` before accepting `M`. Then, it checks (Ⓒ2¹) the subscriber's permission of receiving `M` before delivery. While such two steps of security checks may appear to be reasonable to many, we find they are actually insufficient and incommensurate with the complicated logic of `QoS 2` flows.

Specifically, although the MQTT messaging flows are generally considered asynchronous (the publishing from the publisher to the broker and the delivery from the broker to the subscriber are considered asynchronous, see § 2.2), we find that the delivery timing of a `QoS 2` message is somewhat controllable by the publisher — at least partially breaking the "asynchronous" expectation. More specifically, consider the smart home application of MQTT where there could be users with malicious intentions (a common and well-accepted IoT application scenario [29], [45], [46]), the publisher (e.g., a malicious user) could deliberately delay (step Ⓐ1¹) the `PUBREL` packet in step Ⓢ4¹ to control the time when the broker actually delivers `M` to the IoT device, since the `PUBREL` packet works as a trigger to the message delivery in the broker. If the delivery happens after the administrator (e.g., an Airbnb host) revokes the publisher's (e.g., an Airbnb guest) access right (step Ⓐ2¹), the publisher could still control the IoT device after he loses access right.

Discussion. **Flaw 1** indicates that it is nontrivial to securely implement the MQTT protocol with sufficient security guards in the logic steps of the messaging flows. Hence, there could be many more authorization-related flaws in the real-world MQTT brokers, considering the large number of open-source brokers and the complexity of MQTT messaging flows within brokers (§ 3.1). Further, given the worldwide usage of open-source MQTT brokers in production, it is imperative to systematically study this problem and come up with approaches and readily available tools that can automatically identify the flaws in the brokers at scale.

Problem scope. In this paper, we focus on how to identify authorization-related logic flaws in the open-source MQTT brokers with elevated level of automation. Our goal is, instead of verifying the entire source code space, to design a general approach to abstract formal models from the source code (e.g., MQTT brokers) based on logic perspectives of our interest (i.e., necessary authorization checks and constraints in the messaging process), and verify the derived models against generalized security properties. This will enable us to find type-/domain-specific (e.g., authorization-related) logic flaws, and rule out such categories of issues based on a new level of formal guarantee. Other types of flaws (e.g., flaws leading to crashes because of processing malformed messages) are considered out of scope.

4. System Design and Implementation

In this section, we elaborate on the design and implementation of *MQTTactic*, our semi-automatic tool for

security analysis of MQTT broker implementations.

4.1. Overview

At a high level, MQTT brokers in IoT applications should ensure that any unauthorized client cannot send messages (commands) to other clients or receive messages from the broker. To detect the security flaws for many different open-source MQTT brokers, our proposed general approach is to construct a state transition model facilitated by source code for each MQTT broker, and to use a model checker to verify whether the pre-defined security properties hold in the model. The model checker reports a security flaw for each violation to the security properties. The reported flaws are then validated manually through our end-to-end deployments and execution of the MQTT brokers.

Architecture of *MQTTactic*. As mentioned earlier (§ 3.1), each real-world MQTT implementation is a unique customization to the MQTT specification. As a result, we cannot use a single unified model to describe all MQTT implementations. Therefore, our approach is to customize a unique model for each MQTT broker implementation by extracting implementation-specific information (e.g., control flows) from the broker's source code. Also recall that, due to the large code footprints and complicated control flows of the MQTT implementations, to fully abstract and model the actual behaviors in an MQTT implementation is intractable. Therefore, we abstract the control flows with their logical constraints in the source code that are relevant to the broker state machine (formally defined for MQTT in § 4.2), while dropping those non-related implementation details. Note that, as also shown by prior works [45], [47], one can always enrich a model progressively to account for additional details that were dropped in an initial model for increased coverage (e.g., for extended security properties).

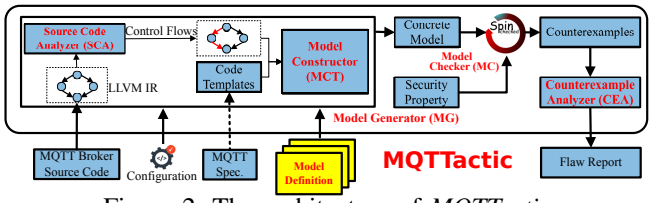


Figure 2: The architecture of *MQTTactic*

To this end, as outlined in Figure 2, we design *MQTTactic*, which includes 3 key components: a general *model definition* (§ 4.2), a *model generator* (**MG**, see § 4.2 and 4.3), and a *model checker* (**MC**, see § 4.4). The `model definition` provides an general state-machine model for MQTT brokers, including the definition of what constitute key states of a MQTT broker's operations, what actions and operations that lead to state transitions (§ 4.2). Following the general model definition which is implementation-agnostic, the **MG** takes as input the broker source code and a `configuration` file (see § 4.5) to generate a specific `concrete model` for the MQTT broker. Specifically, **MG** first extracts model-relevant information from a specific broker's source code (done by a sub-component called *source code analyzer* or **SCA**, based on static analysis and symbol execution). **MG** then automatically translates the model information into a *concrete model* of the broker — done by a sub-component *model constructor* or **MCT**. A `concrete model`, model of the specific broker, essentially is a highly abstracted (or simplified) implementation of the broker in the modeling language, keeping only execution paths that operation on or impact key broker states of interest (defined by the general model definition).

To detect security vulnerabilities, the **MC** then verifies the `concrete model` against the pre-defined security properties (§ 4.4). The **MC** reports a counterexample for each violation to the properties it finds. The counterexample would record the state transitions from the initial state to the abnormal state where the violation happens and the corresponding sequence of actions that drive these state transitions. The counterexamples output by the **MC** are then analyzed by the *Counterexample Analyzer* or **CEA**, which mainly filters duplicated results. Each result is then manually validated through PoC experiment.

4.2. Formal Definition for MQTT Brokers

Definition for a state-machine model. We model an MQTT broker as a state transition system $\mathcal{M}$ = ($\mathcal{V}$, $\mathcal{O}$, $\mathcal{S}$, s_0, $\mathcal{A}$, δ, $\mathcal{SP}$). Here $\mathcal{V}$ is a finite set of variables (e.g., subscription, message queue, listed in Table 2) whose values collectively constitute the states of the MQTT broker; $\mathcal{O}$ is the set of generalized low-level operations (e.g., `read`, `write`, see below) on the variables in $\mathcal{V}$; $\mathcal{S}$ is the set of states, at each of which the MQTT users/clients and MQTT broker can perform actions (e.g., sending a `PUBLISH` request and delivering a message to clients); s_0 ($s_0 \in \mathcal{S}$) is the initial state where no actions have been performed in the system; $\mathcal{A}$ is a finite set of client actions that can be performed by a user/client resulting in a state transition of the broker. $\mathcal{A}$ includes (1) common MQTT actions defined in the specification (e.g., `CONNECT`, `PUBLISH`, `SUBSCRIBE`) and (2) permission-related actions (e.g., add or revoke a user's access to a particular `topic`, detailed below); δ is a transition function that drives the system transit from one state to the next upon actions being executed; $\mathcal{SP}$ is the set of security proprieties where each sp ($sp \in \mathcal{SP}$) needs to be verified in the model $\mathcal{M}$. We further elaborate on the formal definition of key elements in model $\mathcal{M}$ as follows.

• *The variable set* $\mathcal{V}$. We define $\mathcal{V} = \{v_1, v_2, ..., v_n\}$ which include (1) variables defined by the MQTT specification that comprise part of the broker states ; (2) security policy (i.e., an access control policy) maintained by the broker (see the full list in Table 2). For example, based on MQTT specification, the MQTT broker would maintain retained messages associated with MQTT topics. Hence, we use $v_{RetainedMsg}$ in $\mathcal{V}$ to record the retained messages. Also, brokers normally record the security policy (see its model below) for permission checking. Therefore, the $v_{Permission}$ in $\mathcal{V}$ is defined to record the access rights of all clients/users.

Since the variables with their values collectively represent broker states, we call them *state variables*. For finer-grained modeling, we further consider the state

variables as either *session variables* or *global variables* (Table 2): (1) session variables such as $v_{WillMsg}$ are related to individual sessions (a session is specific to and generally identified by the client's ID, see § 2.2); (2) global variables such as $v_{RetainedMsg}$ are broker-global (not session/client specific). In our model implementation (§ 4.3), the broker can have, for example, multiple $v_{WillMsg}$s related to multiple clients/sessions.

• *The operation set* $\mathcal{O}$. An operation o_i in $\mathcal{O}$ is defined generally as `read`, `write`, or `deliver` on a state variable in $\mathcal{V}$ (see Table 3). With respect to the `write` operations, since a state variable can be a list of elements (e.g., the variable v_{Subs} records MQTT `topic`s that the client has subscribed to), `write` can be either `add` or `remove` elements to the variable. The `read` operation means accessing the resource represented by the variable. The `deliver` operation operates on a message related state variable indicating that the broker delivers the message to subscribers. Therefore an operation is denoted as o_{read}, o_{write}, o_{add}, o_{remove}, or $o_{deliver}$. An operation can be further labeled with the variable being operated, such as o_{sub_add} (denoting an `add` operation on the variable v_{Subs}).

• *The state set* $\mathcal{S}$. A state s_j ($s_j \in \mathcal{S}$) for the MQTT broker is defined by the values of all state variables $\mathcal{V}$; $s_j = \{v_1^j, v_2^j, ..., v_n^j\}$. The state changes when the value of any variable changes.

• *The client action set* $\mathcal{A}$. $\mathcal{A}$ starts with a finite set of actions: $\mathcal{A}$ = { `CONNECT`, `PUBLISH`, `PUBREL`, `SUBSCRIBE`, `UNSUBSCRIBE`, `DISCONNECT` } (with two more actions `AUTHORIZE` and `REVOKE` to add, see below). It is a subset of control packets defined in the MQTT specification. We exclude the control packets that are unrelated to authorization. For example, we exclude the `AUTH` packet, which is used for authentication exchange between the client and broker, and is irrelevant to our goal. Note that, the execution of a client action can change the states of the broker, i.e., by resulting in value changes to the variables in $\mathcal{V}$. Intuitively, for example, the client can perform a `SUBSCRIBE` action, which could result in the variable v_{Subs} being updated.

Moreover, we abstract the authorization-related behaviors commonly found in MQTT broker implementations as two general actions — the `AUTHORIZE` action for access right authorization and the `REVOKE` action to remove certain rights from the clients/users, and include them into $\mathcal{A}$. These two actions can change the security policy (abstracted as the state variable $v_{Permission}$) maintained by the MQTT broker.

Further, by abstracting the protocol specification, a client action is represented as an ordered sequence of *operation*s in $\mathcal{O}$ that the broker is supposed to execute in response to the action. For example, the action `SUBSCRIBE` is defined as $o_{sub\_add} \rightarrow o_{retained\_read} \rightarrow o_{deliver}$, which means that, when the broker receives a `SUBSCRIBE` packet from the client (who aims to subscribe to a `topic`), the broker would execute the o_{sub_add} operation to update the subscription (an o_{add} to the v_{Subs} variable), execute the $o_{retained\_read}$ operation to obtain a `Retained` message for the `topic` (read the $v_{RetainedMsg}$ variable), and execute the $o_{deliver}$ to send the `Retained` message to the client. Table 6 in Appendix A.8 details the *operation* sequences for each action.

• *The transition function* δ. We define the transition function as δ: $\mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, which drives the transition from one state to the next with the execution of an action in $\mathcal{A}$. For example, $\delta(s_0,$ `CONNECT`$) = s_i$ (where s_0, $s_i \in \mathcal{S}$ and `CONNECT` $\in \mathcal{A}$) indicates that the MQTT broker transfers from the initial state (s_0) to the state s_i in which the MQTT broker creates a session for the newly connected client.

• *The security property set* $\mathcal{SP}$. $\mathcal{SP}$ records all the security properties that will be verified in the model $\mathcal{M}$ (§ 4.4).

Common authorization model for MQTT implementations. Based on our survey of 7 popular MQTT implementations, almost all MQTT brokers use a security policy that can be modeled as a three-tuple $\{client, \texttt{topic}, r\}$ to indicate that the subject *client* is authorized to access the object `topic` with the access right r.

• *The subject client*. Specifically, several brokers define the *client* as the identifier of the client (`ClientID`), while other brokers use an `RBAC` (role based access control) mechanism — each client is assigned with a role R and the role R is used as the *client* in the $\{client, \texttt{topic}, r\}$.

• *The object* `topic`. Usually, the MQTT brokers use `topic`s to represent the resources. For example, a smart home system may authorize users with different `topic`s for them to access/control different smart home devices.

• *The access right* r. The access right r specifies the action type on the `topic`. The most commonly used access rights are the `SUBSCRIBE` right and the `PUBLISH` right, which indicate whether a client is authorized to send `SUBSCRIBE` and `PUBLISH` requests.

To enforce authorization, MQTT brokers store the 3-tuples as the security policy. Typically, the brokers use a configuration file or a database to store the security policy and provide APIs to check and update the security policy (with the policy abstracted as $v_{Permission}$, $v_{Permission} \in \mathcal{V}$).

4.3. Implementation-Specific Model Construction

This section elaborates on the approach (the **MG** part of *MQTTactic*) to generate Promela code that represents the model for specific brokers. As mentioned in § 3.1), each MQTT implementation is a customization to the protocol. To construct a model that actually describes a specific implementation, we need to abstract the implementation-specific control flows with logical constraints that can impact the broker states and state transitions. Our approach entails first following the high-level model definition (§ 4.2) to extract model-relevant information from a specific broker's source code (§ 4.3.1) and then translating the model information into a `concrete model` (§ 4.3.2).

4.3.1. Extracting Model-Relevant Information from Broker Implementations. *MQTTactic* first extracts (from broker source code) model information that can describe a `concrete model` for a specific broker (based on the model definition). The model information to extract includes (1) unique program-paths (along the control flows, after

pruning invalid paths, detailed below); (2) each program-path characterized with an ordered sequence of operations (e.g., reads/writes on state variables and thus impacting broker states) — we abstract such a unique program-path as Effective Path Type (detailed below). More specifically, on the control flows starting from an entry point of the broker (usually being a handler function corresponding to an MQTT client action a ($a \in A$, see Table 6) such as `PUBLISH` or `CONNECT`), our approach extracts its different program-paths and then Effective Path Types (denoted as a set EPT_a), and for each Effective Path Type ept ($ept \in EPT_a$), abstracts its sequence of operations on the state variables (e.g., [o_{sub_read}, o_{will_read}, $o_{deliver}$]). Note that we remove invalid program paths (unreachable) based on symbolic execution (detailed below). We elaborate on key technical steps as follows.

Step 1: Identifying action entry points and state-impacting code-blocks from implementation. A client action a ($a \in \mathcal{A}$) triggers state transitions, and each broker implementation has a set of entry functions (also called *action handler* or *handler*) for handling specific actions. A typical broker has the handlers for all actions in $\mathcal{A}$. Starting from an action handler, *MQTTactic* analyzes the code basic blocks along the (inter-procedural) control flows (based on its LLVM IR, see § 4.5) to identify those blocks that impact the broker states $\mathcal{S}$, called *Key Basic Blocks*.

Def. 1: Key Basic Block (KBB). A *Key Basic Block* is a basic block [48] in the broker's ICFG [49] that performs at least one operation o ($o \in \mathcal{O}$) on a state variable v ($v \in \mathcal{V}$) (or multiple state variables). Notably, an update to v changes the broker state (see the model definition in § 4.2).

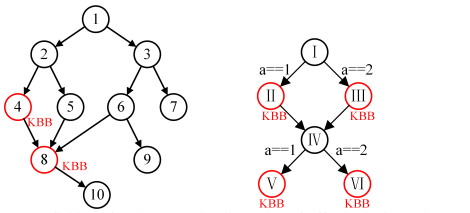
Taking Figure 3 (a) as an example, starting from basic block 1, we identify the basic block 4 and 8 as KBBs. To find all KBBs on the LLVM IR, we leverage context-sensitive pointer analysis (by adopting the off-the-shelf tool SVF [50]) to find out all basic blocks that impacted a variable v ($v \in \mathcal{V}$, *MQTTactic* currently relies on a simple configuration to map out all the 7 variables in $\mathcal{V}$ to variables at the code level, see § 4.5).

Step 2: Modeling implementation-specific control flows for the actions. For each action a ($a \in \mathcal{A}$), our modeling goal is to abstract how the action impacts the broker states (i.e., leading to state transitions or updates to any variable v, $v \in \mathcal{V}$). To this end, *MQTTactic* starts from its handler and abstracts all its control flows with KBBs. In our context, those non-KBB basic blocks are considered not to impact the broker states. Hence, for example, in Figure 3 (a), we consider the program paths `1-2-5-8-10` and `1-3-6-8-10` have the same impact (due to KBB 8) on the broker state. To facilitate the modeling of the implementation-specific actions, we introduce the definition of *Path Type* as follows.

Def. 2: Path Type. For an MQTT broker implementation, one *Path Type* is the category of paths with a unique sequence of KBB(s). Specifically, consider the ICFG in Figure 3 (a) where the basic block `4` and basic block `8` are KBBs. Although there are five paths in total (from block `1` the root to a leaf block), there are only three *Path Type*s: (1) the path that includes the KBB sequence {`4`, `8`}, i.e., the path `1-2-4-8-10`; (2) the paths that include the KBB sequence {`8`}, including the path `1-2-5-8-10` and the path `1-3-6-8-10`; (3) the paths that include no KBB, including the paths `1-3-6-9` and `1-3-7`.

Based on Def. 2, our modeling of each action in the broker implementation comprises all its *Path Type*s. For example, in Figure 3 (a), the action that starts with block `1` is modeled as three unique KBB sequences: {KBB 4, KBB 8}, {KBB 8}, {none KBB}. Notably, our modeling removes a *Path Type* if its underlying program path is invalid or unreachable (based on symbolic execution, see auxiliary step 2 below), and we consider those valid as *Effective Path Type*s.

Further, based on Def. 1, each KBB includes code statements that perform operations on one or more state variables. Accordingly, in our modeling, each Effective Path Type ept, with its sequence of KBBs, are further abstracted as an ordered sequence of operations (e.g., [o_{sub_read}, o_{will_read}, $o_{deliver}$]), with each operation $o \in \mathcal{O}$. We show a real example in Appendix A.10, with code snippets of KBBs in the hmq broker along one of its Effective Path Type, and how the code is abstracted as the ordered sequence of operations. The above are core steps to extract model information that describe implementation-specific messaging flows. We refine and supplement the information with two auxiliary steps below.



(a) Definition of Path Type (b) Illustration of Effective/False Path Type

Figure 3: The KBB and Path Type

• *Auxiliary step 1: Abstracting permission-related path constraints*. As mentioned in § 3.1, although the MQTT protocol lacks security-related specification, real-world brokers implanted implementation-specific permission checks somewhere along the control flows, i.e., in certain KBBs (by checking/reading the security policy represented by variable $v_{Permission}$). Specifically, these permission checks are in the form of invocation to the broker's authorization APIs (see § 4.2). Those permission checks present constraints for the control flows between KBBs. For example, in Figure 3 (a), the KBB `4` includes a permission check against the subscriber for its `SUBSCRIBE` right (e.g., ⓒ2[1] in Figure 1), and the KBB `8` delivers the message (e.g., ⓢ5[1] in Figure 1). Here the control flow from KBB `4` to `8` includes a constraint, which we model as a 3-tuple {*subscriber*, `topic`, `SUBSCRIBE`}, the same as the modeling for security policies (see § 4.2). That is, the broker checks the security policy ($v_{Permission}$) at KBB `4`, and if the $v_{Permission}$ includes the permission — the above constraint, the execution then flows to KBB `8` to deliver the message.

• *Auxiliary step 2: Pruning the Path Types*. As mentioned above, we filter the Path Types that are invalid or unreachable. This is because all program paths are obtained from

the LLVM IR (i.e., ICFG), and certain paths might not be reachable at runtime due to their constraints not being fulfilled. Take the ICFG in Figure 3 (b) as an example, where the basic blocks II, III, V, and VI are KBBs. The figure also shows the runtime constraints (e.g., the constraint of $\{a = 1\}$ on the edge $\{I \rightarrow II\}$). Based on the constraints, the execution path {KBB II, KBB VI} is false (cannot happen at runtime) due to constraint conflicts (with both $a = 1$ and $a = 2$). Hence, we exclude such false paths based on their symbolic constraints, which are obtained by adopting off-the-shelf symbolic execution tool Haybale [51]. More specifically, for each Path Type pt identified above, we perform the symbolic execution on the LLVM IR to look for at least one true path that includes the KBB sequence of pt, such that pt is valid (can happen at runtime). We set the symbolic execution timeout as 30 minutes for each Path Type, which is evaluated to be effective (see § 6).

4.3.2. Translating Model Information to Concrete Model Implementation. Based on the above extracted model information, our approach generates the Promela representation of the `concrete model` for a specific broker $Broker$. The model information includes: (1) the list of supported client actions, denoted as set A_{Broker}, which can include elements such as `CONNECT`, `PUBLISH`, and `PUBREL` ($A_{Broker} \subset \mathcal{A}$); (2) for each action a ($a \in A_{Broker}$), its different Effective Path Types (denoted as a set EPT_a); (3) for each Effective Path Type ept ($ept \in EPT_a$), its sequence of operations on specific state variables, denoted as seq (e.g., [o_{sub_read}, o_{will_read}, $o_{deliver}$]). Hence, the *MQTTactic* will translate this information into a `concrete model` in the Promela language for model checking.

Based on such model information, we summarize the Algorithm 1 and Algorithm 2 to generate the Promela code of the `concrete model`. Particularly, as shown in Algorithm 1, in the `concrete model` implementation (in Promela), each client action a is implemented as either (1) one Promela function (if it has only one Effective Path Type, indicating only one kind of operation sequence impacting broker states/state-variables) or (2) multiple Promela functions corresponding to different Effective Path Types in the broker for the client action. More specifically, Algorithm 2 shows that we translate a specific Effective Path Type ept, i.e., a sequence of operations on specific state variables denoted as seq, into a Promela function by sequentially assembling Promela code templates (pre-defined) corresponding to each operation o ($o \in seq$). Notably, each code template describes the operation (`read`, `write`, or `deliver`) performed on a particular state variable v ($v \in \mathcal{V}$). We show a few real examples of code templates in Appendix A.10 (see the full list of templates online [42]).

Further, by adapting the implementation for the Spin model checker, we define a snippet of skeleton code in Promela (see Listing 7 in Appendix A.10) , which includes the "main" function for Spin to run. Essentially, the skeleton code outlines the client actions that a client can run, such as `CONNECT` and `PUBLISH`; it also defines the handler functions (with just placeholders) of the broker in response to each client action. The placeholders are populated with the above generated (by Algorithm 1 and 2) Promela functions (corresponding to all different Effective Path Types of all supported client actions of the broker). When Spin runs, it exhaustively attempts all the Promela functions of all the client actions (simulating that the client performs those actions as many as possible in diverse orders), and thus traverses the broker states automatically.

To implement the model with multiple concurrent clients, in the skeleton code, we launch 2 processes for 2 publisher clients and 1 process for 1 subscriber client. Spin runs each process (representing each client) in parallel, and helps the client randomly invoke any of the support client actions. Note that one can easily extend the implementation to include more clients. We provide a running example for `concrete model` generation in Appendix A.10.

Algorithm 1 Generate Promela Functions for Client Actions

Input: A_{Broker}, the list of supported client actions;
Input: $\{EPT_{a_i} | a_i \in A_{Broker}\}$, where EPT_{a_i} is the set of Effective Path Types for action a_i.
Output: $\{F(a_i) | a_i \in A_{Broker}\}$, where $F(a_i)$ is the set of generated Promela functions for action a_i (handler functions).

$N \leftarrow length(A_{Broker})$
for $i = 1, 2, ..., N$ **do**
  $F(a_i) \leftarrow \emptyset$
  for ept **in** EPT_{a_i} **do**
    // generate Promela function for each Effective Path Type
    $f \leftarrow translateSingleEPT(ept)$
    $F(a_i) \leftarrow F(a_i) \cup f$
  end for
end for
return $\{F(a_i) | a_i \in A_{Broker}\}$

Algorithm 2 Generate Promela Function for each Path Type

Input: ept, a specific Effective Path Type, which includes a sequence of operations $seq = [o_1, ..., o_n]$, $o_i \in \mathcal{O}$.
Output: f, a single Promela function representing the ept.

function $translateSingleEPT(ept)$
  $C \leftarrow \emptyset$ // C is a set of code snippets
  $seq \leftarrow getOperationSequence(ept)$
  for o **in** seq **do**
    // generate code snippet for each operation in Effective Path Type
    $code\_snippet \leftarrow generateCode(o)$
    $C \leftarrow C \cup code\_snippet$
  end for
  // assemble code snippets to construct a Promela function
  $f \leftarrow assemble(C)$
return f
end function

4.4. Vulnerability Discovery with Model Checking

With an MQTT broker's `concrete model` specified in Promela, we use an off-the-shelf model checker Spin [40] to verify the model against a set of security properties.

Definition of security properties. We generalize the following three security properties, based on our high-level security goal and recent security analyses [15], [29].

- *sp_1: A client C_{rcv} that is receiving a message `m` from the broker should have the right to read the message `m`.*
- *sp_2: A client C_{snd} should have the right to send the message `m` when the broker accepts the message `m`.*
- *sp_3: The client C_{trg} that causally triggers the broker to send a message `m` to other client(s) (subscribers) should have the right to send `m` when `m` is delivered to the MQTT subscribers.* The client C_{trg} is called a *trigger*.

Model checking. A `concrete model` essentially is a highly abstracted (or simplified) implementation of the broker in the modeling language (§ 4.3.2), keeping only execution paths that operate on or impact the state variables (thus impacting broker states of our interest). A `concrete model` includes multiple handler functions (generated by Algorithm 1, corresponding to the client actions): upon a client action a_i (e.g., `PUBLISH`), Spin runs one handler function related to a_i (there can be multiple handler functions for a_i corresponding to different Path Types in the broker, see Algorithm 1), which updates or operates the state variables. Spin can simulate multiple concurrent clients (publishers, subscribers): a client can randomly perform a client action, and Spin runs one of the corresponding handler functions. Hence, Spin helps traverse potentially all different sequences of client actions to visit potentially all possible states of the broker.

To verify authorization related properties, *MQTTactic* enhances the `concrete model` to additionally maintain a new data *D* to keep track of ground-truth of each client's authorized rights. When Spin runs the `concrete model`, *MQTTactic* randomly changes each client's permissions between client actions (similar to `AUTHORIZE` and `REVOKE` actions on a real broker). Due to permission changes, two sequential client actions of the same type (e.g., `PUBLISH`) may not both go through the same state transition (e.g., to the point of $o_{deliver}$ in a handler function) should the execution path include permission related constraint — permission check (Auxiliary step 1, § 4.3.1).

Based on our security properties to discover insufficient permission checks in the `concrete model`, we add two assertions (checking whether the client has the permissions based on ground-truth *D*) at any message delivery ($o_{deliver}$). One assertion requires the message sender or trigger (identified by `ClientID`) with the PUBLISH permission; the other assertion ensures the message recipient with the SUBSCRIBE permission. See real assertion and model code online [52]. When Spin traverses potentially all different sequences of client actions to visit potentially all broker states, any violation to the assertions indicates a flaw.

Taking **Flaw 1** as an example (Figure 1), the publisher client (who performs the action `PUBREL` in ⑭[1]) is expected to have the right `PUBLISH`. However, the model checker find a violating state (counterexample): after the client lost the `PUBLISH` right, it performed the ⑭[1] step and led to the ⑮[1] step of the broker (the broker lacks the permission check Ⓜ[1] for the publisher client and let through the operation to ⑮[1], violating sp_3).

4.5. Implementation Details and Discussion

Code analysis and converting broker source code to LLVM IR. In § 4.3.1, *MQTTactic* performs static analyses on source code. To analyze brokers developed in different programming languages (e.g., C, C++, and Rust), we first convert the broker source code into LLVM IR [53]. Specifically, we adopted off-the-shelf LLVM-based/supported tools to generate the LLVM IR for different languages (Clang/-Clang++ [54] used for C/C++, Gollvm [55] for Golang, and Rustc [56] for Rust). We provide detailed technical guidance and examples for LLVM IR generation online [57], which include environment configuration, necessary commands to run the tool, and real examples (illustrating the translation from real code snippets of MQTT brokers to LLVM IR). *MQTTactic* further adopted SVF [50] and haybale [51] to identify KBB and analyze control flows on the LLVM IR.

Full release of *MQTTactic* source code and concrete model implementation. We fully release artifacts of this study online [42], including (1) the *MQTTactic* source code (5,500 LOC in total), and (2) the `concrete models` implemented in Promela for 7 brokers (see broker names and version in Table 1). [42] further includes a demo to run *MQTTactic* with FlashMQ.

The `configuration` to run *MQTTactic*. To inspect a specific MQTT broker implementation, *MQTTactic* expects a simple `configuration` file to map (1) the 7 state variables (Table 2) defined in the model (§ 4.2) to corresponding variables in the broker code; (2) the 8 client actions (e.g., `CONNECT` and `PUBLISH`, see Table A.8) to handler functions implemented by the broker. For example, for the FlashMQ broker, the configuration file maps the `CONNECT` action to the handler function named $handleConnect$, which is the entry point to handle clients' `CONNECT` actions. We additionally leverage the `configuration` file to list the authorization function(s) in the broker. We show the `configuration` file for the FlashMQ broker in Appendix A.5. We evaluate the practical manual efforts for preparing `configuration` in § 6.1.

Also, the `configuration` for a broker implementation does not expect frequent changes along with broker updates or version changes. In particular, a broker's interface-functions (e.g., the handlers) rarely change (based on the Open-Closed Principle [58]). As an evaluation on 7 brokers, by inspecting more than 350 Git commits including 5 major version updates, we find that their `configuration` expected almost no changes. Also, any alterations to variable names or function names could be easily tracked and captured from Git commits.

One-time efforts for code templates and limited impact of MQTT versions. We leverage one-time efforts to develop Promela code templates for the operations and a skeleton code, which are reused for constructing `concrete models` of different MQTT broker implementations. They are also general across major MQTT versions used in the wild, being usable and effective for both MQTT v3.1.1 and v5.0. Note that the MQTT specification comes with a "Summary of new features in MQTT v5.0", and the new features added in MQTT v5.0, typically supporting backward compatibility, do not change the core messaging flow and operations. It took us about 3 domain-expert days to identify the state variables in the MQTT specification, implement the code templates and skeleton in Promela.

5. Security Flaws and Measurement of Impact

We ran *MQTTactic* on 7 popular open-source brokers developed in 4 different languages, and identified 11

authorization-related logic flaws, including 7 zero-day flaws (see Table 1) and 4 existing flaws (see Appendix A.6). Notably, we only evaluated a subset of the 70+ open-source brokers (see Appendix A.4), selecting only those having implemented dynamical authorization mechanisms and written in the programming languages supported by *MQTTactic* currently (i.e., C/C++/Go/Rust). Further, in our measurement study (§ 5.2), we manually investigated 13 other popular brokers — 7 commercial brokers from leading IT companies and 6 open-source brokers (written in Java, JS, or Erlang). Results have shown the pervasiveness of the 0-day flaws identified by *MQTTactic*.

5.1. New (0-day) Flaws Identified by *MQTTactic*

Flaw 2: Vulnerable delivery retry in `QoS 1` messaging. The MQTT specification defines the `QoS 1` messaging as "at least once delivery" in terms of delivery assurance (see § 2.2), with the following control flow (see Figure 4): if the subscriber is **online**, the broker delivers the message to the subscriber without delay (Ⓢ1²-Ⓢ2²-Ⓢ3²-Ⓢ4²′). If the subscriber is **offline**, the broker would retry to deliver the message when the subscriber reconnects (Ⓢ1²-Ⓢ2²-Ⓢ3²-Ⓢ4²-Ⓢ5²Ⓢ6²). Most brokers add a permission check (Ⓒ1²) before accepting the incoming message `M`, ensuring the publisher/subscriber has the access right to send/receive the message `M`. Still, *MQTTactic* reports a possible violation to the security property sp_2 — when the broker delivers the message `M`, the publisher (i.e., the malicious user) does NOT possess the right to send the message `M` — with the attacking action sequence of Ⓐ1²-Ⓢ1²-Ⓒ1²-Ⓢ2²-Ⓢ3²-Ⓢ4²-Ⓐ2²-Ⓐ3²-Ⓢ5²-Ⓢ6².

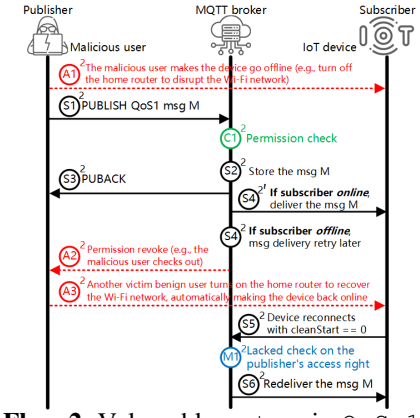


Figure 4: **Flaw 2**. Vulnerable `retry` in `QoS 1` messaging

Flaw 3: Vulnerable delivery retry in `QoS 2` messaging. Due to the "exactly once delivery" feature (see § 2.2), there is a similar `retry` mechanism in `QoS 2` messaging, which has the same security implication of that in `QoS 1` messaging and leads to **Flaw 3** in the `QoS 2` messaging. For simplicity, we omit the redundant description on **Flaw 3** here (see Appendix A.1 for the detailed description).

Flaw 4: Unguarded usage of the `Topic Alias`. MQTT specification v5.0 uses `Topic Alias` to reduce the resource consumption of MQTT messaging in that using the 2-byte integer `Topic Alias` to represent the (possibly long) `topic` name (see § 2.1). As shown in Figure 5, the control flow of the `Topic Alias` enabled MQTT messaging is defined as Ⓢ1⁴-Ⓢ2⁴-Ⓢ3⁴-Ⓢ4⁴-Ⓢ5⁴. Again, though not explicitly specified in the MQTT specification, the brokers would usually add a permission check (Ⓒ1⁴) before accepting and delivering the first message `M1`. However, *MQTTactic* find such security practice has not been completely enforced to guard all the `PUBLISH` packets, reporting a flaw with the attacking sequence of Ⓢ1⁴-Ⓒ1⁴-Ⓢ2⁴-Ⓢ3⁴-Ⓐ1⁴-Ⓢ4⁴-Ⓢ5⁴ in the VolantMQ broker. That is, the broker does NOT check the publisher's permission when processing the subsequent `PUBLISH` packets only carrying the `Topic Alias` (i.e., message `M2`).

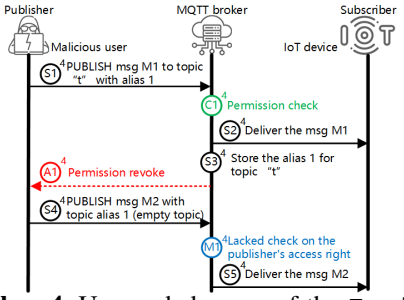


Figure 5: **Flaw 4**. Unguarded usage of the `Topic Alias`

Flaw 5: Insecure session taking over. When receiving a new `CONNECT` request with an already existing `ClientID`, the broker creates a new session (with the same `ClientID`) to take over the existing session, during which, the broker would deliver the `Will` message (if any) of the existing session (see § 2.2). The session taking over process of Mosquitto goes as Ⓢ1⁵-Ⓒ1⁵-Ⓢ2⁵-Ⓢ3⁵-Ⓢ4⁵-Ⓒ2⁵-Ⓢ5⁵ in Figure 6, where Ⓒ1⁵ and Ⓒ2⁵ are the permission checks added by Mosquitto. Despite such security enforcement, *MQTTactic* still reports a flaw during session taking over where a malicious user (who is able to obtain the `ClientID` of others [29]) can trigger the delivery of a `Will` message containing the topic that he is not entitled to access (violating the security property sp_3).

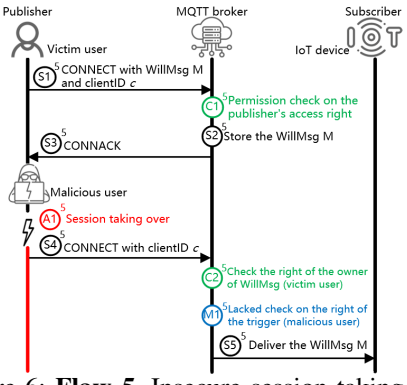


Figure 6: **Flaw 5**. Insecure session taking over

Flaw 6: Unvetted `Will` message. When a client sends a `CONNECT` packet containing a `Will` message, the broker would store the `Will` message and later deliver the `Will` message to the appropriate subscribers when the client goes offline accidentally (e.g., network outage). In the absence of

security guidance, we find the `Will` messaging in several brokers is vulnerable. For instance, hmq [59] neither checks whether the client is authorized to `PUBLISH` message to the `topic` contained in the `Will` message when accepting/storing the `Will` message (Ⓜ1⁶ in Figure 7) nor when delivering it (Ⓜ2⁶), enabling any (unauthorized) client to send commands to any `topic` arbitrarily.

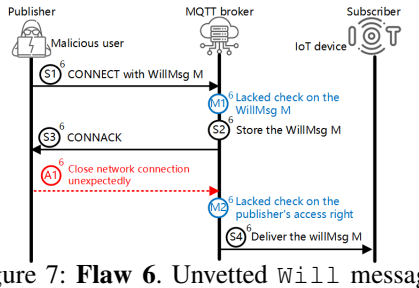


Figure 7: **Flaw 6**. Unvetted `Will` message

Flaw 7: Vulnerable message caching. Mosquitto uses a homegrown `cache-before-process` mechanism to limit the resource consumed by a single publisher during `QoS 2` messaging. In specific, Mosquitto stores the unfinished valid `QoS 2` messages in its *InflightQueue* (with a finite capacity of n, $n = 20$ by default) first (Ⓢ1⁷-Ⓒ1⁷-Ⓢ2⁷-Ⓢ3⁷ in Figure 8). After the *InflightQueue* is full, the new valid `QoS 2` messages will be cached in the *CacheQueue* (Ⓢ4⁷-Ⓒ2⁷-Ⓢ5⁷). After finishing a message in the *InflightQueue* (Ⓢ6⁷-Ⓒ3⁷-Ⓢ7⁷-Ⓢ8⁷), the broker removes the finished message from the *InflightQueue* (Ⓢ9⁷), moves a message in the *CacheQueue* (e.g., `M`) to the *InflightQueue* (Ⓢ10⁷) and continues to process the message `M` (Ⓢ11⁷-Ⓢ12⁷-Ⓒ4⁷-Ⓢ13⁷-Ⓢ14⁷).

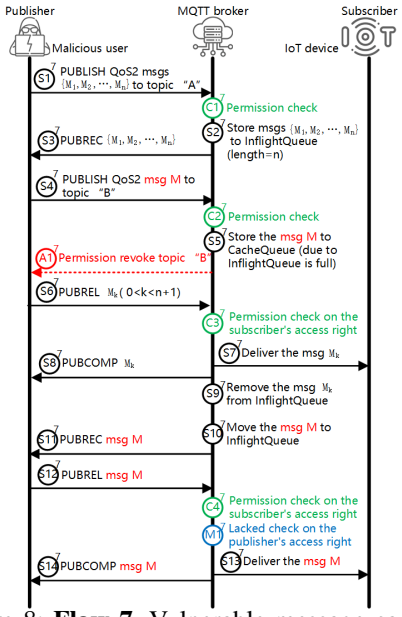


Figure 8: **Flaw 7**. Vulnerable message caching

Mosquitto's `cache-before-process` mechanism is found to be vulnerable: a malicious user could force Mosquitto to cache a malicious message `M` in its *CacheQueue* with n prepositive messages. After Mosquitto revokes the user's permission associated with the message `M`, the user tricks Mosquitto into finishing one of the prepositive messages and eventually delivering the malicious message `M` (see above), which violates the security property sp_3.

PoC exploits and ethical consideration. *We developed PoC exploits for all flaws to confirm their realistic impacts (elaborated online [42]).* For ethical experiments, we deployed the MQTT brokers under inspection and the MQTT clients in our lab environments, without affecting real-world IoT services and users.

Possible mitigation to the flaws. Fully securing the MQTT systems requires the developers to enforce sufficient security checks based on a complete understanding of the security implications of the complex messaging logic, the asynchronous expectation, and the dynamic access control in real-world applications, which is unfortunately not easy for the developers to make it right, in the absence of standardized security guidance. To provide timely protection to the MQTT brokers, we have suggested the necessary security checks to fix the flaws we find, which are illustrated as the "Lacked check" (the steps of Ⓜi^j in blue) in the Figure 1, 4, 5, 6, 7, 8, and 9. Moreover, the current MQTT specification could be enhanced or better clarified to help developers avoid the flaws. It is particularly imperative to enhance the specification and provide guidance on where in the messaging process one should implement a permission check (e.g., Flaw 1 for the lack of authorization after a `PUBREL`). Also, the protocol's specification of certain operations can be more deterministic, e.g., the timing to deliver the `QoS 2` message (see "Ambiguous definition" in § 3.1).

5.2. The Pervasiveness and Magnitude of the Flaws

We list all the flaws identified by our tool *MQTTactic* in Table 1 and 4. As we can see, each broker we inspected has at least 3 (up to 9) flaws and violates all the 3 security properties, indicating the authorization issues in MQTT messaging come from various aspects and are common in even the most popular brokers — there are over 318,000 deployments of the Mosquitto broker in production [10].

Moreover, we manually check 7 popular commercial MQTT brokers and 6 other open-source brokers (implemented in Java and Erlang, which *MQTTactic* does not support currently, see § 7) to evaluate the pervasiveness of the 0-day flaws identified by *MQTTactic*. Not surprisingly, as shown in Table 5, all 6 open-source brokers are vulnerable. More importantly, even brokers of the most popular/leading IoT service providers (i.e., AWS, IBM, Baidu, Alibaba, and Tencent), who have millions of users worldwide, are also vulnerable. Notably, we find no flaw in the Google broker and Azure broker, because they provide simplified MQTT services, e.g., `QoS 2` message is not supported. Given the large number of the MQTT brokers' downstream MQTT customers and end-users, the pervasiveness and the magnitude of the flaws in the brokers pose a significant threat to today's IoT ecosystem.

Responsible disclosure. We report all the identified flaws to the 20 vendors, 13 of which have acknowledged the seriousness of the problems and are taking actions to address

TABLE 1: The 7 zero-day flaws identified by *MQTTactic*

	Violation to which security properties	Mosquitto (C, 6.2K stars)* v2.0.11	FlashMQ (C++, 80 stars)* v0.9.9	Emitter (Go, 3.2K stars)* v3.0	VolantMQ (Go, 879 stars)* v0.4.0	hmq (Go, 1k stars)* v1.5.0	RMQTT (rust, 125 stars)* v0.2.3	Mochi MQTT (Go, 230 stars)* v1.2.3
Flaw 1	sp_2, sp_3	✓	✗	N/A	✓	✗	✗	✗
Flaw 2	sp_2	✓	✓	✓	✓	✗	✓	✓
Flaw 3	sp_2	✓	✓	N/A	✓	✗	✓	✓
Flaw 4	sp_2	✗	N/A	N/A	✓	N/A	N/A	N/A
Flaw 5	sp_3	✓	✓	✗	✓	✓	✓	✓
Flaw 6	sp_2, sp_3	✗	✓	✗	✗	✓	✓	✗
Flaw 7	sp_3	✓	✗	N/A	✗	✗	✗	✗

✓ indicates the flaw was identified in the broker, while ✗ indicates the flaw was NOT identified in the broker. * specifies the broker's programming language and the number of stars in GitHub. N/A indicates the broker does not support the corresponding feature (e.g., `QoS 2` messaging and `Topic Alias`).

them with our help. Mitigation has been deployed or is on the way — Mosquitto fixed **Flaw 7** and partially fixed **Flaw 1**; FlashMQ fixed **Flaw 2**, **Flaw 3**, and **Flaw 9**; EMQX fixed **Flaw 9**; Baidu resolved **Flaw 10**; Alibaba fixed **Flaw 8**.

6. Evaluation

Performance overhead. We run *MQTTactic* on a server running Ubuntu 20.04, equipped with 192GB RAM, and a 40-core CPU of Intel Xeon Platinum 8269CY@3.1GHz. The major tasks of *MQTTactic* are data flow analyses, symbolic execution, and model checking. The time of data flow analyses for a broker is less than 5 minutes. The time for symbolic execution is closely related to the amount of codes in the *action* under inspection. Specifically, for the `CONNECT` action of Mosquitto (2,000 LOC), it takes about 8 minutes with a single process to identify one effective path type, while it takes less than 5 minutes to identify all the effective path types for the `CONNECT` action in VolantMQ (250 LOC). Similarly, the time needed for model checking highly depends on the complexity of the `concrete model` — it takes **MC** about 30 minutes to identify the flaws in Mosquitto while only 2 minutes for VolantMQ. Overall, it takes about 176 minutes for *MQTTactic* to finish analyzing a broker in average.

Completeness of modeling. Our modeling goal for a specific broker is to extract and abstract its control flows with logical constraints impacting the key states of the broker's operations. We consider the broker's state based on the 7 state variables; hence, given huge code footprint (e.g., 10K lines of code), we do not extract model-unrelated implementation details (such as format check or exception handling). To evaluate the completeness of modeling for our goal, we manually inspect all 6 action handlers of FlashMQ and their program paths: our generated model covers 10 out of 11 (91%) unique paths. The missed one is an autonomous messaging flow unique to FlashMQ: regardless of client actions, the broker autonomously delivers a `Will` message if related client session is not alive based on a timer. Hence, our search of paths starting from handler functions (broker entry points of client actions) does not cover this path. However, our general model definition (e.g., operation on `Will` message is considered to impact broker state) enables us to cover such a path in the future work.

End-to-end effectiveness of modeling and verification. We implement proof-of-concept test with the 36 counterexamples reported by *MQTTactic* for FlashMQ and find that all are executed successfully and violate the security properties (some counterexamples with slightly different action orders indicate the same security flaw in nature).

6.1. Comparison with Prior Works

Comparison with MPInspector. We compare the (1) effectiveness of MPInspector [15] and *MQTTactic* (including model definition, construction, and security violation identification) (2) manual efforts in the model construction.

• *Effectiveness*. The model definition and construction of MPInspector and *MQTTactic* are significantly different and thus *MQTTactic* can identify authorization-related violations while MPInspector generally could not (with its focus on authentication and secrecy of message attributes).

Extending MPInspector for access-control problems can be non-trivial and entail serious research efforts. First, MPInspector does not model authorization-related semantics and states, limiting its ability to identify authorization violations. MPInspector does not consider permissions, models system states, or behaviors when access-policies are changed or different, and thus, MPInspector could not identify system states that violate access policies; for example, the broker delivers or handles a message for a client violating access policies. Second, MPInspector is based on Active Automata Learning [60], which does not directly come with semantics for the automata states. To identify authorization violations, MPInspector should supplement authorization-related semantics to the states. Examples of necessary semantics for a state include (1) whether a state indicates a message delivery; (2) who are message recipients and senders; (3) permissions of the related clients in the state. Expanding MPInspector states with authorization-necessary semantics entails thorough design and can be non-trivial. Hence, MPInspector cannot be directly or simply adopted to find authorization mistakes like *MQTTactic* can do. MPInspector does supplement semantics to the states, such as in-message attributes to handle secrecy and authentication properties. Current *MQTTactic* does not focus on secrecy or authentication properties.

• *Manual efforts expected*. In general, *MQTTactic* and MPInspector expect comparable amounts of manual effort for model construction. For a specific broker, manual efforts expected by MPInspector include (1) setting up the broker to run, (2) communication configuration (e.g., MQTT version and raw password), (3) collecting diverse network traffic, (4) configuring LearnLib [60] — the underlying tool used by MPInspector. In contrast, manual efforts expected by *MQTTactic* is to develop a `configuration` file that

maps state variables/actions to program code (§ 4.5). We apply *MQTTactic* and MPInspector for three real-world broker implementations Mosquitto [61], Tuya cloud [62], and FlashMQ [63], and evaluate the actual time (manual efforts) needed to construct models. The results showed that the manual efforts are comparable: *MQTTactic* needs 1.3 to 4.2 hours per broker, compared to MPInspector, which needs 1.6 to 3.8 hours (Table 7 in Appendix details the results).

Comparison with other prior model construction methods. Prior approaches for model construction [15], [30]–[39] generally cannot be directly applied to generate useful models for MQTT broker implementations, because (1) they lack definition for formal models that can describe MQTT states and semantics; (2) they are not suited to abstract operations and logic that are related to broker states. Moreover, building models under the prior works' contexts may or may not require more manual efforts than ours to achieve their verification goals. For example, MPInspector [15], LearnLib [60], NGLL [64], Libalf [65], Tomte [66], ROLL [67], and Scikit-SpLearn [68] leverage semi-automatic model inferring/learning techniques for model construction, although their learning time increases exponentially with the increasing input/output space.

7. Discussion and Future work

Possible false positives/negatives. During the investigation of the 7 open-source brokers (see § 5), we did not come across any false positives. Nevertheless, we present several potential causes that could result in false positives/negatives, which could serve as directions for future optimization.

• *False positives*. (1) Partial symbolic execution: For efficiency in our current experiment configuration, instead of symbolically executing all function calls, we mark the return values of certain function calls (pertaining to system functions like $exit()$ or surpassing the specified callstack depth) as indeterminate symbolic values. This can render certain symbolic constraints not fully unidentified in running *MQTTactic*, over-estimate reachable paths, and lead to false paths extracted. (2) Incorrect `configuration`: An incorrect `configuration` (e.g., specifying incorrect implementation-level variables) could lead to irrelevant basic blocks being identified as KBB and further cause false path types to be added to the `concrete model`.

• *False negatives*. (1) Incomplete pointer analyses results: Pointer analyses of SVF [50] might fail to identify all the KBBs, which could cause certain effective path types not to be added to the `concrete model`, resulting in *MQTTactic* missing the flaws related to such effective path types; (2) Incorrect `configuration`: Incorrect `configuration` (e.g., incomplete mapping) could also cause certain path types not being added to the `concrete model`; (3) Advanced programming language features: Different programming languages features may bring various challenges to control flow extraction. For example, function pointers in C/C++ make it difficult for static analyzers to fully identify control flows. *MQTTactic* mitigated this problem with pointer analyses. However, advanced programming language features in different languages (e.g., Virtual Function in C++ and Interface in Golang) could pose new challenges for extracting complete control flows from the source code. Future efforts are expected to better solve the problem.

Towards fully automated analysis. In the current *MQTTactic*, manual efforts comparable to related work are expected (§ 6.1). In future work, we may leverage *natural language processing* (NLP) based semantic analysis to help automatically identify model-related function and variable names in the source code, thus making *MQTTactic* more automatic.

Applicability. *MQTTactic* cannot analyze a broker without source code. Notably, developers of closed-source brokers can leverage *MQTTactic* to find problems. Further, our idea of extracting only model-relevant information from large code bases for relatively efficient model checking can be applied to other messaging protocols and application domains.

8. Related Work

IoT platform security. The security of IoT platforms has been widely studied, including the coarse-grained capabilities, vulnerable automation control rule detection, delegation problem, etc. [22], [23], [45], [69], [70]. Most of these works typically focus on a specific cloud platform, such as SmartThings [16]–[21], AWS's Alexa platform [22], [23], and IFTTT [24]–[26]. By contrast, our work tries to discover security flaws across multiple brokers with a unified method.

Model-guided vulnerability identification. Model-guided approaches have also been proposed for attack strategy generation (Fuzzing) [32]–[34] or model refining/learning [31]. However, most of these works require substantial manual efforts for either building the model [32], [33] or performing security checks on the model [31]. Moreover, Pacheco et al. [71] proposed an approach for automated attack synthesis by extracting finite state machines from the protocol specification. In contrast, we use both the information in the protocol specification and the implementation-specific details to derive unique concrete models for each implementation, aiming to identify the flaws in the protocol implementations.

9. Conclusion

We report the first attempt to systematically identify authorization-related logic flaws in open-source MQTT broker implementations by formally verifying their source codes. We propose *MQTTactic* to analyze popular open-source MQTT brokers, and discover 7 zero-day flaws with serious security implications. Our findings suggest that the customization of messaging flow and logic can easily go wrong. Our new understandings and findings will provide better protection to today's IoT supply chain.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their insightful comments. This work is supported by the National Natural Science Foundation of China (No. 62372191). Yan Jia is supported by the National Natural Science Foundation of China (No. 62102198) and China Postdoctoral Science Foundation (No. 2021M691673, No. 2023T160335). Luyi Xing is supported in part by NSF CNS-2145675 and CCF-2124225.

References

[1] "2022 survey shows MQTT adoption is high for industry," https://www.hivemq.com/blog/2022-survey-shows-mqtt-adoption-is-high-for-industry/, accessed: 2023-04.

[2] "AWS IoT," https://aws.amazon.com/cn/iot-core, accessed: 2023-04.

[3] "Google IoT," https://cloud.google.com/iot-core, accessed: 2023-04.

[4] "IBM IoT," https://www.ibm.com/cloud/internet-of-things, accessed: 2023-04.

[5] "Azure IoT," https://azure.microsoft.com/en-us/solutions/iot/, accessed: 2023-04.

[6] "Baidu IoT," https://intl.cloud.baidu.com/product/iot.html, accessed: 2023-04.

[7] "Alibaba IoT," https://mqtt.console.aliyun.com/, accessed: 2023-04.

[8] "MQTT implementations," https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations, accessed: 2023-04.

[9] "HiveMQ Customers," https://www.hivemq.com/customers/, accessed: 2023-04.

[10] "ZoomEye: Mosquitto usage," https://www.zoomeye.org/searchResult?q=app:"Mosquitto", accessed: 2023-04.

[11] "MQTT Specification," https://mqtt.org/mqtt-specification/, accessed: 2023-04.

[12] R. Jhala and R. Majumdar, "Software model checking," ACM Comput. Surv., vol. 41, no. 4, pp. 1–54, 2009.

[13] K. Hofer-Schmitz and B. Stojanović, "Towards formal methods of iot application layer protocols," in Proceedings of the 12th CMI Conference on Cybersecurity and Privacy, 2019, pp. 1–6.

[14] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," ACM Transactions on Programming Languages and Systems, vol. 16, no. 5, pp. 1512–1542, 1994.

[15] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu, and R. Beyah, "Mpinspector: A systematic and automatic approach for evaluating the security of iot messaging protocols," in Proceedings of the 30th USENIX Security Symposium, 2021, pp. 4205–4222.

[16] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks," in Proceedings of the 25th USENIX Security Symposium, 2016, pp. 531–548.

[17] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in Proceedings of the 26th USENIX Security Symposium, 2017, pp. 361–378.

[18] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot." in Proceedings of the 26th Annual Network and Distributed System Security Symposium, 2019.

[19] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity, "Contexiot: Towards providing contextual integrity to appified iot platforms." in Proceedings of the 24th Annual Network and Distributed System Security Symposium, 2017.

[20] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in Proceedings of the 25th Annual Network and Distributed Systems Security Symposium, 2018.

[21] B. Yuan, Y. Wu, M. Y. L. Xing, X. Wang, D. Zou, and H. Jin., "Smartpatch: Verifying the authenticity of the trigger-event in the iot platform," IEEE Transactions on Dependable and Secure Computing, vol. 20, no. 2, pp. 1656–1674, 2022.

[22] L. Cheng, C. Wilson, S. Liao, J. Young, D. Dong, and H. Hu, "Dangerous skills got certified: Measuring the trustworthiness of skill certification in voice personal assistant platforms," in Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 1699–1716.

[23] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian, "Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems," in Proceedings of the 41st IEEE Symposium on Security and Privacy, 2019, pp. 1381–1396.

[24] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action iot platforms," in Proceedings of the 25th Annual Network and Distributed System Security Symposium, 2018.

[25] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in iot apps," in Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 1102–1119.

[26] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 1439–1453.

[27] "MQTTSA," https://github.com/stfbk/mqttsa, accessed: 2023-04.

[28] "MQTT-PWN," https://github.com/akamai-threat-research/mqtt-pwn, accessed: 2023-04.

[29] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao, and Y. Zhang, "Burglars' IoT Paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds," in Proceedings of the 41st IEEE Symposium on Security and Privacy, 2020, pp. 465–481.

[30] K. Hofer-Schmitz and B. Stojanović, "Towards formal methods of iot application layer protocols," in Proceedings of the 12th CMI Conference on Cybersecurity and Privacy, 2019, pp. 1–6.

[31] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in Proceedings of the 24th USENIX Security Symposium, 2015, pp. 193–206.

[32] S. Jero, M. E. Hoque, D. R. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in TCP congestion control using a model-guided approach," in Proceedings of the 25th Annual Network and Distributed System Security Symposium, 2018.

[33] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in Proceedings of the 36th IEEE Symposium on Security and Privacy, 2015, pp. 535–552.

[34] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2015, pp. 1–12.

[35] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in Proceedings of the 23rd USENIX Annual Technical Conference, 2018, pp. 147–158.

[36] B. Aziz, "A formal model and analysis of the mq telemetry transport protocol," in Proceedings of the 9th International Conference on Availability, Reliability and Security, 2014, pp. 59–68.

[37] A. Rodriguez, L. M. Kristensen, and A. Rutle, "On modelling and validation of the mqtt iot protocol for m2m communication," in Proceedings of the 39th International Workshop on Petri Nets and Software Engineering, 2018, pp. 99–118.

[38] J. Hcine and I. Ben Hafaiedh, "Formal-based modeling and analysis of a network communication protocol for iot: Mqtt protocol," in Proceedings of the 8th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications, 2018, pp. 350–360.

[39] B. Aziz, "A formal model and analysis of an iot protocol," Ad Hoc Networks, vol. 36, pp. 49–57, 2016.

[40] "Spin," http://spinroot.com/spin/whatispin.html, accessed: 2023-04.

[41] "Promela," https://en.wikipedia.org/wiki/Promela, 2022, accessed: 2023-04.

[42] "MQTTactic," https://github.com/CGCL-codes/MQTTactic/, accessed: 2023-04.

[43] "The MQTT protocol," https://mqtt.org, accessed: 2023-04.

[44] "Topic Alias - MQTT 5.0 new features," https://www.emqx.com/en/blog/mqtt5-topic-alias, accessed: 2023-04.

[45] B. Yuan, Y. Jia, L. Xing, D. Zhao, X. Wang, and Y. Zhang, "Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation," in Proceedings of the 29th USENIX Security Symposium, 2020, pp. 1183–1200.

[46] Y. Jia, B. Yuan, L. Xing, D. Zhao, Y. Zhang, X. Wang, Y. Liu, K. Zheng, P. Crnjak, Y. Zhang, D. Zou, and H. Jin, "Who's in control? on security risks of disjointed iot device management channels," in Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 1289–1305.

[47] R. Ensafi, J. C. Park, D. Kapur, and J. R. Crandall, "Idle port scanning and non-interference analysis of network protocol stacks using model checking," in Proceedings of the 19th USENIX Security Symposium, 2010, pp. 257–272.

[48] "Basic Block," https://en.wikipedia.org/wiki/Basic_block, accessed: 2023-04.

[49] "Control Flow Graph," https://en.wikipedia.org/wiki/Control-flow_graph, accessed: 2023-04.

[50] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in Proceedings of the 25th International Conference on Compiler Construction, 2016, pp. 265–266.

[51] "haybale," https://github.com/PLSysSec/haybale, accessed: 2023-04.

[52] "Security Property," https://github.com/CGCL-codes/MQTTactic/blob/e0090bba956d79791905c68e5e0dce1213de0579/Examples/mosquitto_concrete_model.pml#L1408, accessed: 2023-07.

[53] "LLVM," https://llvm.org/, accessed: 2023-04.

[54] "Clang," https://rustc-dev-guide.rust-lang.org/, accessed: 2023-04.

[55] "Gollvm," https://go.googlesource.com/gollvm/, accessed: 2023-04.

[56] "Rustc," https://clang.llvm.org/, accessed: 2023-04.

[57] "LLVM IR generation," https://github.com/CGCL-codes/MQTTactic/tree/main/LLVM-IR-generation, accessed: 2023-04.

[58] "Open closed principle," https://en.wikipedia.org/wiki/Open-closed_principle, accessed: 2023-04.

[59] "hmq," https://github.com/fhmq/hmq, accessed: 2023-04.

[60] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib: a framework for active automata learning," in Proceedings of the 27th Computer Aided Verification, 2015, pp. 487–495.

[61] "Eclipse Mosquitto," https://mosquitto.org/, accessed: 2023-04.

[62] "Tuya Smart," https://en.tuya.com/, accessed: 2023-04.

[63] "FlashMQ," https://www.flashmq.org/, accessed: 2023-04.

[64] O. Bauer, J. Neubauer, and M. Isberner, "Model-driven active automata learning with learnlib studio," in Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, 2016, pp. 128–142.

[65] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon, "libalf: The automata learning framework," in Proceedings of the 22nd International Conference on Computer Aided Verification, 2010, pp. 360–364.

[66] F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. Vaandrager, "Learning register automata with fresh value generation," in Proceedings of the 12th International Colloquium on Theoretical Aspects of Computing, 2015, pp. 165–183.

[67] Y. Li, Y.-F. Chen, L. Zhang, and D. Liu, "A novel learning algorithm for büchi automata based on family of dfas and classification trees," Information and Computation, vol. 281, p. 104678, 2021.

[68] D. Arrivault, D. Benielli, F. Denis, and R. Eyraud, "Scikit-splearn: a toolbox for the spectral learning of weighted automata compatible with scikit-learn," in Proceedings of the 19th Conference Francophone sur l'Apprentissage Aurtomatique, 2017.

[69] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in Proceedings of the 37th IEEE Symposium on Security and Privacy, 2016, pp. 636–654.

[70] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity, "Contexlot: Towards providing contextual integrity to appified iot platforms." in Proceedings of the 24th Annual Network and Distributed System Security Symposium, 2017.

[71] M. L. Pacheco, M. von Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru, "Automated attack synthesis by extracting finite state machines from protocol specification documents," in Proceedings of the 43rd IEEE Symposium on Security and Privacy, 2022, pp. 51–68.

Appendix A.

A.1. Flaw 3: Vulnerable Delivery Retry in `QoS 2` Messaging

Due to the "exactly once delivery" feature in `QoS 2` messaging (see § 2.2), if the target client is offline, the broker would retry to deliver the message `M` to the client when the client reconnects (i.e., ⑤³-⑦³-⑧³). However, we find the delivery retry mechanism in `QoS 2` has the same problem of that in the `QoS 1` messaging as elaborated in the **Flaw 2**. The exploiting and mitigation to the **Flaw 3** are also similar to that of the **Flaw 2**. For simplicity, we omit the redundant description on the **Flaw 3** and use Figure 9 to illustrate the exploiting of **Flaw 3** (i.e., Ⓐ³-①³-Ⓒ①³-②³-③³-④³-⑤³-⑥³-Ⓐ②³-Ⓐ③³-⑦³-⑧³) and the possible mitigation (i.e., Ⓜ①³).

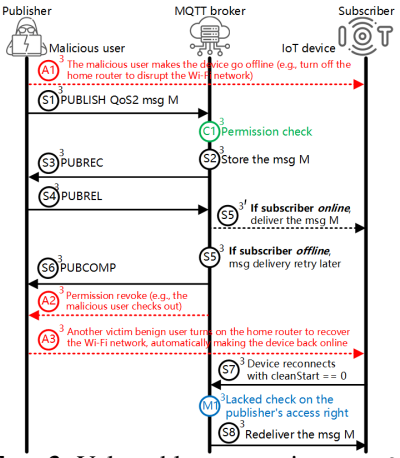


Figure 9: **Flaw 3**. Vulnerable `retry` in `QoS 2` messaging

A.2. The State Variables

TABLE 2: The list of state variables

Type	Variable name	Data recorded in the variable
State	Session: v_{Subs}	The client's subscriptions
	Session: $v_{WillMsg}$	The `Will` message sent along with the `CONNECT` request
	Session: v_{MsgQue}	The (unfinished) message queue
	Session: v_{Msg}	The message being processed
	Session: $v_{Session}$	The session created at the time of connection establishment
	Global: $v_{RetainedMsg}$	The `Retained` message queue
Permission	$v_{Permission}$	The access rights authorized to all the clients/users

A.3. The List of *operation*s

TABLE 3: The list of *operation*s

Operation name	Semantics
$o_{deliver}$	*deliver* the message v_{Msg}
o_{sub_read}	*read* v_{Subs}
o_{sub_add}	$write^+$ v_{Subs}
o_{sub_remove}	$write^-$ v_{Subs}
$o_{Session\_read}$	*read* $v_{Session}$
$o_{Session\_write}$	*write* $v_{Session}$
$o_{msgs\_queue\_read}$	*read* v_{MsgQue}
$o_{msgs\_queue\_add}$	$write^+$ v_{MsgQue}
$o_{msgs\_queue\_remove}$	$write^-$ v_{MsgQue}
$o_{retained\_read}$	*read* $v_{RetainedMsg}$
$o_{retained\_add}$	$write^+$ $v_{RetainedMsg}$
$o_{retained\_remove}$	$write^-$ $v_{RetainedMsg}$
$o_{permission\_add}$	$write^+$ $v_{Permission}$, $write^+$ D
$o_{permission\_remove}$	$write^-$ $v_{Permission}$, $write^-$ D
$o_{permission\_check}$	*read* $v_{Permission}$
o_{will_read}	*read* $v_{WillMsg}$
o_{will_add}	$write^+$ $v_{WillMsg}$
o_{will_remove}	$write^-$ $v_{WillMsg}$

$o_{deliver}$ indicates the broker sending the message identified by the v_{Msg} to the subscriber(s). *write* v_i indicates changing the value of the variable v_i. $write^+$ v_i indicates adding data element to the variable v_i. $write^-$ v_i indicates removing data element from the variable v_i.

A.4. The (Incomplete) List of MQTT Brokers

Open-source MQTT brokers (with 20+ stars in the GitHub, see the complete list at [57]). ejabberd, Emitter, EMQ X, FlashMQ, HBMQTT, HiveMQ, Jmqtt, Moquette, Mosca, Mosquitto, MQTTnet, MqttWk, NanoMQ, RabbitMQ, Cassandana, Apache ActiveMQ, Apache ActiveMQ Artemis, Solace, SwiftMQ, VerneMQ, mainflux, hmq, mqtt, uMQTTBroker, volantmq, mqtt, crossbar, Erl.mqtt.server, gmqtt, smqtt, enmasse, mqtt, gnatmq, gossipd, mica-mqtt, rumqtt, jo-mqtt, gmqtt, sol, mqtt-broker, mqtt, iot-mqtt, hermes, esp-idf-mqtt-broker, mmqtt, JetMQ, mqttools, mqtt-gateway, TinyMqtt, whsnbg, mithqtt, skyline, Aedes, hrotti, KMQTT, Mystique, SurgeMQ, creep, amlen, rmqtt, PronghornGateway, DovakinMQ, io-Broker.mqtt, node-red-contrib-aedes, mqttcpp, LV-MQTT-Broker, haskell-hummingbird, mhub, clima-link, GS.GRID, pyrinas-server-rs, wave, lannister, JoramMQ

Commercial MQTT brokers (24 in total). Alibaba Cloud, AWS IoT Core, adafruit, Azure IoT Hub, CloudMQTT, Google Cloud IoT, Baidu Cloud, Tencent Cloud, Tuya Cloud, Huawei Cloud, Onenet, HiveMQ Cloud, MyQttHub, EMQ X Cloud, CrystalMQ, Yunba, Waterstream, ThingScale IoT message broker, IBM Integration Bus, flespi, Eurotech Everywhere Cloud, Bevywise MQTT Broker, Akiro by Sentienz, Ably MQTT Broker

A.5. The `configuration` File of the FlashMQ Broker

```
config = {
  # Functions
  "handle_connect": "void MqttPacket::handleConnect()",
  "handle_publish": "void MqttPacket::handlePublish()",
  "handle_pubrel": "void MqttPacket::handlePubRel()",
  "handle_subscribe": "void MqttPacket::handlePubRel()",
  "handle_unsubscribe": "void MqttPacket::handleUnsubscribe()",
  "handle_disconnect": "void MqttPacket::handleDisconnect()",
  "handle_authorize": "void Authentication::loadMosquittoAclFile()",
  "handle_revoke": "void Authentication::loadMosquittoAclFile()",
  "premission_check": "AuthResult Authentication::aclCheck(
    const std: :string &clientid, const std: :string &username,
    const std: :string &topic, const std: :vector<std: :string> &subtopics,
    AclAccess access, char qos, bool retain)",

  # variables
  "Subs": "SubscriptionNode::subscribers",
  "RetainedMsg": "RetainedMessageNode::retainedMessages",
  "Session": "Session",
  "WillMsg": "Client::will_topic",
  "MsgQue": "Session::qosPacketQueue",
  "Msg": "MqttPacket",
  "Permission": "Authentication::aclTree",
}
```

Figure 10: The `configuration` of the FlashMQ broker

A.6. Existing Flaws Identified by *MQTTactic*

Jia et al. [29] identified several flaws in different commercial MQTT brokers through manual analyses, while Wang et al. [15] proposed a black-box analysis based method to discover the security flaws in the IoT messaging protocols. Among all the security flaws identified in [15] and [29], four of them are authorization-related flaws (our goal), which are also identified by *MQTTactic*, i.e., **Flaw 8**: Unauthorized subscription via `ClientID` hijacking; **Flaw 9**: Unauthorized trigger of the `Retained` message; **Flaw 10**: Un-updated subscription; **Flaw 11**: Unauthorized trigger of the `Will` message (see Table 4).

TABLE 4: The 4 existing flaws identified by *MQTTactic*

	Violation to which security properties	Mosquitto (C, 6.2K stars)* v2.0.11	FlashMQ (C++, 80 stars)* v0.9.9	Emitter (Go, 3.2K stars)* v3.0	VolantMQ (Go, 879 stars)* v0.4.0	hmq (Go, 1k stars)* v1.5.0	RMQTT (rust, 125 stars)* v0.2.3	Mochi MQTT (Go, 230 stars)* v1.2.3
Flaw 8	sp_1	✗	✗	✗	✓	✗	✓	✓
Flaw 9	sp_2	✓	✓	✓	✓	✓	✓	✓
Flaw 10	sp_1	✗	✗	✓	✓	✓	✓	✓
Flaw 11	sp_2, sp_3	✗	✓	✗	✓	✓	✓	✗

✓ indicates the flaw was identified in the broker, while ✗ indicates the flaw was NOT identified in the broker.
* specifies the broker's programming language and the number of stars in GitHub.

A.7. Flaws Identified by MQTTactic in Other Brokers

TABLE 5: The 0-day flaws identified by *MQTTactic* in other brokers (manually confirmed)

	EMQX# (Erlang, 10.2k stars)* v4.3.11	VerneMQ# (Erlang, 2.8k stars)* v1.12.3	RabbitMQ# (Erlang, 9.8k stars)* v3.10.7	HiveMQ# (Java, 789 stars)* v2021.2	Aedes# (JS, 1.4k stars)* v0.46.3	Moquette# (Java, 2k stars)* v0.15	Baidu +	Google +	AWS+	Azure +	IBM+	Alibaba+	Tencent +
Flaw 1	✗	✗	N/A	✗	✓	N/A	N/A	N/A	N/A	N/A	✗	✗	N/A
Flaw 2	✓	✓	✓	✓	✓	N/A	✓	N/A	✓	N/A	✓	✓	✓
Flaw 3	✓	✓	N/A	✓	✓	N/A	N/A	N/A	N/A	N/A	✓	✓	N/A
Flaw 4	✗	✗	N/A	✓	N/A	N/A	N/A	N/A	N/A	✗	✗	✗	N/A
Flaw 5	✗	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓	N/A	N/A
Flaw 6	✓	✗	✗	✗	✗	✓	✗	N/A	✗	✗	✗	N/A	N/A
Flaw 7	✗	✗	N/A	✗	✗	N/A	N/A	N/A	N/A	N/A	✗	✗	N/A

✓ indicates the flaw was identified in the broker, while ✗ indicates the flaw was NOT identified in the broker.
means the broker is open-sourced. + means the broker is commercial. * specifies the broker's programming language and the number of stars in GitHub.
N/A indicates the broker does not support the corresponding feature (e.g., `QoS 2` messaging, `Topic Alias`, dynamical access control, etc.).

A.8. A Base Semantic Definition of Actions

TABLE 6: Actions definition

Action Name	Parameters/Conditions	Semantics of the action (Operation sequence)
CONNECT	CleanStart==false & ClientID==cid & oldSession(cid)	$o_{msgs\_queue\_add} \rightarrow o_{sub\_add} \rightarrow o_{will\_add} \rightarrow o_{clientid\_write}$
	CleanStart==false & ClientID==cid & !oldSession(cid)	$o_{will\_add} \rightarrow o_{clientid\_write}$
	CleanStart==true	$o_{will\_add} \rightarrow o_{clientid\_write}$
DISCONNECT	reasonCode==0x04(Disconnect with Will Message)	$o_{will\_read} \rightarrow o_{sub\_read} \rightarrow o_{deliver}$
	reasonCode!=0x04	o_{will_remove}
PUBLISH	QoS 0 & retained==false	$o_{sub\_read} \rightarrow o_{deliver}$
	QoS 0 & retained==true	$o_{retained\_add} \rightarrow o_{sub\_read} \rightarrow o_{deliver}$
	QoS 1 & retained==false & online(suber)	$o_{sub\_read} \rightarrow o_{deliver}$
	QoS 1 & retained==false & offline(suber)	$o_{sub\_read} \rightarrow o_{msgs\_queue\_add}$(suber)
	QoS 2 & retained==false	$o_{msgs\_queue\_add}$(puber)
PUBREL	online(suber)	$o_{msgs\_queue\_read}$(puber) $\rightarrow o_{sub\_read} \rightarrow o_{deliver} \rightarrow o_{msgs\_queue\_remove}$(puber)
	offline(suber)	$o_{msgs\_queue\_read}$(puber) $\rightarrow o_{sub\_read} \rightarrow o_{msgs\_queue\_add}$(suber) $\rightarrow o_{msgs\_queue\_remove}$(puber)
SUBSCRIBE	—	$o_{sub\_add} \rightarrow o_{retained\_read} \rightarrow o_{deliver}$
UNSUBSCRIBE	—	o_{sub_remove}
AUTHORIZE	—	$o_{permission\_add}$
REVOKE	—	$o_{permission\_remove}$

A.9. Comparison of Manual Efforts: MPInspector v.s. MQTTactic

TABLE 7: Comparison of manual efforts: MPInspector v.s. *MQTTactic*

	MPInspector				MQTTactic	
	Setting up the broker	Specifying the communication configuration	Collecting traffic	Configuring Learlib	Specifying the 7 variables	Specifying the 9 functions
Mosquitto [61]#	1 h	10 min	40 min	30 min	4 h	10 min
Tuya cloud [62]+	20 min	30 min	1 h	2 h	N/A	N/A
FlashMQ [63]#	30 min	10 min	40 min	20 min	1 h	20 min
Total time	1 h 40 min - 3h 50 min				1 h 20 min - 4 h 10 min	

In this comparison, we exclude one-time efforts, which both approaches need. # means the broker is open-sourced. + means the broker is commercial.

A.10. An Example of Translating the Effective Path Types into Promela code

We use pre-defined (with one-time efforts) code templates of each *operation* o ($o \in \mathcal{O}$) to generate the Promela code for each Path Type ept. Specifically, each code template describes the operation (`read`, `write`, or `deliver`) performed on a particular state variable v ($v \in \mathcal{V}$). For example, o_{will_read} indicates to read the `Will` message from the client's session. Hence, the code template of o_{will_read} is defined as shown in Listing 1. Notably, there are placeholders in the code templates (e.g., "$\{clientID\}$" in Listing 1), which will be populated with the actual values when **MCT** constructs the `concrete model`.

```
msg = Sessions[{clientId}].willmessage
```

Listing 1: o_{will_read} code template

Taking the hmq broker [59] as an example, we will illustrate how to translate one of the Effective Path Types (extracted by **SCA** module) for `DISCONNECT` action into Promela code.

The ept example of `DISCONNECT` action. As shown below (Listing 2, 3, 4), the ept contains 3 operations: o_{will_read}, o_{sub_read}, and $o_{deliver}$.

```
if c.info.willMsg != nil {
    //read will msg variable
    b.PublishMessage(c.info.willMsg)
}
```

Listing 2: o_{will_read} **at** *hmq_sourcecode/broker/client.go:850*

```
// read subscription variable
return this.sroot.smatch(topic, qos, subs, qoss)
```

Listing 3: o_{sub_read} **at** *hmq_sourcecode/broker/lib/topics/memtopics.go:82*

```
for _, sub := range subs {
  s, ok := sub.(*subscription)
  if ok {
    // deliver the msg
    if err := s.client.WriterPacket(packet); err
    != nil {
      log.Error("write message error", zap.Error(
    err))
    }
  }
}
```

Listing 4: $o_{deliver}$ **at** *hmq_sourcecode/broker/broker.go:669*

Generating Promela codes for ept. With the identified ept(s) for the actions, we now generate the Promela codes (see details in § 4.3.2). Firstly, we show the pre-defined Promela code templates of these three operations as follows (Listing 1 for o_{will_read}, Listing 5 for $o_{deliver}$, Listing 6 for o_{sub_read}).

```
Deliver({msg}, {sess});
```

Listing 5: $o_{deliver}$ code template

```
bool hasSubscription = false;
j = 0;
// Traverse the subscription tree of {sess} and
    check if it is subscribed to the topic of
    message
do
  :: j < MAXSUBSCRIPTIONS ->
    if
      :: (Sessions[{sess}].subscriptions[j].topic
    == {msg}.topic) ->
        hasSubscription = true;
        break;
      :: else -> skip;
    fi;
    j = j + 1;
  :: else ->
    goto nextClients;
od;

nextClients:
  skip;
```

Listing 6: o_{sub_read} code template

```
proctype ProcessSubscriber(short index){
  do
    ::
      atomic{
        // placeholders
        CONNECT_{placeholder}();
      }
    ::
      atomic{
        // placeholders
        DISCONNECT_{placeholder}();
      }
    ...
    :: else -> break;
  od;
}
...

init {
  ...
  run ProcessPublisher(0);    //Publisher client 1
  run ProcessSubscriber(1);   //Subscriber client 1
  run ProcessPublisher(2);    //Publisher client 2
}
```

Listing 7: The model's skeleton code

Then, *MQTTactic* will assemble these code templates with the operation sequence of this ept to generate a handler function (the function codes can be found at [42]) in Promela code for the `DISCONNECT` action.The same process will be carried out for other actions and epts. The placeholders in skeleton code (Listing 7) would then be populated with the above generated Promela functions to construct the `concrete model` in Promela.

Appendix B. Meta-Review

B.1. Summary

This paper focuses on the security of MQTT protocol, a widely used IoT messaging protocol. The authors employ static code analysis, formal modeling, and model checking to identify the authorization-related logic flaws in open-source MQTT brokers. They discover several zero-day flaws which have been acknowledged by related parties.

B.2. Scientific Contributions

- Identifies impactful vulnerabilities
- Provides a valuable step forward in an established field
- Creates a new tool to enable future science

B.3. Reasons for Acceptance

1) This paper identifies multiple impactful vulnerabilities. The authors used code analysis and formal analysis to uncover logic flaws, especially the authorization-related flaws, in open-source MQTT brokers. They identified multiple zero-day vulnerabilities which can be exploited to cause serious security implications, such as gaining unauthorized access to IoT devices. The authors reported the identified vulnerabilities to help the related parties (e.g., IoT vendors) fix the issues.
2) The paper provides a valuable step forward in the field of analyzing security flaws in MQTT brokers. The authors fill multiple significant research gaps in prior studies. First, the authors performed the first systematic study on logic flaws in MQTT brokers. Although there are a few work studies the implementation flaws (e.g., memory-related bugs) in MQTT brokers, logic flaws remain unstudied. Second, the authors used static code analysis and formal analysis to mitigate low code coverage and poor scalability problems suffered from the prior studies. As a result, the authors uncover many previously unknown vulnerabilities in MQTT brokers.
3) The paper leads to a new tool named MQTTactic to enable future science. The authors claim that they will fully release artifacts of the paper online, including the source code, setup instructions, and a demo. The artifacts will allow other researchers to evaluate new ideas, develop new tools, and compare them with MQTTactic.

B.4. Noteworthy Concerns

1) Applicability. The authors designed their approaches targeting open-source MQTT brokers, and the implemented tool cannot handle closed-source MQTT brokers. It would be very useful to extend the proposed approach to cover those closed-source MQTT brokers.
2) Manual work. The proposed approach requires manual efforts to configure the tool based on the domain knowledge about different MQTT brokers, and thus the analysis process is not fully automated. It would be good if the authors could adopt other techniques to reduce manual efforts, such as leveraging NLP techniques to analyze the specification of MQTT brokers to automatically extract the required information for configuration.

Appendix C. Response to the Meta-Review

C.1. Applicability and Limitation

MQTTactic analyzes source code to identify security issues. Notably, by analyzing source code when it is available, *MQTTactic* may help more directly and precisely pinpoint issues in the implementation than blackbox approaches. Hence, *MQTTactic* is at least complementary to black-box approaches. Moreover, the developers of closed-source software or brokers can use *MQTTactic* to assess the security of their brokers.

C.2. Manual Work

While *MQTTactic* advances the state of the art for formal security analysis on source code, the manual efforts are comparable to related work (§ 6.1). As mentioned in § 7, we anticipate an NLP-based approach to help automatically identify model-related function and variable names in the source code, thus making MQTTactic more automatic (to generate the configuration).