MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols

Qinying Wang, *Zhejiang University;* Shouling Ji, *Zhejiang University; Binjiang Institute of Zhejiang University;* Yuan Tian, *University of Virginia;* Xuhong Zhang, *Zhejiang University; Binjiang Institute of Zhejiang University;* Binbin Zhao, *Georgia Institute of Technology;* Yuhong Kan and Zhaowei Lin, *Zhejiang University;* Changting Lin and Shuiguang Deng, *Zhejiang University; Binjiang Institute of Zhejiang University;* Alex X. Liu, *Ant Group;* Raheem Beyah, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity21/presentation/wang-qinying

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols

Qinying Wang[†], Shouling Ji[†,¶], Yuan Tian[+], Xuhong Zhang[†,¶], Binbin Zhao[§], Yuhong Kan[†], Zhaowei Lin[†], Changting Lin[†,¶], Shuiguang Deng[†,¶], Alex X. Liu [‡], and Raheem Beyah[§]

[†]Zhejiang University, [¶]Binjiang Institute of Zhejiang University, [+]University of Virginia, [§]Georgia Institute of Technology, [‡]Ant Group

E-mails: {wangqinying, sji}@zju.edu.cn, yuant@virginia.edu, zhangxuhong@zju.edu.cn, binbin.zhao@gatech.edu, {kan_yuhong, leon.linzw}@zju.edu.cn, linchangting@gmail.com, dengsg@zju.edu.cn, alexliu@antgroup.com, rbeyah@gatech.edu.

Abstract

Facilitated by *messaging protocols* (MP), many home devices are connected to the Internet, bringing convenience and accessibility to customers. However, most deployed MPs on IoT platforms are fragmented, which are not implemented carefully to support secure communication. To the best of our knowledge, there is no systematic solution to perform automatic security checks on MP implementations yet.

To bridge the gap, we present `MPInspector`, the first automatic and systematic solution for vetting the security of MP implementations. `MPInspector` combines model learning with formal analysis and operates in three stages: (a) using parameter semantics extraction and interaction logic extraction to automatically infer the state machine of an MP implementation, (b) generating security properties based on meta properties and the state machine, and (c) applying automatic property based formal verification to identify property violations. We evaluate `MPInspector` on three popular MPs, including MQTT, CoAP and AMQP, implemented on nine leading IoT platforms. It identifies 252 property violations, leveraging which we further identify eleven types of attacks under two realistic attack scenarios. In addition, we demonstrate that `MPInspector` is lightweight (the average overhead of end-to-end analysis is ~4.5 hours) and effective with a precision of 100% in identifying property violations.

1 Introduction

Messaging protocol (MP) is critical for IoT platforms, as it connects IoT devices to the Internet and enables the communication between IoT devices, users, manufactures, and IoT app servers. IoT platforms offer customized MP implementations with different security schemes for IoT vendors. For example, Google IoT Core adopts Json Web Token (JWT) for authentication [14]. Unfortunately, MPs are hard to design correctly and several implementation flaws have been identified through ad-hoc manual analysis [45]. These flaws lead to critical consequences, such as denial of service (DoS), sensitive data theft and malicious message injection [37, 54].

So far, IoT platforms still have limited understanding about the security of MPs, since neither industry nor academia has good ways to systemically and effectively evaluate the security of MP implementations. Considering the large amount of diversified IoT platforms, manual analysis that requires significant expert efforts is infeasible. Consequently, the pressing question is how to build an automatic tool to verify the security properties of MP implementations on different IoT platforms effectively? To answer the question, there are two main challenges.

Diverse and customized MP implementations. The MP implementations are diverse. Specifically, there are multiple types of MPs with different message formats and mechanisms, such as MQTT (Message Queuing Telemetry Transport) [46], CoAP (Constrained Application Protocol) [1] and AMQP (Advanced Message Queuing Protocol) [4]. In addition, there are various customized implementations on different IoT platforms with different programming languages for each MP. These diverse and customized MP implementations stress the scalability of the analysis. Even worse, there are always gaps between the customized MP implementations and the standard MP specification, such as the differences on the configuration, parameter semantics, and interaction logic. Therefore, previous work on analyzing the high-level protocol specifications [23, 27, 28, 34] is hardly applicable in the IoT context.

Complex and closed-source MP workflow. Checking the MP implementation requires precisely modeling MP workflow including the exchanged parameters and interaction logic. However, the workflow of MP is complicated, as it connects multiple devices and usually consists of multiple messages. Even worse, MP implementations are closed-source. As an example, the commercial platforms such as AWS IoT Core [5] and Azure IoT Hub [6] do not open their source code on the server side. The closed-source MP implementation requires any testing approach to be black-box and system-agnostic. Accordingly, previous research on program analysis for protocols [25, 40, 41] cannot be used.

Shouling Ji and Xuhong Zhang are the co-corresponding authors.

To handle these challenges, previous research conducts reverse engineering on the firmware and apps [36], which requires large expert knowledge. Therefore, it is not scalable and can be time-consuming. Fuzzing is an alternative solution [38, 39, 43] to detect flaws by monitoring the crashes of the system under test. However, it can hardly cover the full workflow of an MP implementation and cannot discover logic flaws that do not cause crashes.

Our solution. To address the above challenges, we propose and implement `MPInspector`, the first framework to systematically and automatically identify security flaws in MP implementations. We follow a property-driven and model-based testing philosophy. First, we model an MP implementation into a state machine. Second, we gather the security properties that need to be verified from the standard MP specification and refine them based on the learned state machine. Finally, we detect property violations on the state machine by formal verification. Specifically, the extracted state machine includes transition messages and transition logic. Transition messages are the messages that trigger the transition from one state to another, while transition logic is also referred to as interaction logic. To support in-depth inspection of security flaws in MP implementations, `MPInspector` recovers the detailed semantics of transition messages, which refer to as the customized composition of each parameter in the messages. For example, the `ClientID` parameter in MQTT [46] may consist of `ProjectId` and `DeviceId` in a customized MP implementation. As for the interaction logic, we adopt active model learning [21], a framework to construct the state machine of a system by providing inputs and observing outputs. In `MPInspector`, the inputs are messages sent to an MP implementation and the outputs are the relevant response messages or the connection states. Then, `MPInspector` gathers security properties that need to be verified, which include the meta properties concluded from the standard MP specification and the extended properties inferred from the customized MP implementation. After that, we convert the state machine and security properties into Tamarin codes and perform formal verification with Tamarin Prover[17]. In the above procedures, we meet several challenges as follows.

First, extracting message semantics is non-trivial, as some parameters may be encrypted, making their semantics hidden. To tackle this, we construct traffic- and NLP-based methods to identify the crypto function of each encrypted parameter. Then, the semantics of a parameter can be recovered according to the definition of the identified crypto function. Some common crypto functions can be identified by pattern matching on the real traffic, while it is almost impossible to define patterns for the unknown customized crypto functions. Since the parameters with customized crypto functions are usually specified in the IoT manufacturer documents offered by IoT platforms, we further develop a novel NLP-based method to directly extract the semantics of these parameters from the IoT manufacturer documents.

Second, considering the IoT context that involves multiple parties and multiple types of messages, active model learning cannot be directly applied to extract the interaction logic of MP implementations, as it only supports two parties and can be time-consuming when dealing with multiple types of messages. Moreover, when applying model learning to test MP implementations in the real world, they may produce uncertain responses due to uncontrolled factors, e.g., failing to receive an expected response due to timeout. In such a case, model learning may be trapped into an endless learning procedure, thereby failing to construct the state machine. To overcome these issues, we design an enhanced active model learning framework to support observing outputs from multiple parties. Further, to speed up the learning procedure, `MPInspector` cuts down unnecessary input tests. To overcome the uncertainty issue, `MPInspector` stops the learning procedure if the same state machine is constructed more than once.

Third, when performing formal verification, the traditional Tamarin Prover may fail to prove some properties, as some MP implementations have complex state transitions. In order to solve this problem, we design a helping oracle to guide the proof, which is a script that can help Tamarin Prover adjust the order of solving goals during the proof.

Evaluation. We apply `MPInspector` on three popular MPs, MQTT, CoAP and AMQP, implemented on nine leading IoT platforms (e.g., Google IoT Core, Azure IoT Hub) [20]. It successfully recovers the state machines of all the MP implementations and formally verifies their authentication and secrecy properties. The average overhead of end-to-end analysis is 4.5 hours with a precision of 100% in identifying property violations. Specifically, it checks 57 customized security properties and detects 252 property violations, leveraging which we further identify eleven types of attacks. These results and findings are alarming. Each platform at least violates 18 properties, which enables at least one attack. The resulting attacks have serious consequences, e.g., privacy leakage and malicious data injection. Our research further shows that the main root causes of risky MP implementations are: (1) the gap between ad-hoc MP implementations and the standard specification, (2) the undermined security mechanisms under the resource constrained IoT context, and (3) the lack of careful consideration about device sharing, multi-party involved communication situations under the IoT context.

Summary and contributions. Our key contributions are:

- We propose `MPInspector`, the first framework for automatic security analysis of MP implementations. `MPInspector` is precise on the detection of MP implementation flaws and is extensible and configurable to different IoT platforms and different protocols. We release `MPInspector` as an open-source tool for facilitating further studies.
- With `MPInspector`, we evaluate three popular MPs on nine leading IoT platforms and detect 252 property vi-

olations. We also uncover eleven kinds of attacks that exploit the combinations of property violations under practical threat models. We have responsibly reported these vulnerable implementations to the vendors and got acknowledged from vendors such as Tuya Smart.

2 Background

2.1 Cloud based IoT Platforms

Today, most IoT platforms (e.g., AWS and Azure) offer MP implementations, which serve as networking infrastructures for IoT manufactures and also called SaaS (Software-as-a-Service) applications. As shown in Figure 1, the service contains the message broker (can be configured by IoT manufactures), device SDKs (e.g., cameras and lockers) and APP SDKs (designed for terminal users). The device sends telemetry and event messages and receives command messages via MPs, and the user application also sends control commands to the devices remotely via MPs. We regard the device and the application as clients. All the messages between the device and the application are forwarded by the broker on the remote IoT platform. We regard the broker as the server. IoT device manufactures buy and deploy the SaaS application for MP to enable users remotely control their devices.

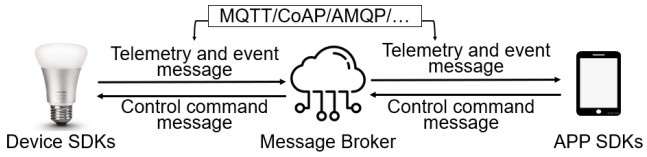


Figure 1: A typical architecture of MP implementations.

Studying the SaaS appliactions for MPs can cover most devices in the real world. A previous survey [19] shows that IoT manufactures simply deploy the SaaS without customization. As a result, security analysis of the SaaS appalications for MP can reflect the real-world threats.

2.2 MP Types and Implementations

Various MPs with distinct message types and formats have been implemented for IoT systems. For example, MQTT has nine key types of messages running over TCP. Among them, CONNECT is one type of MQTT messages, and it has five key parameters including `ClientID`, `Username`, `Password`, `WillTopic` and `WillMessage`. Meanwhile, CoAP has two types of messages running over UDP. Among them, CON is one type of CoAP messages, and it has six key parameters including `Uri`, `MessageId`, `Request`, `Option`, `Token` and `Payload`. For existing MPs, MQTT, CoAP and AMQP are the three most prominent MPs adopted by IoT platforms [20]. For more details and distinctions about these MPs, please refer to their standard specifications [1, 4, 46].

Based on the standard MP specification, MP implementations can be customized by the IoT platforms, including the configuration, the parameters in the messages and the message interaction logic. As for configurations, IoT platforms such as Aliyun Cloud and Tuya Smart optionally adopt the secure session protocol such as SSL/TLS. The configuration of secure session protocol may also be customized by IoT platforms. For example, Google IoT Core and Azure IoT Hub do not support authenticating a client by the certification on the server side. Instead, they adopt customized tokens for authentication. As for parameters, the parameters in messages can have customized semantics. For example, on AWS IoT Core, the `Username` and `Password` are not adopted in the implementation, while on Google IoT core, `Username` in a CONNECT message is composed of `ProjectId` and `deviceId`, e.g., `light123/dev1`. Besides, Tuya Smart assigns a control command and a timestamp to the payload in the PUBLISH message and encrypts these values by a private key using a customized crypto function. Moreover, the message interaction logic can be customized. As an example, Bosch IoT platform allows two clients with the same `ClientID` to be connected with the server at the same time, which is, however, not allowed in the standard MQTT specification.

3 Threat Model

We consider two practical attack scenarios as follows.

Neighbor scenario. In this scenario, the victim and attacker are within the same local network, e.g., in rental homes, and the attacker can perform network-based exploits. We apply the standard Dolev-Yao threat model [31] on the communication channel, under which the attacker can eavesdrop and modify all messages transferred on this channel and can impersonate a legitimate participator to inject messages.

Tenant scenario. Inspired by previous works [35, 36], the tenant scenario characterizes the situations where a victim uses some devices previously used by an attacker. Such cases include second-hand devices [9] and devices in hotels, Airbnb and rental homes [30]. In this scenario, when the attacker owns the device, he/she can collect the device identity including the password of the device or leave a backdoor on the device. After that, when the device is delivered to the victim, the attacker can use the collected identity or the injected backdoor to conduct attacks by sending some malicious command or publishing fake state of the device.

In both scenarios, the goal of the attacker is to exploit the flaws in the client-server interaction to take control of the victim device or monitor/manipulate the victim device data.

4 Design and Implementation

4.1 Overview

At a high level, `MPInspector` aims to automatically verify the security properties of MP implementations on different IoT platforms. Figure 2 provides an overview of `MPInspector`, which includes five modules: message semantics extraction,

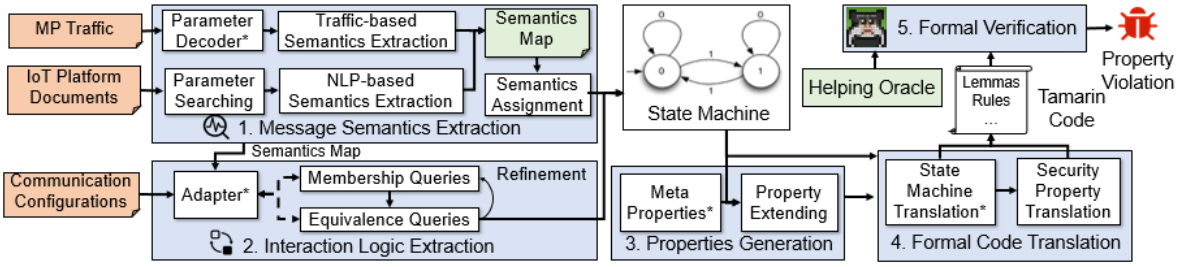


Figure 2: Overview of `MPInspector`. `MPInspector` supports automatically testing of any customized implementation of MQTT, CoAP, or AMQP out of the box. To support a new type of MP, the modules labeled with a star need to be extended.

interaction logic extraction, property generation, formal code translation and formal verification.

The workflow is as follows. First, the message semantics extraction module accepts MP traffic and IoT platform documents as inputs, and extracts the customized composition semantics of each parameter specified in the standard MP specification. Second, the interaction logic extraction module performs active model learning to infer the raw state machine by sending messages to the involved parties in the MP implementation and monitoring their responses. This module requires users to specify the communication configuration in order to generate the messages in the learning process. After these two stages, `MPInspector` adds the message semantics extracted from the first module to the transition messages in the raw state machine inferred in the second module to form a detailed state machine. Third, the property generation module extends the meta properties from the standard MP specification with the extended properties inferred from the detailed state machine to form the final security properties to be validated. Fourth, the formal code translation module translates the detailed state machine and security properties into Tamarin code. Finally, `MPInspector` applies Tamarin Prover to perform formal verification on the Tamarin code. The final outputs are the violated security properties. To make a clearer clarification, we take the MQTT implementation on the Bosch IoT platform as a running example to explain the main process, which is shown in Appendix B.

4.2 Inputs

`MPInspector` takes three inputs: MP traffic, IoT platform documents and communication configurations.

MP traffic. `MPInspector` accepts MP traffic to extract message semantics. The analyst can collect the traffic using his/her device and application to interact with the broker. He/she can set an access point (AP), to which his/her device and application are connected. Then, he/she can apply Wireshark or SSLSplit to record the traffic produced during the interaction. To collect as many different types of messages as possible, the analyst can perform different actions on the client, including sending commands and changing the state of the client.

IoT platform documents. IoT platform documents are supplements to identify the semantics of parameters that cannot be identified from MP traffic. IoT platforms generally offer rich semantics of these parameters in their publicly available documents for IoT manufacturers. However, the downside of the semantics information in the documents is that it might not match the real implementation. Therefore, we treat the documents as a secondary input and only use it when the parameter semantics cannot be extracted from the MP traffic.

Communication configurations. These configurations are required for `MPInspector` to generate real messages to communicate with the broker in the model learning process. They include the MP type and key communication arguments of the device or application, which can be collected from the device's or application's configuration file. Taking MQTT as an example, the key communication arguments are broker address, MQTT version, IoT platform name, raw password, secret key of the device or the application if exists, and the certifications if exist.

4.3 Message Semantics Extraction

The message semantics extraction module aims to extract the composition semantics of parameters in a message, which are of two types. First, a parameter can be a composition of several terms concatenated with delimiters, e.g., parameter `Username` with value `light123/dev1` is composed of `ProjectId` and `DeviceId`. Second, a parameter can be the encryption of several terms by a certain crypto function, e.g., a `Password` with the value of a complex character string can be the encryption of `ProjectId` and `ExpiredTime` by the JWT function [14]. Identifying the semantics of the second type of parameters is not trivial, as the value in the traffic does not have any meaning. To extract these two kinds of semantics, we provide two alternatives in this module. As shown in Figure 2, the message semantics extraction module mainly consists of traffic- and NLP-based semantics extraction. As the semantics extracted from the real MP traffic reflect the actual MP implementation, we prioritize the traffic-based semantics extraction. For the parameters whose semantics cannot be identified from the MP traffic, we resort to the NLP-based semantics extraction. Both of these methods output a semantics map, which maps the parameter values to their corresponding semantics.

For example, the pair {light123:ProjectId} means the semantics of the parameter light123 is ProjectId. In the last step, the two returned semantics maps are merged and fed to the semantics assignment component, which then replace the values in a message with the matched semantics from the semantics map. For parameters having no match in the semantics map, we still need to assign each of them a specific name for the following modeling task. Thus, we sequentially assign them a fake semantics, e.g., V0, V1, V2. Taking the parameter ClientID as an example, its extracted semantics may look like (V0,aud,V2,V3) where the aud means audience. Below, we detail the traffic- and NLP-based semantics extraction process.

For the traffic-based semantics extraction, the parameter parsing component first takes MP traffic as input and decodes the messages from the MP traffic to extract the values of the parameters. For some parameter values, their semantics can be directly inferred from the traffic, e,g., the Payload in a PUBLISH message may contain the format as key:value or key=value, and we can directly extract the key as the semantics of the value. Besides, there are also encrypted parameters whose semantics can only be recovered by identifying the corresponding crypto function. For common crypto functions, we find that the encrypted values have common patterns, e.g., the common pattern for JWT is $ey[A-Za-z0-9\_\backslash\backslash/+-]*\backslash\backslash.[A-Za-z0-9.\_\backslash\backslash/+-]*$). In our implementation, we provide the patterns of nine common crypto functions (e.g., JWT function and Base64 encoding). The semantics extracted from the aforementioned process are also added to the semantics map.

For the parameters whose semantics cannot be extracted from the MP traffic, e.g., the ones encrypted by unknown customized crypto functions, we propose an NLP-based semantics extraction method. Specifically, it extracts the semantics from IoT platform documents, which generally specify the semantics of parameters.

However, IoT platform documents are usually loosely formatted with sentences in different formats, posing challenges to semantics extraction. In our observation, the documents mainly include three types of sentences as shown in Figure 3: (1) structured sentence; (2) unstructured sentence in natural language; and (3) a mixed type sentence that contains both structured and unstructured parts.

Based on the above observation, we take the following steps. The parameter searching component takes IoT platform documents as input and parses sentences from the documents. For each parameter whose semantics cannot be extracted from the MP traffic, this component searches the sentences that contain the parameter. Then, the NLP-based semantics extraction component divides the sentences into the above three types and analyzes the three types of sentences one by one. This component first tries to extract semantics from the structured sentences. If not success, it extracts semantics from the mixed sentences and finally the unstructured sentences. The identified semantics will also be stored into a similar semantics map that will be used in the final semantics assignment component.

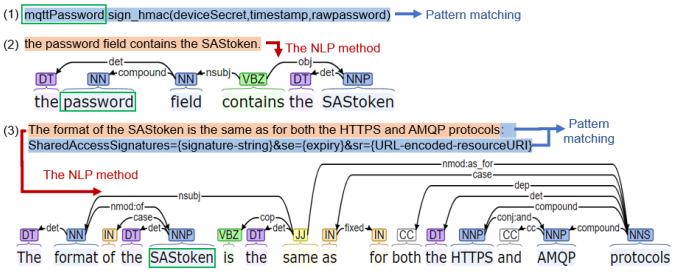


Figure 3: Example sentences of three types, including the structured, unstructured, mixed sentences.

In detail, for structured sentences, they have obvious structure and symbols that indicate the parameter semantics, which can be extracted by pattern matching. For unstructured sentences, the idea is to find a noun or a noun phrase that has an equivalence or inclusion relation with the target parameter. Thus, this module applies the Stanford dependency parser [44] to identify the equivalence relation and Part-of-Speech tagger [44] to identify the part of speech of each word in the sentence. For example, for the unstructured example in Figure 3, we can identify the target parameter password has the inclusion relation with the SAStoken, indicated by the word contain. For mixed sentences, the idea is to find the sentences satisfying two conditions: (1) the subject of the unstructured part is the target parameter, and (2) the structured and the unstructured parts are connected by equivalence symbols such as : and =, which indicate they have equivalence relation. Finally, this component performs pattern matching on the structured part to extract the semantics. For the mixed sentence example in Figure 3, MPInspector first divides the sentence into a structured part in blue and an unstructured part in yellow by the delimiter :. Then MPInspector identifies that the subject of the unstructured part is composed of the target parameter SAStoken, and finally applies the pattern matching to the structured part to identify the semantics of SAStoken.

4.4 Interaction Logic Extraction

This module aims to extract the raw state machine of the MP broker, since it is responsible for processing messages from clients and is closed-source. The state machine includes transition messages and transition logic. Transition messages represent the messages that are used to trigger the transition from one state to another, consisting of the input message to the broker and the response message from the broker. This module adopts active model learning, a framework to construct the state machine of a system by providing inputs and observing outputs. In MPInspector, the inputs are different permutations of message sequences sent to the MP broker

and the outputs are the relevant response message sequences.

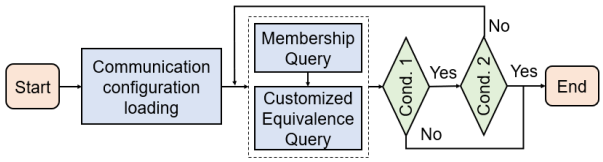
The basic model learning procedure is as follows. First, this approach adopts membership queries (MQs) to collect the responses to the inputs, and generates a state machine (also noted as a hypothesis). Then it performs equivalence queries (EQs) to seek an input that makes the hypothesis state machine and the real system have different outputs. This input is also called a counterexample that distinguishes the inferred state machine and the real system. If there is no counterexample, the inferred state machine is equivalent to the real system and is the final output of the interaction logic extraction module. Otherwise, a new round learning with MQs and EQs will be performed until there is no counterexample.

As shown in Figure 2, we have three components in this module: adapter, MQ and EQ. The adapter is designed to generate different input messages, send input messages to the broker, collect the response messages from the broker, and decode the response messages to identify their types. When generating an input message, the adapter directly uses the parameter values from the semantics map in Section 4.3. However, some parameters have dynamic values, e.g., a timestamp, which need to be generated by referring to their semantics in the semantics map. In addition, there are some dynamic parameters that are encrypted, for which the adapter follows the cryptographic algorithm in the their semantics to generate their values. Specifically, the adapter invokes the corresponding pre-installed encryption interface in `MPInspector`. For example, for `mqttPassword` introduced in Figure 3 from Section 4.3, the adapter invokes the HMAC interface and performs encryption of the timestamp and the raw password to generate the value of the parameter `mqttPassword`.

We implement the adapter for MQTT, CoAP, and AMQP, respectively. Based on the inputs and responses, MQs and EQs can infer the state machine of the broker.

The adapter in existing model learning frameworks usually only supports the communication of two parties, which is not applicable in the IoT context where multiple parties are usually involved. To tackle this, we extend the adapter by the following steps: (1) extending the adapter to support sending all types of messages that can be sent to the broker from all clients, and (2) monitoring the responses of the broker and all clients. Also, there are implicit responses from the broker. For example, in MQTT, the broker may accept the input message but give no response. In addition, the broker may accidentally close the connection without sending any response message. Therefore, we further extend the adapter to monitor the connection state of the broker and map the above two situations to two responses: `EMPTY` and `CONNECTIONCLOSED`, respectively.

Considering there may be many types of messages in the IoT context, the EQ component of existing model learning frameworks, e.g., Chow's W-Method [26], needs to send message sequences for all the permutations of the message types to the broker, leading to a high performance overhead. Therefore, we design a customized EQ component inspired by the previous work [29] to avoid useless queries to improve the efficiency. Specifically, we add a check to see if the connection has been closed when testing a sequence of input messages. If so, our learning procedure stops seeking counterexamples with this particular prefix of message sequences, as the following message sequences with this prefix will receive the same response, namely `CONNECTIONCLOSED`. Thus, it does not make sense to continue searching for counterexamples with this prefix. Our experiments prove that the customized EQ component reduces the query time by 34% compared to Chow's W-Method.



◆ Cond. 1: If an counterexample is found?
◆ Cond. 2: Is the number of same hypotheses greater than the threshold?

Figure 4: The learning procedure of active model learning.

Another challenge is that existing active learning models may be trapped into an endless learning procedure and thus fails to construct the state machine. For instance, when applying model learning in the real world, the targeted broker may produce uncertain responses, e.g., `EMPTY` response caused by timeout, due to uncontrolled factors such as environment. The EQ component may mistakenly take the uncertain response as a counterexample, which may further cause the same hypothesis to be generated repeatedly. To tackle this, we observe that the same hypothesis is generated if and only if it is equivalent to the MP broker. Therefore, we limit the maximum amount of the same hypothesis that is generated repeatedly to help terminate the learning procedure, which is shown in Figure 4. Additionally, we set a time delay to wait for the broker's response for a query, which can mitigate the uncertain response issue when performing MQs and EQs. The thresholds for the amount of the same hypothesis and the time delay can both be specified in the communication configurations.

After model learning, a raw state machine is generated whose transition messages only contain message names, e.g., `CONNECT/CONNACK`. Then, `MPInspector` adds the message semantics extracted from Section 4.3 to the transition messages in the raw state machine. In addition, we check if the MP implementation adopts SSL/TLS. If so, we insert the state transition with `KEYEXCHANGE{sesson_key}` after the initial state to denote the SSL/TLS mechanism, and add the SSL/TLS encryption semantics on the transition messages.

Apart from the inferred state machine, some unobservable internal protocol states called validity predicates can not be extracted by the model learning method and need to be modeled in Section 4.6 for verification. In our study, a validity predicate describes a constraint that a parameter should satisfy in a transition, e.g., the client's signature in a `password` parameter should be valid, or the current mes-

sage ID should be less than the received message ID. Thus, MPInspector extracts the validity predicates by utilizing the adapter to send messages with carefully mutated parameters to the server and observing if they are accepted or not. Particularly, MPInspector supports extracting the validity predicates with the Equality and LessThan constraints. Below are the corresponding mutation strategies. For the parameter with numerical type, MPInspector mutates it by adding or subtracting a random number to it. For other parameters, MPInspector changes one bit of their value for mutation.

4.5 Property Generation

The property generation module generates the security properties that should be verified on the extracted state machine. It aims to generate two groups of properties, including secrecy properties and authentication properties. The secrecy properties are for the confidential goal of certain parameters and the authentication properties are used to check if certain types of messages are authenticated. The parameters and messages that should be checked are first concluded from the standard MP specifications. This initial set of security properties are also called meta properties, including the secrecy properties (e.g., Meta_Sec_Set = {ClietnID, Username, Password,...}) and the authentication properties (e.g., Meta_Auth_Set = {CONNECT, CONNACK, SUBSCRIBE,...}). Second, we filter meta properties, whose targeted messages or parameters do not appear in the inferred state machine, as not all of the messages and parameters from the standard specification are used in IoT implementations. Finally, we add the extended properties based on the inferred state machine, as messages of the same type may have different parameter semantics in an MP implementation. For example, the CoAP implementation on Aliyun Cloud adopts two different CON messages with different parameter semantics for connecting and publishing messages to the broker, respectively. Thus, we add the parameters from such different messages to the secrecy property set and such different messages to the authentication property set. In conclusion, the only hard-coded part in the property generation module is the meta properties from the standard MP specifications. Note that this hard-code effort is required per MP type not per MP implementation. We demonstrate the generated detailed security properties for MQTT, CoAP and AMQP in Appendix A.

4.6 Formal Code Translation

The formal code translation module aims to translate the inferred state machine and security properties into Tamarin code, which can be further analyzed by Tamarin Prover. There are two components in this module including state machine translation and security property translation.

The inferred state machine is translated into rules in Tamarin, where a rule defines a transition in the state machine. A rule has a name and three parts, each of which is a sequence of facts: one for the rule's left-hand side, one for the rule's middle part called action fact, and one for the rule's right-hand side. Taking the simplified transition messages CONNECT/CONNACK that trigger the broker from state A to state B as an example, the transition indicates the broker receives a CONNECT message in state A, which is modeled as two facts including the fact In(connect) and the fact State_A_broker. The above two facts are put into the rule's left-hand side. The transition indicates the broker turns into state B and sends out a CONNACK message, which is modeled as two facts including the fact State_B_broker and the fact Out(connack). The above two facts are put into the rule's right-hand side. The action facts reason about the behaviours in the transition. For example, we use Commit(broker, connect) to reason one of the behaviours of the transition CONNECT/CONNACK. The rule supports let-binding expressions to specify the parameters in the message along with the detailed semantics, e.g., connect = <a,b>. After that, we have a simplified rule of the transition as shown in Listing 1.

We translate the transition messages from the perspectives of both the broker and the client to completely model an MP implementation. For example, CONNECT/CONNACK depicts the transition of the broker that it enters a new state and sends out a CONNACK message after receiving a CONNECT message. It also depicts the two transitions of the client: one describes that the client enters state D from a former state C after sending a CONNECT message to the broker, and another describes that the client enters state E from state D after receiving a CONNACK message from the broker.

```
rule broker_recv_connect_snd_conncak:
  let
    connack = <a>
    connect = <a, b>
  in [ In(connect), State_A_broker ]
  --[ Create('connect',broker),
      Commit(broker,client,connect),
      Running(broker,client,connack) ]
    ->[ Out(connack), State_B_broker]
```

Listing 1: An example rule in Tamarin code.

Additionally, for the validity predicates extracted from Section 4.4, MPInspector models them as a kind of action fact for the related rule's middle part. Particularly, MPInspector adopts the kind of action fact called restriction, which is offered by Tamarin. Restrictions specify constraints that a protocol transition should uphold, e.g., Equal(x,y) and LessThan(x,y). Since some validity predicates have the encryption semantics, MPInspector adds the corresponding encryption function to its action fact, e.g., Equal(verify(sig,m,pubkey),true), where verify(sig,m,pubkey) is a predefined function in Tamarin to verify the signature sig on the received message or parameter m. This action fact indicates that the verify function

equals to the constant `true`.

When translating the state machine, we first implement the initialization rules based on the provided initial state to set up the initial parameters that the broker and clients own. The initialization rule has a sequence of facts that describe the initialization of parameters in its left-hand side and a sequence of facts that describe the initial state in its right-hand side. Then, if the state machine considers the session key negotiation, we hard-code a general rule to model the transition, which is a simplified SSL/TLS key negotiation modeling. Finally, we follow the above translation principle to translate the transition messages into rules.

After state machine translation, the security properties are translated into lemmas, which are first-order logic formulas over time points and action facts, based on the standard security property templates specified from Tamarin Prover documentations [17]. Particularly, for each authentication property, `MPInspector` applies four types of authentication lemmas based on Lowe's taxonomy of authentication goals [42] to make a fine-grained analysis. Lowe defined four kinds of authentication goals including aliveness, weak agreement, non-injective agreement and injective agreement.

Based on the two threat models from Section 3, the formal code translation generates two Tamarin codes, on which Tamarin Prover will perform formal verification, respectively. In the neighbor scenario, the attacker sniffs the traffic and gets to know the session key. Thus, we add a fact to the right-hand side of the session key negotiation rule to indicate that the session key is leaked. In the tenant scenario, the attacker knows the initial parameters that the client owns in the initial state without sniffing the traffic. Thus, we add a fact to the right-hand side of the initial rules to indicate that the initial parameters are leaked.

4.7 Formal Verification

The formal verification module aims to validate the lemmas translated from the security properties on the rules translated from the state machine. In this module, we apply Tamarin Prover, an off-the-shelf tool for property verification. However, in the fully automatic mode of Tamarin, not all lemmas can be proved automatically due to the complex state machine, which is a common limitation of Tamarin Prover [17][22]. This limitation is related to the ranking of unproved goals extracted from the lemma. To overcome this, Tamarin Prover allows a user to supply heuristics called helping oracle to rank the unproved goals and guide the prove procedure. Therefore, we design and implement a new ranking strategy on the helping oracle, which is detailed as follows.

The unproved goals extracted from the lemma include validating the source of a state, the existence of an action fact that the attacker knows some parameters (e.g., secret keys, passwords, encrypted parameters), and other goals. First, we solve the unproved goals to validate the source of a state. Among these goals, the ones that contain a state of a longer trace in the state machine should be solved first, as they can be transformed into the goals that contain a state of a shorter trace. Second, we solve the unproved goals that validate the existence of an action fact indicating the attacker knows secret key or password. Third, we solve the unproved goals that validate the existence of an action indicating the attacker knows an encrypted parameter. This order can avoid the case of finding no proof path when solving the existence of an action that the attacker knows an encrypted parameter. Last, we apply the default ranking from Tamarin Prover for the remaining unproved goals. Our strategy helps Tamarin Prover automatically and efficiently validate the security properties. For instance, we apply our strategy to prove the authentication lemma of the `CONNECT` message on the server-side on AWS IoT Core. While the automatic mode will never be terminated, our helping oracle proves that this lemma is false, whose generated proof only costs 13 steps. As a result, our formal verification module is fully automatic thanks to the proposed helping oracle.

4.8 Extension for New Types of MPs

`MPInspector` has built-in support for security analysis on any customized implementation of MQTT, CoAP and AMQP. As for new customized MP implementations, the amount of work to be done is to offer three inputs including MP traffic, IoT platform documents and communication configurations, which is simple and requires minimum effort. We only need expert involvement when we need to analyze a new MP protocol. First, the message decoder in the message semantics extraction module and the adapter in the interaction logic extraction module need to be re-implemented according to the types and formats of the messages in the new MP. Second, the meta properties for the new MP need to be concluded, which include the necessary messages and parameters that should be authenticated and confidential. Third, the pre-extracted modeling knowledge from the standard MP specification, e.g., the initial states of the clients and the broker, need to be provided for the formal code translation module. All the knowledge required is not tied to a specific MP implementation and can be obtained from the public standard specification of the new MP. Note that the above operations are a one-shot effort for each new MP type. Actually, in real world, the number of popular MPs is limited and usually stable. Therefore, `MPInspector` is directly usable in most scenarios.

5 Evaluations

In this section, we utilize `MPInspector` to explore ten implementations of MQTT, CoAP and AMQP on nine leading IoT platforms. We aim to answer the following research questions:

- **RQ1:** How well do MP implementations on different platforms follow the security properties?
- **RQ2:** What are the reasons for property violations?

• **RQ3:** What kind of attacks can be triggered based on property violations?

• **RQ4:** How efficient and accurate is `MPInspector`?

5.1 Experiment Settings

We perform our experiments on a laptop with a 2.6 GHz 2-core Intel i5 processor and 8GB RAM, using Oracle Java Runtime version 1.8 (64 bit) in its default settings.

Evaluation subjects. To examine the effectiveness of `MPInspector`, we evaluate ten MP implementations from nine leading commercial IoT platforms [8], which are shown in Table 1. These implementations cover three main types of MPs, MQTT (including the widely adopted version V3.1.1 and the latest version V5.0), CoAP and AMQP V1.0. We perform our analysis by buying SaaS applications for MP from the IoT platforms so that the analysis can cover more devices in the real world that use these SaaS applications (see Section 2.1).

Among the ten evaluated MP implementations, five of them adopt SSL/TLS mechanism, including MQTT on Google IoT Core [11], Azure IoT Hub[6], AWS IoT Core[5], Bosch IoT Hub[7], and Aliyun Cloud [3]. We also analyze the secrecy and authenticity properties of MP implementations without SSL/TLS, including MQTT on Tuya Smart [18] and Mosquitto [15], CoAP on Aliyun Cloud [3] and EMQ X [10], and AMQP on ActiveMQ [2]. Because they are widely adopted by the device manufactures [8] and their security flaws may have a large practical impact.

Validation settings. We use the client SDK provided by the SaaS application to build potential victims for vulnerabilities and attack validation. As for Tuya Smart, who has acknowledged our findings, we further validate our findings on the real devices under their permission. We also build up scripts based on JavaScript to exploit the vulnerabilities and perform the validation attacks. Performed as an attacker, we manually check those lemmas guided by the attack paths generated by Tamarin Prover. Specifically, we use our scripts to see if we can acknowledge the secret or impersonate the agents in the communication between a server and a client.

Ethical consideration. Our study conducts active measurement on the real world MP implementations. As a result, we take several steps to ensure that our experiments are ethically sound and do not result in the disruption of other users and IoT platforms. First, we test the SaaS appliactions for MP on our own services bought from the IoT platforms, which does not disrupt other users. Second, when interacting with the broker on the IoT platforms, our messages are based on the normal traffic produced by us in our own SaaS applications, which does not disrupt the IoT platforms. Lastly, we validate our attacks on Tuya Smart with our own devices, which does not influence other devices or the platform.

Table 1: An overview of violated properties (noted as Pr.) in the ten MP implementations. For the checked properties, please refer to Table 6 and Table 7 in Appendix A.

Platform	MPs	Secrecy Pr.		Authentication Pr.	
		Neighbor Scenario	Tenant Scenario	Neighbor Scenario	Tenant Scenario
Google IoT Core	MQTT V3.1.1	**MS{1,3-6}**	MS{1-6}	MA{1-9}	MA{1,3, 5,7,9}
AWS IoT Core	MQTT V3.1.1	MS{1, 3-6}	MS{1, 3-6}	MA{1-10}	MA{1,3, 5,7,9-10}
Azure IoT Hub	MQTT V3.1.1	**MS{1, 3-6}**	MS{1-6}	MA{1-9}	MA{1,3, 5,7,9}
Bosch IoT Hub	MQTT V3.1.1	MS{1, 3-6}	MS{1, 3-6}	MA{1-9}	MA{1,3, 5,7,9}
Aliyun Cloud	MQTT V3.1.1	**MS{1, 3-6}**	MS{1-6}	MA{1-9}	MA{1,3, 5,7,9}
Tuya Smart	MQTT V3.1.1	**MS{1, 3-5}**	MS{1-6}	**MA{1-6, 8-9}**	MA{1,3, 5,7,9}
Mosquitto	MQTT V5.0	MS{1, 3-9}	MS{1, 3-9}	MA{1-11}	MA{1,3, 5,7,9-11}
EMQ X	CoAP	CS{1-6}	CS{1-6}	CA{1-4}	CA{1,3}
Aliyun Cloud	CoAP	**CS{1-4, 7, 9-10}**	CS{1-4, 7-10}	**CA{5-6, 8}**	CA{5,7}
ActiveMQ	AMQP V1.0	AS{1-5}	AS{1-5}	AA{1-13}	AA{1,3,5, 7, 9,11,13}

5.2 Property Validation

This section answers the questions **RQ1** and **RQ2**. We show the identified property violations in Table 1, where we find that all MP implementations encounter various authentication and secrecy property violations, and each MP implementation violates at least 18 properties.

5.2.1 Neighbor Scenario

In the neighbor scenario, `MPInspector` identifies that three out of the ten MP implementations (Mosquitto, EMQ X, and ActiveMQ) violate all the security properties. The rest of these implementations violate at least ten secrecy properties and five authentication properties.

Secrecy properties. We identify that five MP implementations (MQTT on Tuya Smart and Mosquitto, CoAP on Aliyun Cloud and EMQ X, AMQP on ActiveMQ) support transmitting messages in plain text. The other five MP implementations (MQTT on Google IoT Core, AWS IoT Core, Azure IoT Hub, Bosch IoT Hub, and Aliyun Cloud) adopt SSL/TLS but are still facing SSL/TLS interception risks because of wrong configurations. In addition, their messages can still be decrypted by man-in-the-middle attacks. As a result, for all the ten implementations, `MPInspector` identifies that the secrecy properties for the parameters without additional encryption are all failed. Below we discuss the secrecy properties on the parameters with additional encryption. Five MP implementations (Google IoT Core, Azure IoT Hub, Bosch IoT Hub, Aliyun Cloud, and Tuya Smart) deploy additional encryption on some of their parameters. Among them, Google IoT Core and Azure IoT Hub use a secret key to generate JWT and SAS tokens, which are valid before the expired time. In the neighbor scenario, the unexpired token can be reused by an attacker. Aliyun Cloud encrypts a client's secrets

with timestamps by a secret key. The CoAP implementation in Aliyun Cloud additionally encrypts the payload in the `POST_PUBLISH` message with a timestamp by a secret key. However, `MPInspector` validates that the timestamp is not checked by the server, which suggests that the password and payload in Aliyun Cloud can be reused as well. Tuya Smart uses a secret key to encrypt a client's password in the `CONNECT` message and encrypt the payload with a timestamp in the `PUBLISH` message. `MPInspector` identifies that Tuya Smart satisfies the secrecy property for `PUBLISH Payload` but fails the secrecy for the password.

Authentication properties. `MPInspector` validates authentication properties on both the client side and the server side. Table 1 shows the overview of the authentication property violations detected by `MPInspector`.

From the results, three MP implementations without any authentication mechanism (Mosquitto, EMQ X, and ActiveMQ) fail the aliveness goals of all authentication lemmas. Five MP implementations including Google IoT Core, AWS IoT Core, Azure IoT Hub, Bosch IoT Hub, and Aliyun Cloud that adopt SSL/TLS satisfy the non-injective goals on the `CONNECT` message of the server side. However, they still fail the non-injective goals on the `CONNECT` message because of SSL/TLS interception. Their other messages (`SUBSCRIBE`, `UNSUBSCRIBE`, `PUBLISH`, `DISCONNECT` messages) without authentication fail the aliveness goals. The rest two implementations (MQTT on Tuya Smart and CoAP on Aliyun Cloud) do not adopt SSL/TLS but adopt an encryption mechanism on their messages. For Tuya Smart, the `CONNECT` message on the server side satisfies the aliveness goal but fails the weak agreement goal. Therefore, even though the password is encrypted by a secret key, the attacker can still sniff and reuse on the `CONNECT` message. For Aliyun Cloud's CoAP implementation, it has encryption but does not check the timestamp in `CON_POSTAUTH` and `CON_POSTPUBLISH` messages. Therefore, an attacker can connect with the server by replaying the messages he collected from the client previously. As a result, in Aliyun Cloud, authentications on `CON_POSTAUTH` and `CON_POSTPUBLISH` messages satisfy the weak agreement goal but fail the non-injective goal.

5.2.2 Tenant Scenario

In the tenant scenario, `MPInspector` has identified that all the secrecy properties are violated in all the ten implementations. The reason is that the attacker can impersonate the victim to connect with the server and accept all the messages from the server. For authentication properties, `MPInspector` identifies that all the ten implementations violate all the properties on the server side, but meet the properties on the device side. This is due to the differences of the attacker's capabilities to control the device side and the server side. On the device side, the attacker cannot steal the session key as he may not be in the same network with the victim. While on the server side, the attacker can create a fake client to connect with the server using the identities he created when he has access to the device. Then, the server would recognize the fake client as a legitimate one, which allows the attacker to break all the authentication goals.

Table 2: Attacks and relevant property (noted as Related Pr.) violations (●=validated, ◐=partially validated).

Neighbor Scenario	**Affected Protocol**	**Affected Platforms**	**Related Pr.**	**Verified**
Man-in-the-middle	All protocols	All platforms	MA{1-9}, AA{1-13}, CA{1-8}	●
Replay Attack	MQTT V3.1.1	AWS IoT Core	MA{1-9}	●
		Tuya Smart		
	MQTT V5.0	Mosquirro	MA{1-9}, MA{10-11}	●
	CoAP	EMQ X	CA{1-4}	●
	AMQP V1.0	ActiveMQ	AA{1-13}	●
Transfer Sync. Failure	AMQP V1.0	ActiveMQ	AA{1-9}	●
Tenant Scenario	**Affected Protocol**	**Affected Platforms**	**Related Pr.**	**Verified**
Client Identity Hijacking	MQTT V3.1.1	Google IoT Core	MS{1-7}, MA{1,3,5, 7,9}	●
		Azure IoT Hub		
		AWS IoT Core		
		Aliyun Cloud		
	MQTT V5.0	Mosquitto		
	AMQP V1.0	ActiveMQ	AS{1-5}, AA{1.3,5, 7,9,11,13}	●
	CoAP	EMQ X	CS{1-11}, CA{1,3,5,7}	●
		Aliyun Cloud		
Reflection Attack	CoAP	EMQ X	CS1, CA{1,3,5,7}	◐
		Aliyun Cloud		
Malicious Topic Subscription	MQTT V3.1.1	AWS IoT Core	MS{5,7-9}, MA3	●
	AMQP V1.0	ActiveMQ	AS{2,4}, AA9	●
Malicious Topic Publish	MQTT V3.1.1	AWS IoT Core	MS{5,7-9}, MA7	●
	CoAP	EMQ X	CS1,CA3	●
Malicious Response Topic Publish	MQTT V5.0	Mosquitto	MS{5,7-9}, MA7	◐
Unauthorized Will Message	MQTT V3.1.1	AWS IoT Core	MA{1, 10}	●
	MQTT V5.0	Mosquitto		
Unauthorized Retained Message	MQTT V5.0	Mosquitto	MA{8, 11}	●
Illegal Occupation	AMQP V1.0	ActiveMQ	AS1, AA{1, 3}	●

5.3 Attacks based on the Property Violations

This section answers the question **RQ3**. Based on the property violations, we uncover eleven kinds of attacks on the ten MP implementations and display the overview in Table 2. We find that the examined MP implementations are all vulnerable under the two attack scenarios. Each platform is vulnerable to at least one attack, and on average 2.8 attacks. These attacks have serious consequences, such as sensitive data leakage and malicious message injection. We introduce six attacks below (more attacks are available in [52]).

5.3.1 Neighbor Scenario Attacks

Replay attack. This attack is due to the authentication property violations, which suggests that the server accepts the messages that the client has sent before. An attacker only needs to collect them and replays them to the server. We identify that CoAP on EMQ X, AMQP on ActiveMQ, and MQTT on Tuya Smart, AWS IoT Core and Mosquitto are vulnerable to this attack. We launch this attack on Tuya Smart and Mosquitto by sniffing and collecting the traffic in the local network and replaying them to the server. As a result, We successfully replay all the messages, including sending commands and telemetry data.

AMQP sync. failure. `MPInspector` finds that the client and server in AMQP strictly maintain the message ID called Delivery ID when sending TRANSFER messages. Utilizing the authentication property violations on AMQP messages, an attacker can kick the victim offline by sending the messages in wrong orders or forging the TRANSFER messages with synchronized Delivery ID using the victim's identity. We identify the attack on ActiveMQ. We develop an attack script using Ettercap and have successfully launched the attack on ActiveMQ.

5.3.2 Tenant Scenario Attacks

Client identity hijacking. `MPInspector` detects that the secrecy properties on the device side are all violated in the tenant scenario. Additionally, the server side authentication properties are also violated. This suggests that an attacker can impersonate the victim device using its identity to connect to the server. We name this attack as client identity hijacking. Especially, `MPInspector` detects that the MQTT implementations disconnect the existing client when the server receives a second connection request with the same ClientID. Therefore, an attacker can use the victim's identity to connect to the server and kick the victim offline. Last, an attacker can impersonate the device to send messages to the server. We successfully launch this attack on Google IoT Core, AWS IoT Core, Aliyun Cloud, Tuya Smart, Mosquitto, and ActiveMQ. Additionally, we find that the attacker once obtains the credentials of the client, he can perform this attack for a long time as these IoT platforms hard-code the credentials of the clients into device SDKs and cannot dynamically revoke or grant new credentials.

Reflection attack. The reflection attack is specific to the CoAP protocol, which is running over UDP. Utilizing the secrecy property and authentication property violations on the MP implementations in the server side, an attacker can forge messages using the victim's IP address to send to the server. The workflow is shown in Figure 5. We identify the attack of CoAP on Aliyun Cloud and EMQ X. As a consequence, an attacker can forge a fake state to deceive the server. Also, the attacker can forge a message to get a considerable amount of messages sent to the victim and cause a DoS. To validate the attack, we use source address spoofing to forge a CoAP message, and the victim successfully receives the unexpected response message. According to our experiments, the amplification reflection rates are 2.25 in Aliyun Cloud and 0.68 in EMQ X, respectively. The amplification reflection rate here is a conservative estimation because we adopt the basic configuration, where the broker only returns the response code without the device data.

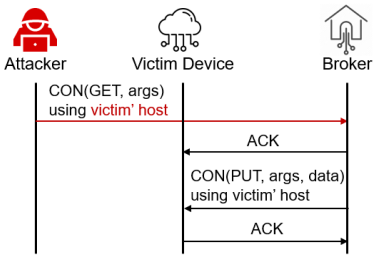


Figure 5: CoAP reflection attack.

Malicious topic subscription. Because of the secrecy property violation on the topic name and the authentication property violation in the `SUBSCRIBE` message, an attacker can subscribe to the victim's topic using his own identity. Taking AMQP as an example, as shown in Figure 6, an attacker uses his own identity `ContainerId` to subscribe to the victim's topic, which is denoted as the target node. When the victim device sends its secret data, the broker transfers the secret data to the attacker. We identify this attack on AWS IoT Core and ActiveMQ and further validate this attack successfully.

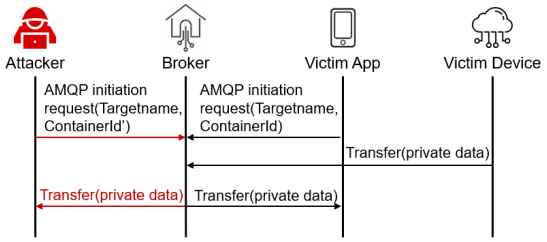


Figure 6: AMQP malicious topic subscription on ActiveMQ.

Unauthorized response message. This attack works for the new request/response mechanism introduced by MQTT V5.0. This mechanism allows the client to publish a message with a response topic and the correlation data. The client who receives this message publishes the correlation data to the response topic. However, an attacker can publish with an unauthorized response topic to the victim, as shown in Figure 7. This attack is based on the secrecy property violation on the victim's topic. It is identified on Mosquitto as it supports MQTT V5.0. To validate the attack, we use our script to simulate the victim and accomplish the request/response mechanism. We successfully launch the attack as the broker does not check the authenticity of the response topic.

Illegal occupation. An attacker can exploit the violated secrecy property on the victim's `ContainerId` and the violated

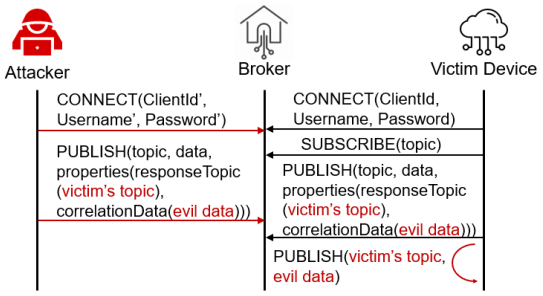


Figure 7: MQTT V5.0 unauthorized response topic publish.

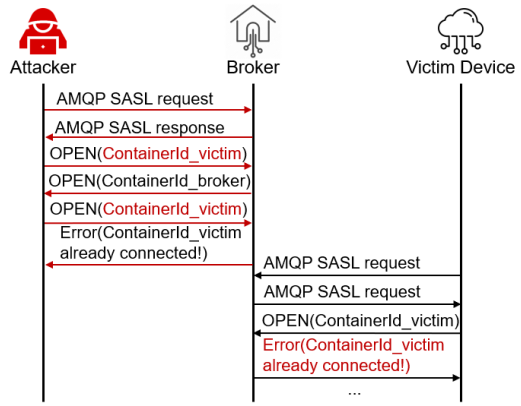


Figure 8: AMQP illegal occupation.

authentication property to perform illegal occupation attacks on AMQP. The server that receives duplicate `OPEN` messages with the same `ContainerId` of the victim closes the connection without updating the session state. When the client reconnects to the server, the server believes that the client with `ContainerId` is online and rejects the victim's connection request. We identify this attack on ActiveMQ, and we believe this attack is severe. As shown in Figure 8, an attacker can collect victims's `ContainerIds` to perform this attack, and make plenty of victims out of service unless the broker resets. We use our script to launch this attack successfully, and the target victim cannot connect to the broker anymore.

5.3.3 Comparisons with Burglars' IoT Paradise Paper

In [36], Jia *et al.* performed a manual analysis on MQTT manually. We compare `MPInspector` with [36], which is shown in Table 3. Our framework is automatic while [36] only analyzed MQTT manually. In addition, `MPInspector` covers four prominent MPs including MQTT V3.1.1, MQTT V5.0, CoAP and AMQP V1.0 while [36] only analyzed MQTT V3.1.1. As for MQTT V3.1.1, we find four new attacks that [36] did not cover. We consider the neighbor scenario and the tenant scenario and [36] only considered the latter. There are two attacks in [36] that `MPInspector` does not cover. However, these two attacks are either not MQTT's implementation flaw or related to the understanding of bit wise parameters, which are out of the current design focus of `MPInspector`.

Instead, `MPInspector` is mainly designed for logic flaw analysis on MP implementations. In conclusion, compared with the previous work [36], `MPInspector` is an automatic approach, covers more MPs and reveals four more new attacks.

Table 3: Our results compared with [36] in MQTT ((●=detected, ○=not detected)).

Scenario	Types of Attacks	[36]	MPInspector
Neighbor Scenario	Man in the Middle	○	●
	Replay Attack	○	●
Tenant Scenario	Unauthorized Will Message	●	●
	Malicious Retained Message	●	●
	Client Identity Hijacking	●	●
	ClientID identification	●	○
	Malicious Topic Subscription	●	●
	Malicious Topic Publish	○	●
	Wildcard-topic Subscription	●	○
	Unauthorized Response Topic Publish	○	●

5.4 Performance

This section answers question **RQ4**. We evaluate the performance of `MPInspector` from three perspectives: (1) state machine modeling, (2) property violation detection, and (3) performance overhead.

Evaluation on state machine modeling. The state machine modeling includes message semantics extraction, interaction logic extraction and formal code translation. We first evaluate the performance of `MPInspector` on message semantics extraction on the ten tested MP implementations. As MP implementations are closed-sourced, it is difficult to get the ground truth of the message semantics for the real MP implementations. Thus, we invite 45 experts with abundant protocol and software reverse engineering experiences to manually validate our results. Since recovering the full message semantics depends on the amount of collected MP traffic and the quality of IoT platform documents, the experts are instructed to only focus on checking the correctness of each parameter semantics extracted by `MPInspector` by checking all the available traffic and documents. Thus, as a precaution, we only report the precision. As a result, the precision of message semantics extraction on Aliyun Cloud is 96%, while the precision on other IoT platforms is all 100%. As the value of parameter `ClientID` from Aliyun Cloud includes some irregular characters, our method cannot handle them and mistakenly extracts wrong terms of the parameter. Additionally, to prove the effectiveness of our NLP-based semantics extraction, we further collect the documents from 20 popular IoT platforms [13] for evaluation. Similarly, our invited experts manually verify the correctness of each extracted parameter semantics by examining the collected documents. Our method yields 94.87% precision. Our method fails to extract the semantics of some parameters, because the sentences that contain these parameter semantics do not belong to the considered sentence types in Section 4.3 and they need to be extracted from several

Table 4: Performance overhead of MPInspector.

IoT Platform	MP	Message semantics Extraction		Interaction Logic Extraction						Formal code Translation	Total Time (h:mm)
		Time (ms)	Precision	States	Time Delay	# of Input Message Types	# MQs	# EQs	Time (h:mm)	Time (ms)	
Google IoT Core	MQTT V3.1.1	115	1.00	3	8s	5	215	373	06:32	0.04	06:32
AWS IoT Core	MQTT V3.1.1	102	1.00	3	3s	5	155	116	02:29	0.06	02:29
AWS IoT Core(will)	MQTT V3.1.1	103	1.00	8	5s	4	727	123	04:37	0.67	04:37
Azure IoT Hub	MQTT V3.1.1	107	1.00	3	8s	5	65	393	05:31	0.04	05:31
Bosch IoT Hub	MQTT V3.1.1	106	1.00	5	9s	5	184	599	09:38	0.03	09:38
Aliyun Cloud	MQTT V3.1.1	105	0.96	3	4s	5	62	1361	07:46	0.08	07:46
Tuya Smart	MQTT V3.1.1	110	1.00	3	8s	5	65	393	04:53	0.03	04:53
Mosquitto	MQTT V5.0	106	1.00	2	1s	5	65	393	00:23	0.03	00:23
Mosquitto(will)	MQTT V5.0	106	1.00	6	5s	4	317	123	03:13	1.26	03:13
Mosquitto(retain)	MQTT V5.0	106	1.00	8	7s	6	727	749	08:02	1.18	08:02
EMQ X	CoAP	928	1.00	1	1s	4	24	420	03:47	125	03:47
Aliyun Cloud	CoAP	2152	1.00	2	1s	3	27	273	04:07	1627	04:07
ActiveMQ	AMQP V1.0	1808	1.00	9	1s	8	728	846	05:11	1917	05:11

Table 5: Performance of MPInspector on property violation detection.

	Google IoT Core	AWS IoT Core	Azure IoT Hub	Bosch IoT Hub	Aliyun Cloud	Tuya Smart	Mosquitto	EMQ X	Aliyun Cloud	ActiveMQ	Average
Protocol	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v3.1.1	MQTT v5.0	CoAP	CoAP	AMQP1.0	/
Precision	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
False Positive rate	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

sentences. For more details on precision of message semantics extraction, please refer to Table 4.

As for interaction logic extraction, we choose four MP implementations for the evaluation, including Mosquitto, EMQ X, ActiveMQ, and Tuya Smart. The first three are chosen because they are open-source, thus our experts can refer to their code for the ground truth. Although Tuya Smart is not open-source, with the help of their security team, we can manually review and confirm the result of Tuya Smart. We cannot validate the other six platforms as we do not have access to their source code. The validation shows that the state machines learned by MPInspector are consistent with these four implementations. As for model translation, we successfully translate all MP state machines into Tamarin code and validate that the codes can successfully run.

Evaluation on property violation detection. Since it is difficult to identify all the security property violations of an MP implementation in practice, we also resort to the 45 experts to manually confirm each of the identified property violations by MPInspector. Therefore, we only report precision. Specifically, the experts act as attackers to perform PoC attacks under the threat models specified in Section 3. For secrecy properties, they try their best to retrieve the values of the parameters specified in the target secrecy properties by reversing the traffic, application and device. If the parameter value can be retrieved, we consider the corresponding secrecy property is violated. As for authentication properties, they try to complete the interactions by forging the messages in the target authentication properties. If the interactions can be completed by them, we consider the target authentication properties are violated. As a result, the average precision of property violation detection on the ten MP implementations is 100%. For more details on the property violation detection, please refer to Table 5.

Performance overhead. We evaluate the overhead of each component in MPInspector and the end-to-end system. The overall overhead of MP implementations is determinated by the time consumption of the interaction logic extraction module, as other modules' overhead is less than 2152 ms. The average overhead of the end-to-end system is ~4.5 hours. Considering the interaction logic extraction is a one-shot task, the overhead of MPInspector is acceptable. For more details on precision of performance overhead, please refer to Table 4.

6 Discussions

6.1 Lessons

Based on our evaluation, we conclude that existing popular MPs do not meet the security requirements mainly for the following three reasons.

Gap between implementations and specifications. Many real-world MP implementations do not completely match the standard specification, which on the other hand might be too complex for developers to follow. Developers usually have their own understanding about MPs, which leads to some conflicting implementations. For example, the MQTT on Bosch IoT Hub allows two clients with the same ClientID to be connected to the broker, while the AMQP on ActiveMQ keeps the connection state of a client even when the client is offline.The above implementations all violate their specifications and can be vulnerable.

Gap between constraint resources and security requirements. Under the resource-constrained IoT context, developers usually cut down some security functions. For example, Google IoT Core does not support authentication on the server-side, and the updated version of MQTT on Tuya Smart does not support authentication based on certifications but leverages a vulnerable PSK algorithm instead. These incomplete security mechanisms are due to that the credential management of numerous devices is challenging and resource-constrained devices cannot support big certificate files.

Gap between the MP security design and adversarial environments. In terms of the MP design, we find that most developers do not carefully consider the adversarial environments. First, the adversarial device-sharing cases are not considered. The devices' credentials in some MP implementations are not updated, which may lead to client identity hijacking. Second, the access control of participants is improper. For instance, the request/response mechanism introduced by MQTT V5.0 does not limit a client's authority on the response topic, which may cause malicious message injection.

Suggestions. With the observations from the security analysis, we make the following suggestions for manufacturers. First, manufacturers should guarantee secure communications. The message integrity and confidentiality should be carefully protected. MP implementations should use SSL/TLS with careful configurations, and additional message encryption is highly recommended. Second, manufacturers need to adopt strict authentication mechanisms. The device and server should not only authenticate the initial connection but also authenticate the messages sent to the agents in every phase. Besides, the timestamp or message sequences should be applied to avoid replay attacks. Third, clients' credentials should be dynamically granted to the device or revoked from the device. Currently, most MP implementations have hard-coded the device credential into the SDKs, which makes it hard for updating the credentials. Last but not least, the client and server should have fine-grained resource access control. In particular, we suggest that the identity of a client and her resource should be carefully protected.

6.2 Limitations and Future Work

A limitation of `MPInspector` is that we only infer the interaction logic and parameter-level semantics of the MP implementations. An interesting future work is to explore the fine-grained testing and more flexible model learning strategies to catch more fine-grained information of MP implementations. To illustrate, a bit-wise mutation of a specific parameter in MP messages can help detect if the implementation has appropriately checked the input messages. In addition, it will also be more efficient to apply NLP techniques to analyze the protocol specifications to extract the meta properties. Also, it is worth mentioning that studying SaaS appliactions might get different results comparing to studying real devices as IoT vendors may configure the SaaS appliactions and introduce some security mechanisms to accomplish the interaction between clients and the server.

7 Related Work

State machine learning. A few literature [40] works on automatically extracting state machines from protocol implementations. While these works are effective under the white box setting where the protocol's source code is available, they are not very helpful for MP implementations as most of them are not open-source. In comparison, `MPInspector` does not use the source code. Model learning has also been applied to analyze TLS in [29]. A similar approach is also used in TLS hostname verification [51].

Formal verification of protocols. In the meanwhile, numbers of verification tools are developed such as ProVerif [24] and Tamarin [17]. Those tools with formal verification have been proved valuable in assessing the security of protocols, such as TLS 1.3 [23, 28], LTE [34] and 5G AKA [22, 27]. By contrast, our framework focuses on the security analysis on protocol implementations. The idea of combining model learning and model checking was applied in the analysis of TCP and SSH protocols [32, 33]. Comparing to these works, we extend this idea in a more automatic way and come up with the first framework for the security analysis of MP implementations.

Security studies on IoT protocols. Researchers have studied the security of IoT communication protocols such as BLE, ZigBee, and Z-Wave [12, 50]. However, little work has been done to understand the security of IoT MPs, such as MQTT, AMQP, and CoAP. There are only a few ad-hoc attacks reported. Previous work [16] reveals that attackers can exploit MQTT by connecting the server without authentication and [47, 53] confirmed the attack in real world. [35] performed security evaluation on IoT devices' interaction applying the "shared devices attack model". [48] presented HomeSnitch to identify a device's behavior in smart home. In addition, Andrea *et al.* [49] constructed a tool called MQTTSA to detect the configuration flaw in MQTT deployments based on the source code. The closest to our work is [36], which performs a manual security evaluation on MQTT and identifies several design vulnerabilities. We compare `MPInspector` with [36] in detail in Section 5.3.3. `MPInspector` is an automatic approach, covers more MPs and reveals four more new attacks.

8 Conclusion

To systematically understand the security of MPs implemented on IoT platforms, we present `MPInspector`, an automatic and systematic framework to recover MP implementations and reveal the gap between protocol implementations and the desired security properties. `MPInspector` achieves

automated and systematic security analysis by combining model learning and formal analysis. We apply `MPInspector` to ten implementations of three popular MPs on nine leading commercial IoT platforms, and identify 252 property violations and eleven attacks. We also present the understanding of the MP implementation flaws and discuss the mitigation and future work. To facilitate future IoT security research, we open source `MPInspector` at [52].

Acknowledgments

We sincerely appreciate our shepherds Omar Chowdhury and Adwait Nadkarni, and all the anonymous reviewers for their valuable comments to improve our paper. We also thank Chenyang Lyu, Yuwei Li, Tianyu Du, Changjiang Li, Yuan Chen, Hong Liang and Han Bao for proofreading this paper.

This work was partly supported by NSFC under No. U1936215, 61772466, and U1836202, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), the State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093) (2020-MS-12), the Zhejiang Provincial Natural Science Foundation under No. LQ21F020010, and the Ant Financial Research Funding.

References

[1] The Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7252s.

[2] ActiveMQ. https://activemq.apache.org/.

[3] Aliyun Cloud. https://iot.aliyun.com.

[4] AMQP Version 1.0. https://www.amqp.org/resources/specifications.

[5] AWS IoT Core. https://aws.amazon.com/iot/.

[6] Azure IoT Hub. https://azure.microsoft.com/services/iot-hub/.

[7] Bosch IoT Hub. https://developer.bosch-iot-suite.com.

[8] Competitive Landscape: IoT Platform Vendors. https://www.gartner.com/en/documents/3983934/competitive-landscape-iot-platform-vendors. Accessed May 22, 2020.

[9] eBay's 2017 Shopping Report Shows Strong IoT Growth. https://www.androidheadlines.com/2018/01/ebays-2017-shopping-report-shows-strong-iot-growth.html.

[10] EMQ X. https://github.com/emqx/emqx-coap.

[11] Google IoT Core. https://cloud.google.com/solutions/iot/.

[12] Honey, I'm home!!- Hacking Z-Wave Home Automation Systems. https://www.blackhat.com/us-13/archives.html#Fouladi.

[13] IoT Cloud Platform Landscape. https://www.postscapes.com/internet-of-things-platforms/.

[14] JSON Web Tokens(JWT). https://tools.ietf.org/html/rfc7519.

[15] Mosquitto. https://mosquitto.org/.

[16] Taking Over The World Through Mqtt Aftermath. https://www.blackhat.com/docs/us-17/thursday/us-17-Lundgren-Taking-Over-The-World-Through-Mqtt-Aftermath.pdf.

[17] The Tamarin Manual. http://tamarin-prover.github.io/manual/.

[18] Tuya Smart. https://en.tuya.com/.

[19] Why Should You Build Your Own IoT Platform. https://medium.com/tomorrow-plus-plus/why-should-you-build-your-own-iot-platform-dff51578c0c.

[20] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Commun. Surveys. Tuts.*, 17(4):2347–2376, 2015.

[21] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. and Comput.*, 75(2):87–106, 1987.

[22] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler. A formal analysis of 5G authentication. In *CCS*, pages 1383–1396, 2018.

[23] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE S&P*, pages 483–502. IEEE, 2017.

[24] B. Blanchet et al. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, volume 1, pages 82–96. Citeseer, 2001.

[25] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity iot. In *USENIX Security*, pages 1687–1704, 2018.

[26] T. S. Chow. *Testing software design modeled by finite-state machines*. 1995.

[27] C. Cremers and M. Dehnel-Wild. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. In *NDSS*, 2020.

[28] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *CCS*, pages 1773–1788, 2017.

[29] J. De Ruiter and E. Poll. Protocol State Fuzzing of TLS Implementations. In *USENIX Security*, pages 193–206, 2015.

[30] R. Dey, S. Sultana, A. Razi, and P. J. Wisniewski. Exploring smart home device use by airbnb hosts. In *Extended Abstracts of CHI Conference on Human Factors in Computing Systems*, pages 1–8, 2020.

[31] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[32] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *CAV*, pages 454–471. Springer, 2016.

[33] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg. Model learning and model checking of SSH implementations. In *SPIN*, pages 142–151, 2017.

[34] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *NDSS*, 2018.

[35] B. Janes, H. Crawford, and T. Oconnor. Never ending story: Authentication and access control design flaws in shared iot devices. In *IEEE Workshop on the Internet of Safe Things*, 2020.

[36] 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 *IEEE S&P*, pages 465–481. IEEE, 2020.

[37] J. Y. less, R. Holz, W. Hu, and S. Jha. Automated analysis of secure internet of things protocols. In *ACSAC*, pages 238–249, 2017.

[38] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, et al. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security*, 2021.

[39] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE Transactions on Cybernetics*, 2020.

[40] D. Lie, A. Chou, D. Engler, and D. L. Dill. A simple method for extracting models from protocol code. In *ISCA*, pages 192–203. IEEE, 2001.

[41] Q. liu, S. Ji, C. Liu, and C. Wu. A practical black-box attack on source code authorship identification classifiers. *TIFS*, 2021.

[42] G. Lowe. A hierarchy of authentication specifications. In *CSFW*, pages 31–43. IEEE, 1997.

[43] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *USENIX Security*, pages 1949–1966, Santa Clara, CA, 2019.

[44] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[45] I. N. McAteer, M. I. Malik, Z. Baig, and P. Hannay. Security vulnerabilities and cyber threat analysis of the AMQP protocol for the internet of things. 2017.

[46] OASIS. MQTT Version 3.1.1. http://docsss.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[47] T. OConnor, W. Enck, and B. Reaves. Blinded and confused: uncovering systemic flaws in device telemetry for smart-home internet of things. In *WiSec*, pages 140–150, 2019.

[48] T. OConnor, R. Mohamed, M. Miettinen, W. Enck, B. Reaves, and A.-R. Sadeghi. Homesnitch: behavior transparency and control for smart home iot devices. In *WiSec*, pages 128–138, 2019.

[49] A. Palmieri, P. Prem, S. Ranise, U. Morelli, and T. Ahmad. MQTTSA: a tool for automatically assisting the secure deployments of MQTT brokers. In *SERVICES*, volume 2642, pages 47–53. IEEE, 2019.

[50] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn. Iot goes nuclear: Creating a zigbee chain reaction. In *IEEE S&P*, pages 195–212. IEEE, 2017.

[51] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *IEEE S&P*, pages 521–538. IEEE, 2017.

[52] 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. https://github.com/wqqqy/MPInspector.

[53] B. Zhao, S. Ji, W.-H. Lee, C. Lin, H. Weng, J. Wu, P. Zhou, L. Fang, and R. Beyah. A large-scale empirical study on the vulnerability of deployed iot devices. *TDSC*, 2020.

[54] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang. Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In *USENIX Security*, pages 1133–1150, 2019.

Appendix

A Security properties.

We present the main evaluated secrecy and authentication properties (both meta and extended properties) in Table 6 and Table 7, respectively.

Table 6: Secrecy properties.

ID	Secrecy Property Description
MS1	Secrecy on MQTT ClientID
⋆ MS2	Secrecy on MQTT Secret Key
MS3	Secrecy on MQTT Username
MS4	Secrecy on MQTT Password
MS5	Secrecy on MQTT Topic
MS6	Secrecy on MQTT Publish Payload
MS7	Secrecy on MQTT User Properties (MQTT V5.0)
MS8	Secrecy on MQTT Publish Response Topic (MQTT V5.0)
MS9	Secrecy on MQTT Publish CorelationData (MQTT V5.0)
AS1	Secrecy on AMQP ContainerId
AS2	Secrecy on AMQP Host Name
AS3	Secrecy on AMQP Transfer Payload
AS4	Secrecy on AMQP Target Node
AS5	Secrecy on AMQP Source Node
CS1	Secrecy on CoAP Uri
CS2	Secrecy on CoAP Token
CS3	Secrecy on CoAP MessageId
CS4	Secrecy on CoAP ACK payload
⋆ CS5	Secrecy on CoAP CON_GET Payload (EMQ X)
⋆ CS6	Secrecy on CoAP CON_PUT Payload (EMQ X)
⋆ CS7	Secrecy on CoAP Random (Aliyun Cloud)
⋆ CS8	Secrecy on CoAP Secret Key (Aliyun Cloud)
⋆ CS9	Secrecy on CoAP AuthToken (Aliyun Cloud)
⋆ CS10	Secrecy on CoAP CON_POSTAUTH payload (Aliyun Cloud)
⋆ CS11	Secrecy on CoAP CON_POSTPUBLISH payload (Aliyun Cloud)

[1] The property with ⋆ is extended property.
[2] MS7-MS9 are only supported in MQTTv5.0, CS5-CS6 are only supported in EMQ X and CS7-CS11 are only supported in Aliyun Cloud in CoAP protocol.

B A Running Example

We take the MQTT implementation on Bosch IoT platform as a running example to clarify how the state machine is generated and how the formal code is translated.

Table 7: Authentication properties.

ID	Property Description
MA1	Authentication on MQTT CONNECT message (server->client)
MA2	Authentication on MQTT CONNACK message (client->server)
MA3	Authentication on MQTT SUBSCRIBE message (server->client)
MA4	Authentication on MQTT SUBACK message (client->server)
MA5	Authentication on MQTT UNSUBSCRIBE message (server->client)
MA6	Authentication on MQTT UNSUBACK message (client->server)
MA7	Authentication on MQTT PUBLISH message (server->client)
MA8	Authentication on MQTT PUBACK message (client->server)
MA9	Authentication on MQTT DISCONNECT message (server->client)
MA10	Authentication on MQTT Will message PUBLISH message
MA11	Authentication on MQTT Retained message PUBLISH message
AA1	Authentication on AMQP SASL message (server->client)
AA2	Authentication on AMQP SASL message (client->server)
AA3	Authentication on AMQP OPEN message (server->client)
AA4	Authentication on AMQP OPEN message (client->server)
AA5	Authentication on AMQP ATTACH message (server->client)
AA6	Authentication on AMQP ATTACH message (client->server)
AA7	Authentication on AMQP FLOW message (server->client)
AA8	Authentication on AMQP FLOW message (client->server)
AA9	Authentication on AMQP TRANSFER message (server->client)
AA10	Authentication on AMQP DISPOSITION message (client->server)
AA11	Authentication on AMQP DETACH message (server->client)
AA12	Authentication on AMQP DETACH message (client->server)
AA13	Authentication on AMQP CLOSE message (server->client)
⋆ CA1	Authentication on CoAP CON_GET message (EMQ X) (server->client)
⋆ CA2	Authentication on CoAP CON_GET message (EMQ X) (client->server)
⋆CA3	Authentication on CoAP CON_PUT message (EMQ X) (server->client)
⋆ CA4	Authentication on CoAP CON_PUT message (EMQ X) (client->server)
⋆ CA5	Authentication on CoAP CON_POSTAUTH message (Aliyun Cloud) (server->client)
⋆ CA6	Authentication on CoAP CON_POSTAUTH message (Aliyun Cloud)) (client->server)
⋆ CA7	Authentication on CoAP CON_POSTPUBLISH message (Aliyun Cloud) (server->client)
⋆ CA8	Authentication on CoAP CON_POSTPUBLISH message (Aliyun Cloud) (client->server)

[1] The property with ⋆ is extended property.
[2] Authentication properties on both client side and server sides are considered. CA1-CA4 are only supported in EMQ X and CA6-CA7 are only supported by Aliyun Cloud in CoAP protocols.
[3] A->B means that A authenticates the message from B.

State machine and property generation. First, `MPInspector` applies message semantics extraction from Section 4.3 to identify the parameter semantics for the key messages specified in the MQTT standard. In particular, `MPInspector` outputs the semantics of nine key MQTT messages using the JSON encoding, e.g., `{"CONNECT":{"ClientID":"","username":{"composition":["authid","tenantid"]},"password":""}}` (an expression `"parameter":""` means that `parameter` does not have extra semantics and is consistent with the standard MP).

Second, `MPInspector` applies interaction logic extraction from Section 4.3 to the MQTT implementation on the Bosch IoT platform. It outputs a raw state machine whose transition messages only contains the message names, e.g., `CONNECT/CONNACK`. Then, it adds the semantics extracted from Section 4.3 to each transition message. After that, we have the inferred state machine as shown in Figure 9. According to the property generation method in Section 4.5, `MPInspector` outputs the secrecy and authentication properties as shown in Appendix A.

State machine translation. First, `MPInspector` generates

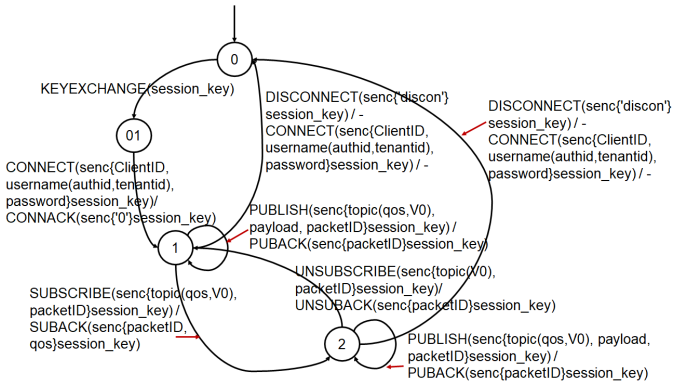


Figure 9: The inferred state machine of the MQTT implementation on the Bosch IoT platform.

the special initial rule and session key negotiation rule. The initial rule defines the initial states of the broker and clients, which is concluded from the MQTT specification. `MPInspector` uses the let-binding expression to specify the parameter semantics in the initial states, as shown in the second line of Listing 2. `MPInspector` generates the transition rule for session key negotiation based on the state machine, which is a simplified SSL/TLS key negotiation modeling. The rule is shown in Listing 3.

```
rule init_client:
    let username = <~authid,~tenantid> in
    [ !SERVER($SERVER), Fr(~ClientID), Fr(~
    authid),Fr(~tenantid), Fr(~password)]--[
    ]->[!DEVICE($SERVER,~ClientID,username,~
    password),!State_0_Serv($SERVER,~ClientID,
    username,~password),!State_0_Dev($SERVER,~
    ClientID,username,~password)]
```

Listing 2: An example of an initial rule in Tamarin code.

```
rule client_serv_negotiate_tls_key:
    let username = <~authid,~tenantid> in
    [!State_0_Serv($SERVER,~ClientID,username,~
    password),!State_0_Client($SERVER,~ClientID
    ,username,~password),Fr(~session_key)]
  --[  ]->[ Dev_Tls_Sym($SERVER,~ClientID,
    username,~password,~session_key),
    Serv_Tls_Sym($SERVER,~ClientID, username, ~
    password,~session_key)]
```

Listing 3: An example of a session key negotiation rule in Tamarin code.

Second, we translate the transition messages from the inferred state machine to rules following the principle described in Section 4.6. Taking the server side transition `CONNECT(senc{ClientID,username(V1,V2),password}session_key/CONNACK(senc('0')session_key))` as an example, we show its translated Tamarin rule in Listing 4. As shown in Listing 4, the rule's left part shows the state that the server receives the `CONNECT` message and its right part indicates the state that the server sends out the `CONNACK` message. The action facts in the rule's middle part indicate the behaviors in the transition, which will be used in the property lemmas for reasoning. For example, `Secret(<'server','password',password>)` means that the `password` is supposed to be secret on the server side.

```
rule serv_recv_connect_snd_conncak:
  let username = <authid,tenantid>
    uername =  <authid,tenantid>
    connack = senc('0',session_key)
    connect = senc{ClientID,username,password}
    session_key
  in [ In(connect), Serv_Tls_Sym($SERVER,
    ClientID,authid, tenantid,password,
    session_key) ] --[ Create('connect','server
    ',$SERVER), Commit($SERVER,username,<'
    server','client',username>), Commit($SERVER
    ,username,<'server','client',ClientID>),
    Commit($SERVER,username,<'server','client',
    password>), Running($SERVER,username,<'
    client','server',<'connack',connack>>),
    Honest(<'client',username>),Honest(<'server
    ',$SERVER>), Secret(<'server','username',
    username>), Secret(<'server','password',
    password>), Secret(<'server','ClientID',
    ClientID>)] ->[ Out(connack), State_1_Serv(
    $SERVER,ClientID,authid,tenantid,password,
    session_key]
```

Listing 4: An example of a transition rule in Tamarin code.

Property translation. Finally, the formal code translation module automatically translates the secrecy properties on password to Tamarin code using the formula shown in Listing 5. `MPInspector` automatically generates four types of authentication lemmas for each authentication property based on the state machine. Taking the injective agreement as an example, `MPInspector` generates the formalization of the injective agreement property on a `CONNECT` message, as shown in Listing 6. Listing 5 and Listing 6 show the property lemmas use the first-order logic formulas over time points and acton facts, based on the standard security property templates specified by Tamarin Prover [17].

```
lemma secret_Password_serv:
"All n #i. Secret(<'server','password',n>) @i
    ==> (not (Ex #j. K(n)@j)) | (Ex A B #j.
    Reveal(A,B)@j & Honest(A)@i)"
```

Listing 5: An example of a secrecy lemma in Tamarin code.

```
lemma injective_agreement_dev_serv_CONNECT:
  "All a b t #i. Commit(a,b,<'server','client',t
    >) @i ==> (Ex #j. Running(b,a,<'server','
    client',t>) @j & j < i & not (Ex a2 b2 #i2.
     Commit(a2,b2,<'server','client',t>) @i2 &
    not (#i2 = #i))) | (Ex C data #r. Reveal(C,
    data)@r & Honest(C) @i)"
```

Listing 6: An example of an authentication lemma in Tamarin code.