Finding Behavior Bugs in IoT Messaging Protocols with Automated Oracle Generation and Monitoring

Qingpeng Du[†], Zhengxiong Luo[✉‡], Yuanliang Chen[†], Fuchen Ma[†], Yu Jiang[✉†]
[†] KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
[‡] National University of Singapore, Singapore, Singapore

Abstract

IoT messaging protocols, which govern critical device communications and resource management, have emerged as a significant attack surface in IoT systems. Behavior bugs—specification violations that manifest without observable crashes—pose severe security risks, potentially enabling unauthorized data access and device compromise. Detecting such bugs requires a comprehensive understanding of protocol specifications for oracle generation and communication semantics for accurate oracle monitoring, a challenge beyond existing approaches.

This paper introduces ARES, a fully automated framework that synthesizes executable behavior oracles from protocol specifications for real-time compliance monitoring. ARES first transforms specifications into structured oracles using LLM-driven behavior filtering and condition supplementation. Then, during real-time monitoring, ARES employs scenario-aware analysis to accurately determine the most appropriate oracle among potentially overlapping candidates. We evaluate ARES on six widely used IoT protocol implementations and identify 25 new bugs, of which 18 have been confirmed or fixed, and 10 CVEs were assigned due to their severity.

Keywords

IoT Messaging Protocol, Protocol Testing

ACM Reference Format:
Qingpeng Du[†], Zhengxiong Luo[✉‡], Yuanliang Chen[†], Fuchen Ma[†], Yu Jiang[✉†]. 2026. Finding Behavior Bugs in IoT Messaging Protocols with Automated Oracle Generation and Monitoring. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3803921

1 Introduction

Internet of Things (IoT) has revolutionized modern technology by connecting billions of devices across industrial automation, healthcare, and smart cities [1, 26]. At the core of these systems, messaging protocols govern critical device communications and resource management, making them a prime attack surface [22, 28]. Due to their complex multi-party interactions and stateful resource dependencies, these protocol implementations are susceptible to behavior bugs—specification violations that maintain apparent functionality without triggering observable crashes or memory errors, yet can lead to unauthorized access to sensitive data or malicious takeover of device control. Detecting these elusive compliance violations before deployment is thus critical for ensuring IoT system integrity.

Nevertheless, this task poses challenges. Unlike conventional testing approaches that rely on generic oracles like program crashes or memory violations, behavior bugs typically do not manifest through such observable anomalies and are highly coupled with specific communication scenarios and contextual conditions. They emerge only under precise combinations of message sequences and field configurations, and even when triggered, determining what the correct behavior should be is difficult (unlike observable anomalies like crashes where violations are self-evident), making traditional testing methods inadequate due to a lack of oracles necessary to validate protocol-specific behavioral requirements. To detect such behavior bugs, current approaches either involve manual construction, which requires substantial human effort and domain expertise to understand the specifications and translate them into executable test oracles, or leverage differential analysis methods, which depend critically on the availability of high-quality alternative implementations and provide no guarantee of conformance and comprehensiveness to the original specification.

This paper presents ARES, an automated framework for creating behavior oracles by analyzing protocol specifications and performing non-intrusive real-time compliance monitoring. We make use of protocol specifications, i.e., RFCs (Request for Comments), since they standardize the intended behavior of IoT messaging protocols and can provide reliable and comprehensive behavioral requirements. To achieve this, we need to address two main challenges: (1) RFCs are written in natural language and not directly machine-readable. Moreover, RFCs typically describe high-level behavioral intentions rather than directly providing concrete actionable test oracles, requiring deep contextual understanding across multiple sections and sophisticated reasoning for such transformation. (2) IoT messaging protocols exhibit inherent behavioral complexity with many special cases tailored for specific scenarios to ensure robustness. This leads to situations where multiple oracles may appear to apply simultaneously to the same network event, yet only one represents the scenario's true nature. This requires accurately distinguishing the essential characteristics of each scenario to determine which behavioral requirements truly apply.

For the first challenge, we utilize LLMs to automate the processing of informal natural language specifications. ARES first extracts the behavioral requirements from the specification. To bridge the gap between high-level natural language descriptions and actionable concrete test oracles, ARES implements condition supplementation through cross-referential analysis, where the LLM is equipped with special tools that enable navigating to oracle-related specification content for resolving implicit prerequisites. Meanwhile, to ensure consistency and executability while combating the untrustworthiness of LLMs, ARES leverages structural information to guide the generation of structured test oracles.

For the second challenge, ARES employs a scenario-aware analysis that annotates oracles with semantic metadata and performs scenario identification to determine which behavioral conditions truly apply. When multiple oracles trigger simultaneously, ARES uses scope-based, type-based, enforcement-level, and severity-based analysis that captures IoT protocol features to identify the appropriate oracle for each scenario accurately.

Zhengxiong Luo and Yu Jiang are the corresponding authors.

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803921

We implement ARES and evaluate it on six widely-used implementations of popular IoT messaging protocols. Our evaluation demonstrates that ARES effectively synthesizes behavior oracles from protocol specification and transforms them into executable test oracles that are directly applicable to real-time network traffic analysis in a non-intrusive manner. To further assess ARES's effectiveness in real-world testing, we integrate these oracles with established testing tools, including Peach [8] and AFLNet [20]. Our experiments show that these tools detect only 4 bugs through observable anomalies like crashes, while ARES can help identify 21 additional subtle behavior violations that deviate from protocol specifications. As of the submission, 18 bugs have been confirmed or fixed by the developers, and 10 CVEs have been assigned, highlighting their importance. Our main contributions are as follows:

- We propose to detect behavior bugs in IoT messaging protocols by systematically analyzing protocol specifications.
- We harness LLMs with specialized tools to autonomously synthesize behavior oracles, and design a scenario-aware nonintrusive monitoring mechanism for bug detection.
- We implement and evaluate ARES on six widely-used IoT protocol implementations. Results show that ARES can efficiently synthesize behavior oracles and seamlessly enhance established testing tools to detect 25 new bugs, including 21 behavior bugs.

2 Background and motivation

2.1 IoT Messaging Protocol

IoT messaging protocols use broker-mediated architectures where multiple clients communicate through a central broker. These protocols exhibit communication patterns that are determined by message sequences and field configurations.

Figure 1 shows an MQTT [2] example with two scenarios under different `retain` control field settings. With `retain=1` (scenario I), the broker would store the message and deliver it to later subscribers (④) after the connection establishment (③). Instead, with `retain=0` (scenario II), the broker does not store the message, so subscribers receive only a `SUBACK` acknowledgment without any retained messages published. We refer to these patterns as *behaviors*—the actions and decisions implementations must exhibit when processing messages under specific conditions. IoT protocol behaviors involve (1) complex interactions between message sequencing, packet types, and control field configurations; (2) multi-role dynamics, where entities act as publishers, subscribers, and resource providers; (3) stateful resources, requiring understanding the historical context of resource creation, modification, and access patterns. Despite this complexity, the correct implementation of protocol-defined behaviors is critical for security and functionality. For instance, mishandling the `retain` flag in the above example can cause an ephemeral message (①) to be stored and delivered to unauthorized subscribers (④), resulting in data leakage. RFCs, standardized through rigorous expert review [6], define precise communication rules and compliance requirements. By extracting message types and control fields from traffic and validating them against RFCs, we can identify scenarios and assess compliance.

2.2 Motivating Example

We illustrate protocol behaviors and detection challenges using a MQTT `Will` bug newly discovered by ARES. A `Will` message is a preconfigured message the broker publishes if the client disappears. In this scenario, Client2 connects with a `Will` message set to activate after a 2-second delay if disconnected ①, disconnects, then

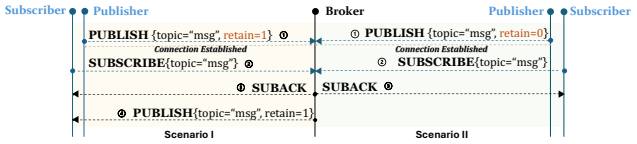


Figure 1: MQTT message flow example showing the impact of `retain` flag on broker behavior: `retain=1` enables message storage and delivery to new subscribers, while `retain=0` results in no message storage.

reconnects with `Clean=1` within 2 seconds ③. The broker acknowledges the new connection ④ and decides whether the `Will` message should be published—a behavior whose correctness is determined by the specification's requirements. In smart-lock deployments, door locks send `Will` messages to notify hosts of tampering or forced disconnections. If mishandled, an attacker could force a lock to disconnect and quickly reconnect with a clean session before the `Will` delay expires—the compromised lock appears to reconnect normally without sending the tampering detected `Will` message. This enables attackers to gain unauthorized property access while leaving hosts completely unaware of the security breach, as the expected intrusion alerts are silently suppressed.

Behavior Bug and Behavior Oracle. As shown above, behavior bugs are specification violations manifesting as incorrect server-side decisions while maintaining apparent functionality. These incorrect decisions typically compromise access control or data confidentiality. To detect such bugs, we formalize behavioral requirements in protocol specifications as behavior oracles. Each oracle consists of two components:

$$\text{Oracle} : \langle \text{Condition C, Assertion A} \rangle$$

where the condition specifies the prerequisite circumstances that must be satisfied, and the assertion defines the required behavioral outcome that implementations must exhibit when those conditions are met. For the MQTT `Will` message scenario, the specification statement s_1: *"The* `Will` *message MUST be published after the Network Connection is closed and either the* `Will Delay Interval` *has elapsed or the Session ends"* decomposes into:

$$\mathrm{C}_{s_1} : \underbrace{\mathit{Conn.}\ \text{Closed}}_{\text{connection closed}} \wedge (\underbrace{\mathit{Delay}_{\texttt{Will}}\ \text{Elapsed}}_{\texttt{Will delay expired}} \vee \underbrace{\mathit{Session}\ \text{End}}_{\text{session ends}})$$

$$\mathrm{A}_{s_1} : \underbrace{\mathit{Message}_{\texttt{Will}}\ \text{Published}}_{\texttt{Will message should be sent}}$$

Similarly, s_2 states that if a new Network Connection for the ClientID is opened before the `Will Delay Interval` has elapsed, the `Will` message MUST NOT be published, forming a second oracle:

$$\mathrm{C}_{s_2} : \underbrace{\mathit{Conn.}_{\text{same clientID}}\ \text{Established}}_{\text{new connection established}} \wedge \underbrace{\neg\, \mathit{Delay}_{\texttt{Will}}\ \text{Elapsed}}_{\texttt{Will delay not expired}}$$

$$\mathrm{A}_{s_2} : \underbrace{\neg \mathit{Message}_{\texttt{Will}}\ \text{Published}}_{\texttt{Will message should not be sent}}$$

Although behavioral requirements are specified, transforming them into real-time checkable oracles introduces two challenges:
C1. Context Searching for Oracle Synthesis: Individual statements often describe behavior in isolation, leaving critical prerequisite conditions implicit. The `Will` message publication requirement implicitly assumes that: (a) a subscriber (Client1) has established a connection and subscription for the relevant topic, (b) the `Will` message was configured during Client2's initial connection, and (c) connection termination has occurred. Only under all these prerequisites can the `Will` message publication be observed during testing.
C2. Accurate Oracle Application for Real-Time Monitoring: Real-time monitoring requires *scenario identification*—determining

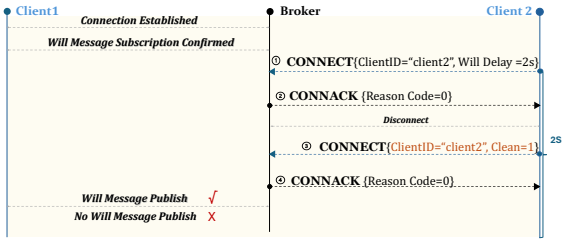


Figure 2: MQTT `Will` message scenario. Client2 reconnects with `Clean=1` before `Will` Delay expires, simultaneously satisfying two scenarios applicable to two oracles: Oracle$_1$ expects `Will` message publication while Oracle$_2$ prohibits it.

which behavior oracle applies to the current situation among multiple overlapping candidates. Unlike conventional generic oracles that universally apply (e.g., non-crashing), protocol-specific behavior oracles are contextually dependent,with different oracles governing different scenarios. Consider the example in Figure 2: Client2's reconnection with `Clean=1` before `Will Delay` expires. The key is identifying the *fundamental nature* of this event: while surface-level observation suggests a "reconnection before delay expires" scenario (triggering Oracle$_2$), `Clean=1` transforms this into a "session termination due to `Clean=1`" scenario (triggering Oracle$_1$). This distinction imposes different requirements —the former suggests `Will` message suppression, while the latter mandates `Will` message publication regardless of timing. Misidentifying the scenario leads to an incorrect oracle application.

3 System Design

Overview. Figure 3 presents the overall framework of ARES, which is composed of two main parts: behavior oracle generation and scenario-aware real-time detection. In the behavior oracle generation stage, ARES leverages LLMs to transform protocol specifications into formalized, real-time checkable oracles by equipping them with tools for grounded context access and formalizing outputs into executable schemas. It begins with Knowledge Base Construction (1), which extracts chapter-level information from protocol specifications. This information is then passed to Behavior Filtering (2), which identifies behavior-related normative statements. These statements are further processed by Condition Supplementation (3), which resolves implicit prerequisites via contextual reasoning and semantic analysis to generate complete triggering conditions. The output conditions, along with guidance from the schema and packet field information, are fed into Oracle Generating (4) to produce structured behavior oracles. During this process, each oracle is annotated with semantic metadata that captures its behavioral scope, type, enforcement level, and severity characteristics. These oracles are instantiated within the real-time monitoring stage via an oracle factory. The Network Traffic Monitor (5) continuously observes protocol messages and matches them against the active oracles. When one or more oracles are triggered, Scenario Identification (6) leverages the oracle metadata to resolve ambiguity, then chooses the appropriate oracle to report.

3.1 Behavior Oracle Generation

Protocol specifications often describe behavioral requirements without explicitly stating their triggering conditions. ARES first transforms protocol specifications into structured knowledge, then identifies normative statements that define observable behaviors. It further analyzes their contextual dependencies to recover implicit conditions and reconstruct complete behavioral logic, ultimately generating real-time checkable oracles.

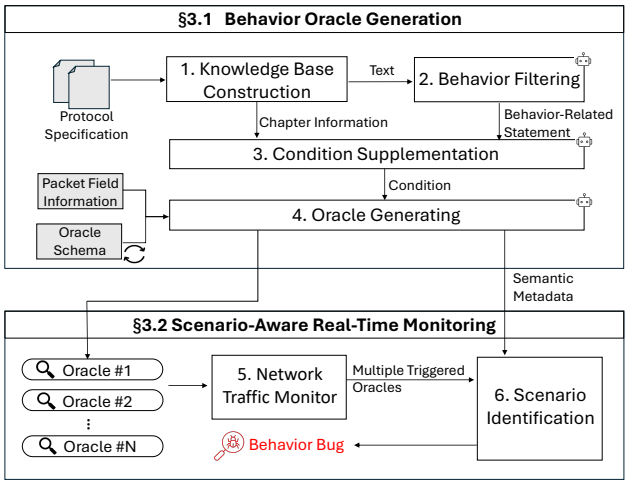


Figure 3: ARES Design Overview. It first synthesizes oracles from protocol specifications (steps 1-4) and then conducts real-time compliance checking (steps 5-6). Step 2-4 are powered by LLMs.

1) Knowledge Base Construction: To equip LLMs with contextual knowledge, we construct a structured knowledge base by segmenting the specification into sections and extracting normative statements for downstream filtering. For each specification section, we extract and store a structured s_i that contains the section's title t_i, content c_i, and hierarchical path p_i. We also record cross-reference relationships between sections, capturing how different parts of the specification reference each other to form a comprehensive semantic context. This structured representation aligns specification structure with protocol message formats and retains cross-references between chapters to support effective retrieval.

2) Behavior Filtering: This module filters extracted normative statements for behavior checking. For each section s_i, we select sentences with conditional and normative keywords, forming a candidate set $N = \bigcup_{i=1}^{n} N_i$. We then use an LLM-based classifier with a structured prompt to identify statements describing meaningful, testable broker behaviors. For each candidate n_i, we obtain a binary label $f(n_i) \in \{0, 1\}$, where 1 indicates generating a behavioral oracle. The filtered corpus $B = \{n_i \mid n_i \in N \wedge f(n_i) = 1\}$ ensures the generated oracles target meaningful, testable behaviors.

3) Condition Supplementation: This module resolves implicit conditions in behavioral statements via cross-referential analysis and outputs the complete prerequisite message sequence that triggers the behavioral assertion. To emulate expert reading of protocol documentations, the LLM uses on-demand retrieval tools for condition analysis, as illustrated in Figure 4.

Specifications distribute requirements across interdependent sections. A statement S in section σ_i references elements $E = \{e_1, \ldots, e_k\}$ whose definitions lie in sections $\sigma_{\text{def}}(e_j) \neq \sigma_i$. Temporal operators ("when/after/before") and state predicates require tracing to those sections to recover triggering conditions, prerequisite states, and precise definitions of X and Y. Our condition supplementation reconstructs the Minimal Oracle Trigger Sequence – the shortest sequence of messages to establish the conditions under which the assertion can be validated – through a three-step analysis process that emulates human expert reasoning patterns. (1) Initial Parsing. The LLM performs analysis to identify key elements such as packet types, field names and attributes, temporal conditions, and stated prerequisites. (2) Active Knowledge Retrieval. Based on identified knowledge gaps, the LLM uses tools to uncover dependencies

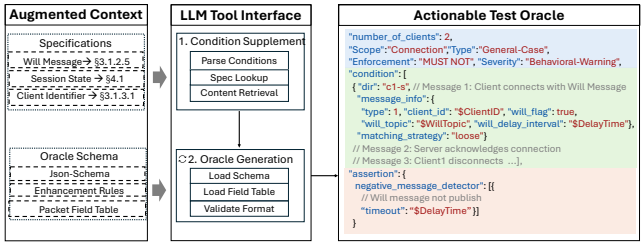


Figure 4: ARES analyzes contextual inputs using specialized tools to autonomously perform condition completion for oracle generation.

and cross-references. We provide three categories of retrieval tools: `find_chapters(keywords)` for navigation by sections based on keyword matches in titles, `search_content(terms)` for detailed content-based retrieval, and `read_section(path)` for specification access. (3) Comprehensive Analysis. The LLM composes the minimal oracle trigger sequence.

4) Structured Oracle Generation: This module transforms trigger sequences into JSON oracles for real-time checking, and enforces consistency via schema validation and protocol field standardization. For schema-driven validation, we define a JSON schema that specifies the required oracle structure, field types, field descriptions, and value constraints. We also maintain a protocol field mapping table to standardize field representations. The LLM reads the schema and field mapping and performs validation through dedicated tools in Figure 4. When validation fails, error details are fed back into the context to trigger LLM self-correction without requiring low-level syntax debugging. To support diverse testing scenarios, our oracles incorporate three critical capabilities: (1) directional message tracking for multi-client scenario capture, (2) variable content placeholders supporting dynamic fields that are resolved in real time, and (3) checking of both mandatory behaviors and prohibited behaviors through distinct assertion types. Taking s_2 from Section 2.2 as an example, the final generated oracle is shown in Figure 4.

3.2 Scenario-Aware Real-Time Monitoring

This module takes generated behavior oracles and produces bug detection by monitoring network traffic. The system instantiates and deploys oracles via a factory, which monitors network traffic and flags violations. When the observed traffic matches multiple oracles, semantic metadata is employed to disambiguate scenarios and select the applicable requirements.

1) Network Traffic Monitor: This module continuously observes protocol messages and matches them against active oracles deployed by the oracle factory. For network traffic analysis, we implement a Scapy-based packet processing pipeline [24]. The pipeline captures TCP segments, extracts structured message information, and performs raw byte retrieval, multi-packet parsing with length calculation, and malformed packet filtering. Monitoring is managed by a centralized rule factory that supports real-time registration and deregistration of oracles, with automatic state reset mechanisms when waiting periods expire or oracle condition requirements are violated. When packets arrive, the monitor evaluates them against all instantiated oracles to identify violations. Multiple oracles may be triggered simultaneously, requiring systematic resolution to determine the most appropriate oracle for the observed scenario.

2) Scenario Identification: Scenario identification requires understanding each oracle's intended context. Each oracle is annotated with metadata along four dimensions: *Scope* categorizes the breadth of behavior affected; *Type* distinguishes between Special-Case oracles handling exceptional conditions and General-Case oracles verifying standard protocol behaviors; *Enforcement Level* captures the normative strength from the specification; *Error Severity* categorizes the potential impact of violations. When multiple oracles are triggered simultaneously, hierarchical prioritization applies: Scope-based prioritization ensures that broader-impact oracles take precedence. Type-based resolution prioritizes special-case oracles. Enforcement-level ordering ensures mandatory requirements supersede recommended or optional behaviors. Severity-based filtering prioritizes protocol-fatal conditions over behavioral warnings. As shown in Algorithm 1, the system applies this hierarchical prioritization through priority resolution and evaluation phases. Triggered oracles are sorted by rank, with the top one kept and the rest suppressed. Only the retained oracle undergoes condition matching against the captured network packets, followed by assertion validation, yielding a deterministic verification result.

Algorithm 1: Real-Time Scenario-Aware Oracle Selection

Input: $O = \{o_1, o_2, \ldots, o_n\}$ – the set of triggered oracles
Input: $M = \{m_1, m_2, \ldots, m_n\}$ – oracle metadata
Input: P – the packet to be evaluated
Output: R – oracle execution results

```
// Priority Resolution Step:
if |O| > 1 then
    sortedOracles ← sort(O, compareByPriority(M));
    selectedOracle ← sortedOracles[0];
    suppressOracles (O \ {selectedOracle});
// Evaluation Step:
R ← {evaluate(o, P) | o ∈ O, ¬isSuppressed(o)};
return R;
```

4 Evaluation

We implemented ARES in Python3, with the LLM backend GPT-4o-2024-11-20 and temperature setting 0.5. In this section, we evaluate ARES to answer the following research questions:

RQ1. Is ARES effective in detecting behavioral bugs in real-world projects when integrated into established testing tools? (§4.2)

RQ2. How effective and efficient is the oracle generation module in terms of cost-efficiency? (§4.3)

RQ3. How do different components of ARES contribute to overall bug detection effectiveness? (§4.4)

4.1 Experimental Setup

Subjects. We selected six open-source IoT messaging protocol implementations: libcoap, FreeCoAP, Californium, NanoMQ, FlashMQ, and async_mqtt. These implementations have been widely studied in prior IoT protocol testing research [14, 19, 32], and are extensively deployed in real-world systems, as reflected by their high popularity and substantial codebases detailed in Table 1.

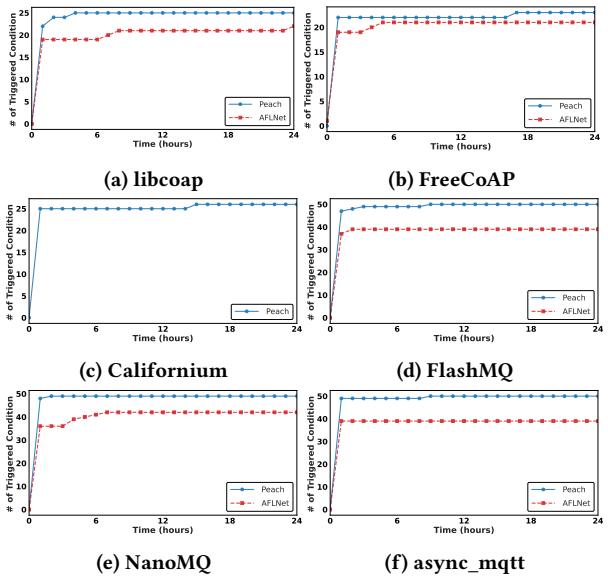
Base Testing Tools and Settings. We utilize two widely-adopted protocol fuzzers, Peach [8] and AFLNet [21], as our base testing tools. Our oracle generation and monitoring framework ARES is integrated into both fuzzers, enabling real-time behavioral validation and immediate detection of protocol specification violations. Each testing campaign runs for 24 hours with 4 CPUs and 8GB RAM.

Table 1: Detailed information of the selected subjects

Subject	libcoap	FreeCoAP	Californium	FlashMQ	NanoMQ	async_mqtt
Github Stars	854	138	747	217	1,924	136
Lines of Code	119k	29k	184k	46k	429k	78k

Table 2: Comparison of bug detection capabilities of different testing tools with or without oracle for the bugs in Table 3.

Tool	Detectable Bugs	Total
Peach	#1, #6-8	4 crash bugs
Peach^{ARES}	#1-25	4 crash bugs, 21 behavior bugs (**+525%**)
AFLNet	#1	1 crash bug
AFLNet^{ARES}	#1, #3-5, #10-18, #20, #24	1 crash bug, 14 behavior bugs (**+1400%**)



(a) libcoap (b) FreeCoAP

(c) Californium (d) FlashMQ

(e) NanoMQ (f) async_mqtt

Figure 5: Number of oracle conditions triggered by different testing tools on six IoT messaging protocol implementations over 24 hours.

4.2 Bug Detection Capability (RQ1)

To answer **RQ1**, we evaluate the effectiveness of ARES-generated oracles in detecting behavior bugs across real-world IoT protocol implementations. Our system extracted 207 oracles from the MQTT specification and 67 oracles from the CoAP specification, covering diverse behavioral requirements. The detailed breakdown of discovered vulnerabilities is presented in Table 3, encompassing 25 previously unknown bugs. For crash bugs, we instrument targets with ASan [25] and UBSan. Simultaneously, our protocol-specific oracles operate in real-time during fuzzing campaigns to detect incorrect protocol behaviors. Since the evaluated subjects have undergone extensive testing, our fuzzing campaigns discovered relatively few crash bugs compared to behavior bugs. However, we were able to uncover previously unknown behavior bugs that conventional testing approaches had not detected. 21 of these are behavior bugs, while 4 represent memory safety violations.

Figure 5 reports oracle triggers over 24h (Californium only with Peach due to Java). AFLNet^{ARES}, relying on seed input mutation, discovered 14 behavior bugs. Peach^{ARES}, which enables more comprehensive exploration of protocol states and cases, identified 21 behavior bugs. Key recurring patterns include: (1) option handling violations, including improper processing of reserved options (bugs #2, #9), critical options (bugs #5, #11, #16), and conditional options (bugs #3, #14); (2) content format and method processing, where implementations fail to reject unsupported content formats (bugs #4, #15) or method codes (bug #17); (3) protocol-specific properties, such as `retain` handling (bugs #18, #21), `Will` message processing (bug #19), and `QoS` handling (bug #25). The concentration around special protocol features reflects complex conditional rules that ARES's oracles effectively capture.

Case Study (Bug #18): MQTT Retain Handling Settings Violation. According to the specification, when a client subscribes with `Retain Handling=2`, the broker must not send retained messages at subscription time. However, the broker incorrectly delivers retained messages regardless of this setting, causing an information disclosure vulnerability where clients receive content they opted out of. An attacker can exploit this by publishing sensitive retained messages in advance. Even with `Retain Handling=2`, clients still receive them—causing information leaks or malicious payloads. This bug received a CVE ID due to its severity.

False Positive Analysis. Our testing campaigns generated 3 false positives across all subjects. Analysis reveals that the root cause is the vagueness in RFC specifications that permits multiple interpretations. For example, the MQTT specification states: *"If the Server does not want to accept the CONNECT, and wishes to reveal that it is an MQTT Server, it MAY [...], and then it MUST close the Network Connection."* The phrase "wishes to reveal" introduces subjectivity; the oracle generator interpreted this as mandating connection closure for every CONNECT rejection, thereby missing the optional nature of the revelation behavior. Considering the inherent ambiguity of natural language, this false positive rate is acceptable.

4.3 Evaluation of Oracle Generation (RQ2)

We analyze the efficiency of our LLM-based oracle generation process. Table 4 presents the performance breakdown of the oracle generation process. In summary, generating a single oracle requires an average of 25.28 seconds and consumes an average of 11,991 tokens, with an average cost of $0.038.

We further analyze the frequency of different actions used during the generation process. Figure 6 shows the distribution of different actions. `read_content` and `find_chapters` are the most frequently used operations, reflecting the intensive text processing required for synthesizing behavioral requirements from specifications. The relatively low frequency of `validate_json` invocations indicates that most generated oracles conform to the required format on the first attempt, achieving a 95.16% first-time success rate.

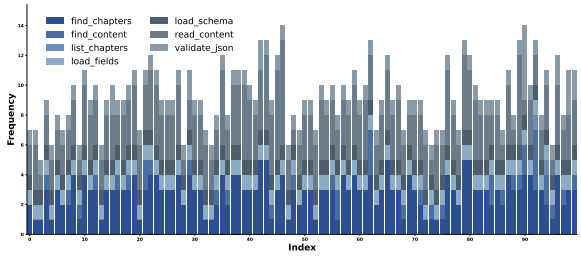


Figure 6: The frequency with which actions are invoked during oracle generation process for individual protocol specification statements.

4.4 Ablation Study (RQ3)

To answer **RQ3**, we conduct an ablation study to assess the contribution of each component by creating two variants: (1) ARES ^{m1}: disabling the condition supplementation component, and (2) ARES ^{m2}: disabling the scenario identification and oracle selection component. We run 24-hour Peach campaigns with these variants.

Table 5 presents the comparative results. Peach^{ARES} ^{m1} reported 11 bugs with 12 false positives. This precision degradation indicates that an incomplete understanding of protocol specifications impairs oracle effectiveness. By analyzing the false positives, we discover that behaviors requiring joint analysis across multiple specification sections cannot be adequately validated by ARES ^{m1}. Disabling the

Table 3: Previously unknown bugs detected when incorporating ARES-generated oracles into Peach and AFLNet.

#	Subject	Bug Type	Condition	Assertion	Threats	Identifier	Status
1	libcoap	Crash	Unspecified scenario	Non-crash	DoS	CVE-2025-51064	Fixed
2	libcoap	Behavior	Reserved option used with `Location-Path`	Broker MUST return `4.02 Bad Option`	DoS, ID	CVE-2025-51065	Confirmed
3	libcoap	Behavior	`If-Match` condition fails	Broker MUST return `4.12 Precondition Failed`	DoS, ID	Issue#1647	Fixed
4	libcoap	Behavior	Unsupported `Content-Format` received	Broker MUST return `4.06 Method Not Allowed`	ID	Issue#1648	Confirmed
5	libcoap	Behavior	Critical option with invalid length	Broker MUST return `4.02 Bad Option`	DoS, ID	Issue#1649	Confirmed
6	libcoap	Crash	Unspecified scenario	Non-crash	DoS, MC	Issue#1683	Reported
7	libcoap	Crash	Unspecified scenario	Non-crash	DoS	Issue#1684	Reported
8	libcoap	Crash	Unspecified scenario	Non-crash	DoS	Issue#1685	Reported
9	FreeCoAP	Behavior	Reserved option used with `Location-Path`	Broker MUST return `4.02 Bad Option`	ID	CVE-2025-51069	Reported
10	FreeCoAP	Behavior	Proxy request received but not supported	Broker MUST return `5.05 Proxying Not Supported`	ID	Issue#44	Reported
11	FreeCoAP	Behavior	Duplicated critical option received	Broker MUST return `4.02 Bad Option`	ID	Issue#45	Reported
12	FreeCoAP	Behavior	`GET` response sent	Response MUST include `Max-Age`	ID	Issue#46	Reported
13	Californium	Behavior	Proxy request received but not supported	Broker MUST return `5.05 Proxying Not Supported`	ID	CVE-2025-51074	Confirmed
14	Californium	Behavior	`If-Match` condition fails	Broker MUST return `4.12 Precondition Failed`	DoS, ID	CVE-2025-51077	Confirmed
15	Californium	Behavior	Unsupported `Content-Format` received	Broker MUST return `4.06 Method Not Allowed`	ID	Issue#2344	Confirmed
16	Californium	Behavior	Duplicated critical option received	Broker MUST return `4.02 Bad Option`	ID	CVE-2025-51081	Confirmed
17	Californium	Behavior	Unsupported method code received	Broker MUST return `4.05 Method Not Allowed`	ID	CVE-2025-51079	Fixed
18	FlashMQ	Behavior	`Retain Handling=2` in subscription	Broker MUST NOT send retained messages	ID	CVE-2025-44068	Fixed
19	FlashMQ	Behavior	`Clean Start=1` before `Will Delay`	Broker `Will` message MUST be published	ID	Issue#122	Fixed
20	NanoMQ	Behavior	`User Property` exceeds `Maximum Packet Size`	Broker MUST NOT send	ID	Issue#1923	Confirmed
21	NanoMQ	Behavior	Publish permission revoked	Broker MUST NOT deliver retained messages	ID	CVE-2025-44065	Confirmed
22	NanoMQ	Behavior	Duplicate `ClientID` received	Broker MUST send `Disconnect` with reason code `0x8E`	ID	Issue#1927	Confirmed
23	NanoMQ	Behavior	Shared subscription	Messages MUST follow sharing strategy	ID	Issue#1930	Confirmed
24	async_mqtt	Behavior	Topic contains null character	Broker MUST reject packet	ID	CVE-2025-44070	Fixed
25	async_mqtt	Behavior	Duplicate `QoS 2` message received	Broker MUST send `DISCONNECT`	ID	Issue#380	Confirmed

* DoS: Denial of Service. ID: Information Disclosure. MC: Memory Corruption.

Table 4: Average (*time/tokens*) breakdown per oracle generated

Stage	Behavior Filtering	Condition Supplementation	Oracle Generation	Total
Average	1.13s / 383	8.26s / 6,058	15.89s / 5,550	25.28s / 11,991

scenario identification and oracle selection component in ARES ^{m2} also reduces precision because multiple oracles can be simultaneously satisfiable, and without contextual disambiguation, the system may apply an inapplicable oracle that produces spurious reports under ambiguous sequences. The results demonstrate that both components provide essential and complementary contributions to ARES's effectiveness. The condition supplementation module ensures accurate oracle generation, while the scenario identification component improves the precision of scenario interpretation and ensures that the oracles correctly interpret the observed behavior.

5 Related Work

IoT Protocol Testing. Researchers have studied IoT protocol security [13, 30] through various testing methodologies. Mutation-based fuzzing approaches [16, 21, 23] enhance fuzzing with state-aware vulnerability discovery. Generation-based frameworks [8, 9, 11, 12, 15] take user-provided protocol models as input and generate packets conforming to protocol specifications. Specialized IoT testing techniques have emerged to address unique protocol characteristics. FUME [19] combines mutation and generation approaches for MQTT broker testing, while ShadowFuzzer [31] introduces shadow broker architectures for client-side vulnerability discovery. MPFuzz [14] enables collaborative packet generation for multi-role protocol fuzzing. However, existing IoT protocol fuzzers mainly focus on crashes, lacking semantic validation of behaviors.

Traditional test oracle generation faces scalability issues. Manually crafted oracles are labor-intensive [3], while automated approaches rely on predefined, fixed abstraction, e.g., sequence diagrams [34], grammar-constrained boolean assertions [17], or regular expression patterns [18]. However, effective behavioral validation for IoT protocols requires flexible, requirement-driven abstraction that adapts to diverse specification contexts and behavioral nuances. ARES automatically derives behavioral oracles leveraging LLMs' reasoning, enabling scalable protocol behavior modeling that captures diverse oracle-related properties, including message types, protocol-party behaviors, and fine-grained field values.

Table 5: Precision (true positives / total alarms) of Peach$^{\text{ARES}}$ variants with different module combinations

Method	Peach$^{\text{ARES}}$	Peach$^{\text{ARES}\ m1}$	Peach$^{\text{ARES}\ m2}$
Precision	21/24 (87.5%)	11/23 (47.8%)	21/30 (70.0%)

Formal Verification. Formal verification offers mathematically rigorous methods for ensuring protocol security through formal modeling of network protocols like TLS [4, 7] and LTE [10], as well as frameworks such as ProVerif [5]. Recent efforts apply formal verification to IoT protocol security. MPInspector [29] combines model learning with verification for black-box testing, while MQTTactic [33] employs static analysis and model checking for MQTT authorization vulnerabilities. However, these approaches target specific bug types and require manual configuration, whereas GPTScan [27] combines LLMs with program analysis for smart contract vulnerabilities. Unlike prior work that relies on predefined models or fixed bug categories, ARES enables scalable, specification-driven detection of protocol compliance violations in real time.

6 Conclusion

This paper presents ARES, a tool that leverages LLMs to automatically synthesize executable behavioral oracles for detecting behavior bugs in IoT messaging protocols. ARES employs LLMs to systematically analyze behavioral requirements in natural language specifications by providing special tools for condition supplementation and oracle format validation. Then, during the real-time monitoring stage, ARES implements real-time scenario identification through semantic metadata analysis to determine applicable requirements when multiple oracles trigger simultaneously. We implemented and evaluated ARES on six IoT protocol implementations. ARES demonstrates significant behavior bug detection capability when integrated into established testing tools, and it successfully detected 25 previously unknown bugs with 10 CVEs assigned.

Acknowledgment

This work was supported by the National Key Research and Development Program of China under Grant 2024YFF1401303.

References

[1] 2021. Industrial Internet of Things and its Applications in Industry 4.0: State of The Art. *Computer Communications* (2021). doi:10.1016/j.comcom.2020.11.016

[2] Andrew Banks, Ed Briggs, Richard Coppen, and Ken Borgendale. 2018. MQTT Version 5.0. OASIS Standard. http://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[4] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*. 483–502. doi:10.1109/SP.2017.26

[5] B. Blanchet. 2001. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001*. 82–96. doi:10.1109/CSFW.2001.930138

[6] S. Bradner. 1996. The Internet Standards Process – Revision 3. https://www.rfc-editor.org/rfc/rfc2026 RFC 2026.

[7] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 1773–1788. doi:10.1145/3133956.3134063

[8] Michael Eddington. 2024. GitLab Protocol Fuzzer Community Edition. *https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce* (2024).

[9] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. [n. d.]. Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. In *Security and Privacy in Communication Networks*, Bhavani Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran (Eds.).

[10] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. 2018. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018*. The Internet Society. doi:10.14722/ndss.2018.23313

[11] jtpereyda. 2024. boofuzz: Network Protocol Fuzzing for Humans. *https://github.com/jtpereyda/boofuzz* (2024).

[12] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. 2010. AspFuzz: A state-aware protocol fuzzer based on application-layer protocols. In *The IEEE symposium on Computers and Communications*. 202–208. doi:10.1109/ISCC.2010.5546704

[13] Zhengxiong Luo, Kai Liang, Zhao Yanyang, Feifan Wu, Junze Yu, Heyuan Shi, and Yu Jiang. 2024. DynPRE: Protocol Reverse Engineering via Dynamic Inference. *NDSS'24*.

[14] Zhengxiong Luo, Junze Yu, Qingpeng Du, Yanyang Zhao, Feifan Wu, Heyuan Shi, Wanli Chang, and Yu Jiang. 2024. Parallel Fuzzing of IoT Messaging Protocols Through Collaborative Packet Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 3431–3442. doi:10.1109/TCAD.2024.3444705

[15] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. BLEEM: packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4481–4498.

[16] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function Code Aware Fuzz Testing of ICS Protocol. *ACM Trans. Embed. Comput. Syst.* 18 (2019), 93:1–93:22.

[17] Davide Molinelli, Alberto Martin-Lopez, Elliott Zackrone, Beyza Eken, Michael D. Ernst, and Mauro Pezzè. 2025. Tratto: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA083 (June 2025), 23 pages.

[18] Manish Motwani and Yuriy Brun. 2019. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 188–199. doi:10.1109/ICSE.2019.00035

[19] Bryan Pearson, Yue Zhang, Cliff Zou, and Xinwen Fu. 2022. FUME: Fuzzing Message Queuing Telemetry Transport Brokers. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*.

[20] Thuan Pham. 2024. AFLNet: A Greybox Fuzzer for Network Protocols. *https://github.com/aflnet/aflnet* (2024).

[21] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[22] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. 2015. Security and privacy challenges in industrial internet of things. In *DAC*.

[23] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Reza Abbasi, and Thorsten Holz. 2022. Nyx-net: network fuzzing with incremental snapshots. *Seventeenth European Conference on Computer Systems* (2022).

[24] secdev. [n. d.]. Scapy: Packet crafting for Python2 and Python3. ([n. d.]). https://scapy.net.

[25] Kostya Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. *2012 USENIX Annual Technical Conference* (2012).

[26] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. 2018. Industrial internet of things: Challenges, opportunities, and directions. *IEEE transactions on industrial informatics* (2018).

[27] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. [n. d.]. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*.

[28] Koen Tange, Michele De Donno, Xenofon Fafoutis, and Nicola Dragoni. 2020. A systematic survey of industrial Internet of Things security: Requirements and fog computing opportunities. *Communications Surveys & Tutorials* (2020).

[29] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yu Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X. Liu, and Raheem A. Beyah. 2021. MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. In *USENIX Security Symposium*.

[30] Feifan Wu, Zhengxiong Luo, Yanyang Zhao, Qingpeng Du, Junze Yu, Ruikang Peng, Heyuan Shi, and Yu Jiang. 2024. Logos: Log Guided Fuzzing for Protocol Implementations. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*.

[31] Huikai Xu, Miao Yu, Yanhao Wang, Yue Liu, Qinsheng Hou, Zhenbang Ma, Haixin Duan, Jianwei Zhuge, and Baojun Liu. 2022. Trampoline Over the Air: Breaking in IoT Devices Through MQTT Brokers. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*.

[32] Junze Yu, Zhengxiong Luo, Fangshangyuan Xia, Yanyang Zhao, Heyuan Shi, and Yu Jiang. 2024. SPFuzz: Stateful Path based Parallel Fuzzing for Protocols in Autonomous Vehicles. *ACM/IEEE Design Automation Conference (DAC)* (2024).

[33] Bin Yuan, Zhanxiang Song, Yan Jia, Zhenyu Lu, Deqing Zou, Hai Jin, and Luyi Xing. 2024. MQTTactic: Security Analysis and Verification for Logic Flaws in MQTT Implementations . In *2024 IEEE Symposium on Security and Privacy (SP)*.

[34] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 307–318. doi:10.1109/ASE.2009.94