Industry Practice of LLM-Assisted Protocol Fuzzing for Commercial Communication Modules

Qiang Fu[1,2†], Changjian Liu[1†], Yuan Ding[2], Chao Fan[2], Yulai Fu[1],
Yuhan Chen[1], Ying Fu[5*], Ronghua Shi[1], Fuchen Ma[3,4], Heyuan Shi[1*]
[1]Central South University, Changsha, China
[2]China Mobile IoT Co.,Ltd, Chongqing, China
[3]Tsinghua University, Beijing, China
[4]Shuimu Yulin Technology Co.,Ltd, Beijing, China
[5]National University of Defense Technology, Changsha, China

***Abstract*—Fuzzing is widely used for software robustness testing. However, its application in commercial communication modules remains limited due to several key challenges, including labor-intensive template generation, lack of coverage collection support, limited testing performance, and inconsistencies between practical hardware and software CI/CD processes. In collaboration with China Mobile IoT, we present FuzzCM, a comprehensive protocol fuzzing framework tailored for commercial communication modules. FuzzCM employs a Retrieval-Augmented Generation (RAG)-enhanced large language model (LLM) to automate template generation and utilizes GPIO-based instrumentation for efficient runtime coverage data collection. Additionally, it leverages a knowledge base constructed from prior tests to guide hybrid mutation strategies and integrates CI/CD across both software and hardware layers, enabling continuous and environment-aware testing. We conducted an industrial practice with FuzzCM on five LTE Cat.1 bis modules, identifying 21 previously unknown bugs, 15 of which have been fixed. The results demonstrate that FuzzCM outperforms both manual methods and the Peach* approach, achieving average coverage improvements of 51% and 29%, respectively, with overall coverage reaching 85%.**

***Index Terms*—Protocol Fuzzing, Communication Module, Large Language Model**

I. Introduction

Communication modules are essential for reliable connectivity in embedded systems. These modules often use lightweight network protocols such as MQTT, HTTP, and NTP to support constrained and intermittent communication scenarios. However, limited computational resources and insufficient runtime introspection make them particularly susceptible to protocol-level faults, which may result in silent failures, degraded performance, or system-level vulnerabilities. [1]–[4]

Fuzzing [5]–[8] has been demonstrated to be effective for software testing across various domains. In communication systems, protocol fuzzing is critical for uncovering vulnerabilities by targeting the structure and semantics of communication protocols. Unlike general-purpose fuzzing, which typically operates on binaries or file inputs, protocol fuzzing generates malformed or unexpected messages to explore edge cases in stateful communication flows. This technique is particularly effective at identifying flaws in network interactions, message parsing, and protocol logic, making it essential for validating the robustness of communication modules.

In recent years, protocol fuzzing has been widely adopted in domains such as web servers, IoT protocols, wireless stacks, and industrial control systems. Tools such as AFLNet [9] and Peach [10] have demonstrated effectiveness in uncovering protocol implementation bugs by systematically mutating message formats and testing complex state transitions. However, directly applying these tools to communication modules, especially in embedded or resource-constrained environments, remains highly challenging. The main obstacles include the large number and complexity of AT commands, the tightly coupled software and hardware execution environments, and the difficulty for existing fuzzers to provide meaningful feedback or adapt to the constantly changing protocol states within communication modules.

To address these limitations, we systematically analyzed the workflow of protocol fuzzing for communication modules and identified four key challenges:

1) **Labor-intensive and expertise-dependent template generation.** Test engineers typically spend over 0.5 person-days creating input templates from product manuals for each function, and a basic module may include more than 20 functions. Limited staffing and varying levels of expertise often result in inconsistent Pit file quality, leading to unreliable fuzzing outcomes. This inefficiency renders traditional testing approaches inadequate for rapid project iteration.
2) **Lack of effective coverage feedback in existing methods.** Current protocol fuzzing approaches typically rely on fixed test durations and define success solely by the absence of crashes or reboots. This practice overlooks code coverage and increases the risk of missing deeper or more complex bugs, especially when poor-quality seeds confine the fuzzer to shallow execution paths. Without visibility into which parts of the codebase have been exercised, testers cannot assess the thoroughness of their efforts or identify gaps in coverage. These limitations underscore the urgent need for effective feedback mechanisms.
3) **Limited testing performance.** While general-purpose pro-

†Both authors contributed equally to this research.
*Corresponding authors: Heyuan Shi (hey.shi@foxmail.com), Ying Fu (fuying1995@foxmail.com)

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00260

tocol fuzzing tools exist, their effectiveness on communication modules is limited due to system-specific complexities, resulting in insufficient coverage and undetected vulnerabilities. Although engineers often identify key problem areas as essential test cases, these insights are not incorporated into the fuzzing workflow, reducing the utility of historical data and limiting the discovery of new issues. Effectively integrating such knowledge remains a significant challenge.

4) **Inconsistencies between hardware and software CI/CD processes.** Current CI/CD pipelines depend heavily on manual operations, resulting in increased labor costs and a higher risk of human error. For example, stress test configurations are typically assembled manually based on specific module versions and test scripts, and even partially automated setups still require engineers to perform manual module switching. Seamlessly integrating fuzzing for communication modules into CI/CD pipelines, while also enabling robust and automated coordination between software and hardware environments, remains a significant challenge in practice.

To address the aforementioned challenges, we present FuzzCM, an LLM-assisted fuzzing framework specifically designed for commercial communication modules. FuzzCM leverages a RAG-enhanced LLM to automatically extract protocol semantics from product manuals and historical test logs, enabling the automated generation of multi-command test templates and corresponding fuzzer scripts. This significantly reduces manual effort. To support feedback-driven fuzzing, we implement a lightweight instrumentation mechanism based on GPIO signals to capture real-time coverage information from embedded devices. Additionally, FuzzCM constructs a knowledge base from historical test data to guide the application of diverse input mutation strategies—including fault injection, random perturbation, and server-side response mutation—thereby enhancing both test diversity and depth. FuzzCM is integrated into an automated CI/CD framework, facilitating continuous, environment-aware testing. This integration allows fuzzing tasks to be triggered seamlessly during software updates, adapt dynamically to varying hardware configurations, and coordinate effectively with external test instruments. In collaboration with China Mobile IoT, we validated in industrial practice the effectiveness of FuzzCM, which combines LLM-driven automation with protocol-aware fuzzing to achieve robust and scalable validation of commercial communication modules. The main contributions of this work are as follows:

- Drawing on industry practice, we identify four challenges in protocol fuzzing for communication modules: labor-intensive template generation, unsupported coverage collection, limited testing performance, and inconsistencies between practical hardware and software CI/CD processes.
- We propose FuzzCM, which leverages a RAG-enhanced LLM to automate template generation and employs a GPIO-based instrumentation module for efficient runtime coverage data collection. Additionally, FuzzCM utilizes a knowledge base constructed from prior test executions to guide hybrid mutation strategies and is integrated into a CI/CD framework to support continuous, environment-aware testing.
- We conducted an industrial practice with FuzzCM on five modules, identifying 21 previously unknown bugs, 15 of which have been fixed. The results show that FuzzCM outperforms both manual methods and the Peach* approach, achieving average coverage improvements of 51% and 29%, respectively, with overall coverage reaching 85%.

II. Background

A. Communication Module

A communication module is an embedded hardware component designed to enable wireless or wired data transmission using standardized network protocols. These modules are commonly employed in IoT systems, mobile terminals, and industrial control devices. Typical protocols supported by communication modules include HTTP, MQTT, NTP, FTP, DNS, CoAP, and GATT. Communication modules are generally controlled via AT commands issued by the host processor, which are translated into calls to the underlying protocol stack within the module.

1) AT Command: AT commands for communication modules are specifically formatted strings sent from the TE (Terminal Equipment) or DTE (Data Terminal Equipment) to the TA (Terminal Adaptor) or DCE (Data Circuit Terminal Equipment). The TE transmits AT commands via the TA to control MS (Mobile Station) functions and interact with network services. Users can employ AT commands to manage features such as voice calls, SMS, phonebook operations, data services, supplementary services, and fax functionality.

Typically, communication modules are operated using AT commands, which are human-readable control strings transmitted from the host processor to the module over serial interfaces such as UART or USB. Initially standardized by ITU-T and later extended by 3GPP and module vendors, AT commands offer a high-level control interface that maps user intent to low-level protocol stack operations. Upon receiving an AT command, the module's interpreter parses the command and translates it into specific actions on the communication stack. This process functions as a semantic bridge between application-layer control and protocol-layer execution.

TABLE I
AT Command Types and Interaction Formats.

Type	Request	Response
Test	AT<*CMD*>=?	<ATCmd, Status>
Read	AT<*CMD*>?	[ATCmd1, ATCmd2, ...]
Set	AT<*CMD*>=<p1>,..	<Status, ATCmd>
Execute	AT<*CMD*>	Status

2) AT Command Example: AT commands adhere to a standardized format and are generally classified into four types: Test, Read, Set, and Execute, each serving a distinct purpose. Table I and Listing 1 illustrate these command types and corresponding output patterns in practice. The

```
%-------------Test Command------------
AT+CPOL=?
% Response
+CPOL: (list of supported <index>s),(list of
    supported <format>s) OK
%-------------Read Command------------
AT+CPOL=?
% Response (if succeed)
+CPOL: <index1>,<format>,<oper1>[,<GSM_AcT1>,<
    GSM_Compact_AcT1>,<UTRAN_AcT1>,<E-
    UTRAN_AcT1>]
+CPOL: <index2>,<format>,<oper2>[,<GSM_AcT2>,<
    GSM_Compact_AcT2>,<UTRAN_AcT2>,<E-
    UTRAN_AcT2>]
... OK
% Response (if fail)
+CME ERROR: <err>
%-------------Set Command-------------
AT+CPOL=<index>[,<format>[,<oper>[,<GSM_AcT>,<
    GSM_Compact_AcT>,<UTRAN_AcT>,<E-UTRAN_AcT
    >]]]
% Response (if succeed)
OK
% Response (if fail)
+CME ERROR: <err>
%-----------Execute Command-----------
AT+COPN
% Response (if succeed)
+COPN: <numeric1>,<alpha1>
+COPN: <numeric2>,<alpha2>
... OK
% Response (if fail)
+CME ERROR: <err>
```

Listing 1. AT Command Usage Examples: Test, Read, Set, and Execute.

Test form (e.g., `AT+CMD=?`) allows the host to query the valid parameter ranges for a command, while the Read form (`AT+CMD?`) retrieves the current configuration. The Set form (`AT+CMD=<params>`) is used to modify settings or activate specific features, and the Execute form (`AT+CMD`) triggers a predefined action without requiring parameters. For example, the Set command `AT+COPS=0,2,"46001"` selects a mobile operator by numeric ID, whereas `AT+COPN` in Execute form retrieves operator names stored on the SIM. These commands return standard success responses, such as `OK`, or error codes like `+CME ERROR: <err>`, both of which are essential for validating input and parsing responses during fuzzing.

B. Protocol Fuzzing

Protocol fuzzing is a core technique in software testing and security analysis, especially for networked systems and embedded devices [11]–[25]. Its main goal is to uncover vulnerabilities by automatically generating malformed or unexpected protocol messages and injecting them into the target system. Unlike traditional fuzzing, protocol fuzzing focuses on the communication layer, which requires a thorough understanding of both protocol syntax and semantics. The complexity and stateful nature of modern protocols bring unique challenges in terms of input modeling, state tracking, and obtaining effective coverage feedback. To address these challenges, two main approaches have been developed in protocol fuzzing: mutation-based and generation-based techniques.

1) Mutation-Based: Mutation-based fuzzers utilize real-world traffic or recorded protocol traces as seed inputs, applying random or guided mutations to explore a wide range of input variations. For instance, AFLNet [9] constructs message sequences by maintaining a pool of messages derived from network traces, which can be inserted into or used to replace existing seeds. DYFuzzing [26] combines seed mutation with the Dolev-Yao (DY) attack model to systematically uncover logic flaws. Frankenstein [27] improves code coverage by rearranging known message sequences, demonstrating the effectiveness of recombining real-world protocol flows to expose corner cases and unexpected behaviors. While mutation-based fuzzing is adept at exercising commonly encountered protocol paths and identifying shallow bugs, its ability to reach deeply nested or rarely exercised states is fundamentally limited by the diversity and completeness of the initial seed set.

2) Generation-Based: Generation-based fuzzers synthesize inputs from scratch using protocol models or specifications, such as state machines, grammars, or message templates. Although this approach requires greater effort during the initial setup phase, it enables a systematic and comprehensive exploration of the protocol's state space and facilitates the generation of highly structured, valid, and semantically rich test cases. By leveraging explicit protocol knowledge, generation-based fuzzers can target rarely exercised paths, edge cases, and complex state transitions that are often inaccessible to mutation-based methods. These fuzzers adapt their input generation strategies based on protocol constraints, allowing precise manipulation of message fields, timing, and sequencing to maximize test coverage.

Industrial efforts [28]–[35] have developed techniques for generating anomalous message sequences by injecting or removing protocol messages within valid sequences derived from state-machine models. Such approaches facilitate the exploration of out-of-order interactions, invalid state transitions, and protocol violations frequently overlooked by traditional methods. Similarly, tools such as [36]–[38] employ explicit state models to monitor transitions and inject syntactically correct packets at semantically invalid states, exposing deep-seated implementation vulnerabilities. Overall, the systematic construction of targeted test scenarios makes generation-based fuzzers indispensable for validating the robustness, correctness, and security of complex communication protocols.

III. Fuzzing Procedures and Targets

A. Fuzzing Procedures

The overall fuzzing process of FuzzCM is shown in Figure 1, which includes three key stages: preparation, execution, and continuous fuzzing.

Preparation: Fuzzing template generation. FuzzCM employs LLMs in conjunction with RAG to automate the generation of communication protocol test templates. This process begins by extracting structured semantic information from

TABLE II
OVERVIEW OF THE SELECTED COMMUNICATION MODULES.

Product	Transmission	Protocols	Application Domains
ML307R-DC	Down: 10 Mbps Up: 5 Mbps	IPv4, IPv6, PING, NTP, DNS TCP, UDP, HTTP, HTTPS, MQTT, MQTTS, FTP	Smart Metering, Smart City, Industrial IoT, Consumer Electronics
ML307M-DL	Down: 10 Mbps Up: 5 Mbps	IPv4, IPv6, PING, NTP, DNS TCP, UDP, SSL, HTTP, HTTPS, MQTT, MQTTS	Smart Metering, Smart City, Industrial IoT
ML307A-DC	Down: 10 Mbps Up: 5 Mbps	IPv4, IPv6, PING, NTP, DNS TCP, UDP, SSL, HTTP, HTTPS, MQTT, MQTTS	Mobile Payment, Internet of Vehicles, Asset Tracking
Product 1	Down: 10 Mbps Up: 5 Mbps	PPP, TCP, UDP, NTP, NITZ, FTP, HTTP PING, HTTPS, FTPS, SSL, FILE, MQTT	Tracker, POS, IPC, Data Card, Smart Security, Industrial-Grade PDA
Product 2	Down: 10.3 Mbps Up: 5.1 Mbps	TCP, UDP, HTTP, HTTPS, MQTT, SSL TLS, IPV6	Meter, POS, Smoke Detector, IPC, Tracker, Security Surveillance, In-Vehicle Device

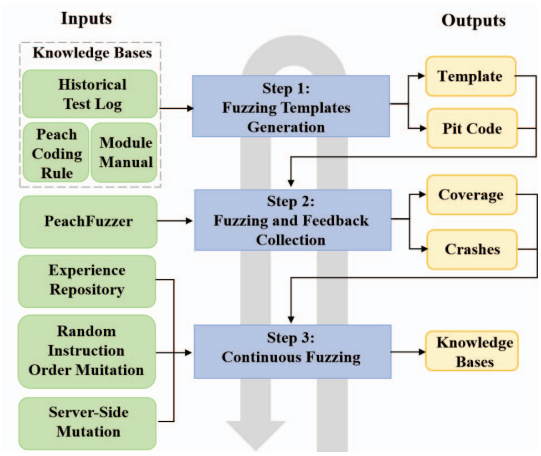

Fig. 1. Workflow of the FuzzCM Fuzzing Framework. (Step 1: Preparation; Step 2: Execution; Step 3: Continuous Fuzzing.)

product manuals, historical test logs, and protocol specifications to build a searchable knowledge base. During template generation, the RAG framework dynamically retrieves relevant content and provides it as contextual input to the LLM, ensuring the outputs are both protocol-compliant and semantically accurate. Based on this context, the LLM generates AT command–conforming test templates, including not only standard input formats for individual commands but also multi-command, ordered, and state-aware sequences that capture inter-command dependencies. The resulting outputs consist of Pit files for the Peach fuzzer and batch scripts to orchestrate command execution logic. All templates undergo lightweight manual validation to ensure correctness, enabling the generation of high-quality automated inputs for fuzzing.

Execution: Fuzzing and feedback collection. FuzzCM supports real-time coverage monitoring through source-level instrumentation that transmits execution signals via dedicated GPIO pins. The fuzzing workflow is automatically triggered by build events, which manage firmware compilation, flashing, and deployment to the target communication module. Upon deployment, protocol-level fuzzing is performed using Peach. It executes the Pit files or batch scripts generated during preparation. Throughout the fuzzing process, FuzzCM continuously monitors coverage data and system behavior, logs anomalies or failures, and integrates them into the issue tracking system.

Continuous Fuzzing: Analysis, coverage optimization, and CI/CD integration. Coverage metrics, runtime exceptions, and crash traces collected during fuzzing are aggregated into a centralized knowledge base that records historical failures, unexplored protocol branches, and edge-case behaviors, supporting in-depth analysis of test effectiveness. FuzzCM leverages this knowledge base to adapt and optimize subsequent fuzzing campaigns through three main mechanisms: (1) reusing historically erroneous AT command parameters to trigger untested states, (2) injecting randomized command sequences to explore unexpected transitions, and (3) synthesizing server-side responses to enable bidirectional mutation. This adaptive fuzzing process is fully integrated into the CI/CD pipeline. With each firmware update, FuzzCM automatically deploys the new version, conducts hardware-in-the-loop testing, and initiates fuzzing with environment-aware instrumentation. Real-time coverage and test results are continuously tracked to support regression testing and long-term quality assurance. By embedding fuzzing into the firmware development cycle, FuzzCM ensures automation, scalability, and continuous feedback of the testing process.

B. Fuzzing Targets

We selected LTE Cat.1 bis as the target communication standard and focused our evaluation on three representative network protocols: MQTT, HTTP, and NTP. The specific product models used in our experiments are detailed in Table II. This selection was guided by several key considerations.

LTE Cat.1 bis communication modules have seen significant market growth, with a 68% year-over-year increase in active connections in the first half of 2024, highlighting their importance in the expanding cellular IoT ecosystem. The three modules selected from China Mobile IoT [39], ML307R-DC, ML307M-DL, and ML307A-DC, represent the core of its LTE Cat.1 bis product line, supporting 10/5 Mbps bandwidth and similar protocol stacks (e.g., HTTP, MQTT, NTP). As shown in Table II, these modules serve various commercial applications, including smart metering, industrial IoT, and mobile payments, with over 50 million units shipped. Fuzzing

TABLE III
CHALLENGES AND SOLUTIONS IN COMMUNICATION MODULE FUZZING.

Idx	Challenge	Solution
1	Template generation is labor-intensive and experience-dependent	RAG-enhanced LLM for automated template and Pit file generation
2	Lack of coverage feedback in communication module fuzzing	Source-level instrumentation for real-time coverage collection
3	Limited testing performance	Hybrid fuzzing using a historical knowledge base
4	Inconsistencies between practical hardware and software CI/CD processes	CI/CD-integrated automated fuzzing framework

these high-volume modules ensures practical relevance and generalizability.

Additionally, we included two benchmark modules from other industry vendors to support comparative evaluation. These modules, with similar communication characteristics, help assess fuzzing robustness across different vendors, protocol stacks, and deployment scenarios. The selected protocols (MQTT, HTTP, and NTP) are widely used in industrial automation and telemetry, providing a solid foundation for evaluating module robustness in typical IoT deployments.

IV. TYPICAL CHALLENGES AND SOLUTIONS

During our industry practice, we discovered several unique challenges in protocol fuzzing for commercial communication modules. We proposed solutions for each challenge, as shown in Table III. The following describes each challenge in detail.

A. Preparation

Challenge 1: Template generation is labor-intensive and expertise-dependent. Fuzzing communication modules presents significant challenges due to their extensive AT command interfaces and complex parameter structures, resulting in an exponentially large input space that is difficult to cover manually. The 3GPP standard alone specifies over a thousand AT commands, which are often further extended by vendor- and application-specific variants. This diversity makes input construction extremely time-consuming. For a single module version with over 20 functional domains, test engineers often spend more than half a day reviewing manuals and developing scripts for each function. Such labor-intensive work renders manual fuzzing inefficient and unscalable, particularly during rapid iteration cycles.

Furthermore, the quality of Pit files [40] used in model-based fuzzing frameworks such as Peach varies significantly across engineers and projects. Their creation depends heavily on individual expertise and domain knowledge, often leading to inconsistencies within teams of varying experience levels. In the absence of standardized tools or automation, manually crafted Pit files may lack structural integrity or semantic completeness, resulting in limited input diversity, reduced protocol coverage, and inconsistent bug discovery. These challenges collectively highlight the need for automated and scalable methods to generate high-quality Pit files and fuzzing templates to enhance the effectiveness of protocol fuzzing.

Solution 1: RAG-enhanced LLM for automated template and Pit file generation. To address the above challenge, we employ a RAG-enhanced LLM to automatically generate fuzzing templates and scripts. Structured protocol documentation, such as MQTT specifications and user manuals, serves as context to ensure that the generated test cases are both syntactically correct and semantically diverse. The framework consists of two key components: Pit file generation and multi-command script synthesis. The LLM first produces Peach-compatible Pit files, leveraging retrieved documentation to guarantee both structural and semantic accuracy. It then generates multi-command scripts for complex, state-aware fuzzing based on dependencies identified in manuals and test logs.

As illustrated in Figure 2, the first stage generates Peach-compatible Pit files that define command structures, parameter types, and mutation strategies for both individual and grouped AT commands. In the second stage, the workflow generates multi-command test scripts that capture inter-command dependencies to support state-aware fuzzing. All Pit files and scripts are manually validated and corrected when necessary, and finalized versions are stored in a Git repository for automated execution within the CI/CD pipeline.

For single-command scenarios, the LLM generates standard fuzzing templates with initialized parameters and embedded mutation strategies, as shown in the yellow section of Figure 2. For more complex tasks, the model employs iterative prompt refinement to construct multi-stage command sequences based on valid protocol state transitions, as depicted in the blue section of the figure. These sequences are replayed during fuzzing to correctly initialize internal module states before injecting mutated inputs.

To implement this framework in practice, FuzzCM's RAG workflow catalogs AT manuals, test logs, and Peach documents into a vector database. The retrieved knowledge guides the LLM to generate structured tables, which are gradually converted into protocol-aware artifacts. Knowledge base maintenance is managed by the company's quality management system (QMS), and firmware release notes and AT command manuals are automatically archived and synchronized as part of daily project management, requiring no additional human intervention. Template generation is automated by the internal LLM service and undergoes light manual verification prior to CI/CD integration to address potential "phantom" risks. When customer-reported issues are incorporated, engineers contribute the buggy command sets, and the finalized Pit files and scripts are stored in the repository for automated execution in the CI/CD pipeline. On average, approximately 0.25 person-days are required to verify the fuzz testing templates and Pit files per release. Overall, the process remains largely

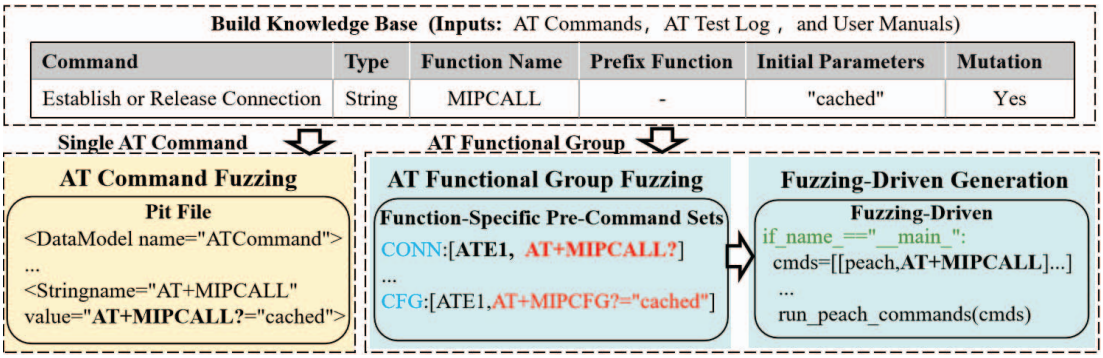

Fig. 2. Knowledge-guided Workflow for Fuzzing-Driven Generation.

automated, with minimal human involvement limited to review and approval, ensuring scalability without requiring significant additional staff investment.

Overall, this LLM-RAG-driven workflow streamlines both Pit file creation and test script generation, substantially reducing manual effort, improving output consistency, and enabling scalable and reproducible fuzzing across a broad range of AT-based communication protocols.

B. Execution

Challenge 2: Lack of coverage feedback in protocol fuzzing for communication modules. Protocol fuzzing for communication modules is particularly challenging due to the lack of effective coverage feedback, which is further constrained by hardware limitations and inherently limited runtime observability. Existing approaches, such as UART-based (Universal Asynchronous Receiver/Transmitter) AT command fuzzing and model-driven methods such as Peach, generally operate in a black-box fashion, evaluating fuzzing progress solely based on observable external behaviors. In practice, fuzzing is often performed for a fixed duration (e.g., 48 hours), with tests typically deemed successful if no crashes or reboots occur. However, without internal feedback mechanisms, it is impossible to determine the extent to which the input space has been explored. Consequently, when low-quality seeds are used, fuzzers may repeatedly exercise only shallow execution paths, failing to reach deeper protocol logic and potentially missing critical vulnerabilities.

Solution 2: Source-level instrumentation for real-time coverage collection. To address the limitations of black-box fuzzing, we developed a real-time coverage collection framework based on source-level instrumentation. Specifically, we instrument the protocol stack source code by inserting probe macros at key control flow locations, such as function entries and exits, conditionals, loops, and exception handlers. These macros are centrally defined and included at compile time to enable systematic coverage tracking.

Using the chip vendor's official development and build environment, the instrumented firmware is compiled and deployed to the communication module. During fuzzing, the module transmits real-time coverage data to the host PC via GPIO signaling, where the data are aggregated and analyzed. Compared to traditional manual inspection, this gray-box approach provides more accurate and dynamic insights into execution paths, significantly enhancing observability and guiding the fuzzing process. Based on this instrumentation, we construct detailed coverage profiles for communication protocols (e.g., MQTT), as shown in Table IV.

TABLE IV
MQTT Coverage Statistics Example. (%)

Function	Statement Cov.	Branch Cov.	Code Cov.
create	100.00	50.00	100.00
append_to_tail	66.67	66.67	76.19
append_to_head	50.00	50.00	54.76
insert_before_index	57.14	62.50	73.47
remove_content	25.00	25.00	19.05
Total	32.65	27.78	64.69

To further improve input quality and protocol compliance, we integrate LLMs for automated test case generation. Given that AT command interfaces and protocol semantics are typically specified in structured user manuals, we employ prompt engineering techniques to extract command logic, parameter constraints, and valid input formats. This knowledge is then used to synthesize protocol-aware test inputs that maximize code coverage and validate robustness. By combining real-time execution feedback with intelligent input generation, our framework provides an efficient and scalable fuzzing solution for protocols of commercial communication modules.

C. Continuous Fuzzing

Challenge 3: Limited testing performance. To evaluate the effectiveness of existing fuzzing tools, we selected three self-developed modules for protocol fuzzing and continuously collected detailed coverage statistics during the testing process. Original stress testing was used as the baseline for comparison with Peach* [41], a widely adopted and extensively validated gray-box fuzzing tool. As shown in Table V, Peach* increased statement and branch coverage for HTTP, MQTT, and NTP protocol functionalities by approximately 30% over the baseline. However, despite this significant improvement, the overall coverage achieved by Peach* remained in the range of 50% to

60%, which is still below the threshold typically required for comprehensive validation in practical engineering applications.

TABLE V
COMPARISON OF COVERAGE ACROSS DIFFERENT METHODS (%).

Coverage	Product	Stress Test	Peach*	FuzzCM
Statement	ML307A-DC	29.04	55.41	93.80
		40.59	56.59	85.04
		37.09	60.14	83.53
	ML307R-DC	27.37	49.24	87.71
		33.59	54.57	89.67
		38.49	66.67	86.42
	ML307M-DL	36.57	63.07	92.08
		37.15	59.26	92.37
		44.07	66.27	89.49
Branch	ML307A-DC	27.59	50.53	82.22
		39.85	53.79	82.21
		34.05	53.66	79.31
	ML307R-DC	34.46	42.62	77.97
		27.83	49.31	84.87
		35.59	60.77	82.80
	ML307M-DL	32.51	57.05	79.76
		31.63	57.11	85.89
		36.28	62.79	85.11

Solution 3: Hybrid fuzzing using a historical knowledge base. To further improve coverage and effectively address the persistent challenge of certain statements and branches being difficult to reach with standard AT test commands, we systematically explored three complementary techniques.

*1) **Erroneous initial parameters**:* Empirical results [42] show that using erroneous initial parameters from a historical knowledge base can significantly improve branch coverage in the tested state machine. Combined with the Peach fuzzer, this approach accelerates the exploration of previously unreachable paths, enhancing both test depth and efficiency.

To support this strategy, we constructed a knowledge base that aggregates error-related parameters collected from internal tests and customer-reported issues. These parameters, prioritized by their historical associations with specific AT commands, are used as seed inputs during fuzzing. Leveraging prompt engineering, the LLM utilizes this knowledge to generate highly targeted and semantically meaningful test cases. Compared to default state-switch parameters, tests guided by historically erroneous inputs achieve higher statement and branch coverage, especially in code regions untouched by conventional stress testing. While highly effective at exposing hidden or rare states, this method is less efficient for broader coverage expansion. Therefore, it is best applied as a supplement to standard fuzzing through experience-driven targeting.

*2) **Random initial states**:* Building on the above, we introduced a random mode for initializing the module state. All AT commands within a given functional domain are randomly combined, and their parameters are randomly mutated. This technique explores whether abnormal command interactions can transition the module into new or unexpected states, thereby increasing statement and branch coverage.

Since most state transitions in traditional testing are derived from normal usage logs, abnormal transitions triggered by random combinations are typically overlooked. To maximize coverage, we first applied several rounds of purely random command sequences to move the module into potentially abnormal states. Next, normal parameters were used to return the module to a known functional state, followed by conventional fuzzing. As shown in Table VI, this method significantly improves coverage compared to the baseline, which achieves only 35% statement coverage and 33% branch coverage.

*3) **Server-side mutation**:* Previous experiments showed that injecting erroneous parameters and introducing randomness yielded only marginal coverage gains (2% to 3%). Further analysis revealed that limited variability in the module's receive-side data, particularly due to fixed server responses, was a key bottleneck. Since such conditions could not be effectively explored through client-side mutations alone, we introduced a server-side mutation mechanism to improve coverage of receive-path logic.

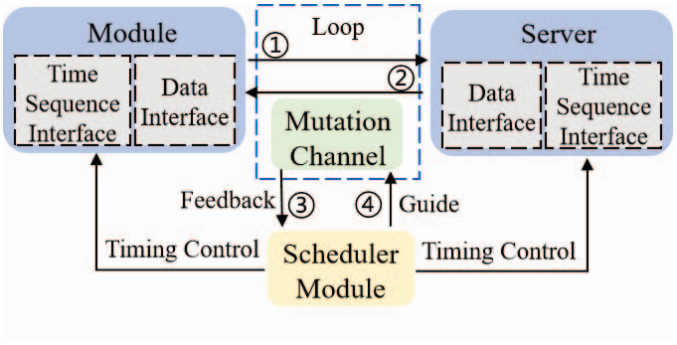

Fig. 3. Server-side mutation scheduling process.

To implement this approach, we designed a dual-end fuzzing framework (Figure 3) that supports bidirectional mutations guided by runtime feedback. The framework operates in three stages: (1) targeted mutation of both client and server data streams, (2) coverage-guided mutation selection using lightweight black-box instrumentation, and (3) event synchronization with a global clock to emulate real-world timing dependencies and expose temporal vulnerabilities.

To effectively explore the complex state models of IoT communication protocols and manage the contextual semantics involved in state transitions, the framework leverages both protocol participants (i.e., the module and the client). Their intrinsic understanding of protocol logic, combined with real-time contextual judgments during testing, is utilized to automatically populate semantic fields, thereby relieving the seed generation component from handling intricate semantics. Concurrently, mutation operations are applied to protocol packet sequences during transmission, and guided by feedback mechanisms, the framework enables automated and efficient exploration of the state model.

As shown in Table VI, server-side mutations provide the largest coverage gains by exposing previously unreachable code paths. The other two methods contribute smaller im-

provements but are closely aligned with client-side scenarios, making them valuable for customized projects. Accordingly, all three methods are retained. Notably, the MQTT protocol achieved full fuzzing coverage, and two previously unknown bugs—triggered exclusively by anomalous server responses—were identified. Overall, by combining these optimization techniques, we improved HTTP protocol statement and branch coverage in the ML307R-DC module by more than 35%, reaching approximately 80%, a level suitable for engineering validation.

TABLE VI
FUZZING CONFIGURATION AND COVERAGE RESULTS. (%)

Invalid Init Params	Random Init State	Server-Side Mutation	Statement Cov.	Branch Cov.
✓			52.75	45.08
	✓		51.23	43.39
		✓	86.67	76.68
✓	✓	✓	87.72	77.98

Challenge 4: Inconsistencies between practical hardware and software CI/CD processes. Effective testing of communication modules requires close coordination with hardware environments, including RF instruments and environmental chambers. In current practice, each module version necessitates a dedicated test setup tailored to its specific scripts, often involving manual configuration and physical switching between modules. This labor-intensive, low-automation process significantly impedes seamless integration of fuzzing into CI/CD pipelines.

Solution 4: CI/CD-integrated automated fuzzing framework. FuzzCM integrates fuzzing workflows with hardware reliability validation by coordinating communication modules, fuzzing templates, environmental chambers, RF test instruments, and real-time coverage monitoring.

As illustrated in Figure 4, the continuous fuzzing framework consists of three main components: a statistical dashboard, a fuzzing workflow, and a resource pool, collectively covering the entire process from input to execution and output. In the input phase, testing artifacts are prepared based on product manuals, source code, firmware, and coverage instrumentation. The execution phase encompasses AT template generation, firmware deployment, test environment simulation, and test case execution driven by the Peach Fuzzer. The output phase focuses on coverage and crash analysis, forming a feedback loop that leverages runtime data to support iterative improvement. The framework also utilizes key resources such as communication modules, code repositories, firmware images, and configuration templates to ensure efficient and sustainable fuzzing execution. All runtime logs, coverage reports, and fault data are unified and visualized through the statistical dashboard to enhance transparency and management efficiency.

This closed-loop architecture enables robust, automated, and scalable fuzzing aligned with practical industrial workflows, particularly for hardware-constrained communication modules. To support reliability-focused fuzzing, we further

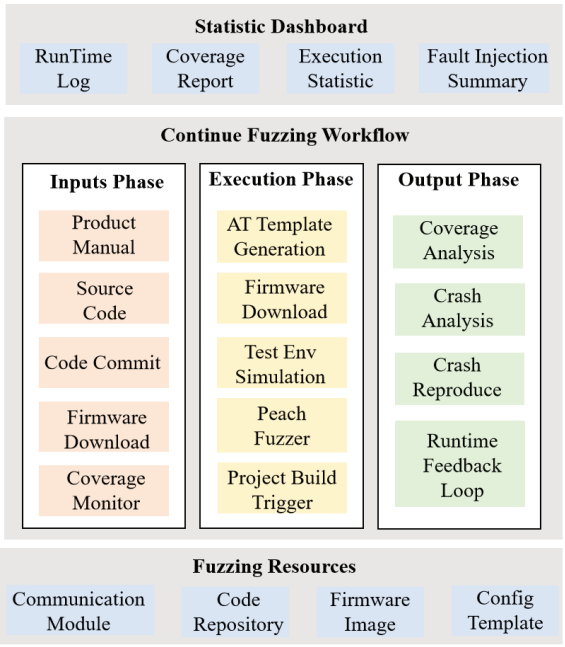

Fig. 4. FuzzCM's CI/CD implementation architecture diagram.

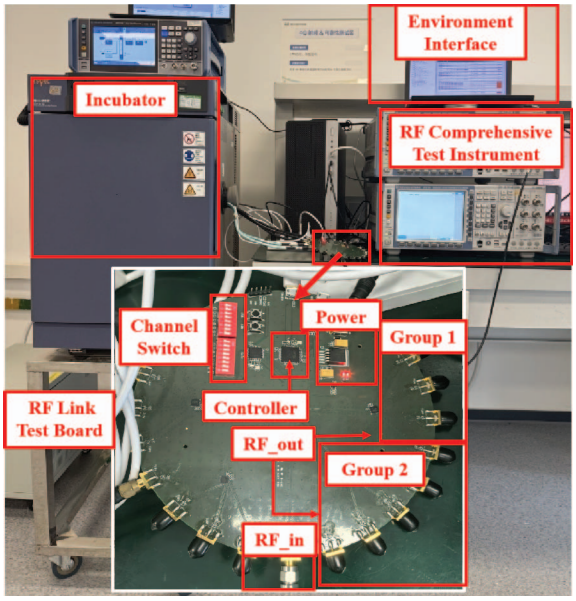

Fig. 5. Overview of the actual fuzzing platform.

developed a hardware-software co-testing environment (Figure 5) comprising four core components:

- RF Comprehensive Test Instrument: Continuously evaluates the radio-frequency performance of cellular communication modules by measuring transmission power, modulation quality, frequency deviation, ACLR (Adjacent Channel Leakage Ratio), and spurious emissions.
- RF Link Test Board: Dynamically switches among multiple module instances, enabling large-scale, 24/7 automated testing within the CI/CD framework (as shown at the bottom

of Figure 5).

- Incubator: Controls environmental stress factors such as temperature and humidity, facilitating the discovery of environment-sensitive defects that conventional software input fuzzing may overlook.
- Environment Interface: Retrieves firmware artifacts, initiates test scripts, manages instrument configurations, and collects both coverage and reliability data in real time.

By integrating LLM-generated fuzzing inputs, dynamic environmental stress testing, and continuous instrument coordination, the framework provides an automated, hardware-aware fuzzing pipeline for communication modules. This approach alleviates resource bottlenecks and improves the comprehensiveness of fuzzing from code commit to defect exposure.

TABLE VII
PREVIOUSLY UNKNOWN BUGS DETECTED BY FUZZCM.

Bug Location	Bug Type	Status
mqtt/tcp_processor.c process_tcp_message()	memory overflow	fixed
mqtt/broker.c handle_large_message()	memory overflow	fixed
mqtt/client.c process_heartbeat_packet()	protocol logic error	fixed
mqtt/queue.c manage_local_queue_length()	resource management	fixed
mqtt/message.c handle_retained_message()	data processing error	fixed
mqtt/qos.c validate_qos_level()	performance issue	fixed
mqtt/subscriber.c parse_subscription_wildcards()	performance issue	fixed
mqtt/will.c send_will_messages()	resource management	acked
mqtt/auth.c enforce_access_control()	security vulnerability	acked
mqtt/encoding.c handle_non_ascii_characters()	character encoding	report.
http/parser.c parse_malformed_response()	data processing error	fixed
http/buffer.c handle_buffer_overflow()	memory overflow	fixed
http/connection.c manage_slow_responses()	memory overflow	fixed
http/security.c prevent_https_downgrade_attack()	boundary condition	fixed
http/cookie.c parse_cookies()	data processing error	fixed
http/cache.c apply_cache_control_directives()	boundary condition	fixed
http/form.c handle_multipart_form_data()	boundary condition	fixed
http/cors.c enforce_cors_policy()	security policy error	fixed
http/redirect.c handle_redirect_loops()	resource management	acked
http/compression.c decompress_data()	protocol logic error	report.
http/html.c parse_html_response()	data processing error	report.

V. RESULTS

After addressing the challenges, we deployed FuzzCM on three commercial communication modules and integrated it into the China Mobile IoT community's continuous testing pipeline. We performed 48-hour fuzzing on each product under real-world conditions.

Bug. Table VII summarizes that FuzzCM identified 21 previously unknown bugs in the MQTT and HTTP protocol stacks. Of these, 15 have been fixed, 3 acknowledged for future patching, and 3 are under triage or pending further analysis.

These bugs span a range of categories, including memory overflows, data processing errors, boundary violations, logic flaws, and security vulnerabilities such as access control bypasses and downgrade attacks. Several identified issues, including memory corruption in `handle_large_message()` and boundary violations in `prevent_https_downgrade_attack()`, could result in severe consequences such as remote denial-of-service or system instability, particularly in resource-constrained embedded environments. Other bugs, such as improper encoding handling and inconsistent protocol parsing, may silently undermine reliability or disrupt downstream processing.

From a theoretical perspective, some shallow memory errors or basic parser flaws could be detected by general-purpose fuzzers such as Peach*. However, our evaluation shows that Peach* did not achieve sufficient execution depth to expose semantic or logic-level vulnerabilities. The integration of historical knowledge, protocol-aware mutation strategies, and hardware-assisted runtime observability in FuzzCM was essential for uncovering these deeper issues.

Listing 2 presents a representative vulnerability discovered by FuzzCM in `tcp_processor.c`. This issue stems from a missing boundary check between the declared content length in the TCP header and the actual payload size. Specifically, when `content_length > tcp_len`, the function `handle_tcp_payload()` attempts to access memory beyond the allocated buffer, potentially resulting in a heap or stack overflow. FuzzCM identified this issue during targeted fuzzing of the TCP handler using a protocol-aware template generated from product documentation. By crafting a packet that inflated the `content_length` field while providing a shorter actual payload, the fuzzer triggered an out-of-bounds read detected through hardware-assisted memory monitoring. This bug remained undetected by conventional fuzzers due to the specific structure and multi-field semantic dependencies required to reach the vulnerable code path.

Coverage. We conducted 48-hour fuzzing campaigns for each communication module, employing source-level instrumentation to monitor real-time coverage. As shown in Table V, FuzzCM achieved substantial improvements in both statement and branch coverage across the three commercial modules.

On average, FuzzCM achieved over 83% statement coverage and more than 77% branch coverage. In contrast, traditional stress testing reached only around 36% and 32%, respectively. Even when compared to Peach*, FuzzCM demonstrated significant gains. For instance, on the ML307A-DC module, FuzzCM achieved 93.8% statement coverage and 92.21% branch coverage, exceeding Peach* by more than 30 percentage points. These results confirm that FuzzCM not only enhances

TABLE VIII
FUZZING EFFICIENCY OPTIMIZATION COMPARISON. (PERSON-DAY)

Protocol	Requirements Analysis		Solution Design		Test Development		Test Verification		Total	
	Stress Test	FuzzCM	Stress Test	FuzzCM	Stress Test	FuzzCM	Stress Test	FuzzCM	Stress Test	FuzzCM
MQTT	0.31	0.05 6.2x	0.13	0.06 2.1x	0.19	0.03 6.3x	0.18	0.04 4.5x	0.81	0.18 4.5
HTTP	0.25	0.04 6.2x	0.06	0.02 3.0x	0.16	0.03 5.3x	0.09	0.04 2.2x	0.56	0.12 4.6
NTP	0.16	0.02 8.0x	0.06	0.02 3.0x	0.08	0.01 8.0x	0.02	0.01 2.0x	0.31	0.06 5.1

```
void process_tcp_message(SlidingWindow *window,
    uint8_t *tcp_data, int tcp_len) {
   int content_length =
       parse_tcp_header(tcp_data);
   ...
-  if (content_length > 0) {
+  if (content_length > 0 && content_length <=
    tcp_len) {
      ...
      handle_tcp_payload(tcp_payload,
          content_length);
   }
}
```

Listing 2. Code analysis example of a memory overflow bug.

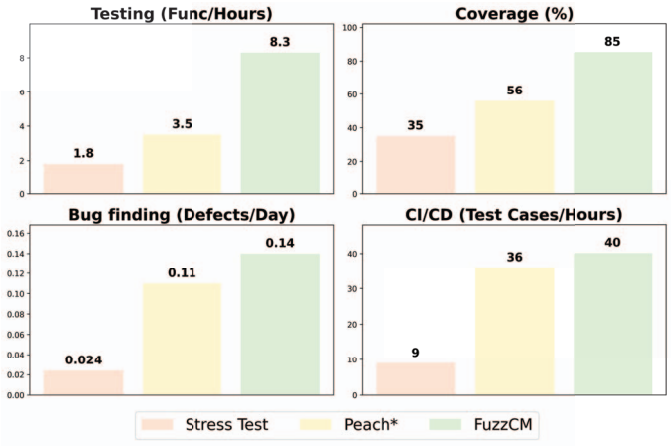

Fig. 6. Comparison of FuzzCM's functional indicators with existing tools.

automation and efficiency but also significantly deepens test coverage, offering a more thorough exploration of protocol logic essential for industrial validation.

Efficiency. Conventional robustness testing is highly dependent on manual script development. Engineers are typically required to extract AT command formats from extensive product documentation and rely on prior experience to define valid parameter ranges and edge cases. On average, this process consumes more than 0.5 person-days per function, introducing a clear bottleneck—especially since modern communication modules often encompass more than 20 functional domains within a single firmware version.

FuzzCM automates this workflow using a RAG-enhanced LLM to generate multi-command test templates and corresponding fuzzer scripts. As shown in Table VIII, the overall testing workload for three typical protocols—MQTT, HTTP, and NTP—was reduced by an order of magnitude. For example, the person-day cost dropped from 0.81 to 0.18 for MQTT (a 4.5× improvement), from 0.56 to 0.12 for HTTP (4.6×), and from 0.31 to 0.06 for NTP (5.1×). Notably, significant time savings were achieved during the early stages—requirements analysis and solution design—where manual interpretation of protocol semantics is most demanding. With LLM assistance, the cognitive load of parsing AT command structures and designing logical test flows was significantly reduced.

Comparison with Existing Tools. FuzzCM adopts a function-oriented fuzzing paradigm with LLM-driven automation. Compared to traditional testing platforms, it offers significant advantages in automation, coverage, and efficiency. As shown in Figure 6, FuzzCM consistently outperforms existing tools across all evaluation metrics, particularly in vulnerability detection and test throughput.

In summary, FuzzCM delivers enterprise-grade fuzzing capabilities by integrating LLM-based test generation, CI/CD workflows, real-time coverage instrumentation, and multi-environment orchestration. These features collectively enable continuous, scalable, and high-coverage protocol fuzzing, providing an effective solution for the rigorous demands of communication module testing.

VI. CONCLUSION

This paper presents FuzzCM, an LLM-assisted protocol fuzzing framework for validating communication protocols in commercial modules. By combining RAG, gray-box instrumentation, and multi-dimensional input mutation, FuzzCM addresses key industrial challenges such as input explosion and limited feedback. Experiments on LTE Cat.1 bis modules achieved over 80% statement coverage and uncovered 21 previously unknown bugs, demonstrating strong practical effectiveness. FuzzCM represents a significant step toward intelligent, automated fuzzing for cellular IoT systems. Future work will explore online learning, automated knowledge extraction from logs, and scalable CI integration.

VII. ACKNOWLEDGMENTS

This research is supported in part by National Key R&D Program of China (No.2022YFB3104003), the NSFC Program (Nos.62472448, 62502254, 62202500), Hunan Provincial Natural Science Foundation (No.2023JJ40772), Hunan Provincial 14th Five-Year Plan Educational Science Research Project (No.XJK23AJD022), Changsha Science and Technology Key Project (No.kh2401027), High Performance Computing Center of Central South University.

REFERENCES

[1] J. Yan and J. Wang, "Environmental testing for electric and electronic products—part 2: Test methods—test db: Damp heat, cyclic (12 h + 12 h cycle)," 2008.

[2] Z. Zhong, G. Liu, and H. Wang, "Environmental testing for electric and electronic products—part 2: Test methods—test a: Cold," 2008.

[3] Z. Zhang, "Environmental testing for electric and electronic products—part 2: Test methods—test b: Dry heat," 2008.

[4] X. Xu, K. Huang, K. Yan, W. Lü, G. Lü, Y. Ni, and Q. Fu, "Environmental testing—part 2: Test methods—test cab: Damp heat, steady state," 2016.

[5] Q. Shi, J. Shao, Y. Ye, M. Zheng, and X. Zhang, "Lifting network protocol implementation to precise format specification with security applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1287–1301.

[6] Y. Shen, Y. Xu, H. Sun, J. Liu, Z. Xu, A. Cui, H. Shi, and Y. Jiang, "Tardis: Coverage-Guided Embedded Operating System Fuzzing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4563–4574, 2022.

[7] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang, and W. Chang, "Rtkaller: State-Aware Task Generation for RTOS Fuzzing," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021. [Online]. Available: https://doi.org/10.1145/3477014

[8] Y. Shen, J. Liu, Y. Xu, H. Sun, M. Wang, N. Guan, H. Shi, and Y. Jiang, "Enhancing ros system fuzzing through callback tracing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 76–87. [Online]. Available: https://doi.org/10.1145/3650212.3652111

[9] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[10] GitLab. (2024) Protocol Fuzzer Community Edition. Accessed Nov. 26th, 2024. [Online]. Available: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce

[11] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[12] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[13] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[14] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "{APICraft}: Fuzz driver generation for closed-source {SDK} libraries," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2811–2828.

[15] C. Zhang, Y. Li, H. Zhou, X. Zhang, Y. Zheng, X. Zhan, X. Xie, X. Luo, X. Li, Y. Liu *et al.*, "Automata-guided control-flow-sensitive fuzz driver generation." in *USENIX Security Symposium*, 2023, pp. 2867–2884.

[16] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[17] M. Tufano, C. Watson, G. Bavota, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 25–36.

[18] F. He, W. Yang, B. Cui, and J. Cui, "Intelligent fuzzing algorithm for 5g nas protocol based on predefined rules," in *2022 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2022, pp. 1–7.

[19] X. Ma, L. Luo, and Q. Zeng, "From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter {IoT} devices," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4783–4800.

[20] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, vol. 2024, 2024.

[21] J. Wang, L. Yu, and X. Luo, "Llmif: Augmented large language model for fuzzing iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 881–896.

[22] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4481–4498.

[23] Z. Luo, J. Yu, Q. Du, Y. Zhao, F. Wu, H. Shi, W. Chang, and Y. Jiang, "Parallel fuzzing of iot messaging protocols through collaborative packet generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3431–3442, 2024.

[24] F. Wu, Z. Luo, Y. Zhao, Q. Du, J. Yu, R. Peng, H. Shi, and Y. Jiang, "Logos: Log guided fuzzing for protocol implementations," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1720–1732.

[25] J. Yu, Z. Luo, F. Xia, Y. Zhao, H. Shi, and Y. Jiang, "Spfuzz: Stateful path based parallel fuzzing for protocols in autonomous vehicles," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[26] P. C. Amusuo, R. A. C. Méndez, Z. Xu, A. Machiry, and J. C. Davis, "Systematically detecting packet validation vulnerabilities in embedded network stacks," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 926–938.

[27] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 19–36.

[28] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," *Communications of the ACM*, vol. 60, no. 2, pp. 99–107, 2017.

[29] G. S. Reen and C. Rossow, "Dpifuzz: a differential fuzzing framework to detect dpi elusion strategies for quic," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 332–344.

[30] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1492–1504.

[31] M. Vanhoef, "Wifuzz: detecting and exploiting logical flaws in the wi-fi cryptographic handshake," *Blackhat US*, 2017.

[32] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Automata-based automated detection of state machine bugs in protocol implementations." in *NDSS*, 2023.

[33] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols." in *NDSS*, 2023.

[34] F. Zuo, Z. Luo, J. Yu, T. Chen, Z. Xu, A. Cui, and Y. Jiang, "Vulnerability detection of ics protocols via cross-state fuzzing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4457–4468, 2022.

[35] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, "Pavfuzz: State-sensitive fuzz testing of protocols in autonomous vehicles," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 823–828.

[36] M. E. Garbelini, C. Wang, and S. Chattopadhyay, "Greyhound: Directed greybox wi-fi fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 817–834, 2020.

[37] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "{SweynTooth}: unleashing mayhem over bluetooth low energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 911–925.

[38] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "{BrakTooth}: Causing havoc on bluetooth link manager via directed fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1025–1042.

[39] C. M. IoT, "China mobile iot developer platform," https://open.iot.10086.cn/v4/en/, 2025, accessed: 2025-07-19.

[40] PeachTech. (2024) Peach Fuzzer Community. Accessed Nov. 26th, 2024. [Online]. Available: https://peachtech.gitlab.io/peach-fuzzer-community/

[41] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "Ics protocol fuzzing: Coverage guided packet crack and generation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[42] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah, "Ems: History-driven mutation for coverage-based fuzzing." in *NDSS*, 2022.