CMFuzz: Parallel Fuzzing of IoT Protocols by Configuration Model Identification and Scheduling

Qi Xu[†], Fuchen Ma[†] ✉, Yuanliang Chen[†], Wanli Chen[‡], Feifan Wu[†], Yanyang Zhao[†] ✉, Heyuan Shi[‡], Yu Jiang[†]
[†] KLISS, BNRist, Software of Software, Tsinghua University, Beijing, China
[‡] School of Electronic Information, Central South University, Changsha, Hunan, China

***Abstract*—IoT protocols are essential for the communication among diverse devices. In real-world scenarios, IoT protocols utilize flexible configurations to meet various use cases. These configurations can significantly impact the protocols' execution paths, with many bugs emerging only under specific configurations. Fuzzing has become a prominent technique for uncovering vulnerabilities in IoT protocol implementations. However, traditional fuzzing approaches are typically conducted using fixed or default configurations, overlooking potential issues that might arise in different settings. This limitation can lead to missing critical bugs that appear only under alternative configurations.**

In this paper, we propose CMFuzz, a parallel fuzzing framework designed to improve fuzzing effectiveness of IoT protocols through configuration identification and scheduling. CMFuzz first constructs a generalized protocol configuration model by systematically extracting configuration items from protocol implementations. Then, based on this model, CMFuzz defines the relations among configuration items and introduces a relation-aware allocation mechanism to distribute them across parallel fuzzing instances. For evaluation, We implement CMFuzz on top of the widely-used protocol fuzzer Peach and conduct experiments on six popular IoT protocols. Compared to the original parallel mode of Peach and state-of-the-art parallel protocol fuzzer SPFuzz, CMFuzz covers an average of 34.4% and 28.5% more branches within 24 hours. Additionally, CMFuzz has detected 14 previously-unknown bugs in these real-world IoT protocols.

***Index Terms*—IoT Protocols, Parallel Fuzzing, Configuration Model Identification and Scheduling.**

I. Introduction

The Internet of Things (IoT) refers to a network of interconnected devices that communicate and exchange data using various protocols to provide advanced services. These protocols enable essential functionalities across smart homes, healthcare, and industrial systems, highlighting their significance in modern technology [1]–[4]. However, IoT protocols are also vulnerable to cyber threats, particularly in critical infrastructure systems, where disruptions can impair operations, cause financial loss, or jeopardize safety [5], [6]. Ensuring the security of IoT protocols is therefore crucial for maintaining uninterrupted services and protecting public safety.

In IoT systems, protocols are inherently configuration-sensitive, as they are designed to operate across diverse scenarios and meet a variety of operational requirements. These protocols often offer extensive and diverse configuration options, where different combinations can influence the system's execution paths and behaviors. Consequently, certain vulnerabilities may only surface under specific configurations. For example, CoAP [7] supports block-wise transfers to handle large payloads by dividing them into smaller, sequential blocks. This feature is an optional configuration that significantly impacts the protocol's execution. When enabled, the protocol introduces additional logic to manage stateful interactions by tracking individual block transfers across multiple requests and responses. This added complexity increases the likelihood of issues such as memory leaks or resource exhaustion.

✉Fuchen Ma and Yanyang Zhao are the corresponding authors.

Fuzzing is a testing technique used to detect vulnerabilities in software by feeding volumes of inputs and observing the system's behavior. It plays a crucial role in identifying security flaws in IoT protocol implementations. Existing protocol fuzzers often build data models and state models to start the fuzzing process. A data model defines the structure and format of protocol inputs. While a state model describes the sequential flow of states that the protocol follows. Protocol fuzzers like Peach [8] and SPFuzz [9] leverage these two models to guide the generation of inputs and manage transitions between protocol states. As a result, they can systematically explore protocol behaviors and uncover a range of vulnerabilities.

However, traditional fuzzers neglect the configuration model of protocols, which sets the basic environment of the protocol instance. Thus, they overlook potential behaviors under alternative configurations in IoT systems. This limitation hinders the effective exploration of deeper protocol vulnerabilities that may arise in non-default settings. Therefore, it is essential to acknowledge the multi-configuration nature of IoT systems and incorporate multiple configurations during fuzzing under the guidance of configuration models. In fact, to explore protocol execution paths under various configurations, a straightforward approach is to run parallel fuzzing instances with different configurations. However, this presents two key challenges:

The first challenge is how to generate high quality configurations for fuzzing. IoT protocols are designed with rich and complex configurations to support a wide variety of use cases and devices. The configurations span multiple domains, including security settings, network parameters, resource management, application-specific options, and so on. Each of these configurations can affect the protocol's behavior and execution path, potentially revealing vulnerabilities. Automatically constructing realistic and comprehensive configurations that reflect real-world scenarios is a difficult task. **The second challenge** is how to divide configuration item combinations between different parallel instances. Configuration items often have dependencies, synergies, or conflicts. Arbitrary assignment across instances can lead to inefficiencies or initialization failures. The challenge, therefore, is to quantify the relations among configuration items and leverage them to group the items into cohesive groups.

To address the above mentioned challenges, we propose CMFuzz. First, in addition to the traditional data model and state model, CMFuzz introduces a generalized configuration model for protocol fuzzing. Given that IoT protocols may define configurations in different forms, such as configuration files and Command-Line Interface (CLI) options, CMFuzz systematically extracts and standardizes these configuration items to build a unified model. This abstraction accommodates different configuration formats and paradigms, ensuring adaptability across diverse IoT protocols. By decomposing protocol configurations into individual items, the model enables precise exploration of parameter spaces. Then, based on

the generalized configuration model, CMFUZZ defines the pairwise relations among configuration items and introduces a relation-aware allocation mechanism to distribute them across parallel fuzzing instances. In this way, CMFUZZ can effectively explore diverse and meaningful configuration spaces with minimal redundancy.

For evaluation, we implement CMFUZZ on top of the widely-used parallel protocol fuzzer Peach. We evaluate it on six real-world IoT protocol implementations. The results show that CMFUZZ improves the total number of branches covered by 34.4% on Peach and 28.5% on SPFuzz. It also achieves a significant speedup in reaching identical code coverage compared to baseline fuzzers. Furthermore, CMFUZZ has detected 14 previously unknown bugs in real-world IoT protocols. These critical vulnerabilities could lead to security breaches, crashes, or denial of service, posing significant risks to devices operating with these protocols in real-world environments.

Our main contributions are as follows:

- We introduce the configuration model to protocol fuzzing and propose a method to construct it, enabling exploration beyond default setups to uncover additional vulnerabilities.
- We consider dependencies between configurations and design a relation-aware allocation mechanism to effectively partition configuration spaces across parallel fuzzing instances.
- We implement and evaluate CMFUZZ on six widely used IoT protocol implementations. The results demonstrate that CMFUZZ outperforms the state-of-the-art fuzzers and has exposed 14 security-critical vulnerabilities.

II. BACKGROUND

A. IoT Protocols

The Internet of Things (IoT) has rapidly expanded, connecting billions of devices across various domains, from home automation and healthcare to industrial control systems. IoT protocols are essential for facilitating communication and interoperability among these devices, with each protocol adapted to meet the distinct requirements of IoT environments. Commonly used IoT protocols, such as MQTT, CoAP, and DDS, are designed to operate efficiently in constrained environments, supporting lightweight messaging, real-time data handling, and flexible Quality of Service (QoS) controls. These protocols typically provide mechanisms to ensure data reliability, adaptability to diverse network conditions, and interoperability across devices with varying capabilities.

Generally, IoT protocols are designed with a high degree of configurability to address the diverse and dynamic nature of IoT environments. Configurations, typically applied through configuration files or command-line interface (CLI) options, allow protocols to be tailored to match specific application requirements, device capabilities, and network conditions. For instance, security settings in these protocols may include options for different authentication methods, encryption algorithms, and key management strategies to ensure data integrity and confidentiality across the network. Quality of Service (QoS) parameters may specify message delivery guarantees, balancing factors such as latency and reliability. Network configurations define addressing schemes and routing protocols that support scalability and efficient communication. Resource management configurations enable protocols to optimize power consumption, bandwidth usage, and processing load to suit devices with limited resources.

B. Protocol Fuzzing

Fuzzing is a widely used technique to identify vulnerabilities in network protocol implementations by generating and injecting unexpected or malformed packets into the protocol to observe how it responds. Based on how packets are generated, fuzzers can be categorized into two types: mutation-based and generation-based. Mutation-based fuzzers generate test cases by randomly mutating existing valid inputs from a predefined corpus, applying various transformations such as bit flipping, field truncation, or inserting unexpected values. This approach simplifies fuzzing setup, as it does not rely on protocol specifications. In contrast, generation-based fuzzers construct inputs according to the protocol's structure and rules, producing well-formed messages that can effectively target specific states or fields. This allows them to thoroughly exercise the protocol's logic and identify vulnerabilities in various scenarios, making them particularly effective for complex protocols with structured interactions.

Traditional protocol fuzzers are typically based on two models: the data model and the state model. The data model defines the structure and format of packet fields, such as field lengths, types, and value ranges, allowing the fuzzer to craft near-valid protocol messages. The state model captures interaction sequences and transitions within the protocol, simulating how states change in response to different inputs. By adhering to protocol data formats, these two models help fuzzers produce highly tailored, protocol-compliant inputs, reaching deeper, less frequently accessed protocol states. This structured approach is especially advantageous for protocols with intricate dependencies or stateful interactions, as it ensures comprehensive coverage of both common and edge-case scenarios.

III. SYSTEM DESIGN

We present an overview of CMFUZZ's design in Figure 1. CMFUZZ starts with a set of inputs comprising the protocol under test and its configurations, defined through various sources such as command-line interface (CLI) options and configuration files. The objective of CMFUZZ is to systematically extract and organize these configurations to enable efficient and diverse parallel fuzzing. The framework outputs a set of cohesive configuration groups, which are then distributed to different parallel fuzzing instances.

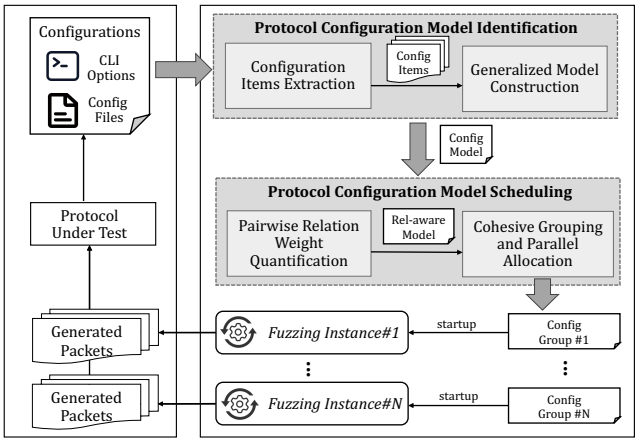


Fig. 1: CMFUZZ System Overview. It mainly consists of two components: (i) a configuration model identification module for extracting configuration items and constructing generalized model; (ii) a configuration model scheduling module for distributing configurations across parallel fuzzing instances.

CMFUZZ introduces a configuration model to explore a broader range of execution paths, supplementing the traditional data and state models used in existing protocol fuzzers. Starting with various protocol configuration formats, CMFUZZ first extracts a comprehensive list of configuration items to capture all adjustable parameters. These items are then organized into a generalized configuration

model, consisting of entities that represent each configuration item. Each entity includes the item's name, type, mutability flag, and a set of typical mutation values. With this configuration model in place, CMFUZZ then quantifies the pairwise relation weights between configuration entities by assessing their impact on startup coverage. Using these weights, CMFUZZ finally employs a cohesive grouping and allocation strategy, grouping interdependent entities together and assigning them to the same parallel fuzzing instance. This relation-aware allocation ensures that each instance explores a distinct and meaningful subset of configurations, thereby enhancing overall coverage and effectiveness in parallel protocol fuzzing.

A. Protocol Configuration Model Identification

1) Configuration Items Extraction: IoT protocol configurations are primarily found in two formats: command-line interface (CLI) options and configuration files. CLI options often follow predictable patterns, such as `--option=value` or `-flag`. To identify and extract these options, CMFUZZ uses a pattern-matching parser. For configuration files, formats vary widely, including key-value pairs (e.g., .ini files), structured hierarchies (e.g., JSON, XML), or unstandardized formats. CMFUZZ applies format-specific static analysis to handle each file structure: (i) For key-value formats, it directly parses each line to extract keys and values. (ii) For hierarchical formats, it recursively parses the structure to retrieve keys and default values based on the file's nested organization. (iii) For custom formats, CMFUZZ uses heuristics and configurable parsing rules to identify adjustable parameters based on keywords and contextual clues.

Algorithm 1: Configuration Items Extraction

Input: $\mathcal{C}_{options}$: CLI Options Configurations
Input: $\mathcal{C}_{files}$: Configuration Files
Output: Set_{CI}: The set of configuration items

1 **Algorithm**
2 $Set_{CI} \leftarrow \emptyset$
3 $Options \leftarrow$ GETCLIOPTIONS($\mathcal{C}_{options}$)
4 $Files \leftarrow$ GETCONFIGFILES($\mathcal{C}_{files}$)
5 `Extracion(` $Options, Files$)
6 **Procedure** `Extracion(` $Set_{Options}, Set_{Files}$)
7 **if** $Set_{Options} \neg \emptyset$ **then**
8 **for** $\mathcal{O}_\mathcal{C} \in Set_{Options}$ **do**
9 $Item \leftarrow$ EXTRACTCLIOPTIONS($\mathcal{O}_\mathcal{C}$)
10 $Set_{CI} \leftarrow Set_{CI} \cup Item$
11 **if** $Set_{Files} \neg \emptyset$ **then**
12 **for** $\mathcal{F}_\mathcal{C} \in Set_{Files}$ **do**
13 $Format \leftarrow$ DETECTFILEFORMAT($\mathcal{F}_\mathcal{C}$)
14 **switch** $Format$ **do**
15 **case** *KeyValue* **do**
16 $Set_{FI} \leftarrow$ EXTRACTKEYVALUE($\mathcal{F}_\mathcal{C}$)
17 **case** *Hierarchical* **do**
18 $Set_{FI} \leftarrow$ EXTRACTHIERARCHICAL($\mathcal{F}_\mathcal{C}$)
19 **otherwise do**
20 $Set_{FI} \leftarrow$ EXTRACTCUSTOMFORMAT($\mathcal{F}_\mathcal{C}$)
21 $Set_{CI} \leftarrow Set_{CI} \cup Set_{FI}$
22 **return**

Algorithm 1 illustrates the detailed workflow of CMFUZZ for extracting configuration items from various formats of IoT protocol configurations. The algorithm begins with two primary inputs: CLI Options Configurations and Configuration Files. First, we initialize an empty set Set_{CI} to store all configuration items extracted from these sources (line 2). Next, we retrieve CLI options and configuration files as structured data from the inputs (lines 3–4). Following this, we extract detailed configuration items according to their specific formats using the `Extraction` procedure. For each CLI option in $Set_{Options}$, we extract relevant details, including the configuration name, value and ranges, and add each item to Set_{CI} (lines 8–10). For each file in $Files$, we determine the format (line 13) and use format-specific extraction functions to parse the file's structure and content accordingly (lines 15–20). Items extracted from each file are then added to Set_{CI}, creating a consolidated set of configuration items for model construction (line 21).

2) Generalized Model Construction: Following the extraction of individual configuration items, CMFUZZ organizes these items into a generalized configuration model, which serves as the core structure within our framework.

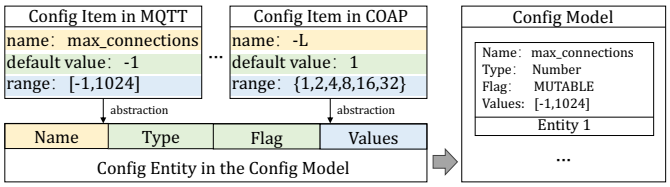


Fig. 2: Entities in the config model derived from the config item

As shown in Figure 2, this model is composed of 4-tuple entities, with each entity encapsulating essential attributes corresponding to a configuration item extracted from the protocol. The $Name$ attribute is inherited directly from the configuration item. The $Type$ attribute is inferred from the item's value patterns, for instance, numeric values are labeled as *Number*, boolean-like values as *Boolean*, and values resembling file paths or URLs as *String*. The $Flag$ attribute indicates whether a value is likely to change during typical protocol operations. Static values, such as paths or system directories, are marked as *IMMUTABLE*, while adjustable values, like numeric ranges or mode settings, are marked as *MUTABLE*. The $Values$ attribute, representing the typical set of values for each configuration, is derived from the item's defined ranges or common values using heuristic-based rules. This standardized configuration model provides a structured foundation for subsequent processes.

B. Protocol Configuration Model Scheduling

1) Pairwise Relation Weight Quantification: To leverage interdependencies among configurations and distribute the configuration model across parallel fuzzing instances, CMFUZZ enhances the original configuration model into a relation-aware configuration model, as illustrated in Figure 3. This transformation involves constructing a weighted relation graph, where each configuration entity is represented as a node, and edges with quantified weights connect pairs of entities. These weights reflect the strength of relationships between entities.

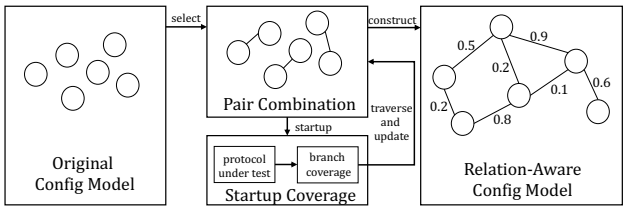


Fig. 3: Enhance an original config model into a relation-aware config model. CMFuzz achieves this by calculating the startup coverage.

When quantifying relations between configuration entities, CMFUZZ utilizes code coverage as a primary metric. Coverage serves as an effective indicator of configuration dependencies, as configurations

with synergistic relations often unlock new execution paths when used together. Conversely, configurations with conflicting relations, such as an encryption mode incompatible with a compression setting, may cause startup failures, resulting in zero code coverage. To improve efficiency, CMFuzz uses startup coverage as a lightweight proxy for overall code coverage. Because configurations are loaded and initialized during the startup phase, initial coverage provides a strong indication of the configuration's impact on later stages of execution.

By analyzing the startup coverage for each pair of configuration entities, CMFuzz then calculates the relation weight between them. Since each configuration entity may contain multiple typical values, CMFuzz explores all possible value combinations for each pair of entities and records the coverage achieved for each combination. To capture the peak interaction effect, CMFuzz selects the highest coverage across all combinations as the final relation weight between the two entities. This strategy ensures that the weight reflects the strongest potential interaction between entities, capturing critical synergies that might only emerge under specific values. Specifically, if the coverage for a pair of entities is zero across all combinations, CMFuzz does not create an edge between these entities. Finally, to ensure consistency and comparability across all relation weights, CMFuzz normalizes the calculated weights, scaling them to a standard range of [0, 1]. This normalization facilitates the grouping and clustering of configuration entities in subsequent phases.

2) Cohesive Grouping and Parallel Allocation: With the Relation-Aware Configuration Model constructed, CMFuzz proceeds to divide the graph into cohesive groups of configuration entities and allocate each group to a distinct parallel fuzzing instance.

Algorithm 2: Allocation Algorithm

Input: $RACModel$: Relation-Aware Configuration Model
Input: $\mathcal{N}_f$: The number of fuzzing instances
Output: $Groups$: The array of configuration groups allocated to each instance

1 **Algorithm**
2 $Groups \leftarrow \emptyset$, $GroupCnt \leftarrow 0$
3 $Set_{Edges} \leftarrow$ SortByWeight($RACModel$)
4 **for** $\mathcal{E}_s \in Set_{Edges}$ **do**
5 GroupNextEdge($\mathcal{E}_s$,$\mathcal{N}_f$)
6 **return** $Groups$
7 **Procedure** GroupNextEdge($\mathcal{E}$,$\mathcal{N}$)
8 $\mathcal{C}_1 \leftarrow \mathcal{E}.C_1$, $\mathcal{C}_2 \leftarrow \mathcal{E}.C_2$
9 **if** $\neg$IsSet($\mathcal{C}_1$) $\wedge$ $\neg$IsSet($\mathcal{C}_2$) **then**
10 **if** $GroupCnt < \mathcal{N}$ **then**
11 $NewGroup \leftarrow \{\mathcal{C}_1, \mathcal{C}_2\}$
12 $Groups \leftarrow Groups \cup \{NewGroup\}$
13 $GroupCnt \leftarrow GroupCnt + 1$
14 **else**
15 **for** $\mathcal{C} \in \{\mathcal{C}_1, \mathcal{C}_2\}$ **do**
16 $BestGroup \leftarrow$ FindBest($\mathcal{C}, Groups$)
17 $BestGroup \leftarrow BestGroup \cup \mathcal{C}$
18 **else if** IsSet($\mathcal{C}_1$) $\oplus$ IsSet($\mathcal{C}_2$) **then**
19 $CurGroup \leftarrow$ IsSet($\mathcal{C}_1$) ? Find($\mathcal{C}_1$) : Find($\mathcal{C}_2$)
20 $CurGroup \leftarrow CurGroup \cup$ (IsSet($\mathcal{C}_1$) ? $\mathcal{C}_2$: $\mathcal{C}_1$)
21 **return**

Algorithm 2 illustrates our allocation strategy, which aims to maximize the relation weights within groups and minimize the relation weights between groups. The algorithm starts with two inputs: the Relation-Aware Configuration Model $RACModel$ and the number of fuzzing instances N_f. First, we initialize an empty set $Groups$ to store the divided configuration groups and a counter $GroupCnt$ to track the number of groups created (line 2). Next, we sort all edges in $RACModel$ by weight in descending order, storing the sorted edges in Set_{Edges} (line 3). This sorting prioritizes edges with higher weights, promoting the formation of tightly connected groups by processing the strongest relationships first. For each edge in Set_{Edges}, we call the `GroupNextEdge` procedure, which determines how the two connected configuration entities (nodes) should be grouped (lines 4-5). In this procedure, if neither of the entities is assigned to a group (line 9), we verify whether the current number of groups $GroupCnt$ has reached the target number of instances $\mathcal{N}$ (line 10). If not, we create a new group with these two entities, add it to $Groups$, and increment $GroupCnt$ by one (lines 11-13). Once the maximum number of groups $\mathcal{N}$ has been reached, each new entity is placed into an existing group that best preserves its relations using the FindBest function (lines 15-17). Alternatively, if one entity is already part of a group, we assign another unassigned entity to the same group, thereby preserving their connection (lines 18-20).

Specifically, to identify the most appropriate existing group for an entity, the FindBest function computes a suitability score for each candidate group G_i based on the strength of the entity's connections to members of the group. The score is defined by:

$$Score(G_i, \mathcal{C}) = \frac{\left(\sum_{\mathcal{C}' \in G_i} w(\mathcal{C}, \mathcal{C}')\right)^2}{|G_i|}$$

where $w(\mathcal{C}, \mathcal{C}')$ represents the relation weight between the new entity $\mathcal{C}$ and each entity $\mathcal{C}'$ in group G_i, and $|G_i|$ denotes the number of entities in group G_i. In this formula, the numerator represents the strength of the relationship between the new node and existing nodes in G_i, while the denominator $|G_i|$ accounts for the balance in the number of entities across groups. The numerator is squared to amplify the effect of stronger connections.

Once cohesive groups have been formed, each group is assigned to a separate parallel fuzzing instance. Each instance then reassembles the configuration entities within its assigned group back into runtime-ready forms, such as configuration files or CLI options. During execution, each instance evaluates the $Flag$ attribute of each configuration entity to determine if the configuration value should be mutated. If mutation is required, it is applied based on the $Values$ attribute, which guides how the mutation should be performed. Mutations are introduced adaptively and are only applied if the current instance's coverage has reached saturation, meaning coverage has not increased over a set duration. Additionally, to prevent cross-contamination between instances, each fuzzing instance operates in an isolated network namespace. Through this structured parallel approach, each fuzzing instance is empowered to fully explore its assigned configuration subset, maximizing the chances of uncovering configuration-specific vulnerabilities.

IV. Evaluation

We have implemented CMFuzz on the state-of-the-art parallel protocol fuzzer Peach [8]. In the *Configuration Model Identification* module, we employ the Python Configuration File Parser [10] along with other specific libraries (such as `json`, `yaml`, `pyparsing`, and others) to convert configuration files into configuration items. For CLI options, we use Python Regular Expression Operations [11] to flexibly identify configuration items. In the *Configuration Model Scheduling* module, we leverage the `networkx` library to construct a weighted graph, where configuration entities serve as nodes and relations as weighted edges. For parallel fuzzing, we use Linux Network Namespaces (netns) via the `ip` networking tool to iso-

late network environments. Additionally, we use Clang to insert `trace-pc-guard` instrumentation, a feature in LLVM SanitizerCoverage [12], to collect branch coverage for pairwise relation weight quantification and parallel execution. For evaluation, we answer the following two research questions:

- **RQ1** Is CMFuzz more efficient than traditional parallel protocol fuzzers when applied to IoT protocols?
- **RQ2** Is CMFuzz effective in exposing previously unknown vulnerabilities in real-world IoT protocols?

A. Experiment Setup

Subjects. We selected six widely used IoT protocols: MQTT [13], CoAP [7], DDS [14], DTLS [15], AMQP [16] and DNS [17]. We use their corresponding popular open-source implementations as the test subjects, including Mosquitto [18], libcoap [19], CycloneDDS [20], OpenSSL [21], Qpid [22] and Dnsmasq [23].

Metrics. We employed three metrics for our evaluation: (i) branch coverage achieved, (ii) speedup in reaching the same coverage level as the compared fuzzers, and (iii) the number of bugs detected. The first metric is commonly used to measure the effectiveness of fuzzers, the second metric assesses the speed of parallel fuzzing, and the third metric indicates vulnerability detection capabilities.

Experiment Settings. Since the fuzzing performance fluctuates to a certain degree due to the inherent randomness, we ran each fuzzing tool on each selected project with a 24-hour time budget and repeated each 24-hour experiment five times to establish statistical significance of results [24]. For fairness, we use the same Pit files that specify the data and state models for each protocol. The experiment is executed in a Docker container with 4 CPU cores and 8 GB of RAM.

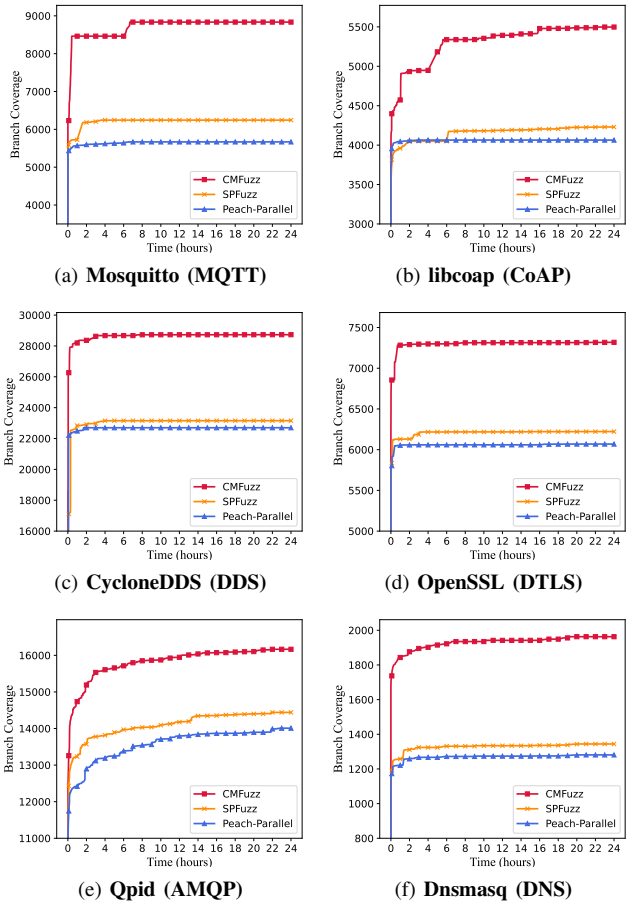
B. Coverage Analysis

To assess the efficiency of CMFuzz, we compared it to the state-of-the-art parallel-mode protocol fuzzers Peach [8] and SPFuzz [9]. Each fuzzer was run with four instances per project. We tracked and analyzed the number of branches covered by each fuzzer over 24 hours and calculated the speedup achieved by CMFuzz in reaching the same coverage level as the other fuzzers. Detailed results and improvements are summarized in Table I. The *Speedup* metric is defined as the baseline fuzzer's time to reach its final coverage divided by the time CMFuzz requires to achieve the same coverage. On average, CMFuzz improved branch coverage by 34.4% over Peach and 28.5% over SPFuzz across all tested protocol implementations. In terms of speedup, CMFuzz achieved an average acceleration of 3,544× compared to Peach and 2,746× compared to SPFuzz. These results reflect CMFuzz's significant enhancements in fuzzing efficiency for IoT protocols.

TABLE I: Average number of branches covered by each fuzzer running with four parallel instances over a 24-hour period.

Subject	CMFuzz	Peach	Improv	Speedup	SPFuzz	Improv	Speedup
Mosquitto	8,835	5,668	+55.9%	2,195×	6,244	+41.5%	30×
libcoap	5,498	4,064	+35.1%	450×	4,230	+30.0%	269×
cyclonedds	28,729	22,698	+26.6%	79×	23,155	+24.1%	146×
OpenSSL	7,318	6,069	+20.6%	8,820×	6,222	+17.6%	7,351×
Qpid	16,166	14,009	+15.4%	117×	14,440	+12.0%	41×
Dnsmasq	1,963	1,284	+52.9%	9,600×	1,348	+45.6%	8,640×
AVERAGE			+34.4%	3,544×		+28.5%	2,746×

Figure 4 illustrates that CMFuzz consistently achieves the highest branch coverage across all six projects, significantly outperforming the baseline tools Peach and SPFuzz. This underscores its effectiveness in exploring diverse execution paths. At the start of each session, all fuzzers exhibit an initial boost in coverage. CMFuzz achieves a considerable early lead because many of its extracted configuration items are loaded at startup, allowing rapid exploration of initial coverage. As time progresses, Peach and SPFuzz reach a saturation point, while CMFuzz continues to increase coverage by dynamically adjusting typical values from $Values$ field in configuration entities. This mechanism is particularly effective in protocols like CoAP, where varying block mode values allow exploration of alternative handling methods. Overall, in projects like Mosquitto and Dnsmasq, CMFuzz shows substantial improvement due to the extensive configurations in these protocols. For instance, Mosquitto supports varied QoS levels, authentication methods, and unique features like bridge connections, enabling CMFuzz to activate diverse paths by simulating different settings. In contrast, CMFuzz 's improvements on OpenSSL, CycloneDDS, and Qpid are modest due to limitations in flexibility. For instance, DTLS relies on fixed cryptographic settings, DDS's structured management restricts configuration diversity, and AMQP's predefined structure limits exploration.



(a) **Mosquitto (MQTT)** (b) **libcoap (CoAP)**

(c) **CycloneDDS (DDS)** (d) **OpenSSL (DTLS)**

(e) **Qpid (AMQP)** (f) **Dnsmasq (DNS)**

Fig. 4: Average number of branches achieved by CMFuzz and baseline parallel fuzzers within 24 hours for 5 repetitions on each IoT protocol implementation. All fuzzers are run with 4 instances.

C. Bug Detection Capability

In addition to the fuzzing efficiency improvement, CMFuzz has also uncovered 14 serious previously unknown vulnerabilities in these target protocols. These vulnerabilities pose significant security risks to devices operating with these protocols, potentially leading to device crashes, unauthorized access, or even complete system compromise. Table II summarizes the vulnerabilities discovered by CMFuzz. Each of them presents critical threats to IoT systems. Specifically, heap-use-after-free allows attackers to reuse freed memory, leading to

unpredictable behavior or remote code execution. SEGV is triggered by invalid memory accesses, and may expose sensitive information or allow code injection. Memory leaks, which cause memory to be exhausted over time, are particularly dangerous in resource-constrained IoT devices, potentially leading to system unresponsiveness and disrupted services. Allocation-size-too-big occurs when an abnormally large memory is allocated, potentially exhausting system resources or leading to denial-of-service attacks. Stack-buffer-overflow and heap-buffer-overflow involve writing data beyond allocated boundaries, potentially overwriting critical memory areas, leading to crashes or enabling attackers to execute malicious code.

TABLE II: Summary of vulnerabilities detected by CMFuzz

No.	Protocol	Vulnerability Type	Affected Function
1	MQTT	heap-use-after-free	Connection::newMessage
2		heap-use-after-free	neu_node_manager_get_addrs_all
3		heap-use-after-free	mqtt_packet_destroy
4		SEGV	loop_accepted
5		memory leaks	multiple functions
6	CoAP	SEGV	coap_clean_options
7		stack-buffer-overflow	CoapPDU::getOptionDelta
8		SEGV	coap_handle_request_put_block
9	AMQP	stack-buffer-overflow	pthread_create
10	DNS	stack-buffer-overflow	get16bits
11		heap-buffer-overflow	dns_question_parse, dns_request_parse
12		allocation-size-too-big	dns_request_parse
13		heap-buffer-overflow	printf_common
14		heap-buffer-overflow	config_parse

Case Study: Bug #8 in libcoap (CoAP). Figure 5 illustrates a SEGV vulnerability exposed by CMFuzz in the CoAP protocol. The bug occurs within `coap_handle_request_put_block` function, where `lg_srcv->body_data` is explicitly set to NULL (line 6) if `lg_srcv` is not found in the current session (line 3). However, if the block transfer process is incomplete or certain expected blocks fail to arrive, `lg_srcv->body_data` may remain NULL without being updated with valid data.

```
1  int coap_handle_request_put_block(...) {
2    ...
3    if (!lg_srcv) {
4      ...
5      // Initialize lg_srcv->body_data to NULL
6      lg_srcv->body_data = NULL;
7      LL_PREPEND(session->lg_srcv, lg_srcv);
8    }
9    ...
10 #if COAP_Q_BLOCK_SUPPORT
11   if (block_option == COAP_OPTION_Q_BLOCK1) {
12     if (check_all_blocks_in(...)) {
13       goto give_app_data;
14     }
15   }
16   ...
17 give_app_data:
18   ...
19   // Potential null dereference
20   pdu->body_data = lg_srcv->body_data->s;
21   ...
22 }
```

Fig. 5: The simplified code snippets related to the Bug#8.

When the non-default Q-Block1 configuration, which enables blockwise transfers, is activated (lines 10-11), the function checks if all blocks have been received (line 12). If they have, the code proceeds to the `give_app_data` label to reassemble and process the complete message body (line 13). At this label, `pdu->body_data` is assigned `lg_srcv->body_data->s` directly (line 20), assuming it is valid. However, if `lg_srcv->body_data` is still NULL, this assignment will cause a null pointer dereference, leading to a segmentation fault. This bug is critical as it potentially leads to server crashes and denial of service when handling blockwise transfers under specific configurations. Notably, it cannot be triggered under the default configuration, as it requires specific blockwise transfer settings to activate the vulnerable code path.

V. Related Work

Protocol Fuzzing. Numerous studies emphasize the extensive use of generation-based fuzzing for protocol implementation testing [8], [25]–[28]. Predominantly, existing research concentrates on enhancing algorithms that generate test packets. For example, Bleem [26] proposes a generation strategy focusing on packet sequences. Conversely, Logos [27] utilizes protocol server logs to conduct deep and efficient fuzzing within black-box environments. PAVFuzz [28] adopts a dynamic method to iteratively learn relationships between packet fields, aiming to improve packet generation. All these approaches focus on refining packet or sequence generation, and their effectiveness can be further bolstered by integrating CMFuzz. Our approach, CMFuzz, can autonomously determine and schedule configurations for parallel protocol fuzzing tasks and improve the effectiveness of the vulnerability detection. Thus, CMFuzz can be integrated with these existing methodologies to significantly boost fuzzing efficiency across multi-core computational resources.

Parallel Fuzzing. The enhancement of fuzzing efficiency is significantly driven by augmenting computational resources and leveraging parallel processing [29], [30]. Current research efforts concentrate on optimizing mutation-based fuzzers through information synchronization mechanisms and the strategic orchestration of fuzzing path scheduling. For example, tools such as AFL [31] and its protocol-oriented variant AFLNet [32] enable parallel fuzzing through synchronized seed sharing; AFLTeam [33] employs a dynamic task allocation model that refines task distribution, while PAFL [34] integrates a real-time synchronization framework for fuzzing status. Furthermore, SPFuzz [9] adopts a state-aware path-based method to enhance autonomous vehicle system scenarios. However, current fuzzing methodologies predominantly depend on standard configurations, neglecting the exploration of potential system behaviors under diverse configurations within IoT environments. This narrow focus restricts the ability to uncover deeper, configuration-specific protocol vulnerabilities that could emerge beyond default settings. Thus, recognizing the inherent multi-configuration complexity of IoT systems and integrating varied configurations into the fuzzing process becomes crucial for comprehensive vulnerability analysis.

VI. Conclusion

In this paper, we propose CMFuzz, a fuzzing framework that enhances parallel fuzzing for IoT protocols by dynamically identifying and scheduling configuration models. CMFuzz constructs a generalized configuration model by extracting protocol configuration items and grouping them based on inter-item relations. This approach enables parallel instances to explore diverse configuration combinations, significantly increasing coverage and the likelihood of discovering configuration-dependent bugs. Experimental results show that CMFuzz improves coverage by an average of 34.4% and 28.5% over Peach and SPFuzz, with significant speedups achieved for both baseline fuzzers. Furthermore, CMFuzz has uncovered 14 previously unknown vulnerabilities in real-world IoT protocols.

VII. Acknowledgment

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), and NSFC Program (No. U2441238, 62021002).

REFERENCES

[1] C. Paul, A. Ganesh, and C. Sunitha, "An overview of iot based smart homes," in *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, 2018, pp. 43–46.

[2] M. N. Bhuiyan, M. M. Rahman, M. M. Billah, and D. Saha, "Internet of things (iot): A review of its enabling technologies in healthcare applications, standards protocols, security, and market opportunities," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10 474–10 498, 2021.

[3] W. Z. Khan, M. Rehman, H. M. Zangoti, M. K. Afzal, N. Armi, and K. Salah, "Industrial internet of things: Recent advances, enabling technologies and open challenges," *Computers & electrical engineering*, vol. 81, p. 106522, 2020.

[4] A. Triantafyllou, P. Sarigiannidis, and T. D. Lagkas, "Network protocols, schemes, and mechanisms for internet of things (iot): Features, open challenges, and trends," *Wireless communications and mobile computing*, vol. 2018, no. 1, p. 5349894, 2018.

[5] I. Stellios, P. Kotzanikolaou, M. Psarakis, C. Alcaraz, and J. Lopez, "A survey of iot-enabled cyberattacks: Assessing attack paths to critical infrastructures and services," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3453–3495, 2018.

[6] K. Tsiknas, D. Taketzis, K. Demertzis, and C. Skianis, "Cyber threats to industrial iot: a survey on attacks and countermeasures," *IoT*, vol. 2, no. 1, pp. 163–186, 2021.

[7] C. Bormann and K. Hartke, "Constrained Application Protocol (CoAP)," RFC 7252, June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252

[8] M. Eddington, "Peach fuzzing platform." https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[9] J. Yu, Z. Luo, F. Xia, Y. Zhao, H. Shi, and Y. Jiang, "SPFuzz: Stateful path based parallel fuzzing for protocols in autonomous vehicles," 2024.

[10] Python, "Configuration file parser," https://docs.python.org/3/library/configparser.html.

[11] ——, "Regular expression operations," https://docs.python.org/3/library/re.html.

[12] LLVM, "Sanitizer coverage," https://clang.llvm.org/docs/SanitizerCoverage.html.

[13] A. Banks, E. Briggs, R. Coppen, and K. Borgendale, "MQTT Version 5.0," OASIS Standard, November 2018. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

[14] O. M. Group, "Data Distribution Service (DDS) Version 1.4," OMG Specification, April 2015. [Online]. Available: https://www.omg.org/spec/DDS/1.4

[15] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, January 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6347

[16] OASIS, "Advanced message queuing protocol (amqp)," OASIS Standard, October 2012. [Online]. Available: https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf

[17] P. Mockapetris, "Domain Names - Implementation and Specification," RFC 1035, November 1987. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1035

[18] Eclipse, "Mosquitto," https://github.com/eclipse/mosquitto.

[19] O. Bergmann, "libcoap," https://github.com/obgm/libcoap.

[20] Eclipse, "Cyclonedds," https://github.com/eclipse-cyclonedds/cyclonedds.

[21] OpenSSL, "Openssl," https://www.openssl.org/.

[22] Apache, "Qpid," https://qpid.apache.org/components/cpp-broker/.

[23] S. Kelley, "Dnsmasq," http://www.thekelleys.org.uk/dnsmasq/doc.html.

[24] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. W. Hicks, "Evaluating fuzz testing," *2018 ACM SIGSAC CCS*.

[25] jtpereyda, "BooFuzz: Network protocol fuzzing for humans," https://github.com/jtpereyda/boofuzz.

[26] 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.

[27] 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*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1720–1732. [Online]. Available: https://doi.org/10.1145/3650212.3680394

[28] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, "PAVFuzz: State-sensitive fuzz testing of protocols in autonomous vehicles," *ACM/IEEE Design Automation Conference (DAC)*, 2021.

[29] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2313–2328. [Online]. Available: https://doi.org/10.1145/3133956.3134046

[30] V.-T. Pham, M.-D. Nguyen, Q.-T. Ta, T. Murray, and B. I. Rubinstein, "Towards systematic and dynamic task allocation for collaborative parallel fuzzing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1337–1341.

[31] M. Zalewski, "american fuzzy lop - a security-oriented fuzzer," *https://github.com/google/AFL*, 2015.

[32] 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.

[33] V. Pham, M. Nguyen, Q. Ta, T. Murray, and B. I. Rubinstein, "Towards systematic and dynamic task allocation for collaborative parallel fuzzing," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering : NIER Track*, 2021.

[34] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: extend fuzzing optimizations of single mode to industrial parallel mode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 809–814. [Online]. Available: https://doi.org/10.1145/3236024.3275525