Parallel Fuzzing of IoT Messaging Protocols through Collaborative Packet Generation

Zhengxiong Luo[†], Junze Yu[†], Qingpeng Du[‡], Yanyang Zhao[†*], Feifan Wu[†], Heyuan Shi[§], Wanli Chang[¶], and Yu Jiang[†*]
[†]KLISS, BNRist, School of Software, Tsinghua University
[‡]Beijing University of Posts and Telecommunications [§]Central South University [¶]Hunan University

***Abstract*—Internet of Things (IoT) messaging protocols play an important role in facilitating communications between users and IoT devices. Mainstream IoT platforms employ brokers, server-side implementations of IoT messaging protocols, to enable and mediate this user-device communication. Due to the complex nature of managing communications among devices with diverse roles and functionalities, comprehensive testing of the protocol brokers necessitates collaborative parallel fuzzing. However, being unaware of the relationship between test packets generated by different parties, existing parallel fuzzing methods fail to explore the brokers' diverse processing logic effectively.**

This paper introduces MPFuzz, a parallel fuzzing tool designed to secure IoT messaging protocols through collaborative packet generation. The approach leverages the critical role of certain fields within IoT messaging protocols that specify the logic for message forwarding and processing by protocol brokers. MPFuzz employs an information synchronization mechanism to synchronize these key fields across different fuzzing instances and introduces a semantic-aware refinement module that optimizes generated test packets by utilizing the shared information and field semantics. This strategy facilitates a collaborative refinement of test packets across otherwise isolated fuzzing instances, thereby boosting the efficiency of parallel fuzzing. We evaluated MPFuzz on six widely-used IoT messaging protocol implementations. Compared to two state-of-the-art protocol fuzzers with parallel capabilities, Peach and AFLNet, as well as two representative parallel fuzzers, SPFuzz and AFLTeam, MPFuzz achieves (6.1%, 174.5×), (20.2%, 607.2×), (1.9%, 4.1×), and (17.4%, 570.2×) higher branch coverage and fuzzing speed under the same computing resource. Furthermore, MPFuzz exposed 7 previously unknown vulnerabilities in these extensively tested projects, all of which have been assigned with CVE identifiers.

***Index Terms*—IoT Messaging Protocol, Parallel Fuzzing, Collaborative Packet Generation**

I. Introduction

INTERNET of Things (IoT) represents a transformative shift in the industrial landscape, marked by the interconnectedness of devices, sensors, and actuators to facilitate seamless data exchange and collaborative decision-making [1]. By merging digital and physical systems, this advanced integration enables industries to boost productivity, streamline maintenance, and elevate operational efficiency. Nonetheless, the widespread adoption of IoT technologies also escalates security concerns, with vulnerable devices presenting potential targets for cyber-attacks, risking serious consequences [2], [3]. Critical to securing and stabilizing the IoT ecosystem, messaging protocols dictate device and user communications. However, these protocol implementations are vulnerable to cyber threats that can compromise data confidentiality, integrity, and availability, as well as disrupt industrial operations. Addressing these security flaws in IoT messaging protocols is imperative to ensure system integrity and avert the severe outcomes of cyber attacks [4].

Fuzzing, an automated software testing method, is promising for identifying security vulnerabilities in real-world software. At a high level, a protocol fuzzer works by continuously generating and sending test packets to the target server to uncover potential anomalies. Based on the packet generation way, fuzzers can be categorized into two categories: mutation-based methods such as AFLNet [5] and Polar [6], and generation-based methods such as Peach [7] and Boofuzz [8]. Mutation-based fuzzers, lacking awareness of the protocol structure, randomly modify existing packets at the byte or bit level, resulting in quick testing but potentially encountering challenges in passing the protocol parsing stage. Conversely, generation-based fuzzers leverage user-defined protocol models to create test packets that adhere to the protocol specifications, offering a more precise but slower approach that may require additional model construction and processing resources.

In IoT systems, numerous users simultaneously communicate with brokers to access and manage distinct production resources. It is essential for the broker to identify resources being accessed by various users and devices, while also maintaining the integrity of these communications. This is achieved by utilizing fields with special semantics, which we call key fields, to track these correlations. Considering this scenario of multiple interconnected communications, single fuzzing instances are insufficient for exhaustive testing of the entire messaging protocol workflow. Therefore, employing parallel fuzzing with multiple instances concurrently is crucial for the comprehensive testing of IoT messaging protocols.

Traditional parallel fuzzing methods for generation-based fuzzers mainly focus on non-conflicting task division across multiple instances, while neglecting the correlation among the packets produced by each instance. However, for the IoT messaging protocols, the key fields in the packets are closely related to each other, rendering the outputs of distinct instances interconnected. Therefore, it is essential to consider this correlation, moving beyond the conventional isolated parallel fuzzing framework. To achieve this, we need to address two challenges. (i) The first challenge is how to effectively synchronize information across parallel fuzzing

* Corresponding authors

instances. Since complicated synchronization schemes and excessive data exchange can cause significant overhead, it is essential to design a lightweight and efficient mechanism. (ii) The second challenge involves the utilization of the shared information by each fuzzing instance. Merely maintaining key fields' consistency across instances falls short of effectively exploring the broker's varied processing capabilities. It is essential to introduce variations for the key fields to explore diverse scenarios while considering their semantics.

To address these challenges, we propose MPFUZZ, a coordinated parallel fuzzing tool for securing IoT messaging protocols. The basic idea is to break the isolation between parallel fuzzing instances and enable them to collaboratively generate test packets by sharing key field information. MPFUZZ first introduces a lightweight synchronization mechanism that enables effective information sharing among parallel fuzzing instances. This mechanism leverages one global field pool and multiple local field pools bonded to each fuzzing instance to synchronize the key field information across instances. Meanwhile, this mechanism designed a structured data format to store the shared information, ensuring minimal data exchange overhead. Then, based on the synchronized information, MPFUZZ introduces a semantic-aware refinement strategy for each fuzzing instance to optimize the generated packets. Considering the distinctive characteristics of IoT messaging protocols, we propose a method to leverage the shared key field information. For fields with distinct semantics, we apply customized mutation operations to refine the packets generated by each instance. This facilitates a collaborative effort toward exhaustive testing of the broker's logic, enhancing the chances of uncovering subtle yet critical vulnerabilities.

For the evaluation, we built the proposed parallel fuzzing method MPFUZZ on top of Peach [7], a widely-used protocol fuzzer, and evaluated it on six widely-used messaging protocol implementations. We compared MPFUZZ against AFLNet [5] and Peach [7], two state-of-the-art protocol fuzzers that support parallel fuzzing, as well as two representative parallel fuzzers, SPFuzz [9] and AFLTeam [10]. The results show that, under the parallel fuzzing mode, MPFUZZ achieves significant coverage increase and speed improvements, achieving (6.1%, 174.5×), (20.2%, 607.2×), (1.9%, 4.1×), and (17.4%, 570.2×), respectively. We also show that MPFUZZ's parallel fuzzing strategy can enhance SPFuzz, with the same base fuzzer as MPFUZZ, by achieving higher coverage. Moreover, to emphasize the importance of parallel fuzzing for IoT messaging protocols, we conducted a comparison with the single modes of Peach and AFLNet using equivalent computational resources over the same time period. MPFUZZ surpassed them in branch coverage by 10.6% and 28.3%, respectively, over 24 fuzzing hours. Furthermore, MPFUZZ has exposed 7 previously unknown vulnerabilities, and AFLNet, Peach, SPFuzz, and AFLTeam only expose 2, 4, 5, and 4 of them, respectively. All these vulnerabilities have been assigned CVEs. MPFUZZ also shows superiority in identifying known vulnerabilities. Our main contributions are as follows:

- We propose the idea of coordinated parallel fuzzing for securing IoT messaging protocols based on collaborative packet generation.
- We propose a lightweight synchronization mechanism for effectively sharing key field information among fuzzing instances and design a semantic-aware refinement strategy to optimize the generated packets.
- We implement and evaluate MPFUZZ[1] on six widely used IoT protocol implementations. The results demonstrate that MPFUZZ outperforms the state-of-the-art and has exposed many security-critical vulnerabilities.

II. Background

A. IoT Messaging Protocols

The Internet of Things (IoT) marks a significant technological shift, connecting numerous "smart" devices via the Internet to exchange data. This network is key to modern innovation, allowing for automation and enhanced functionality across various sectors, including industrial automation, home automation, healthcare, and more.

Figure 1 illustrates a typical IoT system communication architecture. Within this framework, IoT devices (e.g., sensors and cameras) collect event and telemetry data, which they send to the control APP for analysis and processing. The control APP can also send commands to the devices to elicit specific actions. Communication between these entities is not direct but is mediated by a centralized broker, ensuring seamless interaction between the devices and the applications. This design allows for easy integration or removal of devices and apps, enhancing flexibility and scalability for various use cases. In this framework, both the device and the application can be regarded as clients and the broker as a server.

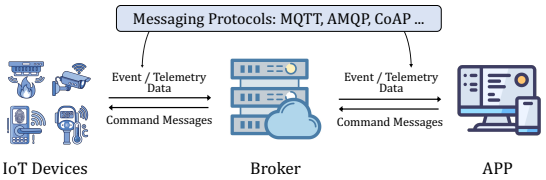


Fig. 1: Typical architecture and application of messaging protocols in IoT systems

Messaging protocols like MQTT, AMQP, and CoAP are vital for ensuring reliable, efficient, and scalable data transmission in this architecture. They are tailored for IoT constraints—low bandwidth, limited energy, and intermittent connectivity—ensuring appropriate service quality. Importantly, these protocols are specifically designed to support multiple devices and applications simultaneously, facilitating their connection and communication via a broker. This multi-connection capability ensures that the system can easily scale up to support a growing number of devices and applications.

B. Generation-Based Protocol Fuzzing

In protocol fuzzing, generation-based fuzzers, which are format-aware, are widely used due to the highly structured nature of protocol packets. They generate packets according to the user-provided test model [7], [8], [11], which includes data

[1]MPFUZZ is available at https://github.com/MPFuzz/MPFuzz.

and state models. The data model defines the packet structure while the state model describes the packet ordering.

Table I details the MQTT `SUBSCRIBE` and `PUBLISH` packet fields, including their descriptions and a Peach description for each. The Peach description, found in the rightmost column of Table I, utilizes Peach Pit [12] to define field characteristics relevant to the state-of-the-art protocol fuzzer Peach [7]. The `PUBLISH` packet comprises eight to nine fields: `Type`, `DUP`, `QoS`, `Retain`, `Length`, `TopicLen`, `Topic`, and `Msg` fields are mandatory, while the `ID` field is optional (needed only for packets with QoS levels 1 and 2). Expounding upon the properties of each field, the Peach definitions cover their crucial attributes like field size, type, value, and mutability. For example, the `Type` field is a four-bit field with a fixed value of 3, indicating the packet type. This field can be described as a *Flags* type and is immutable. The `QoS` field is a two-bit field with valid values of 0, 1, and 2, indicating the *Quality of Service* level for packet delivery. Since it is an enumeration field, we can describe it as a *Choice* type containing three *Flags* elements (with values of 0, 1, and 2, respectively). This field is partially mutable, as the value can be changed to any of the three valid values (so the overall *Choice* is mutable, while the individual *Flags* elements are immutable). Specially, the `Length` and `TopicLen` fields are relationship fields, as their values are determined by the lengths of other fields. These fields are calculated after the referenced fields are generated, and they are typically immutable to ensure the generated packet meets the protocol integrity requirements.

Based on the above description, Peach continuously generates consistent packets. In detail, Peach first analyzes the state model in the provided Pits file to determine the packet order to be sent, e.g., [`CONNECT`, `SUBSCRIBE`, ..., `DISCONNECT`] for MQTT. Then, it generates each packet based on its corresponding format specifications. A typical generation strategy is to randomly select and fuzz a mutable field at a time. Peach provides a variety of mutation operators for each field type. These mutation operators are designed to modify the field's default value while maintaining the field's requirements. Therefore, Peach starts from the top of the format specification and applies all valid mutations to each field element until all possible mutations have been used. Since the combination of the mutable fields and their compliant mutation operators are enumerable, the fuzzing iterations to be performed are finite.

III. Motivation

We use the MQTT [13] architecture and a simplified packet flow, as shown in Figure 2, to illustrate the necessity of collaborative parallel fuzzing for messaging protocols.

A. MQTT architecture

Figure 2 shows a basic MQTT architecture that comprises a broker, a publisher, and a subscriber. In this model, the broker is an intermediary between publishers and subscribers. The clients can be classified into two types: publishers, which disseminate messages, and subscribers, which receive messages. It also provides a basic packet flow: the subscriber first sends a subscription request (i.e., `SUBSCRIBE`) to the broker for a specific topic "SmartLock", which is carried in the `Topic` field of the packet (❶). The broker then accepts this request with a `SUBSCRIBE ACK` packet (❷) and stores the subscription information. Subsequently, when the corresponding publisher sends a `PUBLISH` packet to the broker, which contains the same topic "SmartLock" and provides the message payload in the `Msg` field (❸), the broker processes the packet and forwards the message payload to the subscriber (❹). The processing and forwarding logic (❹) of the broker is crucial for ensuring effective communication between publishers and subscribers. Therefore, the implementation of this logic is typically complex and should be thoroughly tested to identify potential vulnerabilities. Besides, simultaneous processing of disparate requests is an intrinsic characteristic of the protocol broker (as shown in the red lines in Figure 2), whereby data packets received from various publishers are processed by the broker and distributed to distinct subscribers. To effectively cover the broker's processing and forwarding logic, multiple parties should be involved in the testing, which requires parallel fuzzing.

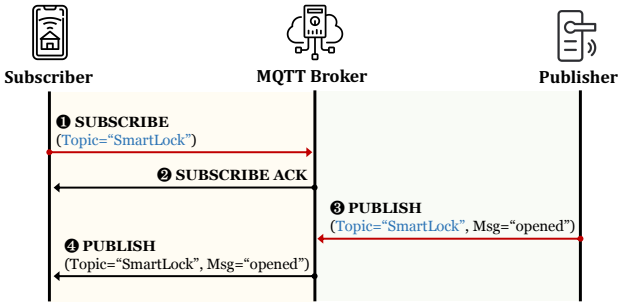


Fig. 2: Simplified MQTT architecture: subscriber, publisher, and broker with basic message flow

B. Limitation of Traditional Parallel Fuzzing

There are some existing fuzzers, such as Peach [7], that support parallel fuzzing. For instance, Peach provides a method to distribute fuzzing tasks across multiple instances, as shown in Figure 3. As mentioned in Section II, given the protocol model under test, Peach first computes all operation sequences to determine the total number of test iterations. Then, to avoid task conflicts, Peach assigns distinct field mutation tasks to each instance, ensuring no overlap in the fuzzing process. This is depicted in Figure 3, where tasks are segmented into N instances. Each one focuses on a different field subset, determined by the execution argument (*Div.Arg* in Figure 3). Specifically, an instance specified as *a/N* takes charge of the a-th portion out of N total portions.

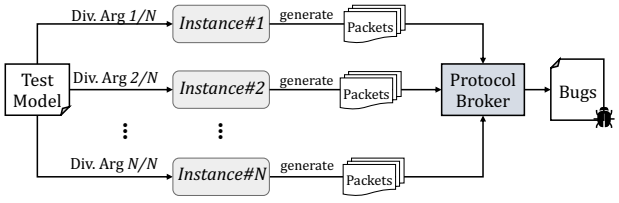


Fig. 3: Parallel fuzzing of Peach fuzzer

TABLE I: MQTT SUBSCRIBE and PUBLISH packet fields and their descriptions for Peach fuzzing

Packet Type	Basic Information		Peach Description			
	Field	Description	Size	Type	Value	Mutable
PUBLISH	`Type`	Indicates the packet type, for PUBLISH it's 3	4	Flags	`3`	○
	`DUP`	Duplicate delivery of a PUBLISH packet	1	Choice{Flags}	`0,1`	◐
	`QoS`	Quality of Service level for packet delivery	2	Choice{Flags}	`0,1,2`	◐
	`Retain`	Whether the packet should be retained by the broker	1	Choice{Flags}	`0,1`	◐
	`Length`	Length of the remaining packet	8×(1-4)	Number	*len(`TopicLen+Topic+ID+Msg`)*	○
	`TopicLen`	Length of the Topic field	16	Number	*len(`Topic`)*	○
	`Topic`	The topic where the message will be published	n	String		●
	`ID`	Packet identifier for QoS 1 and QoS 2 packets	0/16	Choice{Number}		●
	`Msg`	The content to be published	n	String		●
SUBSCRIBE	`Type`	Indicates the packet type, for SUBSCRIBE it's 8	4	Flags	`8`	○
	`Reserved`	Reserved field, using the constant 2	4	Flags	`2`	○
	`Length`	Length of the remaining packet	8×(1-4)	Number	*len(`ID+TopicLen+Topic+QoS`)*	○
	`ID`	Unique identifier for the packet	16	Number		●
	`TopicLen`	Length of the Topic field	16	Number	*len(`Topic`)*	○
	`Topic`	The topic to subscribe to	n	String		●
	`QoS`	Requested Quality of Service for the subscription	8	Choice{Number}	`0,1,2`	◐

* The *Size* column shows the length of the field in bits. The *len* function returns the length of the region in bytes. For the *Mutable* column, ○ indicates that the field is immutable, ● indicates mutable, and ◐ indicates partially mutable (For example, in a *Choice* type containing multiple *Number* elements, i.e., *Choice*{*Number*}, the individual *Number* elements are immutable while the overall *Choice* is mutable).

Although the existing parallel fuzzing method can effectively avoid task conflicts, it is not sufficient for testing the messaging protocol since the correlation between the packets generated by different instances is not considered. This correlation is mainly reflected in the key fields. There are mainly two types of key fields in the messaging protocol: the resource-related fields and the control-related fields.

The resource-related fields primarily establish the producer-consumer dynamics between publishers and subscribers, and these fields impact the dissemination of packets across the broker thus should be carefully managed when generating packets. For example, as illustrated in Figure 2 and Table I, the `Topic` field plays a crucial role. When a client sends a `SUBSCRIBE` packet, this field specifies the topics to subscribe to. The broker then forwards packets related to these topics to the appropriate subscribers. Similarly, the `Topic` field in a `PUBLISH` packet identifies the intended topic for the message's payload. If a subscriber has opted into a topic that a publisher addresses, the broker distributes the message across relevant subscribers based on this topical alignment, using mechanisms like wildcard matching. The absence of a matching topic causes the broker to discard unnecessary messages, avoiding redundant processing or dissemination. Therefore, it is essential to optimize for relevant subscriptions and topic targeting by publishers to reduce ineffective message flow and system strain.

The control-related fields, such as quality of service (QoS) and retention instructions, mainly refer to the fields that control the message delivery process of the broker, and should therefore be carefully crafted to exercise diverse broker logic when generating packets. For instance, MQTT supports multiple QoS levels, affecting how messages are published and subscribed. The broker defaults to the publisher's QoS level if it is lower than that requested by the subscriber for the same topic. This mechanism can trigger additional interactions between the broker and the subscriber under certain QoS conditions. Specifically, this `CONNECT` packet features numerous key fields, with only select data elements depicted in the Table I.

Existing parallel fuzzing methods mainly focus on task distribution across multiple instances to prevent overlap, but they lack customization in the generation strategies of each instance, typically employing basic random methods for packet generation. This method is reasonable for general protocols, since general protocols usually guarantee the independence of different client-server interactions. However, for messaging protocols like MQTT, the packets sent by different clients should be correlated, especially in the key fields, to trigger the broker's processing logic.

C. Insight and Challenges

Insight. To address the above problem, we introduce a collaborative parallel fuzzing strategy designed to overcome the shortcomings of conventional parallel fuzzing methods. This approach aims to synchronize key fields among different fuzzing instances and leverage the shared information to guide packet generation, thereby collaboratively exploring the broker's processing and forwarding logic. To this end, we need to address the following two challenges:

C.1 Lightweight Synchronization. The first challenge is how to synchronize information across different fuzzing instances effectively. On the one hand, one-to-one synchronization incurs significant overhead due to the need for frequent updates to maintain consistency. On the other hand, traditional methods that synchronize complete packets often result in redundancy and substantial post-processing costs [5]. Therefore, it is crucial to develop a lightweight synchronization mechanism that can effectively share essential information across different instances, thereby facilitating parallel fuzzing.

C.2 Collaborative Packet Generation. The second challenge centers on the utilization of shared information by each fuzzing instance to enhance packet generation, thereby collaboratively covering the diverse logic of the broker. Traditional fuzzers, which typically employ random field generation, are less likely to ensure that different instances produce packets with correlated key fields. Besides, merely reusing shared information to maintain key field consistency may not be adequate, as such compliant cases should have been covered in pre-release tests. We need to introduce variations while

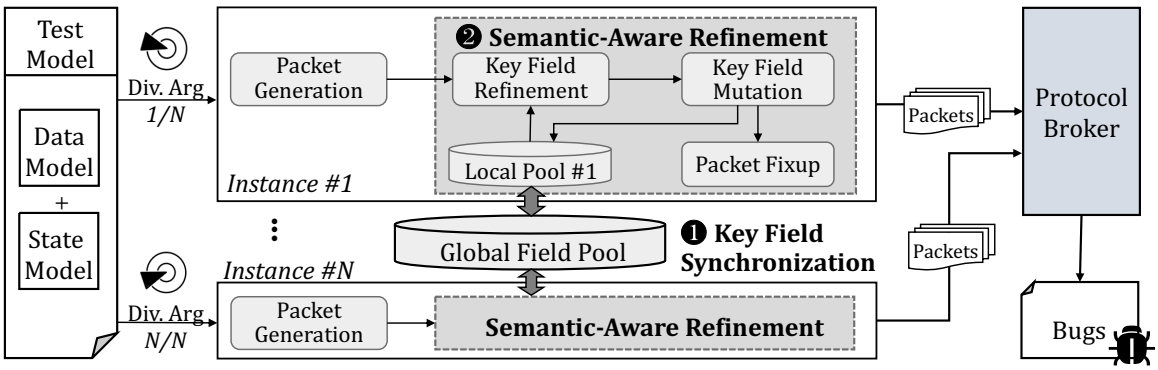


Fig. 4: MPFUZZ System Overview. It mainly consists of two components: (i) a synchronization module for sharing key fields among fuzzing instances and (ii) a semantic-aware refinement module for refining generated packets using shared information.

considering the fields' semantics to help uncover anomalies. This necessitates a generation strategy that can leverage the shared information and incorporate field semantics, guiding the packet generation process toward a collaborative and comprehensive exploration of the broker's logic.

IV. System Design

The MPFUZZ's system overview, as shown in Figure 4, follows a parallel fuzzing process that takes the same input as traditional generation-based fuzzers. This input includes the target protocol broker and protocol test model, which contains data and state models of the target protocol. The system comprises two main components: (i) an information synchronization mechanism for sharing key fields among fuzzing instances through local and global pools and (ii) a semantic-aware refinement module for refining test packets using shared field information. The system is designed to be scalable and efficient, enabling parallel fuzzing across multiple instances. The workflow of the system is outlined as follows:

Key Field Synchronization. Before parallel fuzzing, the global field pool is initialized based on the data model. If there are default values for the key fields, they are combined into a triple of default values, Model_Type and Field_ID, which are then added to the global field pool. If there is no default value, the inherent generation strategy in the data model will be used for generation instead of the default value. The parallelism fuzzing process is initiated, with the fuzzing instances being launched according to the configuration to perform parallel testing on the protocol broker. Every fuzzing instance maintains a local field pool throughout the testing process. During the instance initialization phase, since the local field pool is initially empty, the instance synchronizes all information from the global field pool to the local pool through an information synchronization algorithm.

Semantic-Aware Refinement. The instance starts running in a continuous loop, generating mutated test packets. In state S, the instance selects the data model corresponding to that state and generates an initial packet. The Semantic-Aware Refinement algorithm refines initial packet key fields with the local field pool and applies a semantic-aware mutation. If there is a corresponding index of the field in the local field pool, a value candidate is randomly selected from the pool as a seed. The seed is mutated by the Semantic-Aware mutator with a certain probability. If a new value is generated, it is added to the local field pool. The Fixup module is utilized to repair any new fields that may cause the generated packet to become an invalid state, ensuring its validity. The generated test packet is then sent to the protocol broker, with the processing process monitored to collect abnormal protocol broker states. After a fixed interval of testing iterations, the local field pool synchronizes with the global field pool through an information synchronization mechanism, obtaining key messaging fields from other instances.

A. Key Field Synchronization

In order to efficiently share information between parallel instances, we design a lightweight synchronization mechanism. It reduces unnecessary data exchange while enabling key fields to be shared between fuzzing instances.

Local and Global Field Pools. The local field pool contains key message fields specific to each parallel instance, while the global field pool stores shared key message fields across all parallel instances.

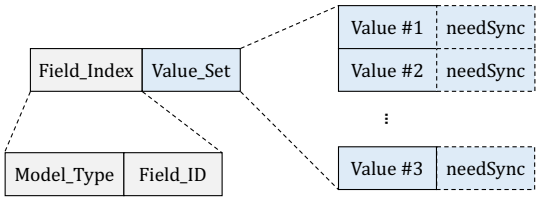


Fig. 5: The structure in Field Pool

The field pool retains key fields for diverse fuzzing instances to utilize and share. Its structure, comprising various field sets, is shown in Figure 5. Each set embodies a tuple structure of $\langle Field_{Index}, Value_{Set} \rangle$, where Field_Index distinctly identifies the shared value set of the key fields. To ensure uniqueness, the tuple composition of $\langle Model_{type}, Field_{ID} \rangle$ serves as the internal data structure for Field_Index. This demarcates disparities amidst identical fields among distinct packet data model types. In the ensuing semantic-aware refinement process, the pertinent field is chosen based on $Model_{type}$. Each pool set possesses a shared Field_Index and consists of varying field values. The value set can be perceived as

the seed set designated in the subsequent refinement process. Every value signifies an instantiated instance of the field. The following semantic-aware refinement module will elucidate comprehensively the utilization of the value set.

Moreover, the local pool can be viewed as a collection of 4-tuple entries, as illustrated in Figure 6. Additionally, the local pool contains a $needSync$ flag for each value, denoting the necessity of synchronization with the global pool the next time. The global pool does not require this flag. Entries with the same $\langle Model_{Type}, Field_{ID} \rangle$ mean they share an identical index of field and hold distinct values, thus establishing a diverse field set. The following synchronization and refinement algorithms will provide further clarification on the utilization and modifications of the field pool.

Model_Type	Field_ID	Value	needSync

Fig. 6: The entry in Field Pool

Synchronization between Local and Global Pools. Leveraging the global field pool, the two steps of the synchronization mechanism can achieve higher scalability and efficiency. Fuzzing instances often need to synchronize distinct key fields with each other, leading to complex sharing relationships. For example, fuzzing instances #1, #2, and #4 share field $Field_\alpha$, while fuzzing instances #1, #3, and #4 share field $Field_\beta$, demonstrating that the synchronized fields for instance #1 differ across its peers. Managing these relationships individually would result in an exponential increase in complexity as more parallel instances are added. To streamline this process, we introduce a global shared field pool that acts as a central hub for communication between pairs of instances. Each fuzzing instance communicates solely with this pool to access or update relevant information without direct interactions with other instances. Consequently, this approach establishes a star topology structure for information synchronization and communication among instances.

Since the information present in the field pool is of paramount importance for generating test packets, it is essential to synchronize the key fields across distinct fuzzing instances. Algorithm 1 describes the process of synchronization between local field pools and global field pools, consisting of two main steps: the Pull step and the Push step.

First, in the Pull step, we pull the fields from the global pool using the function READFROMGLOBALPOOL (Line 2). We then iterate over the global fields. For each $field_g$ in $GlobalFields$, we check if it is already in the local pool $\mathcal{P}_{local}$ (Line 3-4). If $field_g$ is not in $\mathcal{P}_{local}$, we add it to the local pool using the function ADDTOLOCALPOOL with $needSync$ flag setting to `False` (Line 5).

Next, in the Push step, we begin by retrieving fields from the local pool that require synchronization with the global pool using GETNOSYNCFIELDS, storing them in $NoSyncFields$ (Line 7). We then iterate over these unsynchronized fields in a for-loop (Line 8). For each unsynchronized $field$ in $NoSyncFields$, we check if it is already in the global pool $\mathcal{P}_{gobal}$ (Line 9). If $field$ is not in $\mathcal{P}_{global}$, we update the global pool using the function UPDATETOGOBLALPOOL (Line 10). Finally, we mark these fields in the local pool as synchronized by setting their `needSync` flag to false with SETSYNCFLAG (Line 11).

Algorithm 1: Synchronization between local field pools and global field pool

Input: $\mathcal{P}_{local}$: the pool of local fields
Input: $\mathcal{P}_{gobal}$: the pool of global fields
Output: $Sync\mathcal{P}_{local}$: the updated pool of local fields after synchronization

1 **Pull step:**
2 $GlobalFields \leftarrow$ READFROMGLOBALPOOL($\mathcal{P}_{gobal}$)
3 **for** $field_g \in GlobalFields$ **do**
4 **if** $field_g \notin \mathcal{P}_{local}$ **then**
5 ADDTOLOCALPOOL($\mathcal{P}_{local}, field_g, False$)
6 **Push step:**
7 $NoSyncFields \leftarrow$ GETNOSYNCFIELDS($\mathcal{P}_{local}$)
8 **for** $field \in NoSyncFields$ **do**
9 **if** $field \notin \mathcal{P}_{global}$ **then**
10 UPDATETOGOBALPOOL($\mathcal{P}_{gobal}, field$)
11 SETSYNCFLAG($\mathcal{P}_{local}, field, False$)

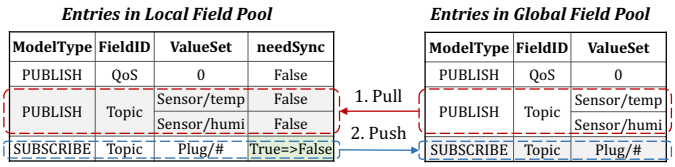


Fig. 7: Illustration of synchronization process

Figure 7 shows an illustrative example for Algorithm 1. A gray background indicates a newly added row after synchronization, while a green background indicates a modified cell. In detail, the Pull step traverses the global field pool and adds the non-existent entry with identification $\langle$PUBLISH, Topic$\rangle$ to the local field pool. Their $needSync$ flags are set to `False` simultaneously. The Push step traverses the local field pool and updates the entries needed to be synchronized, i.e., $\langle$SUBSCRIBE, Topic$\rangle$, to the global field pool. After synchronization, the $needSync$ flag is then set to `False`.

B. Semantic-Aware Refinement

After synchronizing information across parallel fuzzing instances, the instance obtained shared key message fields stored in the local field pool. Leveraging these fields, we implement a semantic-aware mutation strategy to refine the generated test packets. In contrast to traditional packet generation strategies, our algorithm utilizes the shared information provided by different parallel instances to refine the field and applies a semantic-aware mutation, ensuring that the key fields of generated packets are related to each other. Consequently, this method avoids wasting significant time and computing resources on invalid mutations of these fields.

Algorithm 2 provides a detailed illustration of how MP-FUZZ employs semantic-aware mutation to refine data packets based on the shared key fields. The algorithm starts with two inputs: (1) $\mathcal{P}_{local}$, the local key field pool, and (2) S, the current state that activates the algorithm via its output action.

We first obtain the data model DM_S for state S and generate the test data packet *CurPacket*, adhering to the syntax rules described by DM_S (Lines 2-3). Then, we extract the root node $Node_{\mathcal{R}}$ and the type $\mathcal{T}$ of DM_S (Lines 4-5) and recursively refine *CurPacket* based on the element tree by calling the `Refinement` procedure (Line 6).

Algorithm 2: Semantic-Aware Packet Refinement

Input: $\mathcal{P}_{local}$: Local Field Pool
Input: $\mathcal{S}$: State which is current under fuzzing
Output: $NewPacket$: Refined Packet

1 **Algorithm**
2 $\mathcal{DM}_S \leftarrow$ GETDATAMODEL($\mathcal{S}$)
3 $CurPacket \leftarrow$ GENERATEPACKET($\mathcal{DM}_S$)
4 $\mathcal{T} \leftarrow$ GETDATAMODELTYPE($\mathcal{DM}_S$)
5 $Node_{\mathcal{R}} \leftarrow$ PARSEFIELDTREE($\mathcal{DM}_S$)
6 `Refinement(` $Node_{\mathcal{R}}, CurPacket, \mathcal{T}$`)`
7 `// Packet Fixup`
8 $NewPacket \leftarrow$ FIXUP($CurPacket$)
9 **Procedure** `Refinement(` $Node_{\mathcal{T}}, Packet, \mathcal{T}$`)`
10 $ID \leftarrow$ GETIDFROMNODE($Node_{\mathcal{T}}$)
$\mathcal{T}_{Ref} \leftarrow$ GETREFERENCEDFROMMAP($\mathcal{T}$)
11 **if** $\langle \mathcal{T}_{Ref}, ID \rangle \in \mathcal{P}_{local}$ **then**
12 `// Key Field Refinement`
13 $Fields_{\mathcal{C}} \leftarrow$ GETCANDIDATES($\mathcal{P}_{local}, \mathcal{T}_{Ref}, ID$)
14 $Field \leftarrow$ RANDOMCHOICE($Fields_{\mathcal{C}}$)
15 `// Key Field Mutation`
16 $Field_{\mathcal{M}} \leftarrow$ APPLYSEMANTICAWAREMUTATOR($Field$)
17 ADDTOLOCALPOOL($\mathcal{T}, ID, Field_{\mathcal{M}}, False$)
18 $CurPos \leftarrow$ GETPOSITION($Node_{\mathcal{T}}$)
19 $Packet \leftarrow$ ASSEMBALFIELD($Packet, CurPos, Field_{\mathcal{M}}$)
20 **else**
21 $ChildrenNode \leftarrow$ GETCHILDREN($Node_{\mathcal{T}}$)
22 **for** $Node_C \in ChildrenNode$ **do**
23 `Refinement(` $Node_C, Packet, \mathcal{T}$`)`
24 **return**

In the `Refinement` procedure (Lines 9-25), we first identify the current field's *ID* and the type of the reference data packet $\mathcal{T}_{Ref}$ (Lines 10-11). $\mathcal{T}_{Ref}$ and *ID* are combined into a tuple to index the local field pool for retrieval (Line 12). If found, we extract a set of candidate fields, $Fields_{\mathcal{C}}$, from which we randomly select one key field, $Field_S$, as the mutation seed (Lines 14-15). Next, we apply the targeted mutator to perform semantic-aware mutation on this chosen seed, obtaining $Field_{\mathcal{M}}$ (Line 17). We then populate the original generated data packet with the mutated key field and return it (Lines 18-20). If the queried index is absent in the field pool, we collect all child nodes of $Node_{\mathcal{T}}$ and sequentially and recursively apply `Refinement` on these children (Lines 22-24), thus implementing a refinement algorithm based on semantic-aware mutation. Finally, upon recursively traversing the abstract element tree, we obtain the refined packet, and employ the FIXUP process to repair its integrity constraints, e.g., size, length, and checksum, to form a *NewPacket* (Line 8). By combining key field information and semantic-aware mutation, we can generate higher-quality protocol test packets. These refined packets adhere to syntax and semantics. This facilitates a collaborative effort toward exhaustive testing of the broker's logic, enhancing the chances of uncovering vulnerabilities.

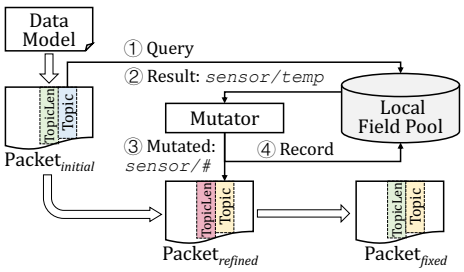


Fig. 8: Illustration of semantic-aware refinement

Figure 8 uses the MQTT PUBLISH packet (as detailed in Table I) as an example to illustrate this process. In detail, MPFuzz first generates the initial packet based on the data model. Then, MPFuzz traverses the packet fields and searches for the key fields for refinement—e.g., the `Topic` field in the PUBLISH packet. MPFuzz queries the local field pool based on `Topic` field's information (①). For one selected result, "sensor/temp" (②), the semantic-aware mutation operator is applied to generate a new value, e.g., "sensor/#". This mutated value, on the one hand, correlates with the key field generated by other instances and thus will be populated to the `Topic` field to generate the refined packet (③). On the other hand, this new value of the `Topic` field will be stored in the local field pool for future synchronization with other instances (④). Meanwhile, direct population to the initial packet may cause conflicts with other fields, e.g., the `TopicLen` field. Therefore, MPFuzz finally applies fixup operations to ensure the correctness of these relationship fields.

C. Implementation

We have implemented a prototype of MPFUZZ based on the widely-used generation-based fuzzer Peach [7]. We leverage the structure of the data model defined in Peach Pits file, adding new element types and attributes to indicate key fields that require synchronization and are subject to semantic-aware refinement within the data model. The *Synchronization Mechanism* is implemented using shared memory to store the global field pool and employing a semaphore based on the "first-come, first-served" principle to ensure only one instance can access the global field pool at a time. Each instance maintains the local field pool within its own process memory and periodically persists it to disk. We utilize the `DllImport` feature in the .Net framework to load libc and invoke Linux POSIX-compliant API calls through it to create, read, and write to shared memory. Interfaces defined in `System.Runtime.Serialization` are used to serialize sets from the field pools into shared memory, facilitating the mapping of objects. This approach supports switching between different serializers, such as BinaryFormatter, Xml, etc. For the *Semantic-Aware Refinement*, we introduced new mutators for the key fields and a new packet generation strategy based on the shared information, as described in Section IV-B. The new mutator operates specifically based on the semantics of the fields, for example, by attempting to incorporate wildcards and keywords specified in the protocol standards.

V. EVALUATION

In this section, we evaluate MPFUZZ to answer the following three research questions:

RQ1 How does the proposed parallel fuzzing strategy compare to the traditional parallel-mode fuzzing?

RQ2 How does MPFUZZ's performance compare to the single-mode protocol fuzzers on identical computing resources?

RQ3 Is MPFUZZ effective in exposing unknown vulnerabilities in real-world IoT messaging protocols?

A. Experiment setup

Subjects. We selected six open-source implementations of three widely-used IoT messaging protocols: MQTT [13], CoAP [14], and AMQP [15]. Table II provides detailed information about the selected subjects.

Compared Tools. We select two widely used protocol fuzzers supporting parallel fuzzing, Peach [7] and AFLNet [5], for comparison. They are representative of generation-based and mutation-based fuzzers, respectively. Peach is the base fuzzer for MPFUZZ, and we compare MPFUZZ with Peach to evaluate the effectiveness of the proposed parallel fuzzing optimization strategy. AFLNet's parallel fuzzing capabilities are inherited from its base fuzzer AFL. They establish parallel fuzzing by synchronizing the packet corpus among different instances. However, due to the lack of format awareness, AFLNet lacks a way to recognize the valuable fields in the synchronized packets for further utilization. Besides, we also compare MPFUZZ with two representative parallel fuzzers, SPFuzz [16] and AFLTeam [10], to evaluate the effectiveness of our optimization strategy for parallel fuzzing. Given that AFLTeam mainly targets traditional applications [17], we customized its code to support protocol fuzzing. For the initialization, since SPFuzz is also built upon Peach, we utilize identical Peach Pit files [12] for both SPFuzz and MPFUZZ. We provide the packets from these Peach Pit files as initial seeds for AFLNet and AFLTeam, ensuring a consistent starting point across all fuzzers following established work [16], [18].

Metrics and Settings. We employed three metrics for our evaluation: branch coverage achieved, speed-up to reach the same coverage as the baseline fuzzers in 24 hours, and the number of unique bugs detected. The first metric is commonly used to measure the effectiveness of fuzzers, the second metric assesses the efficiency of parallel fuzzing and has also been used in previous studies [16], [19], and the third metric indicates vulnerability detection capabilities. Besides, 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 [20]. For fairness, each fuzzing campaign runs on a Docker container with 4 CPU cores and 8G RAM.

TABLE II: Detailed information about the selected subjects

Subject	Protocol	#Stars	#LoC	Description
Mosquitto*	MQTT	8,275	52k	A MQTT implementation in the Eclipse IoT project.
NanoMQ*	MQTT	1,326	262k	Ultra-lightweight, blazing-fast messaging broker/bus for IoT edge and SDV.
Mongoose	MQTT	10,471	6k	Embedded networking library and web server.
libcoap*	CoAP	763	35k	A CoAP implementation ideal for resource-constrained IoT devices.
Californium	CoAP	721	133k	A framework offering modular and scalable IoT application support.
Qpid	AMQP	85	217k	An advanced message queuing protocol aims to unify message passing.

- '*': MPFUZZ exposed unknown bugs in the corresponding project. *#Stars*: the number of stars on GitHub. *#LoC*: the number of lines of code.

B. Efficiency of the Optimized Parallel Fuzzing Strategy

To evaluate the efficiency of our optimization strategy for parallel fuzzing, we first compared MPFUZZ to other state-of-the-art parallel-mode fuzzers. Each fuzzer was run with four instances on each project. We collected and analyzed the number of branches covered by each fuzzer over 24 hours and calculated the speed-up of MPFUZZ to reach the same coverage level as the compared fuzzers. The overall results and improvements are summarized in Table III. On average, MPFUZZ achieves a speed-up of 174.5×, 607.2×, 4.1×, and 570.2× compared to Peach, AFLNet, SPFuzz, and AFLteam, respectively, showing a substantial efficiency improvement in parallel fuzzing. Meanwhile, MPFUZZ achieves a higher branch coverage within 24 hours. Specifically, compared to the baseline fuzzers, MPFUZZ increases branch coverage by an average of 6.1%, 20.2%, 1.9%, and 17.4%, respectively. This coverage improvement is attributed to the efficiency-boosting since MPFUZZ can generate more correlated packets early on, thus covering more branches given the limited time budget.

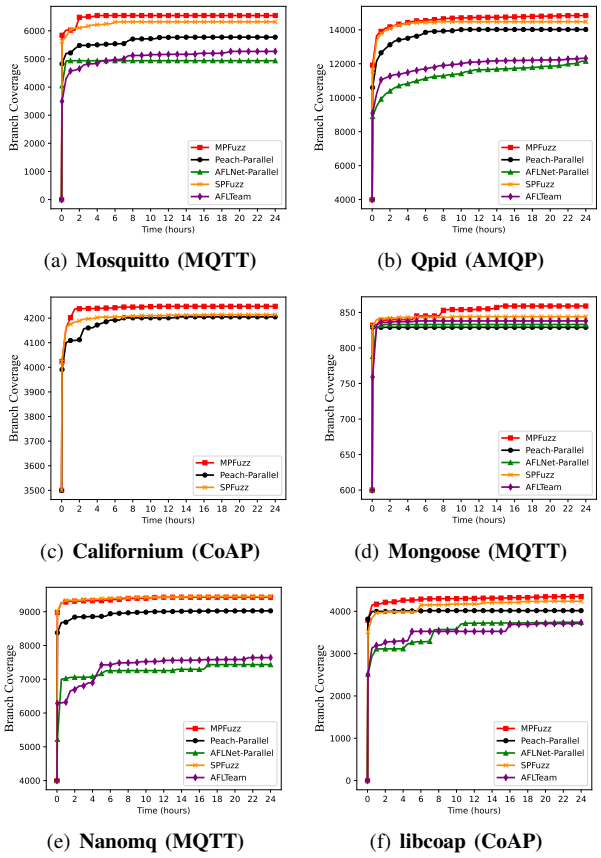
Figure 9 shows the coverage growth trends of each fuzzer over time on the selected projects. The plots indicate that both MPFUZZ and Peach exhibit effectiveness at the beginning of execution, displaying a rapid increase in branch coverage. However, after a certain point, Peach tends to stagnate and reach a state where increasing branch coverage becomes challenging. Table III demonstrates that MPFUZZ achieves the same branch coverage at a speed of 2.6× to 553.8× compared to the original Peach parallel mode. Besides, due to format unawareness and the lack of effective parallelization for IoT messaging protocols, AFLNet also exhibits a slow increase in branch coverage. AFLTeam introduces a task division strategy for parallelized fuzzing, thus achieving a higher branch coverage than AFLNet. However, it is unaware of the correlation between packets generated by different instances and the packet format, leading to a slower increase in branch coverage compared to MPFUZZ. In contrast, powered by the key field synchronizing mechanism and semantic-aware refinement, MPFUZZ consistently delivers high-quality packets generated by collaborative fuzzing instances, which helps alleviate the coverage plateau situation and achieve a sustained increase in branch coverage.

Notably, MPFuzz shows a less pronounced advantage over SPFuzz compared to other baseline fuzzers. This is because SPFuzz also proposes an optimization strategy for parallel

TABLE III: Average number of branches covered by MPFuzz and the baseline fuzzers in parallel mode within 24 hours, all with four fuzzing instances.

Subject	MPFuzz	Comparison with Peach			Comparison with AFLNet			Comparison with SPFuzz			Comparison with AFLTeam		
		Peach	Improv	Speed-up	AFLNet	Improv	Speed-up	SPFuzz	Improv	Speed-up	AFLTeam	Improv	Speed-up
Mosquitto	6,544	5,774	13.3%	553.8×	4,937	32.6%	597.1×	6,320	3.5%	4.8×	5268	24.2%	734.4 ×
Mongoose	859	829	3.6%	2.6×	833	3.1%	22.0×	844	1.8%	1.8×	838	2.5%	7.6 ×
NanoMQ	9,429	9,025	4.5%	391.4×	7,433	26.9%	1514.3×	9,450	-0.2%	0.9×	7641	23.4%	1193.7 ×
libcoap	4,350	4,016	8.3%	77.3×	3,742	16.2%	524.1×	4,238	2.6%	1.37×	3734	16.5%	592.3 ×
Californium	4,248	4,205	1.0%	12.4×	-	-	-	4,214	0.8%	13.6×	-	-	- ×
Qpid	14,843	14,019	5.9%	9.6×	12,152	22.1%	378.8×	14,447	2.7%	2.0×	12333	20.3%	322.9 ×
AVERAGE			6.1%	174.5×		20.2%	607.2×		1.9%	4.1×		17.4%	570.2×

* AFLNet and AFLTeam do not support Californium due to its Java-based implementation.



(a) **Mosquitto (MQTT)** (b) **Qpid (AMQP)**

(c) **Californium (CoAP)** (d) **Mongoose (MQTT)**

(e) **Nanomq (MQTT)** (f) **libcoap (CoAP)**

Fig. 9: Average number of branches covered by MPFuzz and baseline parallel fuzzers within 24 hours on each IoT protocol implementation. All paralyzed fuzzers are run with 4 instances.

protocol fuzzing building on Peach. SPFuzz focuses on minimizing task conflicts and distributing workload across different instances. This method is complementary rather than directly competitive with the optimization of MPFuzz. For the IoT messaging protocols, the key fields generated by different fuzzing instances should be correlated (as per MPFuzz's idea), while non-key field generation tasks should be allocated separately to minimize conflicts (as per SPFuzz's idea). To further show their complementary relationship, we adapt MPFuzz on SPFuzz and evaluate the combined performance. As shown in Table IV, the combined tool achieves an average increase of 3.9% in coverage than SPFuzz. This result further demonstrates the importance of correlated key field generation for IoT messaging protocols in parallel fuzzing.

TABLE IV: Average code branches achieved by adapting MPFuzz on SPFuzz and improvement over SPFuzz in 24 hours

Mosquitto	Mongoose	NanoMQ	libcoap	Californium	Qpid
6778	862	9661	4545	4251	15002
(+7.2%)	(+2.1%)	(+2.2%)	(+7.2%)	(+0.9%)	(+3.8%)

C. Comparison with Single-Mode Fuzzers

To demonstrate the effectiveness of the parallel mechanism, we also compared MPFuzz to single-mode fuzzers with the same computing resources. Specifically, we first set up four MPFuzz instances in parallel mode and collected the number of branches covered in all six targets within 6 hours. Then, to have a fair comparison, we ran Peach and AFLNet in single mode and collected their branch coverage within 24 hours.

Table V shows the branches covered by each fuzzer. We can find that MPFuzz always performed better than other fuzzers in all target protocols. On average, MPFuzz achieves 10.6% and 28.3% higher branch coverage than Peach and AFLNet in single mode, respectively. On all the selected projects, MPFuzz achieves the upper bound in branch coverage, showing a substantial lead on these projects. This substantial improvement in branch coverage emphasizes the importance of parallel fuzzing for IoT messaging protocols since their application scenarios require multiple instances to collaboratively generate test packets to cover diverse functionalities.

TABLE V: Average number of branches covered by MPFuzz (with four instances) in 6 hours and the baselines Peach and AFLNet in single mode within 24 hours

Subject	MPFuzz	Peach	Improv	AFLNet	Improv
Mosquitto	6,544	5,120	27.8%	4,265	53.4%
Mongoose	840	801	4.9%	786	6.9%
NanoMQ	9,328	8,703	7.2%	7,284	28.1%
libcoap	4,288	3,969	8.0%	3,511	22.1%
Californium	4,248	4,011	5.8%	-	-
Qpid	14,843	13,482	10.1%	11,311	31.2%
AVERAGE			10.6%		28.3%

D. Bug Detection Capability

To show the effectiveness of MPFuzz in detecting unknown vulnerabilities, we utilize AddressSanitizer [21] and UndefinedBehaviorSanitizer [22] to enhance the target program and use the crashes identified by MPFuzz to represent its vulnerability detection ability. Furthermore, to eliminate duplicate entries, we utilize the stack traces in the Sanitizer report for bug de-duplication and only consider unique vulnerabilities.

TABLE VI: Summary of the Exposed Vulnerabilities

No.	Subject	Description	AFLNet	AFLTeam	Peach	SPFuzz	MPFuzz	CVE ID
1	NanoMQ	Heap buffer overflow in mqtt_codec.c when handling a PUBLISH request		●	●	●	●	2024-31036
2	NanoMQ	SEGV in nni_msg_set_cmd_type of message.c caused by null pointer	●	●	●	●	●	2023-34491
3	NanoMQ	Heap buffer overflow in the get_var_integer in mqtt_parser.c					●	2024-31040
4	NanoMQ	Null pointer passed as the second argument of strncpy in topic_filtern		●	●	●	●	2024-31041
5	libcoap	Buffer over-read via the function coap_parse_oscore_conf_mem	●	●	●	●	●	2023-35862
6	Mosquitto	Null pointer passed to memcmp in broker handle_publish.c				●	●	2024-31038
7	Mosquitto	Null pointer dereference when handling a CONNECT request					●	2024-31039
8	NanoMQ	Heap buffer overflow via the function nmq_subinfo_decode	●	●	●	●	●	2023-33659
9	NanoMQ	Null pointer dereference in decoding subinfo_decode and unsubinfo_decode		●	●	●	●	2023-29996
10	Mosquitto	Memory leak occurs when handling v5 CONNECT packets	●			●	●	2023-3592
11	Mongoose	Heap buffer overflow when parsing MQTT_CMD_PUBLISH message					●	2023-2905

* CVEs are anonymized for the review. ● means that the fuzzer can detect the corresponding vulnerability. #1-#7 are previously unknown vulnerabilities exposed by MPFuzz and we also show whether other fuzzers can detect them. #8-#11 are known high-severity vulnerabilities in the tested subjects.

Along with the code coverage and fuzzing speed improvement, MPFuzz has also exposed 7 serious previously unknown vulnerabilities on the selected subject projects. All the vulnerabilities have been assigned with CVE identifiers and have been fixed by their respective vendors. We also collected statistics on whether other baseline fuzzers can detect these vulnerabilities. Table VI summarizes the new bugs exposed by MPFuzz (#1-#7). Specifically, AFLNet, AFLTeam, Peach, and SPFuzz can only expose 2, 4, 4, and 5 bugs, a strict subset of the bugs exposed by MPFuzz. These implementations are widely used in IoT devices and have been thoroughly tested by the community, as shown in Table II. Even so, MPFuzz can still detect new vulnerabilities that may pose serious threats to protocol security. Besides, we collected 4 high-severity CVEs previously detected in the selected subjects from the National Vulnerability Database to further evaluate each fuzzer's bug-finding ability. As shown in Table VI (#8-#11), MPFuzz can reproduce all these known bugs, while other fuzzers can only reproduce 2, 2, 2, and 3 CVEs, respectively. These bugs pose potential hazards for devices running these protocols. We provide a case study as follows.

Case Study: Bug #1 in NanoMQ. Figure 10 illustrates a heap-buffer-overflow vulnerability discovered by MPFuzz in NanoMQ. The bug manifests as a heap-buffer-overflow error via the `read_byte` function (invoked in Line 12) in *MQTT_code.c* module of NanoMQ. When processing a malformed MQTT message generated by MPFuzz, the program attempts to read beyond the allocated buffer via the pointer `buf->curpos` in Line 5. This bug can pose a serious threat to the security of IoT devices and attackers may use it to conduct a Denial of Service attack.

```
1  /* Function to read a byte from a buffer */
2  int read_byte(struct pos_buf *buf, uint8_t *val) {
3      if ((buf->endpos - buf->curpos) < 1)
4          return MQTT_ERR_NOMEM;
5      *val = *(buf->curpos++);
6  }
7  /* Function to decode properties from a buffer */
8  property * decode_buf_properties(uint8_t *packet,
       uint32_t packet_len, uint32_t *pos, uint32_t *len,
       bool copy_value) {
9      ...
10     // parsing the properties section of the message
11     while (buf.curpos < buf.endpos) {
12         if (0 != read_byte(&buf, &prop_id)) {
13             property_free(list);
14             break;
15         }
16         property *cur_prop = NULL;
17         property_type_enmu type =
               property_get_value_type(prop_id);
18         cur_prop = property_parse(&buf, cur_prop,
               prop_id, type, copy_value);
19         property_append(list, cur_prop);
20     }
21 out:
22     current_pos += (prop_len);
23     *pos = current_pos;
24     *len = prop_len;
25     return list;
26 }
```

Fig. 10: The simplified code snippets related to the Bug#1.

VI. Related Work

A. Protocol Testing

Protocol Fuzzing. Fuzzing has gained widespread acceptance as a testing approach for various protocol implementations, intending to automatically detect vulnerabilities [6], [11], [23], [24]. Protocol fuzzing can be divided into two main categories: generation-based and mutation-based. Mutation-based approaches [5], [25]–[28] treat the entire packet sequence as a single seed where new packet sequences are generated by mutating existing seeds. However, unaware of the protocol format, trivial mutation operations remain limited in generating valid packets and cannot explore a range of protocol implementation logics. Instead, generation-based fuzzers are format-aware and thus can generate valid packets with high probability. Our work focuses on optimizing parallel fuzzing for generation-based approaches.

Generation-based approaches [7], [8], [29] take a user-provided protocol model as input and generate packets that conform to the protocol model [30]. Recent works optimize generation-based protocol fuzzing from different perspectives. Based on the original format, Bleem [18] introduces a packet-sequence-oriented generation strategy. By analyzing the output packet sequence, Bleem provides an effective feedback mechanism in the blackbox setting and supports guided fuzzing based on runtime state-space tracking. Meanwhile, Bleem generates protocol-logic-aware packet sequences by leveraging the observed interactive traffic. Snipuzz [31] automatically infers packet format in IoT scenarios by employing the server response to enhance fuzzing efficiency. It uses a hierarchical clustering strategy to infer the grammatical role of each packet byte by analyzing the server response. ChatAFL [19] extends the existing model by utilizing the knowledge from large language models (LLMs) to construct grammars for protocol

message types. This enables it to mutate messages or predict subsequent messages in a sequence, thereby improving test input generation. Peach* [23] optimizes packet generalization by integrating coverage guidance. It collects coverage information during tests to identify valuable packets that uncover new execution paths. These packets are then dissected and used to generate higher-quality test packets. Charon [11] enriches the transition details of the state model by employing state guidance. It optimizes cross-state code coverage in the fuzzing of ICS protocols, surpassing the conventional emphasis on individual states. These approaches primarily focus on protocol testing in single fuzzing instances, while our work concentrates on parallel fuzzing. Therefore, these optimization strategies are orthogonal to the technique we proposed and can be applied to MPFUZZ to obtain better fuzzing performance.

Messaging Protocol Testing. There are also some works focusing on testing messaging protocols. MPInspector [32] is designed to enhance the security of IoT messaging protocols by combining model learning with formal analysis. It operates in three primary phases: comprehending the functionality of the messaging protocol, identifying applicable security standards, and verifying adherence to these standards. MQTTactic [33] conducts systematic security analysis of open-source MQTT broker based on static analysis and formal model checking. It semi-automatically checks the security of MQTT broker implementations and focuses on verifying MQTT broker implementations against generated security properties, specifically targeting authorization-related issues. These works are mainly about designing properties to detect vulnerabilities. These properties can be integrated into MPFUZZ to enhance the detection capability for the property-related bugs.

B. Parallel Fuzzing

Existing work on optimizing parallel fuzzing has focused on various aspects, such as task division, information synchronization, and task scheduling. OPAFL [34] develops a multi-candidate scheduling system based on input-related tasks. EnFuzz [35] ensembles diverse fuzzers to broaden fuzzing capability. PAFL [9] divides tasks based on the coverage bitmap and synchronizes guiding information across fuzzing instances. AFLTeam [10] further enhances task division and scheduling. It leverages an attributed graph incorporating program call graph and fuzzing information and applies techniques from graph partitioning and search algorithms to optimize the task allocation. SPFuzz [16] leverages the stateful path generated by analyzing the protocol state model for better task division and scheduling. Instead, MPFUZZ leverages the feature of IoT messaging protocols and mainly focuses on the collaborative refinement of certain key fields generated by different instances. Therefore, MPFUZZ can be integrated with these works to further optimize the parallel fuzzing for IoT messaging protocols.

Some research concentrates on distributed fuzzing techniques across multiple machines [36]–[38]. These techniques typically employ a centralized architecture, with the main node responsible for task scheduling and information synchronization. ClusterFuzz, the fuzzing backend of OSS-Fuzz [39], operates on 30,000 computing cores. Unlike these works, our work mainly focuses on parallel fuzzing on a single machine.

VII. DISCUSSION

Adaption to Other Fuzzers. MPFUZZ mainly focuses on optimizing the parallel fuzzing process for the generation-based fuzzers. Since our approach relies on the synchronization of key fields and then adapts customized mutation strategies based on the shared information and field semantics, our approach requires the protocol format to be available. Therefore, it may not be directly applicable to mutation-based fuzzers, which generate test cases by mutating existing test cases without considering the protocol format. If provided with the information about the key fields, our approach can be also adapted to the existing parallel fuzzing mechanism of mutation-based fuzzers.

For generation-based fuzzers, although our optimization is built on the Peach framework, it can be easily adapted to other generation-based fuzzers with parallel fuzzing capabilities. On the one hand, the Peach framework is widely used in the fuzzing community, and many fuzzing tools are built on top of it. Many recent works are proposed to optimize Peach for effective protocol fuzzing from different perspectives, MPFUZZ can be easily adapted to these tools. For example, PAVFuzz [24] optimizes the fuzzing by dynamically learning the relationships between the fields in the protocol packets sent by a single fuzzer. Charon [11] employs cross-state guidance to improve the efficiency of each fuzzing instance. Since all these optimization strategies are focused on individual fuzzing instances, and our optimization is orthogonal to them, MPFUZZ can be easily integrated into these tools by integrating the proposed key field synchronization module and semantic-aware refinement strategy to further improve their performance in parallelly fuzzing IoT messaging protocols. On the other hand, other generation-based fuzzers that do not currently support parallel mode, such as Boofuzz [8], follow the same packet generation process as Peach, as described in Section II. Therefore, once they support parallel fuzzing, our optimization can be also easily adapted to them.

Scalability to Distributed Parallel Fuzzing. The current design of MPFUZZ is based on the assumption that the fuzzing instances are running on the same machine. Some recent works focus on distributed fuzzing techniques that utilize multiple machines, such as P-Fuzz [36] and UltraFuzz [37]. The proposed synchronization mechanism (in Section IV-A) based on global and local field pools remains applicable to distributed frameworks. The distributed fuzzing technique typically adopts a controller-worker architecture [40], where the controller node oversees and coordinates the worker nodes during fuzzing. Therefore, the global field pool can be implemented on the central controller node while each worker node maintains its local field pool. The synchronization between the global and local field pools can be implemented by following Algorithm 1, which involves two steps, i.e., the Pull and Push steps, as illustrated in Figure 7. During synchronization, the interaction operations, like GETNOSYNCFIELDS and READFROMGLOBALPOOL in Algorithm 1, can be implemented

using Remote Procedure Call (RPC). The invoking results are the entries from the field pools. To support its exchange among machines via the network, we can design a protocol based on Protobuf [41] that adheres to the structure of a field pool entry. Although network communication introduces overhead, the structure of the field pool and the synchronization algorithm we propose help minimize the required data exchange.

VIII. Conclusion

In this paper, we proposed MPFuzz, a generation-based parallel fuzzing tool designed for securing IoT messaging protocols. MPFuzz employs an information synchronization mechanism to share key fields in IoT messaging protocols among different parallel instances facilitated by the global field pool during the fuzzing process. The synchronized information and characteristics of the IoT messaging protocol are used by the fuzzing instance to implement a semantic-aware mutation strategy for refining the generated test packets. Compared to the traditional parallel fuzzing strategy, MPFuzz is capable of bridging the gap between isolated fuzzing instances, allowing data packets generated by different instances to be related and thereby significantly enhancing fuzzing efficiency. Our experiments show that, compared to the state-of-the-art protocol fuzzers, MPFuzz achieves higher branch coverage, and the parallel fuzzing efficiency is also significantly improved. Furthermore, MPFuzz exposed 7 previously unknown bugs.

Acknowledgement

We thank the anonymous reviewers for their constructive feedback and suggestions. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No.92167101, 62021002).

References

[1] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial internet of things: Challenges, opportunities, and directions," *IEEE transactions on industrial informatics*, 2018.

[2] "Industrial Internet of Things and its Applications in Industry 4.0: State of The Art," *Computer Communications*, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366420319964

[3] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial internet of things," in *DAC*, 2015.

[4] K. Tange, M. De Donno, X. Fafoutis, and N. Dragoni, "A systematic survey of industrial internet of things security: Requirements and fog computing opportunities," *Communications Surveys & Tutorials*, 2020.

[5] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," *2020 ICST*.

[6] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ICS protocol," *ACM TECS*, 2019.

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

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

[9] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "PAFL: extend fuzzing optimizations of single mode to industrial parallel mode," *ESEC/FSE*, 2018.

[10] 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 *ASE*. IEEE, 2021.

[11] 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 Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2022.

[12] P. Tech., "Peach fuzzer configuration file (Peach Pit)." Website, https://peachtech.gitlab.io/peach-fuzzer-community/v3/PeachPit.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] C. Bormann and K. Hartke, "Constrained Application Protocol (CoAP)," RFC 7252. [Online]. Available: https://tools.ietf.org/html/rfc7252

[15] OASIS, "Advanced message queuing protocol (AMQP)." [Online]. Available: http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf

[16] 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 *Design Automation Conference (DAC)*, 2024.

[17] MelbourneFuzzingHub, "AFLTeam," https://github.com/MelbourneFuzzingHub/aflteam.

[18] 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," *2023 Usenix Security Symposium*.

[19] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *NDSS*, 2024.

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

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," *2012 USENIX ATC*.

[22] Clang, "Clang 15.0.0 documentation, undefinedbehaviorsanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[23] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "ICS protocol fuzzing: Coverage guided packet crack and generation," *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[24] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, "PAVFuzz: State-sensitive fuzz testing of protocols in autonomous vehicles," *DAC*, 2021.

[25] R. Natella, "StateAFL: Greybox fuzzing for stateful network servers," *ArXiv*, vol. abs/2110.06253, 2021.

[26] S. Schumilo, C. Aschermann, A. Jemmett, A. R. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," *Seventeenth European Conference on Computer Systems*, 2022.

[27] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," *2021 USENIX Annual Technical Conference*.

[28] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," *2022 Usenix Security Symposium*.

[29] 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 *International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

[30] Z. Luo, K. Liang, Y. Zhao, F. Wu, J. Yu, H. Shi, and Y. Jiang, "DynPRE: Protocol reverse engineering via dynamic inference," in *NDSS*, 2024.

[31] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference," *2021 ACM SIGSAC CCS*.

[32] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu, and R. Beyah, "MPInspector: A systematic and automatic approach for evaluating the security of IoT messaging protocols," in *USENIX Security*, 2021.

[33] B. Yuan, Z. Song, Y. Jia, Z. Lu, D. Zou, H. Jin, and L. Xing, "MQTTactic: Security analysis and verification for logic flaws in mqtt implementations," in *IEEE Symposium on Security and Privacy*, 2024.

[34] S. Li, R. Li, J. Ye, and C. Tang, "Adaptive parallel fuzzing with multi-candidate task scheduling," *Journal of Physics: Conference Series*, 2020.

[35] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *USENIX Security Symposium*, 2018.

[36] C. Song, X. Zhou, Q. Yin, X. He, H. Zhang, and K. Lu, "P-fuzz: a parallel grey-box fuzzing framework," *Applied Sciences*, 2019.

[37] X. Zhou, P. Wang, C. Liu, T. Yue, Y. Liu, C. Song, K. Lu, Q. Yin, and X. Han, "UltraFuzz: Towards resource-saving in distributed fuzzing," *IEEE Transactions on Software Engineering*, 2020.

[38] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida, and H. Bos, "CollabFuzz: A framework for collaborative fuzzing," *European Workshop on Systems Security*, 2021.

[39] Google, "OSS-Fuzz," https://github.com/google/oss-fuzz.

[40] Microsoft, "Microsoft Writing Style Guide, master/slave," https://learn.microsoft.com/en-us/style-guide/a-z-word-list-term-collections/m/master-slave.

[41] Google, "Protocol Buffers - Google's data interchange format," https://github.com/protocolbuffers/protobuf.