WINGMUZZ: Blackbox Testing of IoT Protocols via Two-dimensional Fuzzing Schedule

Xiaogang Zhu[†*], Enze Dai[‡*], Xiaotao Feng[‡], Shaohua Wang[§**], Xin Xia[¶], Sheng Wen[‡**], Kwok-Yan Lam[‖], Yang Xiang[‡]
[†]Adelaide University, Adelaide, Australia. xiaogang.zhu@adelaide.edu.au
[‡]Swinburne University of Technology, Melbourne, Australia. {edai, xfeng, swen, yxiang}@swin.edu.au
[§]Central University of Finance and Economics, Beijing, China. davidshwang@ieee.org
[¶]Zhejiang University, Hangzhou, China. xin.xia@acm.org
[‖]Nanyang Technological University, Singapore. kwokyan.lam@ntu.edu.sg

***Abstract*—The Internet of Things (IoT) is widely used in various sectors but is often prone to vulnerabilities. With the proprietary nature of IoT devices, their source code and firmware are frequently unavailable for open review, rendering blackbox fuzzing a viable approach. However, the effectiveness of blackbox fuzzing is often challenging due to the lack of feedback, especially the information of code coverage. In this paper, we propose WINGMUZZ to provide blackbox fuzzing of IoT protocols with effective feedback. The key is to guide blackbox fuzzing by utilizing runtime information from greybox fuzzing on counterpart open-source code. This is based on our observation that IoT protocols and open-source code conform to the same specifications, indicating that inputs exploring different code regions on open-source code may also discover new coverage on IoT protocols. WINGMUZZ uses a two-dimensional fuzzing schedule to optimize the process of fuzzing IoT protocols. The first dimension involves scheduling open-source implementations, referred to as *wingmates*, so that similar ones are preferred to guide blackbox fuzzing. The second dimension utilizes coverage-guided greybox fuzzing to test open-source code. This solution can bridge the performance gap between blackbox fuzzing and greybox fuzzing on IoT protocols. We evaluate the performance of WINGMUZZ across eight IoT protocols and compare it with six widely-used blackbox fuzzers. On average, WINGMUZZ can discover 42.1%, 26.92%, 25.01%, 34.95%, 23.56% and 11.63% more edges than Boofuzz, Spike, Peach, SNIPUZZ, Pulsar and ChatAFL, respectively. Additionally, WINGMUZZ exposes 10 bugs in IoT protocols while other fuzzers expose no more than 3 bugs. It also exposes 2 new protocol vulnerabilities in IoT devices while other fuzzers cannot identify any.**

***Index Terms*—Fuzzing, Blackbox Testing, IoT Protocols**

I. INTRODUCTION

Internet of Things (IoT) devices are widely used in various sectors such as energy, healthcare, mining, agriculture, and transport. Predictions indicate that by 2030, the world could see over 21 billion interconnected IoT devices, with some estimates reaching as high as 64 billion [1]. In recent years, there has been a significant increase in cyber-attacks on IoT devices, with incidents rising from 32.7 million in 2018 to 112.3 million in 2022 [2]. This surge in attacks underscores the critical security risks posed to society, exacerbated by a notable absence of practical testing techniques for IoT devices.

To test abnormal behaviors (*e.g.*, crashing) in IoT protocols, the main challenge is the absence of source code and firmware (binary code). Due to the commercial considerations, vendors of IoT devices may completely block debug interfaces, which prevents the acquisition of firmware [3]. Therefore, techniques that require firmware (such as static analysis [4] and rehosting [5]) or source code [6] cannot be applied in this situation. A practical solution is blackbox fuzzing, which generates a large number of testcases to test targets, via network [7], [8]. However, blackbox fuzzing is less powerful than greybox fuzzing because blackbox fuzzing has limited or no information about the internal execution of IoT protocols. Existing blackbox fuzzers rely on human expertise or perform mutations in a random manner due to the lack of feedback (*e.g.*, code coverage) [9]–[11]. A recent blackbox fuzzer SNIPUZZ [7] regards responses from IoT devices as coarse coverage, but the number of unique responses is too small to efficiently guide fuzzing. All of them lack an efficient feedback to guide blackbox fuzzing.

To bridge the performance gap between blackbox fuzzing and greybox fuzzing, we propose to *use runtime information from greybox fuzzing on open-source code to guide blackbox fuzzing.* An IoT protocol is often developed based on its corresponding specifications such as Request for Comments (RFC) files. Due to the complex and tedious task of developing IoT network applications, many IoT protocols use open specifications. For example, 84.1% of application layer protocols in IoT have open-source implementations, indicating that they use open specifications. Consequently, protocols commonly used in IoT network applications often have corresponding open-source code, enabling the use of runtime information from greybox fuzzing to guide blackbox fuzzing.

We propose a framework using two-dimensional fuzzing schedule for the blackbox testing of IoT protocol implementations. To use the information from open-source code, the first dimension is the schedule of open-source implementations while the second dimension is the schedule of seeds (*i.e.*, testcases that increase code coverage). The challenge is that we do not know the exact implementation in a target device. Thus, the first dimension is to infer and prefer open-source implementations that are similar to the target IoT protocol

[*] Xiaogang Zhu and Enze Dai contribute equally to the paper.
[**] Co-corresponding authors: Shaohua Wang and Sheng Wen.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00212

implementations. We call such an open-source implementation as a *wingmate* for the target IoT protocol. After we select a wingmate, the second dimension is to schedule seeds by coverage-guided greybox fuzzing (CGF). Benefiting from detailed information like magic values and control flow graphs, CGF has demonstrated its power in uncovering bugs [12]–[14]. As a result, *golden seeds, which contain runtime information, are obtained from greybox fuzzing to guide efficient blackbox fuzzing.* The golden seeds generated by CGF explore different code regions in wingmates, and it is likely that these seeds will also cover diverse code regions in IoT protocol implementation. Consequently, blackbox fuzzing is effectively guided by greybox fuzzing through the use of golden seeds. While this paper focuses on IoT protocols, the idea of using greybox fuzzing as feedback can be utilized in many blackbox scenarios, such as testing image applications in IoT.

We implemented our prototype, called WINGMUZZ, to fuzz real-world application protocols in IoT devices. We apply our WINGMUZZ on existing blackbox fuzzers to guide the seed generation of them. We evaluate WINGMUZZ on eight diverse IoT protocols, and compare it with six widely-used blackbox fuzzers, Boofuzz [11], Spike [15], Peach [10], SNIPUZZ [7], Pulsar [16] and ChatAFL [6]. It is worth noting that ChatAFL in the evaluation operates in a blackbox manner, with the specific method detailed in Section IV. The experiment results show that WINGMUZZ can discover new edge coverage and bugs faster and more than the blackbox fuzzers. Specifically, WINGMUZZ discovers 42.1%, 26.92%, 25.01%, 34.95%, 23.56% and 11.63% more edges than Boofuzz, Spike, Peach, SNIPUZZ, Pulsar and ChatAFL, respectively. On the protocol `Tinydtls`, WINGMUZZ even discovers 225.6%, 128.09%, 124.32%, 158.49%, 91.76% and 50.26% more edges than Boofuzz, Spike, Peach, Snipuzz, Pulsar and ChatAFL, respectively. WINGMUZZ also exposes 10 bugs in protocols while the vanilla blackbox fuzzers expose no more than 3 bugs. Moreover, WINGMUZZ identifies 2 new bugs in protocols in real-world devices BMW NBT EVO and GL.iNet Smart Router, respectively. However, other fuzzers cannot expose any bugs in the devices. The contributions are as follows.

- We propose to guide blackbox fuzzing via the runtime information obtained from greybox fuzzing. We develop a two-dimensional fuzzing schedule, including schedule of wingmates and schedule of seeds, to efficiently perform blackbox testing of IoT protocols. With the guidance from greybox fuzzing, the blackbox fuzzing of IoT protocols can obtain the power of greybox fuzzing.
- We implement the framework WINGMUZZ[2] to test protocols. We apply our WINGMUZZ on existing blackbox fuzzers, and the evaluation demonstrates the effectiveness and efficiency of it in coverage and bug discovery.
- WINGMUZZ exposes 8 known and 2 unknown bugs in IoT protocols, and two new vulnerabilities in real-world IoT devices, which are confirmed by vendors.

[2]https://anonymous.4open.science/r/wingmuzz-FEED

TABLE I
DIFFERENT TYPES OF PROTOCOL IMPLEMENTATIONS

Types	Protocol Examples
Single-Maintainer Protocols	NTP, AFP, RDP
Multiple Well-Known Implementations	DNS, SSH, RTSP, DICOM
Diverse Custom Implementations	FTP, Modbus, SIP, BGP, SNMP, SMTP

II. MOTIVATION

IoT network protocols define rules and standards for data exchange in computer networks, with functional specifications such as RFC files. These documents provide detailed formats for request messages. For example, the RFC files define formats including IP address, data fields, and timestamps. Although different implementations of the same protocol may vary in performance or interaction logic, their core functionality remains similar. For example, DNS maps domain names to IP addresses, with implementations like Bind9 and DNSMASQ adhering to the same RFC standards despite differences in code. *This insight motivates us to use the runtime information from testing open-source implementation to guide the testing of blackbox implementation.*

Existence of Open-Source Code. We aggregate protocols from three papers [17]–[19] and Wikipedia[3], take their union, and deduplicate them, resulting in 90 protocols. Our scope is restricted to IoT protocols in the application, transport, and network layers. We then exclude 14 obsolete protocols and 7 blockchain-related protocols, retaining 69 in total: 63 application layer, 1 transport layer, and 5 network layer protocols. The obsolete protocols typically lack commercial incentives or community support, leading to complete abandonment. For each of the 69 protocols, we search for open-source implementations on GitHub, GitLab, SourceForge, and the respective official protocol websites using protocol names as keywords. *Our results show that 59 out of 69 protocols (85.5%) have open-source implementations*, while the remaining are proprietary protocols maintained by major vendors (*e.g.*, Google, Apple). Notably, all proprietary protocols fall within the application layer, indicating that 53 out of 63 application layer protocols (84.1%) have open-source implementations. To facilitate reproducibility, we provide our results in the code repository as *open_source_existence.md*, which lists each protocol and its available open-source software.

We also analyze that, in certain cases, even when IoT protocols have publicly available implementations, it remains challenging to identify appropriate or sufficiently similar wingmates. Specifically, even though some protocols are publicly defined, they are maintained by commercial entities that impose restrictive access models, like Zigbee. Thus, we cannot obtain the full implementations. For evolving protocols with large differences, such as HTTP/1.1 and HTTP/2, they coexist in the same implementations. When we test a specific version, we may only be able to use a subset of the open-source software. For multi-purpose applications, they employ multiple

[3]https://en.wikipedia.org/wiki/Application_layer

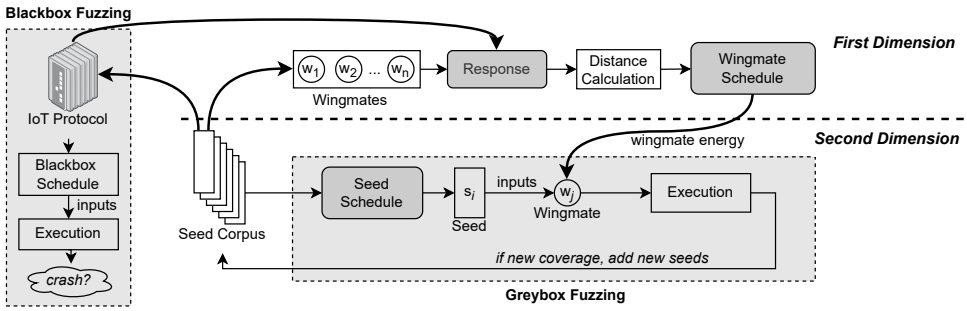

Fig. 1. The two-dimensional framework of WINGMUZZ. It first schedules open-source implementations (wingmates), which links the running of blackbox fuzzing and greybox fuzzing. The second dimension is to schedule seeds in a greybox manner.

protocols, accepting inputs through diverse channels, making it difficult to identify the channel to be tested. For example, RabbitMQ integrates both AMQP and STOMP protocols as its communication interface. However, among the 69 protocols, such cases account for only 4 (5.8%).

IoT Protocol Implementations. We classify open-source implementations into three categories, as shown in Table I. The first type is the single-maintainer protocols (*e.g.*, NTP). These protocols are maintained by a specific organization or official publisher, with few or no alternative implementations in the market. Because different versions exist, identifying the exact version used in IoT devices can be challenging. The second type has multiple well-known implementations (*e.g.*, DNS). This category includes protocols with several widely used implementations, such as dnsmasq, nsd, and bind9 for DNS, alongside many smaller implementations. While all adhere to the same specifications, they may have variations in execution. The third type has diverse custom implementations (*e.g.*, FTP). Many implementations exist for these protocols, with manufacturers often creating their own based on their specific needs. Due to firmware restrictions, identifying the exact protocol implementation within an IoT device is often impossible. *Therefore, the presence of multiple versions and implementations makes blackbox fuzzing essential, as the exact code used in IoT devices is typically unknown.*

III. DESIGN OF WINGMUZZ

Due to the blackbox setting, existing fuzzers cannot get the internal execution states, leading to ineffective feedback or even no feedback. Our key idea is to guide blackbox fuzzing of IoT protocols based on the greybox fuzzing of open-source code. This allows us to obtain coarse or even accurate internal execution states, and significantly solves the challenge of blackbox fuzzing in IoT. To achieve this, we propose a framework with two-dimensional fuzzing schedule. As shown in Figure 1, the first dimension schedules open-source implementations (wingmates) based on the response distance, which indicates how similar two protocol implementations are. At a high-level, WINGMUZZ prefers wingmates that have larger similarity between a selected wingmate and the target IoT protocol. As for the second dimension, WINGMUZZ applies CGF on a selected wingmate. In this dimension, WINGMUZZ utilizes the power of CGF to guide the coverage-guided input generation. Golden seeds generated in the second dimension will be used for blackbox fuzzing of IoT protocols. Therefore, the greybox fuzzing achieves guiding blackbox fuzzing by using those golden seeds. With the two-dimensional schedule, blackbox fuzzing can obtain the power of greybox fuzzing, mitigating the gap between blackbox and greybox fuzzing.

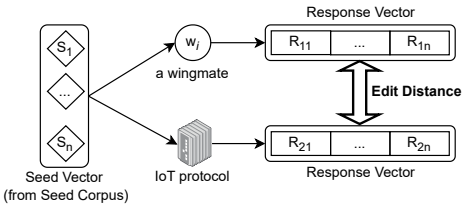

Fig. 2. Response distance, which is the edit distance of all responses.

A. First Dimension: Wingmate Schedule

As shown in Figure 1, the first dimension links blackbox and greybox fuzzing, which is the key to bridge the performance gap between blackbox and greybox fuzzing. To guide blackbox fuzzing, the best performance is achieved when using an implementation that exactly matches the target protocol. However, we do not have access to the code of IoT protocol; thus, we use similarity, quantified by response distance, to prefer wingmates. Due to the absence of sufficient responses to precisely calculate similarity between implementations, the one deemed most similar might significantly differ from the target IoT protocol. Therefore, we employ multiple wingmates to mitigate this problem. As a result, WINGMUZZ prefers wingmates that are more similar to the target IoT protocol. To schedule wingmates, we calculate response distance to assign energy to wingmates. The energy of a wingmate is the allocated time that fuzzing tests the wingmate.

1) Response Distance for Wingmates: The distance of responses that originate from the same input can reflect the similarity between a wingmate and the target IoT protocol. Therefore, we calculate the distance by comparing the response vectors. We regard each response as a string, and a response vector concatenates multiple responses together to form a longer response (string). As shown in Figure 2, we use the seeds to get the response vectors from wingmates and the target protocol. We then calculate the response distance based

on the edit distance of response vectors. Edit distance is used to quantify the dissimilarity between two strings by counting the minimum number of operations needed to transform one string into the other. The distance is defined as

$$\mathcal{D} = edit\_dist(\overrightarrow{R_1}, \overrightarrow{R_2}), \tag{1}$$

where $\overrightarrow{R_1}$ is the response vector for a wingmate, and $\overrightarrow{R_2}$ is the response vector for the target protocol. The value of $edit\_dist(\bullet)$ is the edit distance calculated by comparing the two vectors. We use Levenshtein distance in WINGMUZZ. Note that, the seeds utilized to calculate response distance are from the shared seed pool so that wingmates have the same size of response vectors.

2) Energy Schedule for Wingmates: The energy schedule assigns the duration time that fuzzing tests a wingmate. As aforementioned, WINGMUZZ prefers wingmates that are similar to the target protocol. That is, WINGMUZZ assigns more energy to wingmates that have smaller response distance. To achieve this, the energy assigned for a wingmate is

$$\begin{aligned} E^w_{cur} &= E_{base} \times \frac{\mathcal{S}_{cur}}{\sum \mathcal{S}} \\ \mathcal{S} &= \frac{1}{\max(\mathcal{D}, 1)}, \end{aligned} \tag{2}$$

where E^w_{cur} is the energy assigned to the current wingmate, E_{base} is the total energy for all wingmates, which is predetermined for the current fuzzing round. $\mathcal{S}$ is the similarity for a wingmate, and $\mathcal{S}_{cur}$ is the similarity for the current wingmate. $\mathcal{D}$ is the response distance calculated in Equation 1. $\max(\mathcal{D}, 1)$ returns the maximum value between $\mathcal{D}$ and 1. If two responses are different, $\mathcal{D} \geq 1$; otherwise, if two responses are the same, $\mathcal{D} = 0$. Thus, according to the equation, if two responses are different, $\mathcal{S} \leq 1$; otherwise, if two responses are the same, $\mathcal{S} = 1$. Therefore, wingmates with smaller distance are assigned with larger energy.

3) Update of Response Distance: During fuzzing, new seeds will be discovered and the response distances of wingmates will change. Therefore, we need to update their response distances so as to adjust the energy assignment for different wingmates. To improve the efficiency of fuzzing, the response distance is updated when all wingmates have been fuzzed in the same fuzzing round. A fuzzing round is the time period for all wingmates to complete their testing. That is, before WINGMUZZ assigns energy to all wingmates, it will update response distances for all wingmates. For example, if there are 10 fuzzing rounds, WINGMUZZ will update the response distance 10 times. In each round, energy for all wingmate is assigned at the beginning of the round.

B. Second Dimension: Seed Schedule

In the two-dimensional fuzzing schedule, the second dimension is the schedule of seeds for a selected wingmate, which is basically a coverage-guided fuzzer. CGF generates new testcases by mutating seeds, and if a generated testcase discovers new code regions, the testcase is retained as a new seed. Our WINGMUZZ follows this process in the second dimension. All seeds retained in the seed corpus will be used to test IoT protocol in later fuzzing trials.

1) Shared Seed Pool: Although different wingmates may differ from their functionalities, they are still developed for the same protocol. Thus, different wingmates are likely to share seeds. WINGMUZZ uses the *shared seed pool* (the Seed Corpus in Figure 1), a union set of seeds from all wingmates, to collect all seeds. In later fuzzing trials, when a wingmate is selected, WINGMUZZ uses the shared seed pool to update seeds for the wingmate. This improves efficiency because fuzzing does not need to discover the updated seeds again. Additionally, the shared seed pool is used in both the calculation of response distance and energy assignment.

2) Energy Schedule for Seeds: The energy assigned to a seed is influenced by its frequency. The *frequency* of a seed is the number of times the seed is mutated to test targets. Inspired by AFLFast [20], WINGMUZZ balances the frequencies of seeds by assigning more energy to seeds that are less frequently tested. Therefore, the energy assigned to a seed is calculated as

$$E^s_{cur} = E^s_{ave} \times \frac{f_{min}}{f_{cur}}, \tag{3}$$

where f_{min} is the minimum frequency of all seeds while f_{cur} is the frequency of current seed. E^s_{ave} is the average energy assigned to a seed, calculated as $E^s_{ave} = E^v_{cur}/n$, where n is the number of seeds for the current wingmate. The seed schedule of WINGMUZZ will result that most examined code regions are tested at roughly average frequency, *i.e.*, their assigned energy is close. WINGMUZZ will select and test all seeds until the wingmate energy is exhausted.

3) Prioritization of Seeds: In a fuzzing round, all seeds are supposed to be selected and fuzzed. However, due to the limited energy for a wingmate, some seeds may not be selected in a fuzzing round. Therefore, WINGMUZZ prioritizes to select newly added seeds. During fuzzing, new seeds will be discovered by testing the current wingmate. To test the potential of the newly added seeds, WINGMUZZ will prioritize to test them in later fuzzing rounds. WINGMUZZ maintains a queue of seeds and new seeds are added to the head of the queue. When a new round of fuzzing starts, the seeds at the head of the queue will be fuzzed first. This can also improve the efficiency of fuzzing.

C. Blackbox Fuzzing of IoT Protocols

The goal of the two-dimensional fuzzing schedule is to test the target IoT protocol by generating inputs based on greybox fuzzing. Therefore, the blackbox fuzzing of IoT protocols obtains the power of CGF. As a result, WINGMUZZ can roughly explore code coverage of IoT protocols efficiently. As shown in Figure 1, to guide blackbox fuzzing, the seeds obtained during greybox fuzzing are utilized to test IoT protocol. New inputs are generated by mutating the seeds in the seed corpus.

Our WINGMUZZ can perform as a plugin for existing blackbox fuzzers, improving their efficiency and using their blackbox mutations. Many blackbox fuzzers for protocols

TABLE II
PROTOCOLS FOR EVALUATING FUZZERS IN THE SIMULATED SETTING.

Target	Protocol	Wingmates
Opensips (d156371)	SIP	Opensips v3.0.5*, v3.2.3, v3.3.6, v3.4.1 Kamailio v5.8.3, v5.7.0, v5.5.0, v5.2.7
Proftpd (5fcaf10)	FTP	Bftpd v6.2*; Lightftp v2.2; vsftpd v3.0.5; pureftpd v1.0.50 Proftpd v1.3.5e, v1.3.6a, v1.3.7, v1.3.8
OpenSSH (8aa3455)	SSH	Dropbear v2015.68*, v2017.75, v2019.78, v2022.82 OpenSSH 7.9p1, 8.5p1, 8.8p1, 9.5p1
DNSmasq (0fa7e62)	DNS	Unbound v1.16.2*; knot v3.1.2; nsd v4.6.0rc1; bind v9.16.6 DNSmasq 2.79, 2.83, 2.88, 2.90
Ntp-official (v4.2.8p10)	NTP	Ntp-official v4.2.5*, v4.2.7, v4.2.8p13, v4.3.95 NTPsec v1.2.3, v1.1.7, v1.1.1, v0.9.8
Live555 (ceeb4f4)	RTSP	Mirco-rtsp*; gst-rtsp-server v1.16.1, v1.14.1, v1.10.1 Live555 v1.11, v1.08, v1.00, v0.95
Dcmtk (a137f1a)	DICOM	Dcmtk v3.6.8*, v3.6.6, v3.6.5, v3.6.5+_20191213 GDCM v2.6.5, v2.8.6, v3.0.1, v3.0.15
Tinydtls (dad6344)	DTLS12	mbedtls 92152dc*; Tinydtls 8c55636; gnutls v3.6.8, v3.6.10 Tinydtls v0.9-rc1, v0.8.2, v0.8.1; mbedtls 3.3.0

* Set 1, includes only one implementation. xxx Set 2, includes 4 implementations. Set 3, includes all the 8 implementations.

mutate seeds based on data fields. Typically, protocol programs have strict requirements concerning input formats. Random mutations of characters or bytes within these inputs could lead most inputs to be halted by the protocol program's syntactical verification mechanisms. However, by using the data field as the foundational test unit, each mutation test can maintain the integrity of other data fields, thereby effectively testing the code regions associated with the chosen data field.

IV. EVALUATION

We have implemented WINGMUZZ with over two thousand lines of Python and C/C++ code. It incorporates the wingmate schedule for the first dimension, the greybox fuzzing for the second dimension, and the blackbox fuzzing for target protocols. For the second dimension, our WINGMUZZ is developed based on the greybox fuzzer AFLNet [21] and modifies its component of seed schedule including seed prioritization and energy assignment. To gather run-time information (*e.g.*, coverage information) for wingmates, we utilize AFL's Clang-based instrumentation approach.

Representativeness of IoT Protocols. The essence of WINGMUZZ is to assist in better blackbox testing of target applications by fuzzing other open-source applications that are the same or similar in a greybox setting, thereby generating high-quality seeds. However, due to the wide variety of IoT protocols in the market and their different ecosystems, it is challenging to practically evaluate fuzzing tools. Therefore, in our experiments, *we use three representative types of IoT protocols and their distinctly different software ecosystems to demonstrate the effectiveness of* WINGMUZZ*, which are described in Section II.* The first type includes NTP that only has few alternative implementations. The second type includes DNS that has multiple well-known implementations. The third type includes FTP that has diverse custom implementations.

Benchmark Fuzzers. We select six representative blackbox fuzzers, Boofuzz [11], Spike [15], Peach [10], SNIPUZZ [7], Pulsar [16] and ChatAFL [6], as the benchmark fuzzers for testing IoT protocols. Installing as a Python library, *Boofuzz* enables users to create fuzzing scripts by manually initializing seeds of protocol messages. *Spike* is a C-based fuzzer creation kit that includes scripting capabilities through simple text files (*i.e.*, *.spk* files) containing Spike primitives. Writing these files is crucial for protocol fuzzing and requires knowledge of network protocol formats. Like Spike, *Peach* requires the creation of Pit files in XML syntax to define the data's structure, type, and relationships for fuzzing. It allows security researchers to develop customized strategies. SNIPUZZ, a state-of-the-art automatic blackbox fuzzing approach, runs as a client communicating with the network protocol services and infers message snippets for mutation based on the responses. Each snippet reflects an approximate data field. *Pulsar* is a stateful blackbox fuzzing approach equipped with automatic protocol learning and simulation capabilities. It models protocol message formats and states from observed traffic, such as *.pcap* files, and effectively explores the state space to uncover deep vulnerabilities. ChatAFL is an LLM-assisted fuzzer that extracts machine-readable protocol information from human-readable specifications. ChatAFL is a greybox fuzzer, but for fair comparison, we apply its LLM-assisted strategies in a blackbox manner, called *ChatAFL*$_{bm}$. Specifically, we run AFLNet in dumb mode while retaining two key strategies in ChatAFL, which are grammar-guided mutation and enrichment of initial seeds.

We equip the benchmark fuzzers with our WINGMUZZ to get the corresponding fuzzers *Boofuzz*-WINGMUZZ, *Spike*-WINGMUZZ, *Peach*-WINGMUZZ, SNIPUZZ-WINGMUZZ, *Pulsar*-WINGMUZZ and *ChatAFL*$_{bm}$-WINGMUZZ. For a comprehensive evaluation, we carefully select fuzzers for comparison based on their availability and relevance. Labrador [22] is excluded from the comparison due to the unavailability of its code. Moreover, its concept of response-guided fuzzing is already incorporated in SNIPUZZ, which is included in our evaluation. LLMIF [23] is excluded as its available implementation is limited to Zigbee. As previously mentioned, ChatAFL is originally a greybox fuzzer; thus, we adapt it for blackbox testing to ensure a fair comparison with the up-to-date fuzzers leveraging LLM techniques.

The evaluation answers the following research questions:

- **RQ1**: Can WINGMUZZ efficiently discover new coverage and bugs?
- **RQ2**: What is the performance of each dimension and different configurations?
- **RQ3**: Can WINGMUZZ significantly optimize initial seed corpus for blackbox fuzzing?
- **RQ4**: Can WINGMUZZ effectively discover new vulnerabilities in protocols in IoT devices?

A. RQ1: Performance of WINGMUZZ in Simulated Setting

To evaluate the effectiveness of WINGMUZZ versus the benchmark fuzzers, we measure the extent to which protocol fuzzers cover code regions of protocol implementations. Code coverage is a basic metric to evaluate fuzzers, and the ultimate

TABLE III
DISCOVERY OF EDGE COVERAGE. WINGMUZZ PERFORMS BETTER THAN ITS VANILLA FUZZERS ON ALL PROTOCOLS.

Targets	# of Edges											
	Boofuzz	**Boofuzz -WINGMUZZ**	**Spike**	**Spike -WINGMUZZ**	**Peach**	**Peach -WINGMUZZ**	**SNIPUZZ**	**SNIPUZZ -WINGMUZZ**	**Pulsar**	**Pulsar -WINGMUZZ**	**C.A.$_{bm}$**[1]	**C.A.$_{bm}$ -WINGMUZZ**
Opensips	4,598	4,928 (7.17%↑)	4,800	5,117 (6.60%↑)	4,566	5,011 (9.75%↑)	4,881	5,223 (7.01%↑)	3,941	4,689 (18.98%↑)	5,103	5,346 (4.76%↑)
Proftpd	5,008	5,644 (12.69%↑)	4,689	4,901 (4.52%↑)	5,267	5,796 (10.04%↑)	N/A[2]	N/A	5,177	5,669 (9.50%↑)	5,465	5,799 (6.11%↑)
OpenSSH	2,471	2,477 (0.24%↑)	2,226	2,392 (7.46%↑)	2,301	2,377 (3.30%↑)	2,515	2,673 (6.28%↑)	2,113	2,357 (11.55%↑)	2,731	2,747 (0.59%↑)
DNSmasq	1,514	1,625 (7.33%↑)	1,452	1,601 (10.26%↑)	1,396	1,481 (6.09%↑)	1,589	1,754 (10.38%↑)	1,526	1,605 (5.18%↑)	1,676	1,735 (3.52%↑)
Ntp-official	1,802	1,851 (2.72%↑)	1,484	1,569 (5.73%↑)	1,833	1,861 (1.53%↑)	1,816	1,864 (2.64%↑)	1,683	1,791 (6.42%↑)	1,914	1,982 (3.55%↑)
Live555	779	1,017 (30.55%↑)	856	1,071 (25.12%↑)	812	943 (16.13%↑)	907	1,133 (24.92%↑)	802	1,006 (25.44%↑)	1,252	1,378 (10.06%↑)
Dcmtk	1,764	2,654 (50.45%↑)	2,091	2,667 (27.55%↑)	1,872	2,413 (28.90%↑)	N/A	N/A	1,834	2,195 (19.68%↑)	2,439	2,784 (14.15%↑)
Tinydtls	82	267 (225.61%↑)	89	203 (128.09%↑)	74	166 (124.32%↑)	106	274 (158.49%↑)	85	163 (91.76%↑)	195	293 (50.26%↑)
AVG	-	**42.10%↑**	-	**26.92%↑**	-	**25.01%↑**	-	**34.95%↑**	-	**23.56%↑**	-	**11.63%↑**
TRM[3]	-	**18.49%↑**	-	**13.79%↑**	-	**12.37%↑**	-	**12.15%↑**	-	**15.26%↑**	-	**7.03%↑**

[1] C.A.$_{bm}$: ChatAFL$_{bm}$, ChatAFL in a blackbox manner. [2] N/A: Not Applicable. [3] TRM: TRimmed Mean, which removes the maximum and minimum values before calculation.

goal of WINGMUZZ is to reveal more bugs in a shorter amount of time. Thus, we also analyze the bug detection capability of WINGMUZZ. Because we do not have access to firmware of IoT devices, we cannot get the coverage information of IoT protocols. To evaluate the performance of coverage and bug discovery, we install an open-source protocol implementation, which is randomly selected, on general operating systems and regard it as the target IoT protocol. This setting is feasible because many protocols are shared between general operating systems and IoT embedded systems. In this way, we can evaluate the performance of WINGMUZZ in terms of coverage discovery, as well as bug discovery.

1) Experiment Settings: The experiment is conducted on two different computers capable of communicating with each other over the same local area network (LAN). Two machines are required because running protocols will occupy ports, preventing a machine from running the same protocol with different implementations. By installing virtual machines on the two computers, protocols and fuzzers will run in virtual machines equipped with Ubuntu 20.04. The machine, hosting macOS Sonoma with an Intel Core i5 @1.4GHz processor, will perform blackbox fuzzing on IoT protocols in the virtual machine. The other machine, hosting Windows 11 with an Intel Core i7 @2.60GHz processor, will perform fuzzing on wingmates in the virtual machine. Information exchange required by WINGMUZZ, such as updated new seeds and responses, will be communicated via LAN between the two machines. Each fuzzer uses one CPU core to run one target program for 10 hours, and each trial is repeated 5 times. We use the average value to demonstrate the experiment results. To ensure a fair comparison, fuzzers use the same initial seeds, all of which are sourced from either AFLNet or ProFuzzBench [24]. Besides, WINGMUZZ only runs either the main process or the wingmate at a time. When a wingmate is running, the main process is waiting.

With the same initial seeds, for Boofuzz, we use its Python API to configure initialization and mutation strategies (*e.g.*, s_initialize, s_random). SPIKE and Peach require grammar files, and we generate the files based on the initial seeds. SPIKE uses *.spk* files (plain text) to define message formats and initial values, while Peach uses *.pit* files (XML) to define initial values, data models, and state models. Pulsar learns format and protocol state from network traffic, which we obtain from public GitHub repositories or manually capture as *.pcap* files. SNIPUZZ and ChatAFL do not require special configurations. The configurations are the same for both baseline fuzzers and their corresponding WINGMUZZ-quipped fuzzers.

Target Protocols and Wingmates. As detailed in Table II, our selected target protocols and their implementations are widely-used and carefully chosen, encompassing both practical application protocols such as FTP, NTP, RTSP, and DICOM, as well as protocols for secure and efficient transmission like SIP, SSH, DNS, and DTLS12. These protocols are representative of IoT protocols and belong to different types mentioned before. Protocols NTP, SIP and SSH belong to the first type. Protocols DNS and DTLS12 belong to the second type. Protocols FTP, RTSP and DICOM belong to the third type. As for the target implementation utilized in blackbox fuzzing, we randomly select it to show the effectiveness of WINGMUZZ. To select *Wingmates*, we use three strategies: (i) different versions of the same implementation, like `Opensips` for the SIP protocol; (ii) completely different implementations, exemplified by four diverse FTP protocol implementations; (iii) a combination of the first two strategies, as seen with `Tinydtls` for the DTLS12 protocol. In Table II, the complexity of protocols vary, with the lines of code ranging from 22K to 1249K.

Evaluation Metrics. We measure the coverage based on the number of edges discovered (*# of Edges* in Table III). To quantify the improvement of WINGMUZZ over the baselines, we report the percentage increase in terms of edge coverage in 10 hours. WINGMUZZ's bug detection capability is evaluated in terms of effectiveness and efficiency, where effectiveness is measured by unique bugs or crashes, while efficiency is measured by time to the first crash that witnesses a bug (*TTB* in Table IV). When analyzing unique bugs, we utilize the top-three rule to deduplicate bugs. If the top three backtrace entries are identical, we classify them as the same bug; otherwise, they are considered distinct bugs.

2) Coverage Analysis: As shown in Table III, WINGMUZZ achieves the most significant improvement on `Tinydtls`, with an increase of up to 225.61% compared to Boofuzz, likely benefiting from having a wingmate with high similarity to the test target. Conversely, on `OpenSSH`, the increase in edge coverage is minimal, ranging from just 0.24% to 7.46%. We attribute this modest improvement to the quality of the wingmates. For instance, `Dropbear`, a wingmate for

TABLE IV
THE TIME TO BUG (TTB) OF FUZZERS. WINGMUZZ HAS THE BEST BUG DETECTION CAPABILITY.

Bugs	Boofuzz		Spike		Peach		SNIPUZZ		Pulsar		ChatAFL$_{bm}$		Type	Protocol
	-Ori.[1]	-W.M.[2]	-Ori.	-W.M.	-Ori.	-W.M.	-Ori.	-W.M.	-Ori.	-W.M.	-Ori.	-W.M.		
CVE-2017-13704	2h37m	2h5m	n/a[3]	n/a	n/a	n/a	2h21m	1h51m	n/a	2h14m	2h26m	**1h39m**	IO[4]	DNS
CVE-2018-7183	n/a	n/a	n/a	n/a	n/a	**2h8m**	n/a	n/a	n/a	n/a	n/a	n/a	BO[5]	NTP
CVE-2019-8936	**53m**	1h2m	n/a	n/a	1h36m	1h5m	n/a	n/a	n/a	n/a	n/a	n/a	NP[6]	NTP
CVE-2023-28097	n/a	n/a	2h35m	**1h29m**	n/a	n/a	2h13m	1h44m	2h9m	2h20m	n/a	n/a	SF[7]	SIP
CVE-2023-51713	n/a	n/a	n/a	n/a	n/a	**3h57m**	n/a	n/a	n/a	n/a	n/a	4h33m	OR[8]	FTP
CVE-2024-34508	n/a	**3h42m**	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	SF	DICOM
CVE-2024-34509	n/a	**1h13m**	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	SF	DICOM
Unknown	n/a	2h34m	n/a	n/a	n/a	n/a	3h29m	2h17m	n/a	n/a	2h35m	**1h54m**	SF	DTLS12
CVE-2019-9215	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	3h53m	**2h46m**	SF	RTSP
Unknown	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	**3h11m**	BO	RTSP
# Total	**2**	**+3**	**1**	**+0**	**1**	**+2**	**3**	**+0**	**1**	**+1**	**3**	**+2**	-	-

[1] -Ori.: The original blackbox fuzzer. [2] W.M.: WINGMUZZ. [3] n/a: The vulnerability was not found within 10 hours. [4] IO: Integer Overflow.
[5] BO: Buffer Overflow. [6] NP: NULL Pointer Dereference. [7] SF: Segmentation Fault. [8] OR: Out-of-bounds Read. Note: The targets are detailed in Table II.

`OpenSSH`, is a lightweight SSH implementation tailored for embedded systems, focusing on basic features and omitting some lesser-used functionalities, which limits its ability to aid `OpenSSH` in exploring more code regions. These observations suggest that selecting high-quality wingmates is crucial for achieving significant improvements. However, even with less ideal wingmates, WINGMUZZ's performance can still meet or exceed that of benchmark fuzzers.

We employ three strategies for selecting wingmates as aforementioned. `Opensips`, `Dcmtk`, and `Ntp-official` use different versions as their wingmates and all see coverage improvements ranging from 1.53% to 50.45%. `Ntp-official` shows only minor progress, likely because it is a simple protocol with limited potential for increased coverage. `Proftpd`, `OpenSSH`, `DNSmasq`, and `Live555` use completely different implementations as wingmates, achieving increases from 0.24% to 30.55%. The last strategy, a mix of the first two, is exclusively used by `Tinydtls` which experiences the highest gains, from 50.26% to 225.61%.

As shown in Table III, on average, WINGMUZZ achieves edge coverage improvements of 42.10%, 26.92%, 25.01%, 34.95%, 23.56% and 11.63% over Boofuzz, Spike, Peach, SNIPUZZ, Pulsar and ChatAFL$_{bm}$, respectively. Due to the significant variation in coverage improvements, we also present a trimmed mean (*TRM* in Table III) to better represent the central tendency of the data. TRM removes the maximum and minimum values before calculating the mean value. According to the TRM, WINGMUZZ shows edge coverage enhancements of 18.49%, 13.79%, 12.37%, 12.15%, 15.26% and 7.03% compared to Boofuzz, Spike, Peach, SNIPUZZ, Pulsar and ChatAFL$_{bm}$, respectively. The evaluation results show the effectiveness and efficiency of WINGMUZZ in coverage discovery of IoT protocols.

3) Bug Analysis: To assess bug detection capabilities, we use two metrics: Time to Bug (TTB) and the number of unique exposed vulnerabilities. TTB measures the efficiency of bug exposure, while the latter evaluates the effectiveness of bug discovery. We manually analyze each vulnerability using GDB to eliminate duplicates stemming from the same root cause. Table IV reveals 10 unique vulnerabilities, four of which are exclusively detected by WINGMUZZ across four real-world open-source protocol implementations. For Table IV, the target implementations and their versions are detailed in Table II. Among the 10 vulnerabilities, WINGMUZZ achieves the best TTB performance in 9 cases. For `CVE-2019-8936`, Boofuzz discovers it faster than WINGMUZZ but only 9 minutes faster. This can be attributed to WINGMUZZ's communication mechanism; it communicates with wingmates at one-hour intervals to allow them to provide high-quality seeds to the black-box side. As a result, WINGMUZZ's vulnerability exposure efficiency is not enhanced within the first hour. However, `CVE-2019-8936` is discovered in approximately one hour. After one hour, WINGMUZZ generally surpasses benchmark fuzzers in exposure efficiency. For `CVE-2017-13704` and `CVE-2023-28097`, WINGMUZZ has reduced the TTB by 29.8% and 31.0% respectively, compared to the best benchmark fuzzers. The unknown vulnerability in `Tinydtls` can be discovered by WINGMUZZ, SNIPUZZ and ChatAFL$_{bm}$, but WINGMUZZ discovers it 35.9% faster than the best benchmark fuzzer (ChatAFL$_{bm}$). Our analysis also shows that, for the discovered vulnerabilities in Table IV, 4 of 10 vulnerabilities are related to vulnerable wingmates while the rest 6 are not. This indicates that our WINGMUZZ's ability to discover bugs does not rely on vulnerable wingmates.

To our pleasant surprise, when we analyze the root cause of the vulnerabilities, the two segmentation fault vulnerabilities in `Dcmtk` that WINGMUZZ identifies are assigned CVE numbers on May 5th, just a few days before our confirmation. This demonstrates that WINGMUZZ can effectively identify timely vulnerabilities in IoT protocols. Additionally, the segmentation fault that WINGMUZZ discovers in `Tinydtls` and the buffer overflow in `Live555` do not correspond to any known CVE, indicating that they may be newly discovered bugs.

4) Extrapolation of Experiment: Since a fuzzer can run almost forever, the trade-off between coverage and resource utilization is advisable. We apply the extrapolation methodology proposed in [25] to predict the coverage performance of WINGMUZZ on longer time scales. Assume that the maximum number of edges currently covered by a program is E_{max}, and the threshold coverage rate U_c is defined as $0.01 \times E_{max}$ per hour. That is to say, we consider that if the coverage can improve by 1% within an hour, the fuzzing campaign is still

TABLE V
EXTRAPOLATION OF 24 HOURS COVERAGE BASED ON 10 HOURS DATA.

Targets	# of Edges (10 Hours/24 Hours)[1]						
	Boofuzz -WINGMUZZ	**Spike -WINGMUZZ**	**Peach -WINGMUZZ**	**SNIPUZZ -WINGMUZZ**	**Pulsar -WINGMUZZ**	**ChatAFL$_{bm}$ -WINGMUZZ**	**ChatAFL$_{bm}$**
Opensips	4,928 / 5,184	5,117 / 5,303	5,011 / 5,159	5,223 / 5,528	4,689 / 4,914	5,346 / 5,939	5,103 / 5,139
Proftpd	5,644 / 5,919	4,901 / 4,970	5,796 / 5,876	N/A	5,669 / 5,943	5,799 / 6,118	5,465 / 5,480
OpenSSH	2,477 / 2,552	2,392 / 2,480	2,377 / 2473	2,673 / 2,746	2,357 / 2,421	2,747 / 2,897	2,731 / 2,750
DNSmasq	1,625 / 1,674	1,601 / 1,713	1,481 / 1,523	1,754 / 1,897	1,605 / 1,717	1,735 / 1,901	1,676 / 1,690
Ntp-official	1,851 / 1,981	1,569 / 1,686	1,861 / 1,914	1,864 / 1,970	1,791 / 1,878	1,982 / 2,115	1,914 / 1,925
Live555	1,017 / 1,088	1,071 / 1,099	943 / 956	1,133 / 1,197	1,006 / 1,090	1,378 / 1,560	1,252 / 1,267
Dcmtk	2,654 / 2,866	2,667 / 2,776	2,413 / 2,581	N/A	2,195 / 2,225	2,784 / 3,124	2,439 / 2,443
Tinydtls	267 / 273	203 / 204	166 / 171	274 / 290	163 / 170	293 / 333	195 / 197
Avg CR[2]	**0.37** U_c	**0.26** U_c	**0.21** U_c	**0.42** U_c	**0.35** U_c	**0.73** U_c	**0.05** U_c

[1] The data X/Y: X means edges covered in 10 hours, while Y means the extrapolation of edges covered in 24 hours.
[2] Avg CR: The average coverage rate. A higher coverage rate indicates greater improvement in coverage per unit of time.

TABLE VI
CONTRIBUTION OF EACH DIMENSION IN THE ABLATION STUDY.

Targets	# of Edges			
	Boofuzz[1]	**WINGMUZZ -SeedSchd**	**WINGMUZZ -MateSchd**	**WINGMUZZ**
Opensips	4,598	4,716 (4.30%↓)[2]	4,837 (1.85%↓)	4,928
Proftpd	5,008	5,199 (7.88%↓)	5,479 (2.92%↓)	5,644
OpenSSH	2,471	2,475 (0.08%↓)	2,472 (0.20%↓)	2,477
DNSmasq	1,514	1,567 (3.57%↓)	1,601 (1.48%↓)	1,625
Ntp-official	1,802	1,832 (1.03%↓)	1,829 (1.19%↓)	1,851
Live555	779	901 (11.41%↓)	962 (5.41%↓)	1,017
Dcmtk	1,764	2,431 (8.40%↓)	2,403 (9.46%↓)	2,654
Tinydtls	82	118 (55.81%↓)	186 (30.34%↓)	267
# of Bugs	**2**	**2**	**3**	**5**

[1] WINGMUZZ here is developed based on Boofuzz.
[2] The percentage decrease is compared with WINGMUZZ.

TABLE VII
EVALUATION OF DIFFERENT WINGMATE SELECTION STRATEGIES

Targets	# of Edges			
	Boofuzz[1]	**WINGMUZZ -DiffVer**	**WINGMUZZ -DiffImpl**	**WINGMUZZ -Hybrid**
Opensips	4,598	4,928 (7.17%↑)[2]	4,843 (5.33%↑)	5,031 (9.42%↑)
Proftpd	5,008	5,475 (9.33%↑)	5,644 (12.69%↑)	5,680 (13.42%↑)
OpenSSH	2,471	2,646 (7.08%↑)	2,477 (0.24%↑)	2,542 (2.87%↑)
DNSmasq	1,514	1,687 (11.43%↑)	1,625 (7.33%↑)	1,699 (12.22%↑)
Ntp-official	1,802	1,851 (2.72%↑)	1,835 (1.83%↑)	1,897 (5.27%↑)
Live555	779	1,136 (45.83%↑)	1,017 (30.55%↑)	1,095 (40.56%↑)
Dcmtk	1,764	2,654 (50.45%↑)	2,268 (28.79%↑)	2,677 (51.76%↑)
Tinydtls	82	281 (242.68%↑)	225 (174.39%↑)	267 (225.61%↑)
TRM[3]	**-**	**21.88%↑**	**14.42%↑**	**22.11%↑**

[1] WINGMUZZ in the ablation study is developed based on Boofuzz.
[2] The percentage increase in coverage is compared with Boofuzz.
[3] TRM: TRimmed Mean, which removes the maximum and minimum values.

worthwhile. As advised by the paper [25], when the coverage rate falls below the threshold rate U_c, the fuzzing campaign should be terminated. As shown in Table V, WINGMUZZ cannot improve much coverage and the rate is lower than U_c, whereas even the best performed benchmark ChatAFL$_{bm}$ is approaching saturation, indicating that it is reaching a bottleneck in further improving coverage. Therefore, terminating the fuzzing campaign at the 10th hour is reasonable. On a longer time scale, the expected coverage rate of WINGMUZZ is still higher than that of benchmarks.

B. RQ2: Ablation Study

Performance of Each Dimension. The two dimensions in WINGMUZZ are tightly coupled. Without the first dimension (the wingmate schedule), the second dimension (the seed schedule) has no targets to fuzz; without the second dimension, WINGMUZZ cannot produce golden seeds for the blackbox fuzzing to use. Thus, to perform ablation study, we evaluate the contribution of each schedule in the two dimensions, namely the wingmate schedule and seed schedule. To evaluate the contribution of the first dimension, we disable the wingmate schedule and only use energy schedule for seeds (WINGMUZZ-SeedSchd). To evaluate the contribution of seed schedule, we disable the seed schedule and only use energy schedule to wingmates (WINGMUZZ-MateSchd). Specifically, when disabling the wingmate schedule, we allocate the average energy to each wingmate for generating test cases. When seed schedule is disabled, we assign average energy to seeds, but new seeds are still prioritized. When performing the ablation study, we use the WINGMUZZ developed upon Boofuzz.

Table VI shows the performance of WINGMUZZ in terms of edge coverage and vulnerability exposure capability under different strategies. The results demonstrate that the absence of wingmate schedule has a more significant impact on performance. WINGMUZZ-SeedSchd shows worse performance than WINGMUZZ-MateSchd on five of eight targets. Of the remaining three, two groups of wingmates are different versions of the same implementation, indicating that wingmate schedule may be less efficient but still effective when wingmates are similar. In terms of vulnerability exposure, there is no improvement comparing WINGMUZZ-SeedSchd and Boofuzz. WINGMUZZ-MateSchd discovers one more vulnerability than WINGMUZZ-SeedSchd. The complete WINGMUZZ discovers 5 vulnerabilities while others only discover less than 3.

Evaluation of Selection Strategies. We use three strategies to select wingmates. For the different version strategy (WINGMUZZ-DiffVer), we select four wingmates with the same implementation as the target but different versions; for the different implementation strategy (WINGMUZZ-DiffImpl), we select four wingmates with different implementations from the target. For the hybrid strategy (WINGMUZZ-Hybrid), we select two wingmates from each of the above strategies. All candidate wingmates are listed in Table II. Table VII shows the evaluation results of three strategies, and the hybrid strategy shows the best performance of 22.11% increase. Different versions provide higher similarity, achieve a 21.88% improvement, while different implementations improves the

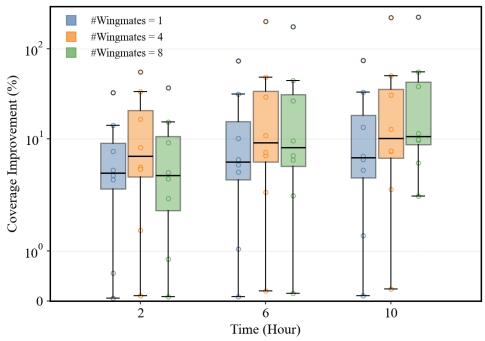

Fig. 3. Coverage improvement under different numbers of wingmates. The appropriate number of wingmates depends on the available fuzzing time.

least. This is because *DiffImpl* has the least similarity with the target protocol. Therefore, in the blackbox setting, where the target is unknown, it is reasonable to diversify by selecting wingmates with different implementations, such as the *Hybrid*. This increases the chance to select similar ones.

Evaluation of the Number of Wingmates. We design three configurations for the number of wingmates: 1, 4, and 8. The detailed grouping for each protocol is summarized in Table II. As shown in Figure 3, coverage improvement is compared with Boofuzz at each time point, with results aggregated across all protocols, where each circle denotes the improvement of one protocol. The horizontal line in the box represents the median value. In 2 hours, the overall performance of the eight-wingmate group is lower than that of both the four-wingmate and one-wingmate groups. In 6 hours, it still lags behind the four-wingmate group. By 10 hours, however, it surpasses the four-wingmate group by 3%. The experiment results show that, the number of appropriate wingmates depends on the running time. Within 6 hours, four wingmates represent the appropriate choice, as each can be allocated sufficient fuzzing time. In contrast, using eight wingmates within the same timeframe is ineffective, since each wingmate receives too little running time. However, when the available time exceeds 10 hours, eight wingmates become more suitable, as each can then be provided with adequate testing time. In general, longer running time allows for more wingmates to be selected, whereas shorter runs require reducing their number accordingly.

C. Evaluation on Protocols in IoT Devices

In this section, we will evaluate WINGMUZZ on protocols in real-world IoT devices. Because we cannot get the internal execution information of IoT devices, we evaluate WINGMUZZ by its ability to optimize seed corpus and discover new bugs.

1) Experiment Settings: The fuzzing is performed on Ubuntu 20.04 with Intel Core i7 @2.80GHz. During the experiment, all the fuzzing instances are performed on separate processes. The fuzzers are then used to test protocols in IoT devices. The target protocols are the widely-used NTP, DNS, and FTP. We choose the widely-used or official implementations as the wingmates. Specifically, as shown in Table VIII, for NTP, we choose the official implementation, and selectively choose 13 versions from 4.2.7 to 4.3.98. For DNS, we select two different implementations, Bind9 and DNSMASQ, and choose 6 versions for each. Since FTP does not have a common implementation, we choose 6 different implementations, which are knftpd 1.0.0, vsftpd-3.0.5, WU-ftpd 2.6.2, proftpd 1.3.8, pureftpd 1.0.50, and glftpd v2.13a.

TABLE VIII
WINGMATES FOR PROTOCOLS IN IOT DEVICES.

Protocols	Wingmates
NTP	4.2.7; 4.2.7p50; 4.2.7p100; 4.2.7p200; 4.2.7p250; 4.2.7p300; 4.2.7p350; 4.2.7p400; 4.2.7p450; 4.2.8p15; 4.3.0; 4.3.50; 4.3.98
DNS-DNSMASQ	2.7; 2.73; 2.77; 2.81; 2.85; 2.89
DNS-Bind9	v9.8.8; v9.10.8; v9.12.4; v9.14.12; v9.16.40; v9.19.17
FTP	knftpd 1.0.0; WU-ftpd 2.6.2; proftpd 1.3.8; pureftpd 1.0.50; glftpd v2.13a; vsftpd-3.0.5

The blackbox targets are listed in Table IX, which covers routers, infotainment systems in smart cars, embedded systems for routers, and Network Attached Storage (NAS). The BMW NBT EVO system runs on the QNX 6.5/6.6 operating system and includes components like NTP. Volkswagen's infotainment system is featured in their latest ID.x series.

2) RQ3: Effectiveness on Optimizing Seed Corpus: Because blackbox fuzzers cannot retain seeds based on code coverage information, the seeds pre-determined for them are critical to the performance of blackbox fuzzing. Blackbox fuzzers such as Boofuzz, Spike and Peach manually defines data fields and mutate the fields. Such workflow is similar to generating new inputs based on RFC files, which are the specifications of protocols. Therefore, to evaluate the ability of optimizing seed corpus, we generate thousands of seeds via traversing the data fields in RFC files. Then, WINGMUZZ is utilized to reduce the size of seed corpus to show the ability of seed optimization. RFC files describe data fields and their possible values. To generate the seed corpus, we traverse the data fields and possible values to generate seeds. For some data fields that only describe data type, such as an integer, we generate edge values for them, such as 0. This is to simulate the seed generation process of blackbox fuzzers, such as Boofuzz. Since they cannot remove seeds (no coverage information for them), they have to use every RFC-generated seed to generate inputs, which decreases the efficiency because many of the seeds examine the same code.

By performing the RFC-based seed generation, we generate 8600 seeds for FTP, 24000 for NTP, and 38000 for DNS. We use WINGMUZZ to retain seeds and the result is shown in Figure 4. On average, NTP retains 86.8 initial seeds, DNS-DNSMASQ retains 185.3 ones, DNS-Bind9 retains 151 ones, and FTP retains 40.7 ones. By running WINGMUZZ, we remove over 99% seeds for each wingmate, *i.e.*, the number of the retained seeds is less than 1% of RFC-generated seeds. This will significantly improve the efficiency of fuzzing because it reduces time on repeatedly testing the same code regions. Because other blackbox fuzzers such as Boofuzz cannot distinguish inputs, all the RFC-generated inputs are the seeds for them. This will significantly decrease the efficiency of blackbox fuzzers because most of the RFC-generated seeds repeatedly test the same code regions.

Figure 4 also shows that, different wingmates for the

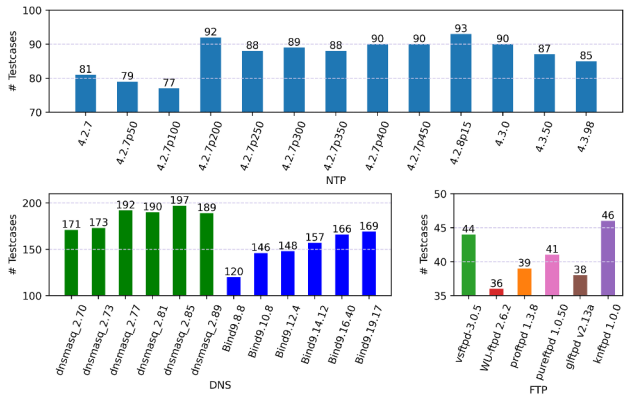

Fig. 4. The number of initial seeds after running WINGMUZZ.

```
1  static void write_variables(struct recvbuf *rbufp, int restrict_mask)
2  {  const struct ctl_var *v; int ext_var; char *valuep; long val;
3     ...
4     val = 0;
5     ...
6     // valuep is set to NULL in ctl_getitem()
7     while ((v = ctl_getitem(sys_var, &valuep)) != 0) {
8        ext_var = 0;
9        ...
10       errno = 0;
11       // valuep is dereferenced
12       if (!ext_var && (*valuep == '\0' || (val = strtol(valuep, NULL, 10), errno != 0))) {
13 ...
```

Fig. 5. The illustrative code for the vulnerability in BMW NBT EVO. The variable `valuep` is dereferenced when its value is NULL.

same protocol have different initial seeds. This indicates the discrepancies of program logic in different versions or implementations. The figure also shows that the discrepancy between different protocol projects is larger than the discrepancy within the same protocol project. In other words, even though different implementations follow the same specification, they use different program logic to achieve the same goal. Therefore, it is critical to schedule wingmates because the exact implementation used in target devices is unknown. We need to infer the similar program logic that is implemented in target devices, which can achieve better performance.

3) RQ4: Discovery of Vulnerabilities in IoT Devices: We perform fuzzers on real-world devices for 10 hours and only WINGMUZZ exposes two new vulnerabilities in target devices. As shown in Table IX, WINGMUZZ exposes a new vulnerability in BMW NBT EVO and a new one in GL.iNet Smart Router. These vulnerabilities have been reported to the vendors and have been confirmed by them. We analyze the vulnerabilities and find that they are similar to the existing publicly released open-source protocol vulnerabilities. Note that, although the exposed vulnerabilities are similar to the existing ones, they are still new in IoT devices.

The Vulnerability in the Smart Car. Our WINGMUZZ discovers a new vulnerability in BMW NBT EVO, which is similar to the vulnerability `CVE-2019-8936` in NTP. The code in Figure 5 is from `CVE-2019-8936` but can demonstrate the vulnerability discovered in the smart car. As shown in Figure 5, in line 7, the value of `valuep` is set to NULL in the function `ctl_getitem()`. Thus, when `valuep` is dereferenced in line 12, it leads to an issue of

TABLE IX
REAL-WORLD TARGETS OF IOT DEVICES.

Device/System	Type	Protocol	Discovered Bug
Xiaomi Mi WiFi R3G	Router	DNS	None[3]
GL.iNet 4G Smart Router	Router	DNS	Heap Buffer Overflow
BMW NBT EVO	Smart Car	NTP	NULL Pointer Dereference
Volswagen	Smart Car	NTP	None
Qnap TS-262C	NAS[1]	FTP	None
Synology DS920+	NAS	FTP	None
OpenWrt iStoreOS 22.03.5[2]	Router	FTP	None

[1] NAS: Network Attached Storage [2] Embedded System for Router
[3] None: No bugs found in the device

```
1 // extract_name() returns 0
2 if (d == 0 && extract_name(header, plen, p, buff, 1, 0)) return to_wire(buff);
3 else{
4    // signed comparison
5    if ((end - *p) < d) d = end - *p; // d is negative
6    if (d != 0){
7       memcpy(buff, *p, d); // wild copy
8       *p += d;}
9    return d;}
```

Fig. 6. The illustrative code for the vulnerability in GL.iNet Smart Router. The function `memcpy()` will copy memory with a large length.

NULL pointer dereference. This security issue can cause the NTP daemon to SIGSEGV. As a result, attackers can exploit this vulnerability to perform a Denial of Service (DoS) attack. This vulnerability can be triggered by sending our crafted packets to the device, causing the NTP functionality to crash. As a result, the device becomes unresponsive to requests.

The Vulnerability in the Router. Another new vulnerability exposed by WINGMUZZ is the one in the GL.iNet Smart Router. Our analysis shows that the vulnerability is similar to the vulnerability `CVE-2020-25683` in DNS. The code in Figure 6 is from `CVE-2020-25683` but can demonstrate the vulnerability discovered in the smart router. This issue stems from missing length verification, which can be manipulated to prompt the code to execute `memcpy()` with a negative size in `get_rdata()`. As shown in Figure 6, the variable `d` is assigned with a negative value in line 5. Because it does not check if `d` is larger than 0 in line 6, the value of `d` in line 9 is negative, which will be translated into a large positive value when executing `memcpy()`. Therefore, a heap buffer overflow occurs, which can lead to DNSMASQ crash. Because routers are responsible for managing traffic between different networks, this vulnerability can breakdown a whole local network. Sending our crafted message can cause the router to crash, potentially leading to a DoS attack.

V. DISCUSSION

Threats to Validity. For external validity, our experimental results may not hold for subjects outside of this study. However, we categorize three types of protocols and select some protocols in each type. In this way, we can mitigate the bias of the selection of target protocols. We also simulate the blackbox setting for more protocols, which further mitigates the bias of protocol selection. For internal validity, our evaluation indicates that the selection of wingmates may significantly influence the evaluation results. With wingmates that are more similar to target protocol, WINGMUZZ can perform better on code coverage and bug discovery. Otherwise, WINGMUZZ will perform worse. In the evaluation, although we select

diverse wingmates to mitigate the bias, the performance of WINGMUZZ on some protocols still indicates the influence of different wingmates. Therefore, we use multiple wingmates to mitigate such problem. For construct validity, we minimize the impact of irrelevant factors onto the evaluation of our main hypothesis. In the evaluation, we implement our WINGMUZZ based on existing tools and compare it to its counterpart tools, *e.g.*, Boofuzz-WINGMUZZ versus Boofuzz. Thus, the improvement can be attributed fully to the changes.

Suggestions for Selecting Wingmates. We suggest selecting wingmates according to several criteria to maximize their relevance and effectiveness. First, preference should be given to well-known, popular, or widely used projects, as these are more likely to have mature and reliable implementations. For repositories hosted on GitHub, a high star count can serve as an additional indicator of quality and community endorsement. Version diversity should also be considered, selecting major versions that exhibit substantial functional differences to increase the likelihood of covering code similar to that of the target protocol. Finally, deployment similarity is important; for instance, if the target operates on a small IoT device, a wingmate with a comparably small codebase is preferable to ensure environmental alignment.

Future Work. Our WINGMUZZ is a framework of using greybox fuzzing to guide blackbox fuzzing. The framework includes three components that are blackbox fuzzing, greybox fuzzing and the component linking the two fuzzing components. As a framework, the components can be replaced without much engineering efforts. On the greybox side, we can replace AFLNet with other protocol greybox fuzzing. Recent research [26] shows that the states of protocols are critical in discovering bugs in protocol implementations. Some issues can only be discovered when considering states, such as the unexpected state transition. Therefore, in the future, we can research on using enhanced stateful greybox fuzzing to guide the blackbox testing. On the blackbox side, recent research [6] shows that we can use large language models (LLMs) to generate and mutate inputs. We can improve the mutation in blackbox side by using LLMs in the future, as performed in the evaluation. Note that, the inputs generated by LLMs will have a large number of duplicated inputs, *i.e.*, they examine the same code region. Fortunately, our WINGMUZZ can trim the size of input corpus to improve fuzzing efficiency. However, other blackbox fuzzers cannot achieve this.

VI. RELATED WORK

The most related research to this paper is the blackbox network protocol fuzzing, which focuses on testing protocols in a blackbox manner [9]–[11], [27], [28]. These research works mainly intend to construct correct testcases (*i.e.*, messages) to test target protocols. They also construct messages by parsing corresponding specifications. The mutation process is mostly performed in a random manner. For example, IoTFuzzer performs blind mutation based on data domains [29]. To improve the effectiveness of fuzzing, some blackbox fuzzers provide the functionality of allowing users to customize the mutations [10], [11]. In other words, these fuzzers rely on human expertise about target protocols. Another strategy is to learn input formats from existing test cases [30], [31], based on which they can generate valid test cases or guide fuzzing. However, these fuzzers cannot determine the quality of inputs due to the lack of effective feedback. Since our scenario lacks the open-source or even the binary code, another related research is the use of proxy code [32], [33]. Due to the lack of target code, these works conduct testing on the equivalent code, whose results are further utilized to test target code. However, they do not use runtime information from the testing proxy code to guide fuzzing.

This paper regards execution states from greybox fuzzing as the feedback of testing IoT protocols. Existing blackbox fuzzers propose different solutions to obtain the feedback for fuzzing protocols, such as message response and state machine [7], [34]–[37]. SNIPUZZ enhances the mutation process of blackbox fuzzing by leveraging response-based feedback [7]. Yet, the number of responses may be too small to guide fuzzing. Some blackbox fuzzers build state machines based on specifications of protocols, which can further guide the process of fuzzing [34]–[37]. However, states in state machine only reflect the coarse information of execution.

On the other hand, greybox fuzzing of protocols uses code coverage and/or protocol states as feedback, which provides more information of program execution [6], [21], [38], [39]. As greybox fuzzing methods, these fuzzers require the source code of target protocols. For instance, IoTHunter adopts a greybox strategy to fuzz IoT firmware's stateful protocols, enabling the constant exploration of both protocol states and code coverage [38]. AFLNet, which uses both protocol state and code coverage as feedback, functions as a client and perpetually replays varied original message sequences to a target [21]. ChatAFL, whose feedback is state and code coverage, uses large language models (LLM) to generate valid and rich messages for testing protocols [6]. However, blackbox fuzzing does not have the internal execution information.

As for the coverage-guided fuzzing or directed fuzzing for non-protocol applications, existing research usually instruments source code or binary code to obtain the information of code coverage [12], [40], [41]. Coverage-guided fuzzers focus on the optimization of fuzzing schedule by formulating fuzzing process [20], [42], [43]. On the other hand, directed fuzzers aim to reach and test pre-determined code locations [44]–[48].

VII. CONCLUSION

In this paper, we propose to use runtime information from greybox fuzzing to guide blackbox fuzzing, which is achieved by the two-dimensional fuzzing schedule. In the first dimension, WINGMUZZ prefers wingmates that are similar to target IoT protocols. The second dimension is greybox fuzzing, which schedules seeds to optimize energy assignment. We have implemented WINGMUZZ to demonstrate the efficacy of this method. The evaluation results show the efficiency of WINGMUZZ in both code coverage and bug discovery.

REFERENCES

[1] Australia, "The voluntary code of practice," https://www.homeaffairs.gov.au/reports-and-publications/submissions-and-discussion-papers/code-of-practice, 2023.

[2] A. Petrosyan, "Global annual number of IoT cyber attacks 2018-2022," https://www.statista.com/statistics/1377569/worldwide-annual-internet-of-things-attacks, 2023.

[3] X. Feng, X. Zhu, Q.-L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on IoT device firmware: A survey," *IEEE/CAA Journal of Automatica Sinica*, 2022.

[4] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 480–491.

[5] A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *Proceedings of the 29th USENIX Security Symposium (USENIX '20)*, 2020.

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

[7] 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," *in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[8] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, 2022.

[9] wireghoul, "Doona," https://github.com/wireghoul/doona, 2019.

[10] Peachtech, "PEACH: The PEACH fuzzer platform," 2024, Accessed: 2024-01. [Online]. Available: https://www.peach.tech/products/peach-fuzzer/

[11] J. Pereyda, "boofuzz: Network protocol fuzzing for humans," https://boofuzz.readthedocs.io/en/stable/, 2024, Accessed: 2024-01.

[12] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2169–2182.

[13] K. Zhang, X. Zhu, X. Xiao, M. Xue, C. Zhang, and S. Wen, "Shapfuzz: Efficient fuzzing via shapley-guided byte selection," in *Network and Distributed System Security (NDSS)*, 2024.

[14] K. Zhang, X. Xiao, X. Zhu, R. Sun, M. Xue, and S. Wen, "Path transitions tell more: Optimizing fuzzing schedules via runtime program states," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1658–1668.

[15] S. Hamlaoui, "Spike fuzzer," https://github.com/SofianeHamlaoui/Spike-Fuzzer, 2024, Accessed: 2024-01.

[16] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 330–347.

[17] A. Saleem, S. Shah, H. Iftikhar, J. Zywiołek, and O. Albalawi, "A comprehensive systematic survey of iot protocols: Implications for data quality and performance," *IEEE Access*, 2024.

[18] B. B. Gupta and M. Quamara, "An overview of internet of things (iot): Architectural aspects, challenges, and protocols," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 21, p. e4946, 2020.

[19] V. E. Quincozes, S. E. Quincozes, J. F. Kazienko, S. Gama, O. Cheikhrouhou, and A. Koubaa, "A survey on iot application layer protocols, security challenges, and the role of explainable ai in iot (xaiot)," *International Journal of Information Security*, vol. 23, no. 3, pp. 1975–2002, 2024.

[20] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[21] 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.

[22] H. Liu, S. Gan, C. Zhang, Z. Gao, H. Zhang, X. Wang, and G. Gao, "Labrador: Response guided directed fuzzing for black-box iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 127–127.

[23] J. Wang, L. Yu, and X. Luo, "Llmif: Augmented large language model for fuzzing iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 196–196.

[24] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 662–665.

[25] D. Liyanage, S. Lee, C. Tantithamthavorn, and M. Böhme, "Extrapolating coverage rate in greybox fuzzing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[26] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3255–3272.

[27] Fitblip, "Sulley," https://github.com/OpenRCE/sulley, 2019.

[28] FORTAR, "bestorm overview," https://www.beyondsecurity.com/products/bestorm/, 2023, accessed on 26-March-2023.

[29] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[30] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Learning-based fuzzing of iot message brokers," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 47–58.

[31] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 59–67.

[32] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 722–733.

[33] K. Serebryany, M. Lifantsev, K. Shtoyk, D. Kwan, and P. Hochschild, "Silifuzz: Fuzzing cpus by proxy," *arXiv preprint arXiv:2110.11519*, 2021.

[34] S. Jero, M. E. Hoque, D. R. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in tcp congestion control using a model-guided approach," in *NDSS*, 2018.

[35] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2523–2540.

[36] H. Park, C. K. Nkuba, S. Woo, and H. Lee, "L2fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 343–354.

[37] 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.

[38] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[39] 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," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 489–502.

[40] K. Ye, X. Zhu, X. Xiao, S. Wen, M. Xue, and Y. Xiang, "Bazzafl: Moving fuzzing campaigns towards bugs via grouping bug-oriented seeds," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[41] X. Zhu, X. Feng, X. Meng, S. Wen, S. Camtepe, Y. Xiang, and K. Ren, "Csi-fuzz: Full-speed edge tracing using coverage sensitive instrumentation," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 912–923, 2020.

[42] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[43] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

[44] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," 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, pp. 2329–2344.

[45] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 2095–2108.

[46] S. Wang, X. Jiang, X. Yu, and S. Sun, "Kcfuzz: Directed fuzzing based on keypoint coverage," in *Artificial Intelligence and Security: 7th International Conference, ICAIS 2021, Dublin, Ireland, July 19–23, 2021, Proceedings, Part I 7*. Springer, 2021, pp. 312–325.

[47] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 36–50.

[48] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: a directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2440–2451.