RIoTFuzzer: Companion App Assisted Remote Fuzzing for Detecting Vulnerabilities in IoT Devices

Kaizheng Liu
Southeast University
Nanjing, Jiangsu, China
kzliu18@seu.edu.cn

Ming Yang
Southeast University
Nanjing, Jiangsu, China
yangming2002@seu.edu.cn

Zhen Ling
Southeast University
Nanjing, Jiangsu, China
zhenling@seu.edu.cn

Yue Zhang
Drexel University
Philadelphia, PA, USA
yz899@drexel.edu

Chongqing Lei
Southeast University
Nanjing, Jiangsu, China
leicq@seu.edu.cn

Junzhou Luo
Southeast University
Nanjing, Jiangsu, China
jluo@seu.edu.cn

Xinwen Fu
UMass Lowell
Lowell, MA, USA
xinwen_fu@uml.edu

Abstract

Due to the diversity of architectures and peripherals of Internet of Things (IoT) systems, blackbox fuzzing stands out as a prime option for discovering vulnerabilities of IoT devices. Existing blackbox fuzzing tools often rely on companion apps to generate valid fuzzing packets. However, existing methods encounter the challenges of bypassing the cloud server side validation when it comes to fuzz devices that rely on cloud-based communication. Moreover, they tend to concentrate their efforts on Java components within Android companion apps, limiting their effectiveness in assessing non-Java components such as JavaScript-based mini-apps. In this paper, we introduce a novel blackbox fuzzing method, named RIoTFuzzer, designed to remotely uncover vulnerabilities of IoTu3C

code coverage, particularly for devices that only rely on remote communication through cloud servers (40% in [1]). Expanding their capabilities to encompass server-based fuzzing presents a series of new challenges, including effective crafting of fuzzing packets capable of evading cloud server validation. Second, these fuzzing methods primarily target Java components of companion apps, thereby restricting their effectiveness in assessing non-Java components such as JavaScript-based mini-apps within the companion apps. However, numerous IoT platforms, such as Xiaomi (with 654.5 million connected IoT devices [40]), Jingdong (encompassing more than 4,000 products from 1,000 brands [44]), Huawei (more than 30 million registered users [13]), and Tuya (supporting 2,700 types of smart devices globally [32]), offer mini-apps within their comprehensive all-in-one apps. That is, the mini-app operates on top of an All-in-one App (or the host app), and control commands can be originally sent either directly from the All-in-one App or indirectly through the mini-app. This flexibility enables IoT device manufacturers to create JavaScript-based mini-apps for IoT device control.

Our Approach. To address the limitations of existing methods, we introduce a novel method called RIoTFuzzer ("R" stands for the remote side of IoT devices) for blackbox fuzzing IoT devices remotely, to automatically discover vulnerabilities in IoT firmware. What sets RIoTFuzzer apart from existing methods is our ability to achieve genuine remote fuzzing. That is, the fuzzer and the IoT device are connected to different networks and communicate through a cloud server. We are also able to identify the appropriate mutation point for the control command generated by mini-apps. RIoTFuzzer allows us to uncover vulnerabilities in code related to the remote control functionality of the All-in-one App powered IoT platform, a domain that existing methods overlook.

RIoTFuzzer faces three major challenges: (i) RIoTFuzzer is a blackbox fuzzer and code coverage assessment is a grand challenge. Previous works [4, 25] indicate that high code coverage can be achieved by enumerating control commands from the companion apps. Existing methods such as manual app execution or using tools like monkeyrunner [14] are time-consuming. (ii) mini-apps are often obfuscated [45], making it difficult to analyze their control packets generation. Data transformation can happen in a companion All-in-one App [25] as well, and pinpointing the correct location in the companion app (i.e., the All-in-one App level or the mini-app level) for mutation is crucial to create effective fuzzing packets. (iii) Remote fuzzing faces the challenge of cloud-side checking, which may reject fuzzing packets can be rejected if they do not pass server-side validation. This is very different from traditional local fuzzing methods. To create fuzzing packets that can reach IoT devices, we have to comprehend the cloud server's validation policy. Unfortunately, the cloud server typically operates as a black box, and its validation policy is not publicly accessible.

We address these challenges in RIoTFuzzer as follows. (i) To address the challenge of code coverage, we first extract control commands from the official document using regular expressions and then manual analysis is used to refine the extractions. These documents are collected through network synchronization traffic and the official document search engine. (ii) We employ hybrid app analysis to identify the appropriate *mutation point*, which is the function situated between data encoding and data transferring. This process involves two phases: the first phase entails static app analysis and large language model [43] through ChatGPT to identify candidate Java interface functions. The second phase employs dynamic instrumentation to discover the actual *mutation point* based on the identified candidate Java interface functions. (iii) We introduce a side-channel method to overcome cloud server validation. By analyzing the response time, we can infer the validation policy of the blackbox cloud server. This information guides the construction of fuzzing packets, enabling them to bypass server-side validation.

We implement the prototype of RIoTFuzzer and apply it to 27 IoT devices from four prominent IoT platforms: *Xiaomi*, *Jingdong*, *Huawei*, and *Tuya*. We discovered 11 vulnerabilities across 10 IoT devices and 8 of them have been confirmed by corresponding vendors. To uncover these vulnerabilities, we only need to send an average of 113 fuzzing packets, with a maximum of 746. We also evaluate the side-channel-guided fuzzing approach, revealing a significant improvement in effectiveness, with an average increase of 76.62% and a maximum increase of 362.62%. RIoTFuzzer outperforms the state-of-the-art blackbox IoT device fuzzer—DIANE [25] by detecting 11 more vulnerabilities.

Contribution. We make the following major contributions:

- **Tool Advancing Existing IoT Area:** We introduce RIoTFuzzer, the first remote blackbox fuzzer designed to discover vulnerabilities in IoT devices though the cloud server. RIoTFuzzer is capable of handling All-in-one App powered IoT platforms, an area often overlooked by existing methods.
- **Techniques with Domain Insight:** We propose the document-based control command extraction, hybrid analysis-based mutation point identification, and side-channel-guided fuzzing to effectively construct the fuzzing packets for the IoT device that can bypass the cloud server-side validation.
- **Vulnerabilities with Real-world Impacts:** We evaluate RIoTFuzzer with 27 IoT devices from 4 popular IoT platforms, i.e., *Xiaomi*, *Jingdong*, *Huawei*, and *Tuya*. We have discovered 11 vulnerabilities among 10 IoT devices and all of them have been acknowledged by the corresponding vendors. 8 of them have been confirmed and 4 CVE IDs are assigned, i.e., CVE-2024-3764, CVE-2024-5095, CVE-2024-32268, CVE-2024-32269.

2 Background

This section introduces the IoT system architecture and platforms.

2.1 IoT System Architecture

IoT refers to a network of physical objects or "things" that are equipped with sensors, software, and connectivity capabilities, allowing them to collect and exchange data with other devices and systems over the internet or various communication networks. As depicted in Figure 1, an IoT system can be conceptually divided into three distinct components: the controller, the IoT device, and the cloud server.

- **Controller:** The controller is a device such as a smartphone that has a companion app installed for a specific IoT application and remotely send control commands to the IoT device via the app to execute specific functions.

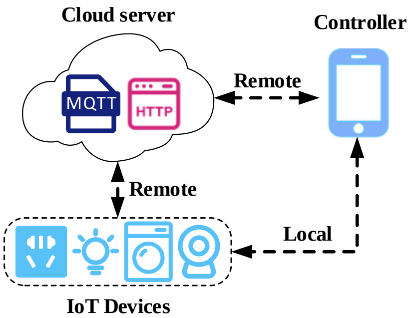

Figure 1: IoT System Architecture

- **IoT device:** An IoT device often has hardware sensors and specific software that can transmit data over the internet or other networks.
- **Cloud server:** Acting as a facilitator for communication between IoT devices and controllers, the cloud server may store data generated by IoT devices, manage device configurations, and enables remote access and control.

A controller may control a device locally or remotely. In local control, both the controller and the IoT device reside within the same network and the two parts communicate with each other directly. Remote control relies on the cloud server as an intermediary to communicate for the controller and the IoT device.

2.2 IoT Platforms

IoT platforms are software solutions or ecosystems designed to simplify and streamline the development, management, and deployment of IoT devices and applications. They offer a comprehensive solution for device connectivity, data management, security, and application development, enabling businesses to harness the potential of IoT technology more effectively and efficiently. Prominent examples in this realm include industry giants such as *Xiaomi* and *Tuya*. For example, to easy app development, *Tuya* provides a SDK to help quickly build a control mini-app with JavaScript running with the *Tuya* All-in-one App. The essential functionalities of various IoT devices are managed by the All-in-one App, including tasks like control command encoding and network communication.

Table 1: Comparison of Major IoT Platforms

Platform	Release Date	All-in-one App	Remote Control	SSL/TLS
Tuya	2014	4	4	4
Amazon	2015	7	4	4
Xiaomi	2016	4	4	4
Google	2017	7	4	4
Jingdong	2018	4	4	4
Huawei	2019	4	4	4

In Table 1, we present a comprehensive comparison of popular IoT platforms. These platforms share a common feature: support for remote control, which allows users to track, monitor, and manage their IoT devices remotely. Additionally, security is of utmost importance to these IoT platforms, and they all implement SSL/TLS encryption to safeguard data transmission across various components of the IoT system. Furthermore, we have identified

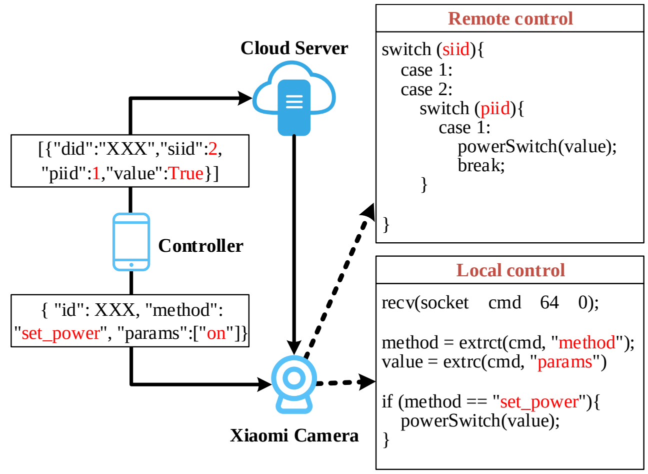

Figure 2: *Xiaomi* Camera Control Command Difference in Local and Remote Scenarios

a distinction among these platforms. To simplify implementation, four IoT platforms—*Xiaomi*, *Jingdong*, *Huawei*, and *Tuya*—have introduced the concept of an All-in-one App: all IoT devices based on the IoT platform can be set up and controlled with mini-apps instead of building standalone companion apps. For Amazon and Google, although they introduce control apps, for some IoT devices standalone companion apps are required to initialize IoT devices first. Only after this initial setup can IoT devices be controlled by apps such as Amazon Alexa and Google Home.

3 Motivation and Challenges

In this section, we will introduce the motivation, thread model, scope of the paper and challenges. Solutions to those challenges are briefly summarized while the details will be presented in §4.

3.1 Motivation

To detect vulnerabilities in IoT devices, companion app-assisted blackbox fuzzing methods have been proposed [4, 25]. All these existing methods focus on fuzzing IoT devices within a local network. However, according to previous research [1], many IoT devices can only be controlled remotely via the cloud server. Even if the IoT device supports both local control and remote control, the implementation in the device can be different. Figure 2 gives an example. The "power on" control commands for Xiaomi camera in local and remote scenarios are presented. The control commands are different for these two scenarios and the different code branches are used in the IoT device for local control and remote control. Therefore, the existing local blackbox fuzzing is insufficient.

As discussed in §2.2, some IoT platforms introduce JavaScript to build the frontend mini-app running on the All-in-one App. Existing blackbox IoT device fuzzing methods [4, 25] primarily rely on identifying input sources at the Java level (e.g., the data generated from the user input). However, in the case of the All-in-one App, control commands may originate from mini-apps and then undergo encoding at the Java level before being transmitted over the network. In this scenario, existing methods may encounter difficulties in pinpointing the data source necessary for mutation, rendering them inadequate for the task.

3.2 Thread Model and Scope

Thread model. Our objective is to test IoT devices for vulnerabilities. We fuzz our own devices as authenticated users while discovered vulnerabilities may be exploited by attackers in various ways. For example, an insider attacker may exploit such a device and plant malware on the device. The discovered vulnerabilities may be exploited bypassing authentication.

Goal and Scope. We focus on devices that can be remotely controlled and communicate with cloud servers through Wi-Fi. There are over 4 billion connected IoT devices worldwide that are Wi-Fi-enabled [27]. We target the four platforms with the All-in-one App (as illustrated in Table 1) as those platforms that support companion app communication have not been extensively investigated. While IoT platforms typically provide both Android and iOS apps, we focus on Android, which has a share of 71.5% on the mobile operating system market [30].

3.3 Challenges and Solutions

For IoT platforms scuh as *Xiaomi* and *Huawei*, cloud server-side verification and use of mini-app introduce new challenges as summarized below when adapting existing methods to such scenarios. We also briefly discuss how we address those challenges.

(C-I) High code coverage for black-box fuzzing. During the fuzzing process, it is imperative to attain the highest possible code coverage. However, in the context of IoT devices, which often operate as black boxes, determining code coverage directly from the device is infeasible. This limitation hinders our ability to guide the mutation process effectively toward achieving high coverage. In the companion app-assisted scenario, high code coverage can be achieved by enumerating all control commands on the controller side. Existing blackbox fuzzing methods [4, 25] have attempted to address this challenge by manually running the app once and replaying UI inputs using tools like RERAN [11] or adopting monkeyrunner [14] to generate UI inputs by generating random events. However, this approach is time-consuming. Therefore, there is a pressing need for innovative methods that can enhance fuzzing techniques to reach high code coverage in these scenarios easily.

Control Command Extracting(§4.2)

To address this challenge, we find that for devices based on IoT platforms, the control commands on the companion app side can be extracted from the IoT platform's document search engine and synchronization network traffic.

(C-II) Formatted control command source identification. In the case of devices on IoT platforms of interest, the control command is generated in mini-apps and then transferred to the Java level of the All-in-one App for transformation and transmission. Constructing fuzzing packets intuitively involves identifying the data source within the mini-app by analyzing the JavaScript code. However, this approach faces two significant challenges. First, as reported in [45], most mini-apps employ obfuscation techniques, making it difficult to analyze and discover the data source within mini-apps. Second, the control commands generated within the mini-app may undergo the transformation in the Java code of the All-in-one App before being sent, with the details below.

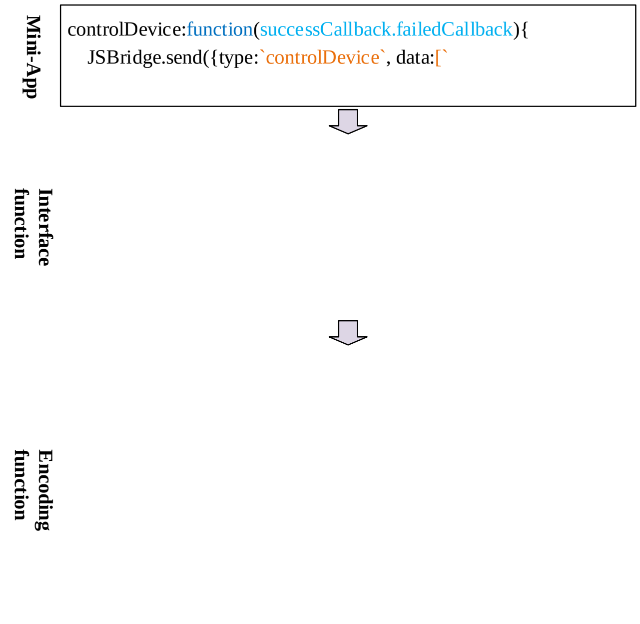

Figure 3: Example of Control Command Data Flow from Mini-app to Message Sending

As illustrated in Figure 3, consider the example of controlling a camera to turn to the left. The control command generated from the mini-app is represented as a string array containing "ptz" and "left". This command is then transferred to the Java layer of the All-in-one App. Subsequently, the string array control command undergoes transformation and is transformed into a key-value pair (e.g., `{127:0}`) by the "`deviceControl`" function. Finally, it is sent to the cloud server using the "`sendToServer`" function. In this case, the "`sendToServer`" function is the *mutation point* and we can construct the fuzzing packets by mutating the argument passed to the "`sendToServer`" function. Identifying this *mutation point* is crucial for efficiently constructing well-structured fuzzing packets containing control commands.

Hybrid Analysis Based Mutation Point Finding (§4.3)

We can first identify the so-called border functions that receive control commands from the mini-apps at the Java level with static app analysis and the large language model (i.e., ChatGPT). We can then employ the hybrid analysis of the All-in-One app with both static and dynamic analysis to identify and confirm the *mutation point* where fuzzing data is generated.

(C-III) Black-box cloud server verification inference. Cloud server side verification can be present in the context of remote

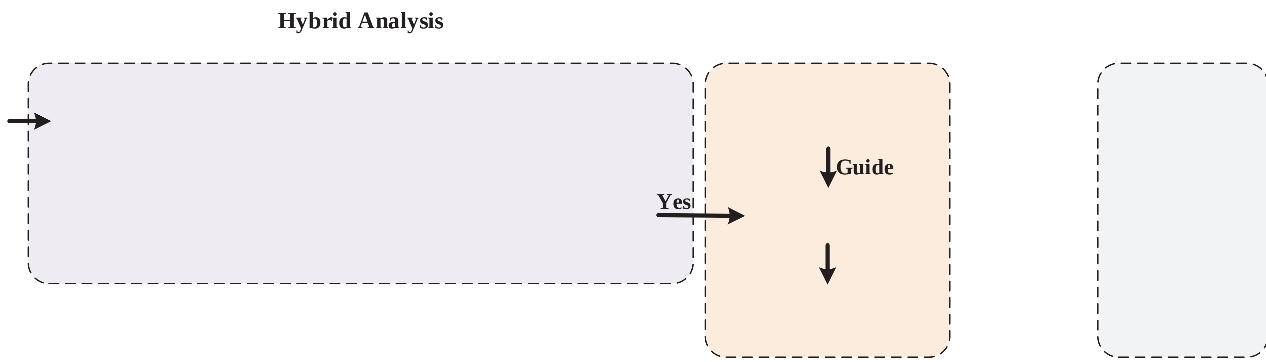

Figure 4: System Overview

control while there is no such challenge in local fuzzing research [4, 9, 21, 25]. While the local control scenario allows all fuzzing packets sent from the companion app to reach the IoT device, remote fuzzing introduces the possibility that a fuzzing packet may not bypass cloud server verification, and the packet could be rejected directly by the cloud server and cannot reach the IoT device, resulting in lower fuzzing efficiency. To develop an efficient remote fuzzing tool, it is essential to first understand and uncover the cloud server's verification process. This is a significant challenge as the cloud server is essentially a black box and its verification procedures are not publicly disclosed.

Side-Channel-Guided Fuzzing (§4.4)

Since the cloud server operates as a black box, we cannot directly discern its verification policy. However, we find that the verification policy can be deduced through side channels. By examining the response time between sending packet and receiving response at the companion app side, we have observed significant differences between packets that can bypass cloud server verification and those that cannot. Leveraging this insight, we can infer the cloud server's verification policy by generating fuzzing packets with varying payload data.

4 RIoTFuzzer Design

To address the challenges in §3.3, we introduce RIoTFuzzer, a method that leverages hybrid app analysis and blackbox fuzzing for remotely discovering vulnerabilities in IoT devices through the cloud server. We will first introduce the four components of RIoTFuzzer and then present the detailed design of each component.

4.1 System Components

Figure 4 shows the four components of RIoTFuzzer.

- **Document based control command extracting (§4.2).** To tackle the challenge in **C-I**, we first extract control commands of the companion app from the document. We employ two methods for obtaining the control command document: utilizing the official document search engine of a vendor or analyzing the synchronization network traffic between the companion app and the cloud server for JSON documents and others. Second, we can also manually run the app to extract control commands and •

```
"characteristics": [
    {
        "characteristicName": "on",
        "characteristicType": "bool",
        "method": "RW",
        "enumList":
        [
            {
                "enumVal": 0,
            },
            {
                "enumVal": 1,
            }
        ]
    }
]
```

Figure 5: Document Snippet of "`Chint`" Plug

the cloud server with control commands and it is generally in the format of JSON or XML. This synchronization is crucial for preventing state inconsistencies. We have noted that these control commands can be extracted directly from the synchronization packets.

To extract the synchronization document, we explore the payload of the synchronization packets. This involves analyzing the network traffic between the companion app and the cloud server, which is usually protected by SSL/TLS encryption. To circumvent this protection, we utilize a man-in-the-middle (MITM) proxy tool, namely mitmproxy [6], along with a self-signed certificate to conduct the MITM attack. We bypass the companion app side certificate verification and SSL/TLS protection through dynamic instrumentation [28] so as to obtain the payload. If there are fields that are protected cryptographically in the payload, we hook the corresponding cryptographic APIs to capture both the input arguments (plaintext) and return value (ciphertext/hash). By comparing the cryptographic text detected in the payload with the hooked return values, we can unveil the plaintext. This reverse-engineering process allows us to decipher the synchronization packets and unveil the synchronization document.

Table 2: Analysis of "`Chint`" Plug Document Snippet

Key	Function	Detail Explanation of Value
`characteristicName`	Operation	Switch the plug
`characteristicType`	Data type	Data type in the command is `bool`
`method`	Permission	R stands reading and W stands writing
`enumList`	Data range	0 stand poweroff and 1 stand poweron

For instance, consider the snippet from the document of a plug manufactured by "Chint" based on the *Huawei* IoT platform, as shown in Figure 5. This snippet provides crucial details about control commands for switching on/off the plug, including the control command key, permission, data type, and valid value range. We present the analysis result for the document snippet in Table 2. In Figure 6, we present a demo control packet sent by the companion app to power off the "`Chint`" plug through the cloud server and the corresponding control command { "on": 0 } can be discovered.

4.3 Hybrid Analysis Based Mutation Point Finding

A simple idea addressing Challenge **C-II** is to mutate the control command generated in the mini-app to construct the fuzzing packet

```
{
    "body": {
        "on": 0
    },
    "header": {
        "method": "POST",
        "requested": "XXX",
        "mode": "ACK",
        "accessToken": "XXX",
        "timestamp": "20231127T101634Z",
        "to": "/devices/XXX/services/switch",
        "from": "/users/XXX"
    }
}
```

Figure 6: "`Chint`" Plug Power off Control Packet

Table 3: Example of the Corresponding Output for Filtering the IoT Device Controlling Non-irrelevant Function

Case #	Interface Functions	Explanation by LLM
1	`controlRuleActive`	Can be used to activate or deactivate rules
2	`deleteDevice` `deleteDeviceById`	Might be involved in removing a device, which indicates they are related to device management

3	`getDeviceConfig`
	`setDeviceInfo`
	`setDeviceInfoWithProdId`
	`setDeviceInfoWithoutCallback`
		Likely involved in retrieving and setting configurations for devices

4	`updateGroupMemberDevice`	Can be relevant if the plug is part of a group

5	`getDevInfo`
	`getDevInfoAll`
	`getDevInfoWithProdId`
		Retrieve device information could be related to check the current state of the plug

After we discover all the potential border functions from the smali code with regular expression, we find there are many functions irrelevant to IoT device control in the list of potential border functions. This may lead to 5svate5svatnumtober ofeularvatmay

to the IoT device cont9ol f8om the list of potential bo9der functionse we do not find the *mutation point*, we recursively perform step 3. This step is represented by line 13, and line 20.

4.4 Side-Channel-Guided Fuzzing

During the construction of fuzzing packets, our primary focus is mutating the key-value pair based on the control command keys extracted from the document to enumerate all the functionalities of the IoT devices that can be triggered by the companion app. This process begins with a random selection of the control command key. Then, a value corresponding to a randomly chosen data type from the following data types is generated:

- **Value of numeric data type:** For numeric data types, including "`Int`" and "`Float`", we generate two categories of values aimed at inducing integer overflow or out-of-bound accesses. The first approach involves generating values that adhere to the data constraints specified for the respective data type in the device documentation. This ensures that the values are within the expected range for each type. The second approach is more arbitrary, involving the generation of large or negative values at random.
- **Value of "`Bool`":** For "`Bool`", we randomly select "`True`" or "`False`" to construct the fuzzing packet.
- **Length of "`String`":** For "`String`", we focus on constructing fuzzing packets with varying lengths. Given that documents typically do not specify length restrictions, it is not feasible to determine a valid length directly from the documentation. Therefore, we employ a strategy to randomly generate strings ranging from 1 to 10,000 characters in length to identify vulnerabilities like buffer overflow.

Black-box cloud server verification inference. As highlighted in **C-III**, randomly mutating control commands to generate fuzzing packets poses a challenge to passing the cloud server's verification. We observe the response time to a request observed at the companion app is different in two scenarios: (i) a packet passes server verification; (ii) a packet fails server verification. Notably, if a fuzzing packet can not pass the cloud server's verification, the server usually immediately sends a response back to the client. This results in a significantly reduced response time, in contrast to the scenario where a fuzzing packet successfully passes the cloud server verification, reaches the IoT device and then waits for a response from the IoT device. With this side channel, we can find the cloud server verification policy and construct the fuzzing packet that can pass the cloud server verification.

A threshold for the response time is needed to determine whether a packet can pass the server verification. We train the threshold by sending labeled packets to the cloud server and record their response time. It is designed that some packets pass the cloud server verification and others cannot not. A suitable threshold such as the Bayesian decision threshold can then be chosen to determine whether the fuzzing packets successfully circumvent the cloud server's verification.

After obtaining the threshold, we can find data types of a specific control command key that can pass the cloud server verification. Network jitters may cause false positives, that is, the response time for the packet that does not pass the cloud server verification may be larger than the threshold. To reduce false positives introduced by network jitters, we send fuzzing packets for a specific control command key with a specific data type multiple times. If all packets for a specific control command key and data type successfully reach the IoT devices, it indicates that this data type can pass the cloud server verification for that control command key and can be used in fuzzing. The total number of packets sent to identify valid data types for all control command keys is $5(G) = G \times \sim \times 4$ (G is the number of control command keys, ~ is the number of packets for a specific data type for a specific control command key).

4.5 Network Behavior Based Crash Monitoring

Our approach involves a network behavior b a net9e65r b i□c controli`0 0-ypes for a

Table 4: Summary of IoT Devices under Testing. "-" means we can not discover the release time of the IoT product. "*" means the firmware version cannot be discovered in the App. "\" means the device model can not be discovered in the device or App. "NA" means we can not discover vulnerabilities in the IoT device within 12 hours. "7" means DIANE fails to generate the fuzzing packets for the device

Platform	#	Vendor	Device Type	Release Time	Model	Firmware Version	Mini-app Control	RIoTFuzzer # of Issues	RIoTFuzzer # of Packet	DIANE Time [hours]
Xiaomi	1	Yeelight	Bulb	2022	yeelink.light.color8	2.1.7_0041	4	0	NA	7
	2	Yeelight	Bulb	2018	yeelink.light.color2	2.0.6_0065	4	0	NA	7
	3	Philips	Bulb	2019	philips.light.cbulb	2.0.8_0004	4	0	NA	7
	4	Xiaomi	Gateway	2022	lumi.gateway.mcn001	1.0.7_0019	4	0	NA	7
	5	Xiaomi	Camera	2022	mxiang.camera.mod11	5.1.5_0035	4	0	NA	7
	6	Xiaomi	Camera	2020	chuangmi.camera.ip029a	4.3.4_0425	4	0	NA	7
	7	Xiaomi	Camera	2021	isa.camera.hlc7	4.3.2_0220	4	1	8	7
	8	Xiaovv	Camera	2022	xiaovv.camera.q2lite	5.1.5_1434	4	0	NA	7
	9	Imilab	Camera	2023	chuangmi.camera.046d02	5.1.7_0408	4	0	NA	7
	10	Xiaomi	Humidifier	2022	deerma.humidifier.jsq2g	2.2.2.0012	4	0	NA	7
	11	Xiaomi	Plug	2022	cuco.plug.v3	1.0.8.0018	4	0	NA	7
	12	Gosund	Plug	2022	cuco.plug.cp1md	2.1.3_0010	4	1	176	7
	13	Gosund	Plug	2022	cuco.acpartner.cp6	2.1.3_0012	4	1	746	7
	14	Xiaomi	Remote control unit	2017	lumi.acpartner.v2	1.4.1_156.0148	4	0	NA	7
Jingdong	15	Bull	Plug	2022	GN-Y2011	*	4	0	NA	>12h
	16	DELIXI	Plug	2022	CD98I-MXWE2	57	4	0	NA	>12h
	17	Jingdong	Plug	2019	SPW01	1.3	4	0	NA	>12h
Huawei	18	Chint	Plug	2022	Sunrise 6-111W	1.0.0.116	4	1	21	>12h
	19	wanyesw	Plug	2019	ZCZ001	1.0.2	4	1	17	>12h
	20	SANSI	Bulb	2018	C21BB-TE27-8W-D	1.01	4	1	82	>12h
	21	YKK	Remote control unit	2021	YKK-1011	1.4.6	4	0	NA	>12h
Tuya	22	Sagewe	Plug	2023	F2s501-GB	V1.3.5	4	0	NA	7
	23	Tuya	Bulb	2023	BD-A60	v1.2.16	4	0	NA	7
	24	Haojiaojing	Camera	-	\	V3.2.9	4	2	48 & 142	7
	25	Yonganda	Camera	-	YAD-LOJ	V3.0.561	4	1	5	7
	26	zsviot	Camera	2022	\	V8.26.31	4	1	10	7
	27	Tuya	Camera	-	U6N	V3.2.5	4	1	47	7

- *RQ3:* How efficient is RIoTFuzzer compared to the state-of-the-art method (§5.4)?

Table 5: APPs of Platforms under Testing

Platform	Android APP Package Name	APP Version
Xiaomi	com.xiaomi.smarthome	9.0.605.4059-64-DEV
Jingdong	com.jd.iots	V1.9.2
Huawei	com.huawei.smarthome	13.1.0.320
Tuya	com.tuya.smart	3.25.0 (international)

5.1 Experiment Setup

We implement a prototype of RIoTFuzzer based on Frida [23] and Androguard [2]. RIoTFuzzer is designed to be compatible with four widely used IoT platforms, namely *Xiaomi*, *Jingdong*, *Huawei*, and *Tuya*, enabling the identification of vulnerabilities in their IoT devices. We use two Android phones (Redmi 10A with Android 9 and Redmi 10A with Android 10) as controllers to install the companion apps of these four IoT platforms and construct the fuzzing packets. The companion apps used in this paper is shown in Table 5. The call graph construction for the target companion app is conducted on a Linux server with a 2.4 GHz Xeon CPU and 128 GB memory. We use an Ubuntu computer with a 3.4 GHz Core CPU and 8 GB memory equipped with a USB WiFi adapter as the monitor by setting up an access point (AP) and configuring IoT devices connected to the AP. The Side-channel-guided fuzzing is conducted with a Ubuntu server with a 2.1 GHz Xeon CPU and 64 GB memory. Moreover, to ensure minimal any potential disruption to the cloud server and prevent excessive load, we schedule the transmission of fuzzing packets with randomized intervals ranging between 10 and 15 seconds.

Before fuzzing, we first determine an appropriate threshold to assess whether a fuzzing packet passes cloud server verification. In this study, we construct 1000 packets for each of the two scenarios. For the packets that do not pass server verification, 998 out of 1000 had a time interval of less than 300ms. Conversely, all 1000 packets that can pass the cloud server verification had a time interval longer than 300ms. Therefore, setting the threshold to 300ms effectively distinguishes between the two scenarios.

5.2 Vulnerability Detection (RQ1)

We conduct an extensive vulnerability detection assessment using RIoTFuzzer, examining a total of 27 IoT devices across various categories, including the camera, gateway, and humidifier, across the four IoT platforms as shown in Figure 7. Our fuzzing process involves sending fuzzing packets to each device continuously for 12 hours while monitoring network behavior on the IoT device side to identify potential vulnerabilities. To ensure the accuracy of our findings and avoid false positives, we employ a rigorous validation process. Upon detecting abnormal network behavior indicative of potential vulnerabilities, we repeat the transmission of the exploiting packet three times. Only when the abnormal behavior is consistently observed we confirm the presence of a true positive vulnerability in the IoT device.

As a result, RIoTFuzzer successfully identifies a total of 11 vulnerabilities across 10 IoT devices with *Xiaomi*, *Huawei*, and *Tuya* IoT platforms as shown in Table 4. These devices encompass a

Figure 7: All IoT Devices Used in Our Experiments

range of products, including the camera, plug, and bulb. Notably, our vulnerability identification process requires 1,291 fuzzing packets in total, averaging 117 packets per vulnerability. All identified vulnerabilities have been reported to their respective vendors, with 8 of them having already been confirmed.

Case Study. The "`isa.camera.hlc7`" (#7) home security smart camera with 2k resolution produced by Xiaomi is operated via the companion Android app "Mi Home". To explore potential vulnerabilities within this camera, we initially bind the camera to the companion app and we can control the camera with the app. Subsequently, we obtain the device's official documentation from Xiaomi's official document search engine [39] using the camera's model, i.e., "isa.camera.hlc7".

Utilizing `RIoTFuzzer`, we detect that a vulnerability is induced when the controller app sends a message, as depicted in Figure 8, resulting in the camera temporarily going offline. In this scenario, the value linked to the *signature* key is used for payload integrity verification. This value is generated via HMAC, employing a secret key received from the server during the binding phase, with the remaining payload in the packet serving as input. Additionally, the *nonce* value is randomly generated, *did* signifies the device ID, while *siid* and *piid* represent the service ID and property ID of the IoT device, respectively. The combination of *siid* and *piid* corresponds to a specific functionality of the device, i.e., the control command key. The *value* field, in turn, denotes the input data when executing this specific functionality. As presented in the official documentation, the functionality defined in Figure 8, i.e., *siid : 6* and *piid : 6*, pertains to voice download. The input data type for this functionality is designated as string without a length limitation in the document. When we exercise the functionality with *-216.49537562852015* as input, the camera will temporarily offline, and a DoS vulnerability is discovered.

To delve deeper into the cause of the crash, we tear down the camera and access its UART port to obtain the console, which is identifiable on the device's circuit board. This port is then connected to a computer using a UART-to-USB bridge, configured at a baud rate of *57600*. Upon establishing the UART connection, we can capture the system log during the camera running. As revealed in Figure 9, we can discover the vulnerability triggered by the control packet is caused by a page fault, leading to the discovery of a memory corruption.

```
signature:4WSlV29P36LN0I0YgkeN5778HA2SNZfaD4br/PYI8aI=
nonce:eAQ3dn+5Qp8Br8sa
data:
    {"params":
        [
            {
                "did":"1074697170",
                "siid":6,
                "piid":6,
                "value":-216.49537562852015
            }
        ]
    }
```

Figure 8: Vulnerability Discovery Fuzzing Packet Example

```
[assis] WDG CMD FEED DOG!!!!
[670.23] do_page_fault()#2:sending SIGSEGV to miot-serv for invalid read access from
[670.23] 00000000 (epc == 772c6928, ra == 00468a60)
[670.26] jxq03p stream off
[670.44] codec_codec_ctl: set CODEC TURN OFF...
[670.64] codec_codec_ctl: set CODEC TURN OFF
```

Figure 9: Page Fault Log of the "isa.camera.hlc7" Camera

5.3 Effectiveness (RQ2)

To pass the cloud server side verification, we introduce a side-channel-guided fuzzing approach in this paper. We demonstrate the efficacy of this method by comparing it with a simpler approach that mutates the control command arbitrarily to generate fuzzing packets.

In our experimentation, we find there are two types of verification in the cloud server that may prevent the relay of fuzzing packets to the IoT device: one for data type verification and another for validating the value range. The later verification is particularly challenging since we may have to enumerate all potential values to test if they can be successfully transferred to the IoT device.

We now present our experiments on inferring and passing the data type verification. To ensure that packet relay failure is attributed to data type rather than data range mutation, we analyze Xiaomi's documentation to discover the valid value range for each data type. However, these ranges may vary across different control command keys. For this study, we randomly select a range for each data type. We configure the value ranges for "`Int`" ([0,10]), "`Float`" ([0.0, 10.0]), and "`Bool`" (True, Flase). For "`String`", the valid value range is not explicitly stated in the documentation and we limit the length to 4 characters to reduce the possibility of rejection by the cloud server due to excessive length.

For comparison, we separately construct 5,000 distinct fuzzing packets and record their response times for both the raw method without side-channel guidance and `RIoTFuzzer`. To construct these fuzzing packets, we first randomly select a valid control command key, then choose an appropriate data type for this control command key, and finally generate a valid value for the chosen data type. For `RIoTFuzzer`, we adopt the data type inference by sending packets with a specific data type of a certain control command key 10 times to find the valid data type for the specific control command key that can pass the server's verification. We only mutate the data of the valid data type associated with that control command key in fuzzing packet construction. The outcomes of this evaluation are detailed in Table 6, which highlights several critical findings.

Table 6: Effectiveness of Side-Channel-Guided Fuzzing

Platform	#	Total packets	Raw	RIoTFuzzer	Improvement
Xiaomi	1	5000	100.00%	100.00%	0.00%
	2	5000	91.36%	99.34%	8.73%
	3	5000	20.72%	94.34%	355.31%
	4	5000	48.10%	95.08%	97.67%
	5	5000	46.78%	76.04%	62.55%
	6	5000	57.14%	92.58%	62.02%
	7	5000	53.34%	94.74%	77.62%
	8	5000	41.64%	78.34%	88.14%
	9	5000	39.72%	75.48%	90.03%
	10	5000	25.86%	91.18%	252.59%
	11	5000	32.90%	80.46%	144.56%
	12	5000	51.50%	95.94%	86.29%
	13	5000	40.18%	83.42%	107.62%
	14	5000	20.44%	94.56%	362.62%
Jingdong	15	5000	100.00%	100.00%	0.00%
	16	5000	100.00%	100.00%	0.00%
	17	5000	100.00%	100.00%	0.00%
Huawei	18	5000	100.00%	100.00%	0.00%
	19	5000	70.64%	99.34%	40.63%
	20	5000	50.06%	95.52%	90.81%
	21	5000	39.10%	94.40%	141.43%
Tuya	22	5000	100.00%	100.00%	0.00%
	23	5000	100.00%	100.00%	0.00%
	24	5000	100.00%	100.00%	0.00%
	25	5000	100.00%	100.00%	0.00%
	26	5000	100.00%	100.00%	0.00%
	27	5000	100.00%	100.00%	0.00%
Average					76.62%

- **Effectiveness of side-channel-guided fuzzing.** As indicated in Table 6, side-channel-guided fuzzing significantly enhances the effectiveness of fuzzing, resulting in an average improvement of 76.62% and a maximum improvement of 362.62% calculated with the following formula.

  ((* RIoTFuzzer − (* '0F)/(* '0F

- **Diverse verification among IoT platforms.** The evaluation also highlights the variance in cloud server-side verification policies among different IoT platforms. Specifically, for *Jingdong*, we discover the absence of a data type verification in the cloud server, allowing all fuzzing packets to be transmitted to the IoT device via the cloud server.
- **Intra-platform diverse verification.** Our experiments reveal that even within the same IoT platform, such as *Xiaomi*, the policies for cloud server verification vary among different IoT devices. For instance, the bulb ("yeelink.light.color8") can successfully 100% receive fuzzing packets through the cloud server. However, for other IoT devices on the same platform, the fuzzing packets encounter restrictions in passing through the cloud server-side verification.

5.4 Baseline Comparison (RQ3)

To demonstrate the efficiency of RIoTFuzzer, we conduct a comparative analysis against the state-of-the-art blackbox IoT device fuzzing methods. Four existing blackbox fuzzing methods are similar to our work including SNIPUZZ [9], HubFuzzer [21], IoTFuzzer [4], and DIANE [25]. SNIPUZZ relies on gathering the API-testing program of each target IoT device to generate initial seeds for fuzzing, which may not always be accessible [21]. HubFuzzer is a hub-based fuzzer designed to target IoT devices that communicate with the hub using ZigBee or Z-Wave protocols. However, this paper focuses on WiFi-based IoT devices, which are not covered by HubFuzzer. IoTFuzzer, on the other hand, lacks publicly available source code. Meanwhile, DIANE has made significant improvements over IoTFuzzer, addressing some of its limitations. Consequently, DIANE is selected as the baseline for comparison.

We extend DIANE, which was designed for fuzzing IoT devices within the local network, to remotely fuzz the IoT device through the cloud server for comparison. Initially, we configure the companion apps and IoT devices connecting to different networks. We then establish a monitor between the IoT device and the cloud server to detect crashes with abnormal network behavior. We set up another monitor between the controller and the cloud server to capture the network traffic sent from the companion app during the setup phase of the DIANE.

We utilize DIANE to analyze the apps listed in Table 5 and the IoT devices shown in Figure 7. Before fuzzing, DIANE first needs to locate the network packet-sending functions and proper mutation points in the apps. Meanwhile, the RERAN [11] is adopted to replay UI inputs. In our experiments, it fails to identify these functions in the apps of *Xiaomi* and *Tuya* due to crashes during replay UI inputs with RERAN which prevents further fuzzing of the IoT devices controlled by these two apps. For the apps of *Jingdong* and *Huawei*, DIANE can finish the setup phase and the further fuzzing can be performed. After 12 hours of fuzzing of each device, the evaluation results are displayed in the last two columns of Table 4. We have successfully conducted fuzzing on 7 IoT devices using DIANE and fail to discover any vulnerabilities.

After analyzing DIANE's source code and runtime logs, we have pinpointed three main reasons for its failure to detect vulnerabilities in *Jingdong* and *Huawei* IoT devices. Firstly, DIANE fails to correctly identify mutation points due to the incomplete call graph. It backward identifies mutation points starting from message sending functions based on the call graph. Secondly, since DIANE fails to identify the correct mutation points, we further configure the candidate message sending functions as the mutation points for DIANE. However, DIANE fails to generate valid fuzzing packets due to invalid data format and message authentication value in the packet payload. As a result, the fuzzing packets with the altered data will be directly rejected by the cloud server instead of relaying to the IoT device. This prevents DIANE from uncovering vulnerabilities in the IoT device through the cloud server. Third, we try to configure the JAVA interface functions as the mutation points. However, we observe the mini-app sends a JSON-format control command to the JAVA component when controlling the IoT device. The mutation policy of DIANE neglects the significance of the data format and it prevents DIANE from constructing fuzzing packets. These findings show that the existing IoT device blackbox fuzzing methods may exhibit reduced effectiveness in remote fuzzing scenarios.

6 Discussion

Ethical Concerns. We carefully conducted all the experiments to ensure we did not cause any harm to the cloud server or other users and follow the ethical and legal boundaries similar to [22, 36]. First, all the experiments were conducted on our own purchased IoT devices and accounts. Second, following the existing research practice [3, 7, 46], we set a proper time interval to avoid affecting the service of the cloud server and not causing excessive load during the fuzzing. Particularly, fuzzing packets were sent every

Table 7: Comparison of IoT Fuzzing Tools

Fuzzers	Type	Release Time	App Assisted	Firm. Free	Zero-day Detection	Remote Fuzzing
IoTFuzzer [4]	Blackbox	2018	✓	✓	✓	✗
Firm-AFL [47]	Greybox	2019	✗	✗	✓	✗
DIANE [25]	Blackbox	2021	✓	✓	✓	✗
SNIPUZZ [9]	Blackbox	2021	✗	✓	✓	✗
EQUAFL [48]	Greybox	2022	✗	✗	✓	✗
HubFuzzer [21]	Blackbox	2023	✗	✓	✓	✗
Greenhouse [31]	Greybox	2023	✗	✗	✓	✗
RIoTFuzzer	Blackbox	2024	✓	✓	✓	✓

10-15 seconds. Third, all the vulnerabilities we identified have been promptly reported to the respective vendors. We have received acknowledgments from them, who have no issues about our vulnerability discovery. For example, *Tuya* confirms the vulnerabilities discovered in *Tuya* camera (#24) are located within their SDK and have been fixed in the latest SDK.

Threats to Validity. In this paper, we implement RIoTFuzzer and exami6t7zw 0 037.141 82uzzer4 037.141 82uzzerIfo33r. 037.1AItenhgh 037.1wehavn guided fuzzing method to infer the cloud server validation policies and facilitate the construction of fuzzing packets that successfully reach the IoT device. Our evaluation of RIoTFuzzer involved 27 IoT devices from four IoT platforms, resulting in the discovery of 11 vulnerabilities across 10 devices. Furthermore, our evaluation demonstrates that the side-channel guided fuzzing method significantly improves the success rate of delivering fuzzing packets to IoT devices, with an average increase of 76.62% and a maximum increase of 362.62%. Our experiment results highlight the effectiveness and efficiency of RIoTFuzzer.

Acknowledgement

We thank the anonymous reviewers for their valuable suggestions and comments. This research was supported in part by National Natural Science Foundation of China Grant Nos. 62072103 and 62232004, by US National Science Foundation (NSF) Awards 1931871 and 2325451, by Jiangsu Provincial Key R&D Programs Grant Nos. BE2022680 and BE2022065-5, by Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant No. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*. San Francisco, CA, USA, 590–604.

[2] Anthony Desnos. 2012-2024. Androguard. https://github.com/androguard/androguard.

[3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking Security Properties of Cloud Service REST APIs. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification (ICST'20)*. Porto, Portugal, 387–397.

[4] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*. San Diego, California, USA, 1–15.

[5] Tumbleson Connor and Wiśniewski Ryszard. 2024. ApkTool. https://ibotpeaches.github.io/Apktool.

[6] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. 2010–. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/. Version 7.0.

[7] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the 22th USENIX Security Symposium (USENIX Security'13)*. Washington, DC, USA, 605–620.

[8] Eric Sesterhenn and Martin J. Muench. 2015. Bruteforce Exploit Detector. https://github.com/wireghoul/doona.

[9] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *Proceedings of the 28th Conference on Computer and Communications Security (CCS'21)*

In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P'11)*. Berkeley, California, USA, 465–480.

[37] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. 2019. Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. Santa Clara, CA, 1151–1167.

[38] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. 2020. Plug-N-Pwned: Comprehensive Vulnerability Analysis of OBD-II Dongles as A New Over-the-Air Attack Surface in Automotive IoT. In *Proceedings of the 29th USENIX Security Symposium, (USENIX Security'20)*. Virtual Event, 949–965.

[39] Xiaomi Inc. –. Xiaomi Miot Spec. https://home.miot-spec.com/.

[40] Xiaomi Inc. 2023. Xiaomi 2023 Q2 Adjusted Net Profit Surges 147% to RMB5.1 Billion. https://www.mi.com/global/discover/article?id=3008.

[41] Yuqing Yang, Chao Wang, Yue Zhang, and Zhiqiang Lin. 2023. SoK: Decoding the Super App Enigma: The Security Mechanisms, Threats, and Trade-offs in OS-alike Apps. *arXiv preprint arXiv:2306.07495* (2023).

[42] Yuqing Yang, Yue Zhang, and Zhiqiang Lin. 2022. Cross Miniapp Request Forgery: Root Causes, Attacks, and Vulnerability Detection. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*. Los Angeles, CA, USA, 3079–3092.

[43] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* (2024), 100211.

[44] Yuchuan Wang. 2019. JD's IoT Smart Housing Solution Brings the Future of Living to Hundreds of Residential Compounds Across China. https://jdcorporateblog.com/jds-iot-smart-housing-solution-brings-the-future-of-living-to-hundreds-of-residential-compounds-across-china/.

[45] Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. 2021. A Measurement Study of Wechat Mini-Apps. *ACM on Measurement and Analysis of Computing Systems* 5, 2 (2021), 14:1–14:25.

[46] Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Don't Leak Your Keys: Understanding, Measuring, and Exploiting the AppSecret Leaks in Mini-Programs. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*. Copenhagen, Denmark, 2411–2425.

[47] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. Santa Clara, CA, USA, 1099–1114.

[48] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. 2022. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *Proceedings of the 31st International Symposium on Software Testing and Analysis (ISSTA'22)*. Virtual Event, South Korea, 417–428.

[49] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. 2019. Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. Santa Clara, CA, USA, 1133–1150.

[50] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2019. Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps. In *Proceedings of the 26th ConferencelatfyLn Zuo, Haohuang Wen, Zhiqiang Lin,*