

國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

利用應用程式和封包分析引導物聯網黑箱模糊測試

IoT Blackbox Fuzzing Guided by App and Packet Analysis

宋哲寬

Che-Kuan Sung

指導教授：蕭旭君 博士

Advisor: Hsu-Chun Hsiao, Ph.D.

中華民國 112 年 5 月

May, 2023

國立臺灣大學碩士學位論文
口試委員會審定書

利用應用程式和封包分析引導物聯網黑箱模糊測試

IoT Blackbox Fuzzing Guided by App and Packet Analysis

本論文係宋哲寬君（學號 R09944073）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國一百一十二年五月二十五日承下列考試委員審查通過及口試及格，特此證明。

口試委員：

蕭旭君

（簽名）

（指導教授）

田維誠

黃俊穎

黃世昆

所長：

鄭卜壬





Acknowledgements

特別感謝蕭旭君教授的指導，提供了許多想法與建議，讓我獲益良多。也要感謝網路安全實驗室的同学提供的幫助與鼓勵。





摘要

物聯網設備在現今生活中變得普及，引起了安全問題的重要性。模糊測試是一種用於檢測物聯網韌體中安全漏洞的常用技術。在各種類型的模糊測試中，黑盒模糊測試成為相對有效的解決方案，因為它不需要韌體的獲取和仿真。然而，如何生成可以被目標設備接受的有效輸入成為一個關鍵問題。此外，一般的突變策略不適合在保持結構良好的輸入的同時高效突變請求，因為涉及到複雜的數據格式。

在本文中，我們提出了一個名為 APAfuzzer 的自動化黑盒模糊測試框架，旨在克服先前提到的問題。在以往的研究中，實現結構良好的輸入通常涉及繁重的應用程序分析或使用文檔生成種子的方法。然而，這些方法往往難以在效果和自動化之間取得平衡。相比之下，我們的工作利用應用程序生成種子和封包進行突變，還使用勾子函數來攔截加密功能，從而實現了效果和自動化的雙重目標。

我們將 APAfuzzer 與三種最先進的黑盒模糊測試器進行了比較，包括 Diane、Boofuzz 和 Snipuzz。我們在模擬設備和實際設備上對我們的模糊測試器進行了評估。結果顯示，APAfuzzer 能夠觸發已知的 CVE 漏洞，並發現新的漏洞。此外，與其他模糊測試器相比，它表現出更高的效率，並成功觸發更多的漏洞。

關鍵字：模糊測試、物聯網





Abstract

Internet of Things (IoT) devices have become prevalent in nowadays life and bring up the importance of security issues. Fuzzing is a popular technique to detect security vulnerabilities in IoT firmware. Among various types of fuzzing, black-box fuzzing becomes a relatively effective solution because it requires no firmware acquisition and emulation. However, how to generate valid input, which could be accepted for the target devices, becomes a critical problem. In addition, general mutation strategies are not suitable for efficiently mutating the requests while preserving the input structure due to the complex data format.

In this paper, we proposed an automated blackbox fuzzing framework called APAFuzzer to overcome the previously mentioned problems. In previous work, achieving well-structured input often involved heavyweight app analysis or the use of documents to generate seeds. However, these approaches often struggle to strike a balance between effectiveness and automation. In contrast, our work utilizes the app to generate seeds and packets for mutation;

hooking encryption functions, allowing us to achieve both effectiveness and automation.

We compared APAfuzzer to three state-of-the-art black-box fuzzers, i.e., Diane, Boofuzz, and Snipuzz. We evaluated our fuzzer on both emulated devices and real-world devices.

Our results show that APAfuzzer could trigger well-known CVEs and also discover new bugs. Also, compared to other fuzzers, it demonstrates higher efficiency and successfully triggers more vulnerabilities.

Keywords: Fuzzing, IoT



Contents

	Page
Verification Letter from the Oral Examination Committee	i
Acknowledgements	iii
摘要	v
Abstract	vii
Contents	ix
List of Figures	xiii
List of Tables	xv
Chapter 1 Introduction	1
Chapter 2 Background & Related Work	5
2.1 Introduction of IoT devices	5
2.1.1 Importance of well-structured requests in IoT fuzzing	5
2.1.2 IoT communication model	6
2.2 Related work	7
2.2.1 App-based fuzzer	7
2.2.2 API-based fuzzer	8
2.3 Goals and Challenges	9
2.4 Motivation	11



Chapter 3	Design	13
3.1	Target and Assumption	13
3.2	Initial seed creation	14
3.2.1	Proxy creation	14
3.2.2	Encryption	15
3.2.3	Certificate unpinning	15
3.3	App analysis	16
3.3.1	Certificate module hooking	16
3.3.2	Cipher module tracing	17
3.3.3	Keyword extraction	17
3.4	Mutator	18
3.4.1	Request clustering	18
3.4.2	Mutation workflow	19
3.4.3	Encrypted component	19
3.4.4	Mutation strategy	20
3.5	Response monitor	21
3.5.1	Response monitor	21
3.5.2	Crash detection	21
Chapter 4	Implementation and Evaluation	23
4.1	Implementation	24
4.2	Dataset and Environment setup	24
4.2.1	Known bug IoT devices collection	24
4.2.2	Unknown-bug IoT devices collection	25



4.2.3	Environment setup	26
4.2.4	Experiment design	26
4.3	RQ1. Experiment result on emulated devices	27
4.4	RQ2. Experiment result on real-world devices	28
4.5	RQ3. Performance result of each fuzzer	29
Chapter 5	Discussion And Limitations	31
5.1	Discussion	31
5.1.1	APAfuzzer vs. Snipuzz	31
5.1.2	APAfuzzer vs. Diane	32
5.1.3	APAfuzzer vs. Boofuzz	32
5.1.4	Case study: Belkin Wemo smart plug	33
5.2	Limitations	33
5.2.1	Different types of vulnerability	33
5.2.2	Test coverage	34
5.2.3	Custom cryptographic functions	34
5.2.4	Communication mode.	34
5.2.5	Manual interaction.	35
5.3	Quantifying required manual effort	35
Chapter 6	Conclusion	37
	References	39





List of Figures

2.1	IoT device workflow, which shows the importance of well-structured input to trigger function handle	6
2.2	IoT communication model, blue arrow indicates the delegated communication, black arrow indicates the direct communication	7
2.3	Different types of fuzzer's architecture	11
3.1	Architecture of APAfuzzer	14
3.2	Architecture of Proxy creation	16
3.3	Mutation workflow of APAfuzzer	18





List of Tables

2.1	Comparison between two methods	8
3.1	Hooked cryptographic APIs	17
4.1	Summary of emulated IoT devices	24
4.2	Summary of real world devices	25
4.3	Experiment Result. Time of each fuzzer to trigger bugs in real-world devices	28
4.4	Experiment Result. Time of each fuzzer to trigger the CVE of emulated devices	29



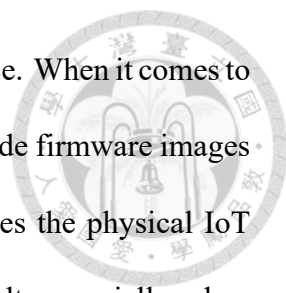


Chapter 1 Introduction

In recent years, Internet of Things (IoT) devices have gained immense popularity, primarily due to their ability to enhance convenience in our daily lives. With more and more IoT devices entering the market, a recent report projected that there will be more than 20 billion active IoT device connections by 2024 [5]. However, the widespread adoption of IoT devices has led to a surge in attack incidents targeting them [1]. Notably, the Mirai botnet and its associated variants infected tens of thousands of vulnerable IoT devices and orchestrated them to launch a series of high-profile Distributed Denial of Service (DDoS) attacks since 2016 [13].

To enhance IoT security, fuzzing has emerged as a powerful automated technique to efficiently identify potential vulnerabilities in IoT devices. A fuzzing tool, or a fuzzer, sends randomly-generated inputs to an IoT device, aiming to detect abnormal behavior that may signify the presence of vulnerabilities. Fuzzing can be broadly categorized into three types based on the level of information available during the target program's execution: whitebox, greybox, and blackbox, each offering unique advantages.

In this work, we focus on the blackbox approach, a widely adopted technique by many IoT fuzzers [3, 7, 11, 16], largely due to its compatibility with a diverse range of IoT device types. In contrast, greybox and whitebox techniques often necessitate firmware



acquisition and emulation, which can present serious barriers in practice. When it comes to firmware acquisition, it is noteworthy that many vendors do not provide firmware images or source code for their IoT devices. Even if an individual possesses the physical IoT device, extracting and unpacking the firmware image can be difficult, especially when dealing with encryption or specific proprietary formats. Moreover, the process of firmware emulation is hindered by the strong dependency of firmware on hardware configurations, making it challenging to emulate the firmware accurately without human intervention.

In order to find out potential vulnerabilities efficiently, IoT fuzzing is an automated technique to find flaws and vulnerabilities in IoT devices. Researchers proposed grey-box techniques to test firmware efficiently [19]. However, those techniques suffer from firmware acquisition and emulation problems. First, most vendors do not provide images or source codes of the IoT firmware. Secondly, unpacking firmware images may be difficult because firmware images might be encrypted or using a specific type of format. Lastly, firmware emulation often encounters the problem of lack of support various types of hardware architecture. Most firmware has strong dependency on hardware configurations, running the firmware correctly usually requires human intervention. Although researchers [6], [9] proposed different techniques to mitigate this problem, these works still could only support specific types of firmware.

For the above reasons, blackbox IoT fuzzing becomes a substitute solution to conquer those emulation problems, and a lot of IoT fuzzers use a blackbox approach [11], [7], [3]. Blackbox fuzzing has relatively higher compatibility compared to greybox ones, as it can effectively test devices even with different types of firmware. It can also avoid the issues associated with emulation, which tends to be slow and error-prone. However, without knowing the protocols of the target devices, fuzzing would be a challenging problem. Be-

cause of the lack of well-structured initial seeds, previous blackbox fuzzing usually needs manual intervention or heavyweight app analysis. Also, most IoT devices do not have a fixed data format, so randomly and blindly mutate the seed could be very inefficient for preserving the right format of the message.

Our methods. We utilize the idea of IoTfuzzer [7] that companion apps contain the information to interact with the IoT devices. However, we try to solve the heavyweight app analysis problem in this framework. We extract the packets from the network communications between the app and the IoT device and use a lightweight method to mutate these packets to trigger bugs. Also, to preserve the same encryption methods of the request messages, we hook the app's cryptographic APIs to collect runtime information. Different from other blackbox IoT fuzzing strategies, our method utilizes the information from app but also uses a lightweight manner to guide the fuzzing.

Contributions. In summary, we make the following contributions:

- We leverage companion-app communications to create well-formatted inputs, enabling effective fuzzing of IoT devices.
- We leverage the app's logic to bypass encryption of the message in order to extract the plaintext for mutation while preserving the message format.
- We propose strategies to detect abnormal behavior of cloud-based IoT devices, which most black-box fuzzers cannot handle.
- The code developed for this work publicly available: <https://github.com/snoopysfriend/APAfuzzer/>





Chapter 2 Background & Related Work

2.1 Introduction of IoT devices

2.1.1 Importance of well-structured requests in IoT fuzzing

In this section, we introduce the importance of well-structured requests and the architecture of IoT systems nowadays. The three main components of an IoT ecosystem are companion apps, IoT devices, and IoT clouds [20]. As depicted in Fig. 2.1, IoT devices usually have three steps to handle incoming requests parsing, handling, and responding. First, the server parses the incoming request; if the format of the request does not match the IoT device's expectation, the request will simply be rejected. Next, the parser would trigger the corresponding function to handle the request. The message inputs would be transferred into parameters for the execution of the target program, which usually is the corresponding CGI. Last, the server handles the execution result of the function and returns the response to the user. This means without the correct format, the request would be dropped by the IoT's parser and cannot trigger deeper bugs in the firmware. Therefore structure-preserving mutation becomes an important goal in our paper.

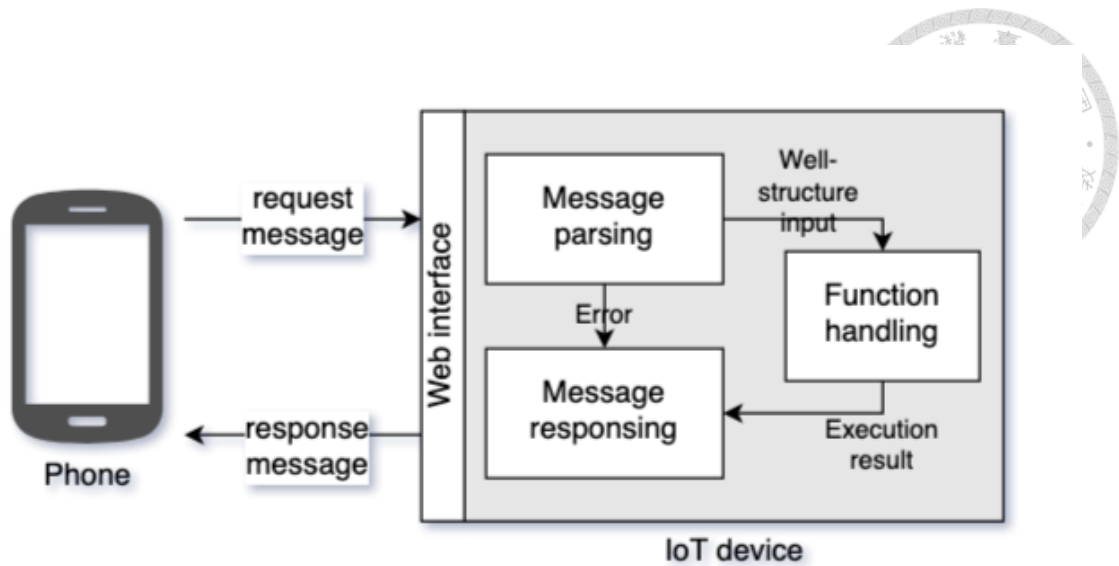


Figure 2.1: IoT device workflow, which shows the importance of well-structured input to trigger function handle

2.1.2 IoT communication model

In this section, we introduce two connection modes of IoT devices, which might influence the design of our fuzzer. As mentioned in the paper of IoTfuzzer [7], the connection mode between the IoT device and the app could be a direct connection or delegated connection. The main difference between the two modes is whether the packets sent out are forwarded by the IoT cloud. As an example in Fig. 2.2, in a direct connection mode, the app sends the control requests to the IoT device directly, which might be using LAN or WAN. In a delegated connection mode, the app communicates with an extra IoT cloud provided by the vendor. The cloud might simply relay the command or support remote device management.

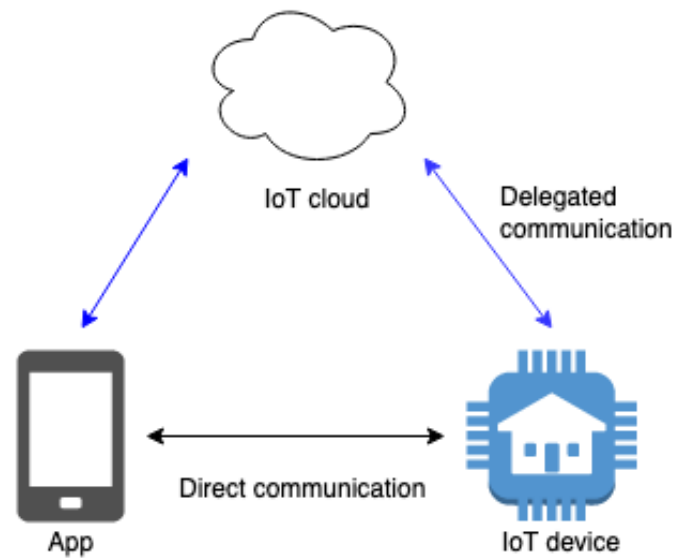


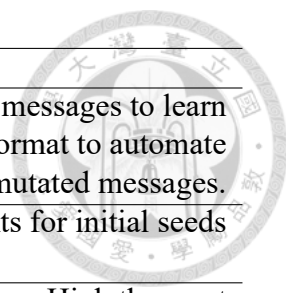
Figure 2.2: IoT communication model, blue arrow indicates the delegated communication, black arrow indicates the direct communication

2.2 Related work

Blackbox IoT fuzzer could be mainly categorized into two types: App-based and API-based. We summarize the two types in Table 2.1 and discuss their methods in detail.

2.2.1 App-based fuzzer

Because most vendors provide companion apps to control IoT devices, app-based fuzzers leverage the app's logic to generate valid messages. IoTfuzzer [7] first proposed a framework which analyzes companion apps' logic and retrieves the functions of message creation. Hooking these functions and mutating the parameters of these functions could generate valid requests to IoT devices. Diane [16] improves the work of IoTfuzzer by reducing the misconfiguration of fuzzing functions. Firstly, each iteration of fuzzing requires triggering a series of UI events, which is often inefficient. This leads to lower



Method	APP-based	API-based
Idea	Mutates the function parameters at app runtime to automate construct mutated requests	Use response messages to learn the format to automate construct mutated messages.
Requirements	Companion app for interacting with IoT devices	API documents for initial seeds
Advantages	Construct well-structured format messages	High throuput
Limits	Slow iteration with UI emulation, Heavy App-reverse on data-flow analysis	Lack of initial seed, Heavy message format analysis
Papers	IoTfuzzer[7], Diane[16]	Snipuzz[10], FuzzDocs[18]

Table 2.1: Comparison between two methods

throughput for each fuzzing iteration, and the functions triggered are limited by the actions of the UI event. Second, because reversing the app’s code is hard work, if the app is obfuscated, the fuzzer might fail to determine the right functions to fuzz.

2.2.2 API-based fuzzer

Snipuzz [10] utilizes response monitoring to guide the mutation strategy of the fuzzer. Snipuzz splits the message into snippets and uses a hierarchical clustering strategy to optimize mutation strategies. However, these methods suffer from valid input creation problems. For example, Snipuzz [10] needs human users to create valid messages by reading API documents or public test suites, which has an average cost of five hours per test case. Fuzzdocs [18] reduces human efforts by leveraging a NLP model to read API documents automatically. However, it is still limited by only being available to those devices that provide API documents. Also, these works only support fuzzing IoT devices that do not encrypt their messages. If the protocol is encrypted, the mutation would be ineffective in finding bugs.



2.3 Goals and Challenges

Our goal is to achieve two main objectives: efficiency and automation. In terms of efficiency, because most greybox IoT fuzzers could not complete one mutation per second, we aim to achieve a mutation rate of at least less than 1 second per mutation. Additionally, in terms of automation, we aim to eliminate the need for heavyweight manual analysis of message values or formats. This section presents the challenges that we might face when achieving these goals.

A running example. To better understand the format of the request messages of IoT devices, we use a message sent by the app of TP-Link Tapo as a running example to understand the challenges of mutating messages. Below is a request message sent by the TP-Link app to turn on the Tapo smart plug. As we can see, this message is using the JSON format containing different protocol fields. However, in this message, we can see that the value of the "params" is encrypted (which is marked as red color).

```
{ request: "securePassthrough",  
  params: "NURj9dr1446nCVUUQYiZAHkLoaZfUnXzV+Gef  
Rs97No9iO131LXaUfULuHooefNLby/  
fD2clQgu+1Hekyxbw          6x1TQ5JZcuppcTIJe0mM/  
yWxbXxZZIKeUfLI/          jpNbJtSAF7  
6S6kE66FmJwKpq5VpQ8J1AsEMTyHNkpePOfHaSsGKEE8  
cK4Cg8Xx1XqNAdh" }
```

And the plaintext message of the red color encrypted message is as below. As we can

see, it performs the task of turning on the device in the field of "params".

```
{ "method": "set_device_info", "params": { "device_on": true }, "requestTimeMils": 1669350405337, "terminalUUID": "6E765FDFB26093E90C9C98742D0E95CB" }
```



In order for the IoT device to correctly decrypt the encrypted message, we should mutate the contents of the message before it is encrypted, not the ciphertext. Hence, ensuring that the generated requests can pass the request parsing of IoT devices by using the same encryption method is one of the challenges to address. It can also be seen that the message has a special format (JSON) and fields. Only by being able to know these formats automatically can we efficiently generate valid test cases.

Challenges. However, our approach has some challenges. Firstly, there may be a lack of initial seeds available for testing. Secondly, when it comes to mutation, maintaining well-structured mutations can be challenging due to the presence of diverse data formats in requests and the use of different encryption algorithms. In summary, we have the following challenges to designing an efficiency and automation fuzzer:

- **Challenge 1: Lack of well-formed initial seeds.** Fuzzer usually mutates the messages to generate various inputs. However, without the knowledge of the protocol or test cases, only relying on mutation to construct the right format is difficult.
- **Challenge 2: Diverse data formats.** Each IoT devices usually use diverse formats such as JSON, XML.... Mutating the right field of the message could preserve the format. On the other hand, mutating the wrong field might be rejected by the IoT device in the request parsing phase.

Challenge 3: Diverse encryption methods Most IoT devices encrypt their mes-

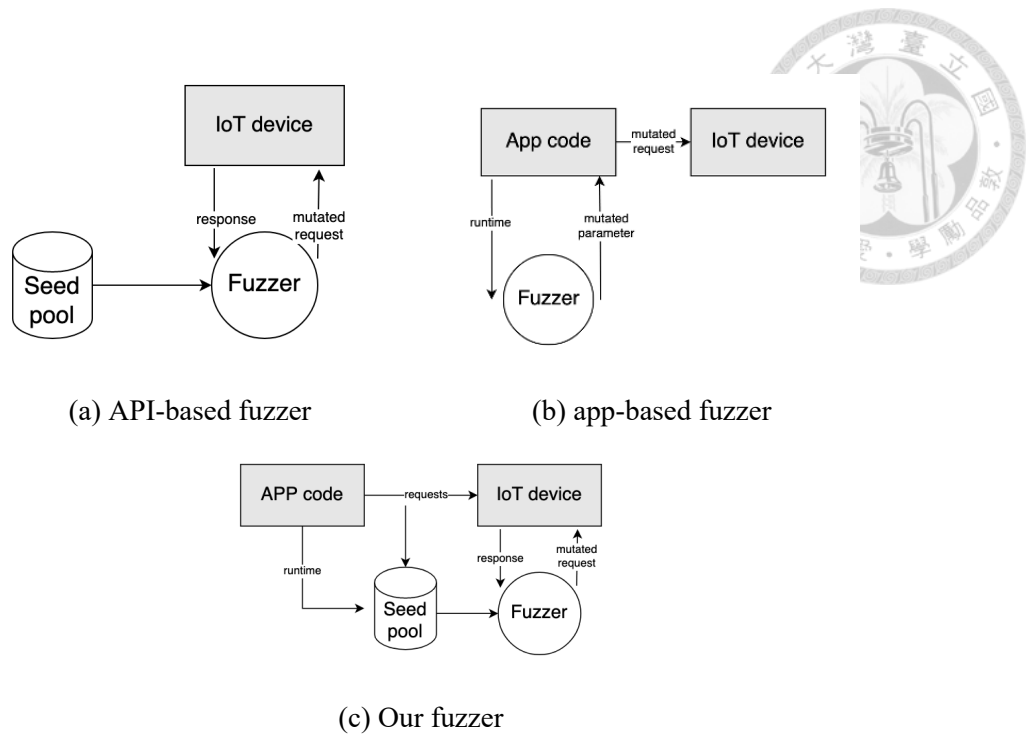


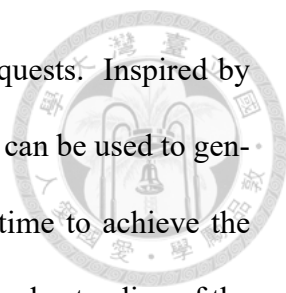
Figure 2.3: Different types of fuzzer's architecture

sages, therefore if we want to mutate the message, the IoT device needs to decrypt the messages and will reject the invalid ones. So we need to mutate the unencrypted message and use the same encryption algorithm to encrypt it, in order to preserve the format.

2.4 Motivation

In this paragraph, we try to solve the challenges mentioned in previous section. First, we analyze the differences between the app-based and API-based fuzzer's architectures. As shown in Figure 2.3 the app-based architecture utilizes the app to generate requests, but the drawback is that it tends to be slower in execution. On the other hand, the API-based architecture generates requests using a seed pool, but the challenge lies in how to effectively generate the seed pool.

Therefore, our idea is to utilize the app to generate the seed pool and then leverage the



API-based mutation approach to rapidly generate well-structured requests. Inspired by the App-based fuzzer concept, we know that the execution of the app can be used to generate valid packets. However, to mutate parameters in the app runtime to achieve the effect of malformed packets, it is often necessary to have a sufficient understanding of the semantics of the apk execution content in order to judge a reasonable mutation function (Diane[16] designs a series of heuristic algorithms to try to solve this problem), but we can retrieve the transmitted content, we can get an effective initial seed. Mutating the content in the packets can void the heavyweight app analysis. We have also seen many techniques on other API-based fuzzers that can effectively fuzz. Also, we used the assumption of the app using official encryption APIs. When faced with the encryption of the protocol, we only need to hook some official functions. In order to get the dynamic runtime encryption information, then our fuzzer could utilize that information automatically to decrypt the encrypted message



Chapter 3 Design

This section presents the detailed design of our fuzzer as illustrated in Fig. 3.1. Our fuzzer mainly works in two phases, Pre-analysis and Fuzzing. In the Pre-analysis phase, a network proxy is established to obtain the communication information between the app and IoT devices. In addition, this module analyzes the encryption algorithms used by the app to obtain useful runtime information. After this phase, our fuzzer will get the valid initial seed and runtime information for mutation.

In the Fuzzing phase, our fuzzer will split each request into different tokens and use the runtime information to mutate tokens with different mutation strategies. Later, the fuzzer monitors the responses from the devices to determine if a memory corruption-related bug has been triggered.

3.1 Target and Assumption

First, we present the target IoT devices our fuzzer could handle. Our target devices are those IoT devices that provide companion apps for users to manipulate them, which is a common assumption nowadays. In our work, we use Android apps for research due to their openness. Our fuzzer could fuzz the devices which use direct connection or cloud-based connections. However, in our assumption, the IoT cloud only checks the format

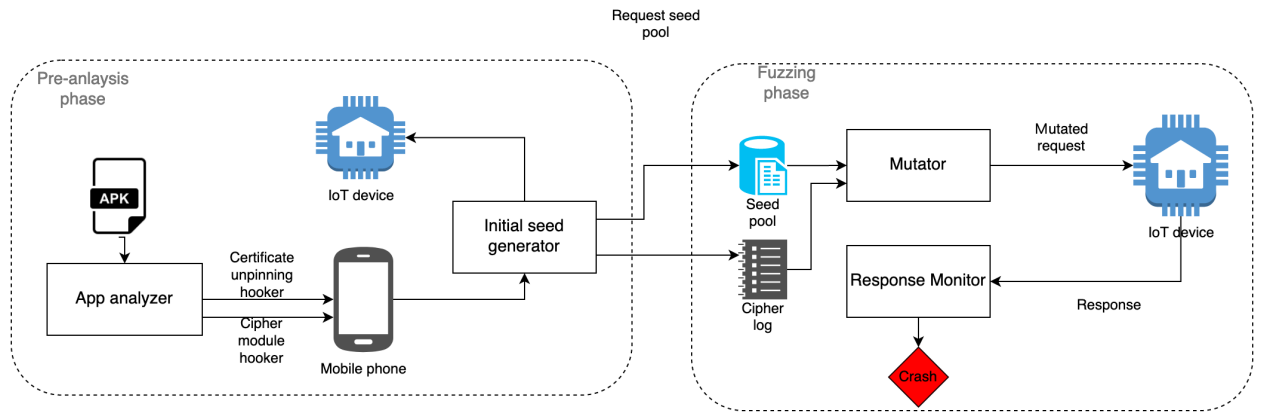


Figure 3.1: Architecture of APAfuzzer

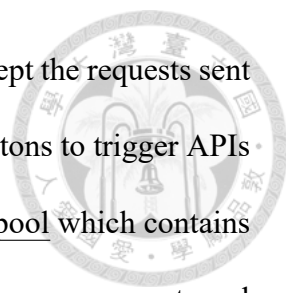
of the request and relays it to the device. The cloud does not read the semantics of the request, so the fuzzing target could still be our IoT devices. According to the paper [17], more than 76.3% of devices support a peer-to-peer connection which shows that it is a common assumption.

3.2 Initial seed creation

At this stage, we design and establish a proxy between the mobile phone and the IoT device to obtain valid requests. After obtaining the requests, we could use them as the initial seed for the fuzzing phase.

3.2.1 Proxy creation

Firstly, we manually set up the IoT device pairing with our phone, such as registering or logging into an online account. After, we created a proxy to intercept the requests sent by phone to the IoT device. A proxy is created to relay and record the network traffic between the cellphone and the IoT device as depicted in Figure 3.2. By setting the proxy



in the cellphone's Wifi to the IP address of our proxy, we could intercept the requests sent by the phone and record them. Then the user presses the app's UI buttons to trigger APIs from the app. The recorded requests would be collected in a request pool which contains all different requests sent from the phone. Different from other works, our request pool not only contains a specific type of UI operations but all the different types of APIs.

3.2.2 Encryption

In order to protect the privacy of communications, most apps use TLS/SSL to encrypt network traffic. We need to decrypt them in order to access the contents of the communication. We implanted a certificate into the user's phone to launch a MITM (man-in-the-middle) attack to decipher the encrypted messages. With the self-signed certificate defined by our proxy is stored in "/system/etc/security", the traffic between our phone and IoT could be decrypted. We implemented our proxy upon the framework mitmproxy [8] using the Python plugin.

3.2.3 Certificate unpinning

However, after a series of experiments, we found that most apps will do certificate pinning (public key pinning). These apps only tacitly recognize the certificate signed by the root certificate in Network Security Configuration. This means they would not trust a new certificate that they didn't recognize before. So our implanted certificate could not bypass the certificate authenticated module. To bypass this, we use the dynamic hooking tool Frida [4] to override the certificate module to bypass certificate pinning dynamically. Our Frida script hooks all certificate modules and makes all the certificate-checking meth-

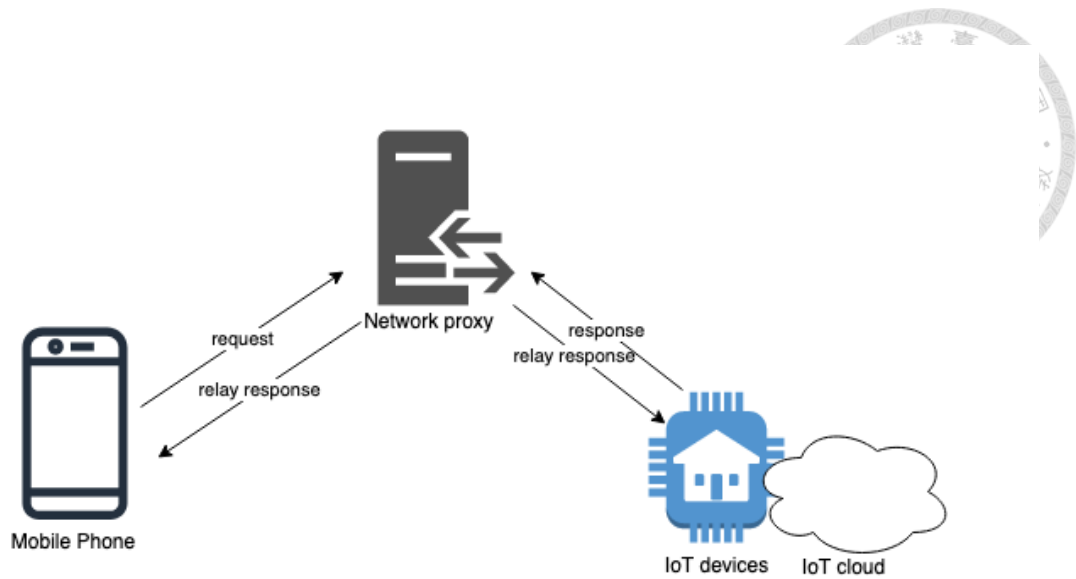


Figure 3.2: Architecture of Proxy creation

ods do nothing.

3.3 App analysis

In order to obtain the app's logic for constructing and encrypting the message, we hook some important Java standard library modules.

3.3.1 Certificate module hooking

As mentioned in Section 3.2.2, in order to bypass the certificate pinning, we would hook well-known certificate modules in the app and replace them with our self-defined functions. In this way, the original function of checking the certificate in the apk will be replaced, and let our self-defined certificate work to achieve the effect of MITM. In this way, it becomes possible to successfully decrypt the encrypted protocol and obtain the the plaintext of outgoing requests.



3.3.2 Cipher module tracing

API name	API function
<code>javax.crypto.init</code>	init the cipher module
<code>javax.crypto.update</code>	update the key
<code>javax.crypto.getInstance</code>	initial key or iv
<code>javax.crypto.doFinal</code>	encrypt or decrypt

Table 3.1: Hooked cryptographic APIs

As shown in the running example in Section 2.3, some of the fields of the message might be encrypted but not on the protocol itself. Here, we refer to the idea of the work of AutoForge [21] and try to make a cryptographic-consistent message on the fuzzer by reimplementing cryptographic functions out of the box. To mitigate this, try to dynamically record the runtime information that the app uses to encrypt messages so that the information before encryption can be restored. we assume the author of the app complies with the rule of not using homemade cryptographic functions. So the encryption functions could be detected by hooking the Java standard library (`javax.crypto.cipher`). We dynamically hook the cryptographic APIs and record the parameters of these functions. With those parameters, we could rebuild the cryptographic functions out-of-box and replay them with arbitrary values in order to generate new valid-encrypted messages (the details explained in Section 3-C). The details of the hooked APIs are listed in Table 3.1;

3.3.3 Keyword extraction

In our finding, because the app utilizes special strings during runtime to construct valid requests, these strings as primarily found within the parameters of the request, making them suitable as units for mutation. We could assume that the message contains words from the apps. We use the tool `strings` to extract the words from apps to build a dictionary

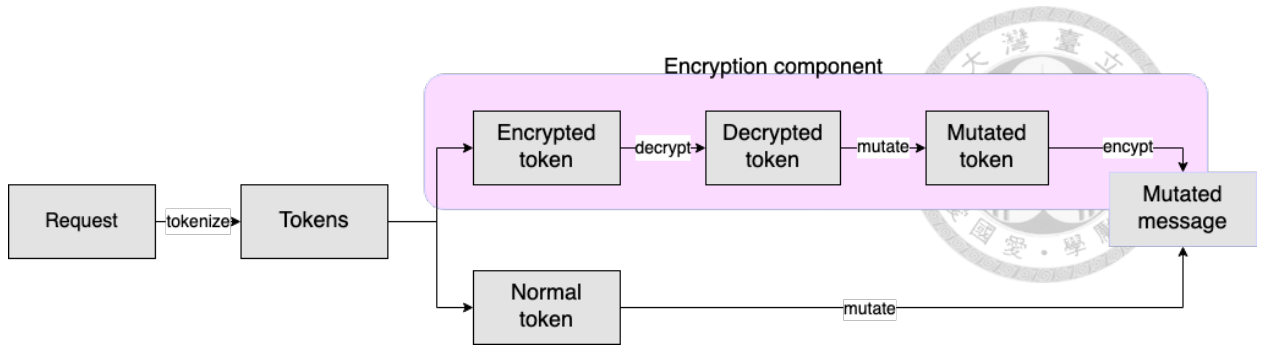


Figure 3.3: Mutation workflow of APAfuzzer

that helps us tokenize our message in Section 3-C.

3.4 Mutator

With the Pre-analysis step, we acquired the plaintext content from the request messages. In this step, we split the content of each request into tokens and mutate each token with different mutation strategies.

3.4.1 Request clustering

In our framework, we aim to gather as many requests as possible during the pre-analysis phase. Unlike other blackbox fuzzers that typically use an action as an iteration of fuzzing [16] [7], we adopt a different approach. In each iteration of our fuzzing process, we select a request from our request pool. However, we encounter numerous requests that are either identical or very similar in nature. Randomly picking requests may result in an excessive number of attempts on common functions. To address this, we propose clustering the requests to ensure equal chances for different functions to be executed. We employ the edit distance to calculate the similarity score between each request. If a request exhibits high similarity with a cluster, it is added to that cluster. Conversely, if a request significantly differs from others, a new cluster is created specifically for that request.



3.4.2 Mutation workflow

Our workflow of mutation is depicted as Figure 3.3. To segment the content of each request, we will utilize the dictionary obtained through static analysis. Initially, we compare the longer strings in the dictionary to check if they match any content in the request. If a match is found, the corresponding string is extracted as a token. If there are no matches, we later compare strings that are exactly the same. Subsequently, the tokens are classified into encrypted or normal tokens. To identify encrypted tokens, we leverage information from the "cipher module tracing." Specifically, we extract the encrypted tokens using the Android Cipher API's doFinal function, which performs encryption or decryption on a byte array. By comparing the doFinal function result from the cipher module log with the content in each request, we determine if a token is encrypted. If there is a match, the token is classified as an encrypted token. These two types of tokens will undergo different mutation methods. Encrypted tokens will be subjected to encryption and decryption using the Encryption component, which will be elaborated upon in the next subsection.

3.4.3 Encrypted component


In order to mutate the encrypted tokens, our fuzzer would try to run the same cryptographic function in our fuzzer for reproducing the encrypted result. We would parse the cipher API's log in the previous phase and pass the parameters of these APIs (such as init, getInstance, update) to the Python crypto module. First, if the result of the doFinal function is matched with the content in request, we would mark this cipher object as interested. Then, we would backtrace this cipher object. For those functions that init or update the cipher runtime object, the arguments would record. After the backtrace, we know how the

cipher object should be initiated i.e., algorithm, key, value, block mode...), then we would initiate it with the version of Python object. This object would perform encryption when we need to mutate the token in the mutation stage. Taking the below diagram as an example, when we find that the result of doFinal matches our encrypted tokens, we will begin to backtrace the corresponding object, which is `javax.crypto.Cipher@426c4d4`. Then, we will search for the functions that use this object instance, which are `getInstance` and `init`. Here, we will record the algorithm used (AES) and the corresponding key value,

```
[Cipher.getInstance():type:AES/CBC/PKCS7Padding, cipherObj: javax.crypto.Cipher@426c4d4  
  
[Cipher.getInstance(): type:AES/CBC/PKCS7Padding,  
cipherObj:javax.crypto.Cipher@472967d  
  
[Cipher.init():mode:Encrypt,key_val:[34,75,56,7,62,-61...],  
cipherObj: javax.crypto.Cipher@426c4d4  
  
....  
  
....  
  
[Cipher.doFinal():cipherObj:javax.crypto.Cipher@426c4d4
```

3.4.4 Mutation strategy

In each iteration, we will randomly select the token for mutation of the content of the request. For our mutation strategy, we refer to the method similar to Diane[16]’s work to achieve the effect of triggering memory corruption-type bugs. We use the following strategies:

- 
- **Numerical values** Use the boundary value of int, long, or float to trigger the case of boundary conditions.
 - **Empty value** Empty values usually trigger a null pointer dereference bug.
 - **Random value** Random size of strings might trigger a buffer overflow bug or other bugs.
 - **Predefined value** Use some word in the dictionary built from the Pre-analysis phase to replace the token.

3.5 Response monitor

In this step, we cluster the response messages from IoT devices in order to detect abnormal behavior. In our model, we use the techniques that black-box fuzzing usually adopts.

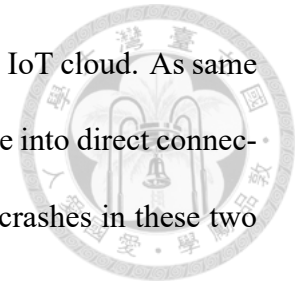
3.5.1 Response monitor

During this step, we will record all the responses received, similar to how we handled the request cluster, and categorize them into different types. Whenever the mutated request generates a new response, we will consider it as a new seed and add it to the seed pool for future use.

3.5.2 Crash detection

After sending the mutated request to the IoT devices, we need to detect whether the mutated value triggers a bug. Although previous work usually use response monitors to

detect bugs. However, they do not handle the model of containing the IoT cloud. As same as the assumption in IoTfuzzer [7], we could categorize the IoT device into direct connection and delegated connection. We use different strategies to detect crashes in these two different cases.



Direct connection. We monitor the response to our requests to determine whether our input triggers potential bugs. If the response from the IoT devices satisfies one of the following conditions, our fuzzer will issue an alert.

- **Connection dropped** If the connection dropped from the IoT device, the device might be corrupted.
- **HTTP server internal server error** The 500 error code means the server encountered an error during the execution, which we could consider the device has triggered a bug.
- **Irregular response time** If the response time of the message exceeds the average response time of variance, the IoT device might be handling some faulty state.

Delegated connection. Simply using the response of the IoT cloud to determine the status IoT device may be wrong because the response from the IoT cloud may not represent the status of the IoT cloud. We solve this problem by changing the monitor target to the network communication between the IoT cloud and IoT device. We use a heartbeat message to trigger a normal request from the IoT cloud to the device. Then, we run **tcpdump** on the router that the IoT device is connected to and dump the network logs from the IoT devices. Then we analyze whether the network communication is regular or not, if there is no response from the device, we could consider the device is offline.



Chapter 4 Implementation and Evaluation

In this section, we present the implementation details of our fuzzer and the experiment results. We measure our fuzzer with the following three research questions

RQ1 Can our fuzzer find known bugs in IoT devices?

RQ2 Can our fuzzer find unknown bugs in IoT devices?

RQ3 Is our fuzzer efficient compared with other state-of-the-art fuzzers?

To answer the first research question, we run our fuzzer on four different emulated devices which contain previously known vulnerabilities. Our fuzzer triggered all the bugs, in all cases, for less than one hour.

In order to address our second research question, we employed our fuzzer to test five different real-world devices, resulting in the discovery of previously unidentified bugs in one of them.

To address the third research question, we executed various other state-of-the-art

Device Id	CVE Id	Device Type	Vendor	Model	Firmware Version
1	CVE-2019-20082	Router	Asus	RT-N53	3.0.0.4.376.3754
2	CVE-2019-11400	Router	Trendnet	TEW-642BRU	1.00.b12
3	CVE-2018-19240	IP camera	Trendnet	TV-IP121WN	V1.2.2.28
4	CVE-2016-1558	Router	Dlink	DAP2695-	v111-rc044

Table 4.1: Summary of emulated IoT devices

fuzzers on the aforementioned devices. We then evaluated and compared the time each fuzzer took to detect bugs, as well as their respective required pre-processing time for the purpose of fuzzing.

4.1 Implementation

Our fuzzer was developed using approximately 3000 lines of Python code and 2000 lines of JavaScript code. To set up the proxy, we used mitmproxy [8] which enabled us to record the requests and replay them after mutating them. Additionally, we employed Frida [4] to perform dynamic runtime hooking on Android apps, which allowed us to record the hooked APIs.

4.2 Dataset and Environment setup

4.2.1 Known bug IoT devices collection

We gathered several CVEs related to the target devices for the purpose of conducting fuzz testing. However, as most of these CVEs were related to outdated devices in the market, and downgrading their firmware versions was difficult, we choose to use emulated IoT devices to test for known bugs on these CVEs. We use FirmAE [12] to emulate the firmware with a specific version. We collect 4 different devices from their official



Device ID	1	2	3
Device type	Smart plug	Smart plug	Bulb
Vendor	Belkin Wemo	TP-Link	TP-Link
Model	WSP080	Tapo P125	Tapo L530E
Firmware version	4.00.20081009	1.0.5	1.0.9
Format	HTTPs + XML	HTTP + JSON	HTTP + JSON
Has cloud	yes	both	both
Encryption token	no	yes	yes

Device ID	4	5
Device type	Router	NAS
Vendor	Xiaomi	Synology
Model	AX6000	DS418
Firmware version	3.0.0.4.385 20490	7.1-42661
Format	HTTP + URLencoded	HTTP + JSON
Has cloud	no	no
Encryption token	no	no

Table 4.2: Summary of real world devices

websites, and each device could be simulated successfully and accessed from the web interface. We also run several docker containers to test devices parallelly. The detailed information on these devices is described in Table 4.1

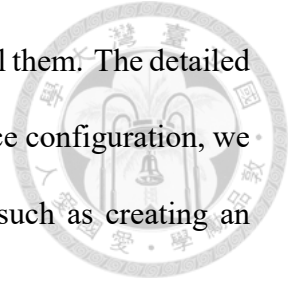
The emulated devices could not connect to the app, so we interact with the browser. In order to collect the network requests, the proxy is set up on the browser which interacts with the emulated devices.

We defined a vulnerability request as the request which would trigger the vulnerable function in the request pool.

4.2.2 Unknown-bug IoT devices collection

We collected different types of IoT devices (smart plugs, bulbs, routers, NAS) and vendors. For unknown-bug devices, we bought five different devices which are similar to

previous papers. All the above devices have companion apps to control them. The detailed information on these devices is described in Table 4.2. For each device configuration, we carried out the initial setup phase manually, which involved tasks such as creating an account on the device and on the Android companion app.



4.2.3 Environment setup

Our fuzzer runs on an Ubuntu Server 22.04 LTS with an AMD Ryzen 9 5950X Processor (16 cores) and 128GB of RAM, and the IoT apps run on ASUS_X01AD with Android 8.1.0.

All of our IoT devices are connected to a router same as with the phone and our fuzzer.

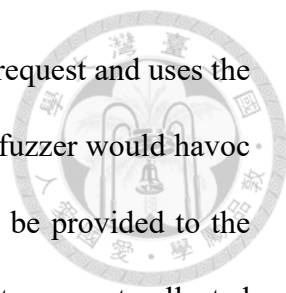
4.2.4 Experiment design

We compared our fuzzer with several state-of-the-art black-box IoT fuzzer tools. We will briefly introduce the core ideas and settings of each fuzzer.

Two modes In order to enrich the benchmark evaluation, we run two modes in these fuzzers.

- **Fuzzer-One.** In this mode, the request pool only contains the vulnerability request.
- **Fuzzer-N.** Compared with **Fuzzer-One.**, the request pool contains all the requests we collected from the pre-analysis step.

Snipuzz [10]. The core idea of Snipuzz is using responses to segment messages into



snippets. In the pre-analysis step of Snipuzz, it flips each byte in the request and uses the response to cluster each byte into snippets. In the fuzzing phase, the fuzzer would havoc the request in the unit of the snippet. However, the messages must be provided to the fuzzer manually, we write a Python script to translate the vulnerability request collected by our fuzzer to the format of Snipuzz.

Boofuzz [3]. Boofuzz is a fuzzer for network-level protocols, which is an improved version of the Sulley[11] framework. The main approach of Boofuzz involves segmenting the message with human guidance, in order to mutate it. For our experiment, we wrote a Python script to convert each request into Boofuzz's code, with the strategy of segmenting each header of the request for mutation. We also set the entire content of the request as one segment of Boofuzz. In this mode, Boofuzz would mutate the entire content of the request.

Diane [16]. Diane is based on the concept of IoTfuzzer and improves the strategy to find the fuzzed functions from apps. The main strategy of Diane is hooking the function triggers which construct the message sent by the app to IoT devices. Diane uses dynamic dataflow analysis to determine the functions which might influence the value of triggering network events. After hooking the function triggers, Diane mutated the arguments in runtime to construct mutated requests to the devices.

4.3 RQ1. Experiment result on emulated devices

We prepared four emulated devices with N-days vulnerabilities. After our fuzzer ran on each device for 12 hours, our fuzzer successfully triggered the CVE of each device.



Device ID	APAfuzzer-N	Diane	BooFuzz-N	Snipuzz-N
1	78.35s	NA	NA	NA
2	>12hr	NA	>12hr	>12hr
3	>12hr	NA	>12hr	>12hr
4	>12hr	>12hr	>12hr	>12hr
5	>12hr	>12hr	>12hr	>12hr

Table 4.3: Experiment Result. Time of each fuzzer to trigger bugs in real-world devices

We observed malformed requests that trigger these device crashes. In these requests, our fuzzer will trigger different kinds of memory bugs with different special values (empty, oversize length) while keeping the format correct. Our fuzzer outperformed other fuzzers in terms of identifying known bugs. Specifically, it was able to trigger four bugs successfully, while Diane was unable to establish contact with the emulated device. Snipuzz failed to trigger any bugs, and boofuzz was able to crash a device, but our fuzzer performed better overall. Therefore, it can be concluded that our fuzzer is highly effective in detecting known bugs.

4.4 RQ2. Experiment result on real-world devices

Table 4.3 presents the results of a 12-hour fuzzing test conducted on several real-world devices. Our fuzzer was the only tool to discover new crashes during this test, while state-of-the-art tools failed to trigger different bugs for various reasons. For the device use cloud relay mode (Device 1), the other fuzzers did not support it, so we marked NA in the experiment column. For more analysis of the experimental results, we will discuss why our fuzzer successfully finds bugs and the behavior of other fuzzers in the subsequent Section.

We communicated our discoveries to the relevant manufacturers, and as far as we know, all identified issues have been addressed.

CVE-ID	APAfuzzer-One	APAfuzzer-N	Snipuzz-One	Snipuzz-N
CVE-2019-20082	11.67s / 44.66s	12.04s / 1m50.72s	22m23.51s / >12hr	319m42.09s / >12hr
CVE-2019-11400	11.90s / 51.43s	12.17s / 3m20.03s	346m33.84s / >12hr	682m12.01s / >12hr
CVE-2018-19240	0.79s / 1m21.38s	0.79s / 19m3.26s	30m47.02s / >12hr	272m36.79s / >12hr
CVE-2016-1558	0.10s / 7m52.62s	0.12s / 52m40.17s	10m57.12s / >12hr	120m12.12s / >12hr

CVE-ID	Boofuzz-One	Boofuzz-N
CVE-2019-20082	NA / 468m41.91s	NA / >12hr
CVE-2019-11400	NA / 6m35.42s	NA / 16m0.07s
CVE-2018-19240	NA / >12hr	NA / >12hr
CVE-2016-1558	NA / >12hr	NA / >12hr

Table 4.4: Experiment Result. Time of each fuzzer to trigger the CVE of emulated devices

4.5 RQ3. Performance result of each fuzzer

We are interested in the performance of our fuzzer in finding known bugs compared with other fuzzers. Our focus is on determining the speed at which each fuzzer detects bugs, which we achieve by recording the time when bugs are triggered by various fuzzers. Additionally, we also record the duration of the pre-processing time for each fuzzer, as it serves as a performance metric for the overall runtime of the fuzzer.

Table 4.4 shows the time to trigger the specific vulnerability in each emulated device. The first number in the column is the pre-processing time of each fuzzer. The second number is the time of fuzzing to trigger the bug. It shows that compared with other fuzzers, APAfuzzer discovers the highest number of bugs and triggers them in the shortest amount of time. Our observations revealed that Snipuzz had a significantly longer pre-processing time compared to our fuzzer. In contrast, our fuzzer leverages the app information to rapidly identify tokens that are susceptible to mutation. Our approach saves a significant amount of time compared to Snipuzz’s method of flipping each byte, observing the response, and categorizing related snippets.





Chapter 5 Discussion And Limitations

This section will begin with a discussion of how APAfuzzer differs from other fuzzers in terms of its effectiveness in bug discovery. We will also present a case study that highlights the distinctive features of our fuzzer. In the second part, we will mention the limitations of our fuzzer and provide recommendations on how to tackle these issues.

5.1 Discussion

5.1.1 APAfuzzer vs. Snipuzz

We identify two reasons, why Snipuzz could not trigger any bugs. First, the probing time of the fuzzer is too long, each time the fuzzer found a new interesting seed, the fuzzer would need to probe the seed to snippets again. Second, the mutation strategy of Snipuzz is too conservative for triggering overflows bugs. Most of the bugs are memory corruption bugs, and triggering them usually needs an overlength value in the correct places. However, we trace the source code of their fuzzer and found out that their mutator only supports the strategies such as bitflip, empty, repeat, random, and dictionary value. The random strategy in Snipuzz only changes each byte to a random value which is still a fixed length value.



5.1.2 APAfuzzer vs. Diane

Our work differs from Diane in several aspects. Firstly, Diane requires the user to select a UI event for each iteration of the fuzzer, while our fuzzer does not. Also the UI event is very heavy-weight, which means that our fuzzer can complete mutations much faster, with an average runtime of 0.79s, compared to Diane's average runtime of 6.16s per mutation.

Moreover, the static analysis tool used by Diane cannot be effectively built on Device 3. When establishing dataflow-flow analysis between classes, it often crashes and cannot run smoothly, whereas our fuzzer does not encounter such problem.

In addition, during our experiments, we observed that Diane often encountered hooking issues and was not as reliable as our fuzzer. When there are too many functions that require hooking, Diane often causes the apk to crash and must be restarted, which is one of the reasons for slowing down runtime performance

5.1.3 APAfuzzer vs. Boofuzz

Although boofuzz is a general-purpose black-box fuzzer, we provide the same information for it as our fuzzer. We also simply split the header and content for it but not the content. Boofuzz can quickly generate a lot of requests to the target device. However, it could only blindly mutate which is very hard to preserve the structure of the request. Boofuzz successfully crashed Device 2 in the emulated device. We will explore why Boofuzz only succeeds on this particular device and fails to trigger bugs on other devices. We observed that the requests successfully triggered on device 2 are relatively shorter com-

pared to other devices. Perhaps, for Boofuzz, shorter content mutations maintain a better well-structured request.



5.1.4 Case study: Belkin Wemo smart plug

The Belkin WSP080 model is a smart plug that could be scheduled to turn on/off by the app. The connection mode of Belkin plug is using the delegated mode. The vendor of Belkin provides public API for a user to control their devices remotely on and off. We found that both the IoT cloud and device do not check the length of the macAddress in the XML. If the length of the macAddress exceeds a specific value, our fuzzer could trigger a crash for the plug which caused the plug offline and could not be accessed by normal requests. In this case, all the requests between the phone and IoT device are forwarded via IoT cloud. So even if the device is offline we still could not judge the status of the IoT device from the network status returned by the cloud. As a result, we monitor the network between the IoT cloud and device and we could detect the connection dropped with abnormal behavior.

5.2 Limitations

5.2.1 Different types of vulnerability

Our fuzzer still focuses on finding memory corruption bugs. However, many IoT devices suffered from different vulnerabilities, such as authentication bypassing or command injection. Using blackbox fuzzing to trigger or detect different types of vulnerabilities can be a challenging task. After all, some vulnerabilities often require observing the runtime

state of the target IoT device to detect them, such as command injection.



5.2.2 Test coverage

Like other App-based fuzzer, our fuzzer could only fuzz the IoT device's network interface opened to the app. Other interfaces or services might not be tested due to the limitation of coverage. How to find out the other services or inputs is an open problem in blackbox fuzzing. Furthermore, different levels of test coverage have been proven effective as the basis for mutations in blackbox fuzzing, as demonstrated in several research papers [15]. By observing whether inputs and outputs achieve different combinations, we can determine whether they should be saved as new seeds.

5.2.3 Custom cryptographic functions

In our assumption, the app only uses the well-known cryptographic API provided by Java. However, if the app uses some homemade function to encrypt the message, our framework could not detect it. We suggest that integrating some heuristic algorithms in detecting encryption algorithms might solve this problem.

5.2.4 Communication mode.

We only support the target IoT devices using the Internet. However, some IoT devices use a different mode of communication such as Bluetooth or Zigbee to communicate

with the user. In the future, we wish to collect those different types of messages by using snoop log from Android devices. We left this work in our future work.



5.2.5 Manual interaction.

In our fuzzer, in the phase of collection requests, we would ask the user to use the phone to interact with the device to trigger as much functionality as possible. However, we suggest that this step could be replaced by Monkey [2] or some automated testing model such as the work of QTypist [14].

5.3 Quantifying required manual effort

The amount of human effort needed to utilize our fuzzer was assessed, and it was found that the fuzzer's workload increases proportionally with the number of devices analyzed. Because the user must repeat the same tasks for each new device analyzed. In our work, We have to manually set up each IoT device. During this stage, the user is required to download the corresponding app and usually create an online account as well. Later, after the proxy is created between the phone and IoT device, the user should control the app to collect requests. The user should try all the UI interactions with the devices. To enhance reproducibility in the proxy creation phase and ensure that the mobile device's behavior is not influenced by the user, we employ a depth-first search (DFS) approach to systematically test each button on the screen. By attempting to operate the screens in a fixed order, we aim to minimize the impact of human factors on the experimental results. Also, the average time we spent on each device was about 15 mins.





Chapter 6 Conclusion

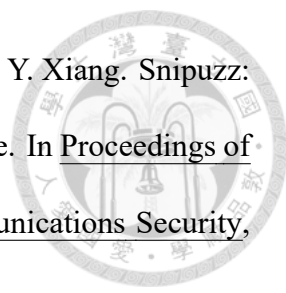
In this paper, we provide a framework for an IoT black-box fuzzer. Unlike other blackbox fuzzers, we provide a method to collect network packets that can be accepted by the device for constructing well-structured seeds automatically. And we also use the additional information of the apk to guide the mutation to conform to the structure that the packet should have, including the encrypted form or data field. We used four known-bug emulated devices and five real-world devices to evaluate our fuzzer. We successfully found one new bug and triggered all the known bugs.






References

- [1] 2017 was 'worst year ever' in data breaches and cyberattacks, thanks to ransomware.
- [2] Android developers. 2017. ui/application exerciser monkey. (september 2017).
- [3] boofuzz: Network protocol fuzzing for humans.
- [4] Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.
- [5] Iot market by component.
- [6] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. In NDSS, volume 1, pages 1–1, 2016.
- [7] 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 NDSS, 2018.
- [8] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 9.0].
- [9] B. Feng, A. Mera, and L. Lu. {P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In 29th USENIX Security Symposium (USENIX Security 20), pages 1237–1254, 2020.

- 
- [10] 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, pages 337–350, 2021.
- [11] Fitblip. Sulley - a pure-python fully automated and unattended fuzzing framework.
- [12] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In Annual computer security applications conference, pages 733–745, 2020.
- [13] C. Koliass, G. Kambourakis, A. Stavrou, and J. Voas. Ddos in the iot: Mirai and other botnets. Computer, 50(7):80–84, 2017.
- [14] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang. Fill in the blank: Context-aware automated text input generation for mobile gui testing. arXiv preprint arXiv:2212.04732, 2022.
- [15] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. Test coverage criteria for restful web apis. In Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, pages 15–21, 2019.
- [16] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In 2021 IEEE Symposium on Security and Privacy (SP), pages 484–500. IEEE, 2021.
- [17] X. Wang, Y. Sun, S. Nanda, and X. Wang. Looking from the mirror: Evaluating {IoT} device security through mobile companion apps. In 28th USENIX Security Symposium (USENIX Security 19), pages 1151–1167, 2019.

- 
- [18] M. You, Y. Kim, J. Kim, M. Seo, S. Son, S. Shin, and S. Lee. Fuzzdocs: An automated security evaluation framework for iot. IEEE Access, 2022.
- [19] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In 28th USENIX Security Symposium (USENIX Security 19), pages 1099–1114, 2019.
- [20] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang. Discovering and understanding the security hazards in the interactions between {IoT} devices, mobile apps, and clouds on smart home platforms. In 28th USENIX security symposium (USENIX security 19), pages 1133–1150, 2019.
- [21] C. Zuo, W. Wang, Z. Lin, and R. Wang. Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services. In NDSS, 2016.