SNIPUZZ: Black-box Fuzzing of IoT Firmware via Message Snippet Inference

Xiaotao Feng
Swinburne University of Technology
Australia

Ruoxi Sun
The University of Adelaide
Australia

Xiaogang Zhu*
Swinburne University of Technology
Australia

Minhui Xue
The University of Adelaide
Australia

Sheng Wen*
Swinburne University of Technology
Australia

Dongxi Liu
CSIRO Data61
Australia

Surya Nepal
CSIRO Data61
Australia

Yang Xiang
Swinburne University of Technology
Australia

ABSTRACT

The proliferation of Internet of Things (IoT) devices has made people's lives more convenient, but it has also raised many security concerns. Due to the difficulty of obtaining and emulating IoT firmware, in the absence of internal execution information, black-box fuzzing of IoT devices has become a viable option. However, existing black-box fuzzers cannot form effective mutation optimization mechanisms to guide their testing processes, mainly due to the lack of feedback. In addition, because of the prevalent use of various and non-standard communication message formats in IoT devices, it is difficult or even impossible to apply existing grammar-based fuzzing strategies. Therefore, an efficient fuzzing approach with syntax inference is required in the IoT fuzzing domain.

To address these critical problems, we propose a novel automatic black-box fuzzing for IoT firmware, termed SNIPUZZ. SNIPUZZ runs as a client communicating with the devices and infers message snippets for mutation based on the responses. Each snippet refers to a block of consecutive bytes that reflect the approximate code coverage in fuzzing. This mutation strategy based on message snippets considerably narrows down the search space to change the probing messages. We compared SNIPUZZ with four state-of-the-art IoT fuzzing approaches, *i.e.*, IOTFUZZER, BOOFUZZ, DOONA, and NEMESYS. SNIPUZZ not only inherits the advantages of app-based fuzzing (*e.g.*, IOTFUZZER), but also utilizes communication responses to perform efficient mutation. Furthermore, SNIPUZZ is lightweight as its execution does not rely on any prerequisite operations, such as reverse engineering of apps. We also evaluated SNIPUZZ on 20 popular real-world IoT devices. Our results show that SNIPUZZ could identify 5 zero-day vulnerabilities, and 3 of them could be exposed *only* by SNIPUZZ. All the newly discovered vulnerabilities have been confirmed by their vendors.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Fuzzing, IoT Firmware, Mutation, and Vulnerabilities

ACM Reference Format:
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 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3484543

1 INTRODUCTION

The Internet of Things (IoT) refers to the billions of physical devices around the world which are now connected to the Internet, all collecting and sharing data. As early as 2017, IoT devices have outnumbered the world's population [39], and by 2020, every person on this planet has four IoT devices on average [23]. While these devices enrich our lives and industries, unfortunately, they also introduce blind spots and security risks in the form of vulnerabilities. We take Mirai [25] as an example. Mirai is one of the most prominent types of IoT botnet malware. In 2016, Mirai took down widely-used websites in a distributed denial of service (DDoS) campaign consisting of thousands of compromised household IoT devices. In the case of Mirai, attackers exploited vulnerabilities to target IoT devices themselves and then weaponized the devices for larger campaigns or spreading malware to the network. In fact, attackers can also use vulnerable devices for lateral movement, allowing them to reach critical targets. For example, in the work-from-home scenarios during COVID-19, Trend Micro has reported that, introducing vulnerable IoT devices to the household will expose employees to malware and attacks that could slip into a company's network [26]. Considering the ubiquity of IoT devices, we believe that these known security incidents and risky scenarios are nothing but a tip of the iceberg.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484543

* Corresponding authors: Sheng Wen and Xiaogang Zhu

IoT vulnerabilities are normally about the implementation flaws within a device's firmware. To launch new products as soon as possible, developers always tend to use open-source components in firmware development without good update plans [1]. This sacrifices the security of IoT devices and exposes them to vulnerabilities that security teams cannot remedy quickly. Even if vendors plan to fix the vulnerabilities in their products, the over-the-air patching is usually infeasible because IoT devices do not have reliable network connectivity [16]. As a result, half of the IoT devices in the market were reported to have vulnerabilities [28].

It is hence crucial to discover such vulnerabilities and fix them before an attacker does. However, most IoT software security tests heavily rely on the assumption of device firmware availability. In many cases, manufacturers tend not to release their product firmware and that makes various dynamic analysis methods based on code analysis [7, 13, 15, 18, 32, 46] (or emulation [8, 10, 20, 50, 51]) difficult. Among the existing defense techniques, fuzz testing has shown promises to overcome these issues and has been widely used as an efficient approach in finding vulnerabilities. Moreover, the ability of IoT devices to communicate with the outside world offers us a new option, and that is to test device firmware through exchanging network messages. Therefore, an IoT fuzzer could be designed to send random communication messages to the target device in order to detect if it shows any symptoms of malfunctioning. Potential vulnerabilities could be exposed if crashes are triggered during execution or the device is pushed to send back abnormal messages.

However, using network communication to fuzz the firmware of IoT devices is very challenging. Since obtaining internal execution information from the device is not possible, most existing network IoT fuzzers [9, 31, 44] work in a black-box manner. This makes optimizing the mutation strategies very difficult. Because the selection of mutated seeds is entirely random, existing black-box IoT fuzzing approaches could become very hard to handle, and sometimes, even become more like brute force crack testing. In addition, IoT devices have strict grammatical specifications for inputs in communication. Most of the messages that are generated by random mutation will break the syntax rules of the input, and will be quickly rejected during syntax validation in the firmware before being executed. A grammar-based mutation strategy [2, 40] can effectively generate messages that meet the input requirements though. This can be done by learning the syntax via documented grammatical specifications or from a labeled training set. However, as shown in Table 1, many non-standard IoT device communication formats are being used in practice. Therefore, preparing enough learning materials for grammar-based mutation strategies is a huge workload, which makes the deployment of grammar-based IoT fuzzing difficult.

Challenges. In this paper, we focus on detecting vulnerabilities in IoT firmware by sending messages to IoT devices. To design an effective and efficient fuzzing method, several challenges have to be overcome.

- *Challenge 1: Lack of a feedback mechanism.* Without access to firmware, it is nearly impossible to obtain the internal execution information from IoT device to guide the fuzzing process (as is done in most typical fuzzers). Therefore, we need a lightweight solution to obtain feedback from device, and optimize the generation process.
- *Challenge 2: Diverse message formats.* Table 1 shows some message formats that are used in IoT communication, including JSON, SOAP, Key-value pairs, string, or even customized formats. In order to be applied to various devices, a solution should be able to infer the format from a raw message.
- *Challenge 3: Randomness in responses.* The response messages of an IoT device may contain random elements, such as timestamps or tokens. Such randomness results in different responses for the same message, and diminishes the effectiveness of fuzzing because the input generation of Snipuzz relies on responses.

Table 1: Format requirements of IoT Devices.

#	Device Type	Vendor	Model	Firmware Version	Format
1	Smart Bulb	Yeelight	YLDP05YL	1.4.2_0016	JSON
2	Smart Bulb	Yeelight	YLDP13YL	1.4.2_0016	JSON
3	Smart Bulb	Philips	A60	1.46.13_r26312	JSON
4	Smart Bulb	LIFX	Mini C	v3.60	Custom Byte
5	Smart Bulb	FloodLight	BR30	35.V7.63.7189-A	Custom Byte
6	Home Bridge	Philips	Hue	1935144040	JSON
7	Home Bridge	Alro	Base Station	1.12.2.8_9_fc4b603	JSON
8	Smart Plug	Tplink	HS100	1.5.2	JSON
9	Smart Plug	Tplink	HS110	1.5.2	JSON*
10	Smart Plug	Belkin WeMo	F7C027au	2.00.1821	SOAP
11	Smart Plug	Meross	MSS310	2.1.14	JSON*
12	Smart Plug	Orvibo	B25AUS	v3.1.3	JSON
13	Smart Plug	Konke	Mini US	us1.1.0	String
14	Smart Plug	Broadlink	SP4L-AU	v57209	Custom Byte
15	Router	Netgear	R6400	1.0.1.46	SOAP*
16	TA Assistant	ZKteco	WL10	ZLM-FX1-3.0.23	Custom Byte
17	Camera	Alro	Alro Pro 2	1.125.14.0_34_1189	JSON*
18	Camera	Foscam	F19821W	2.21.1.127	JSON*
19	NAS	QNAP	T-131P	4.3.6.0959	Key-value pairs
20	Universal Remote	BroadLink	RM mini 3	v44057	Custom Byte

*: have randomness in response.

Our approach. In this paper, we propose a novel and automatic black-box IoT fuzzing, named Snipuzz, to detect vulnerabilities in IoT firmware. Different from other existing IoT fuzzing approaches, Snipuzz implements a snippet-based mutation strategy which utilizes feedback from IoT devices to guide the fuzzing. Specifically, Snipuzz uses a novel heuristic algorithm to detect the role of each byte in the message. It will first mutate bytes in a message one by one to generate probe messages, and categorize the corresponding responses collected from device. Adjacent bytes that have the same role in the message form the initial message snippets, which is the basic unit of mutation. Moreover, Snipuzz utilizes a hierarchical clustering strategy to optimize mutation strategies and reduce the misclassification of categories caused by randomness in the response messages and the firmware's internal mechanism. Therefore, Snipuzz, as a black-box fuzzer, can still effectively test the firmware of IoT devices without the support of grammatical rules and internal execution information of the device.

Snipuzz resolves **Challenge 1** by using responses as the guidance to optimize the fuzzing process. Based on the responses, Snipuzz designs a novel heuristic algorithm to initially infer the role of each byte in the message, which resolves **Challenge 2**. Snipuzz utilizes edit distance [42] and agglomerative hierarchical clustering [43] to resolve **Challenge 3**. We summarize our main contributions as follows:

- **Message snippet inference mechanism.** The responses from IoT devices are related to code execution path in firmware. Based on responses, we infer the relationship between message snippets and code execution path in firmware. This novel mutation mechanism enables that Snipuzz does not need any syntax rules to infer the hidden grammatical structure of the input through the device responses. Compared with the actual syntax rules that determine the input string format, the result of snippet determination proposed by Snipuzz has a similarity of 87.1%.
- **More effective IoT fuzzing.** When testing IoT devices, the number of response categories is positively correlated with the number of code execution paths in the firmware. In the experiment, the number of response categories explored by Snipuzz far exceeded other methods on most devices, no matter how long the analysis duration was (in 10 minutes or 24 hours).
- **Implementation and vulnerability findings.** We implemented the prototype of Snipuzz.[2] We used it to test 20 real-world consumer-grade IoT devices while comparing with the state-of-the-art fuzzing tools, *i.e.*, IoTFuzzer, Doona, Boofuzz, and Nemesys. In 5 out of 20 devices, Snipuzz successfully found 5 zero-day vulnerabilities, including null pointer exceptions, denial of service, and unknown crashes, and 3 of them could be exposed **only** by Snipuzz.

2 BACKGROUND

2.1 Fuzz Testing

Fuzzing is a powerful automatic testing tool to detect software vulnerabilities. After decades of development, fuzzing has been widely used as a base in several security testing domains, such as the OS kernel [12, 36], servers [33], and the blockchain [3].

In general, fuzzing feeds the target programs with numerous mutated inputs and monitors exceptions (*e.g.*, crashes). If an execution reveals undesired behavior, a vulnerability could be detected. To discover vulnerabilities more effectively, fuzzing algorithms optimize the mutation process based on feedback of executions (*e.g.*, coverage knowledge), instead of using a purely random mutation strategy. Moreover, fuzzers can judge from the feedback mechanism whether each test case generated by seed mutation is "interesting" (*i.e.*, whether the test case has explored unseen execution states). If a test case is interesting, it will be reserved as a new seed to participate in future mutation. With the feedback, many fuzzers [4, 5, 29, 41, 49] steer the computing resources towards the interesting test cases and achieve higher possibility to discover vulnerabilities.

2.2 Generic Communication Architecture of IoT Devices

To react with external inputs, most IoT devices implement a similar high-level communication architecture. As per the pseudo code example presented in Figure 1, a typical implementation of the communication architecture may consist of four parts: 1) Sanitizer, 2) Function Switch, 3) Function Definitions, and 4) Replier.

When an IoT device receives an external input, Sanitizer starts parsing the input and performs regular matching. If the input format breaches the syntactic requirements, or an exception occurs during the parsing process, Sanitizer will directly notify Replier by sending a response message describing the input error and terminate the processing of input. If the input is syntactically correct, Function Switch transfers control to the corresponding Functions according to the attribute, `Key`, and corresponding value, `val`, extracted from the input. If `Key` cannot be matched, the processing of this input will be terminated, similarly as done by Replier. When Functions completes the processing, such as `setFlow()`, with the parameter `val`, it notifies Replier to generate the response message. Note that, the implementation of Functions is specific to IoT devices. As described above, Replier is responsible for sending responses to the client (such as the user's APP). Based on the calling situation (indicated by the parameter `code` in the example), Replier determines the content of response message to be sent.

[2]Publicly available at https://github.com/XtEsco/Snipuzz.

3 MOTIVATION

3.1 Response-Based Feedback Mechanism

The interactive capabilities of IoT devices make it possible to test security of device firmware through the network. However, there are also some challenges when testing IoT devices using network-based fuzzers. Since most network fuzzing methods cannot directly obtain execution status of the device, it is hard to establish an effective feedback mechanism to guide the fuzzing process. Without feedback mechanism, the fuzzing tests could be blind in the selection of mutation targets, and may lean to a brute force random test.

As discussed previously, due to the lack of open-sourced firmware, it is difficult or even impossible to instrument the IoT devices. Therefore, the response messages returned by the firmware can be regarded as a valuable source of device status information at run-time. The Replier in Figure 1 will use the value of the variable `code` to determine the content of the response messages. The value of `code` comes from many different function blocks in the firmware. Parameters are passed when Sanitizer fails to parse the input or some exceptions are triggered; or when the Function Switch cannot match the key command characters in the input; or after each input is executed in the Functions. Therefore, through the content of the response message, the code block that has been executed in the firmware can be inferred. When the firmware source code is not available, the correspondence between the firmware execution and the response messages cannot be directly extracted. Moreover, the firmware may return the same response messages even executing different functions.

Although the response message cannot be equated to the execution path of the device, it can still play an important role in the black-box fuzz testing for IoT devices. Although it is hard to link the code execution path corresponding to each response message, if the two inputs get different response messages, we can deduce that the two inputs go to different firmware code execution paths.
Our approach. Snipuzz uses the response message to establish a new feedback mechanism. Snipuzz will collect every response, and when a new response is found, the input corresponding to the response will be queued as a seed for subsequent mutation testing.

3.2 Message Snippet Inference

The firmware of the IoT device can be regarded as a software program with strict syntax requirements for input. If the byte-based

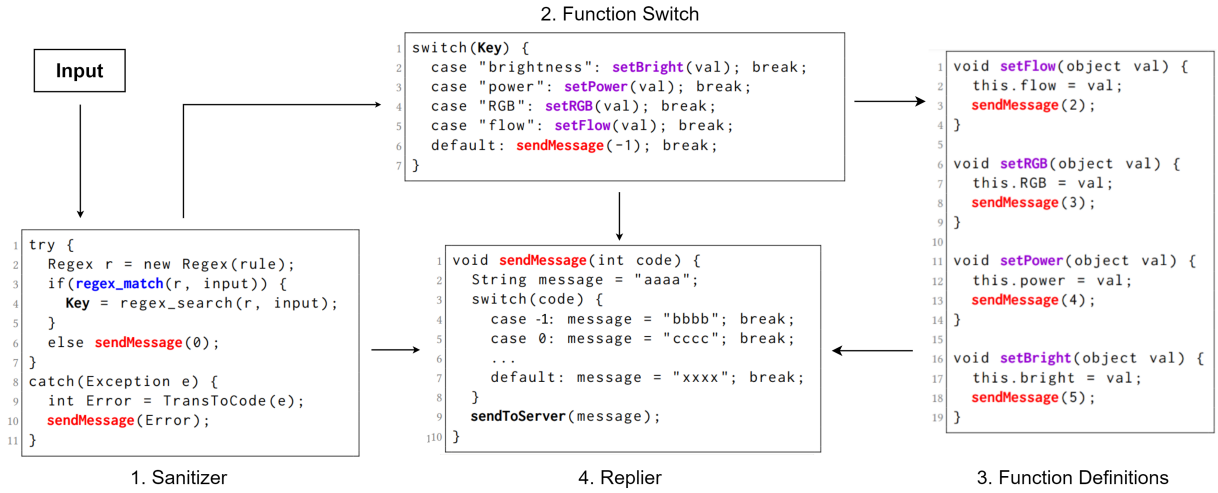

Figure 1: Interaction with IoT Firmware. Most implementations of IoT devices have a similar communication architecture, including Sanitizer, Function Switch, Function Definitions, and Replier. If the Sanitizer and the Function Switch perform correctly, corresponding functionalities will be executed. Except for crashes, the Replier will always send responses to clients.

mutation strategies (such as mutating each byte in the input one by one or randomly selecting bytes for mutation testing) are used in the fuzz testing, the generated test cases could be rare to meet the input syntax requirements. The grammar-based fuzzers utilize detailed documents or a large training data set to learn the grammatical rules and use it to guide the generation of mutation [34, 40]. In many cases, the input syntax in IoT devices is diverse or non-standard. Table 1 shows the communication format requirements used in 20 IoT devices from different vendors. Some of them are using well-known formats such as JSON and SOAP, but some use Key-value pairs or even custom strings as communication format. Therefore, it is difficult to provide grammar specifications or establish training data sets that cover communication formats on a large scale for the grammar-based mutation strategy.

The best grammar guidance originates from the firmware itself. Responses from IoT devices suggest the execution results of messages. If we mutate a valid message byte by byte (*i.e.*, breaching the format), we will get many different responses. If mutation of two different positions in the valid message receives the same response, these two positions have a high possibility that they are related to the same functionality in firmware. Therefore, those consecutive bytes with the same response can be merged into one snippet. This method of inferring message snippets can clearly reflect the utility of each byte after entering the firmware. In addition, mutation based on message snippets can largely reduce the search space and improve the efficiency of fuzzing.

Our approach. SNIPUZZ merges consecutive bytes with the same response into one snippet. We also propose different mutation operators performing on snippets.

4 METHODOLOGY

In order to clearly present our approach, we first introduce some notations while explaining the fuzzing process of SNIPUZZ. At a high level, SNIPUZZ performs as a client which sends a ***message sequence*** M to request certain actions from IoT devices. Any message $m \in M$ requests the IoT device to perform a certain functionality, and all the messages $\bigcup_k m_k = M$ work together to request an action (or actions). Similarly to the typical fuzzers, we initialize a ***seed*** S with an ***initial message sequence***, and a seed corpus C with all the seeds (Section 4.1). Meanwhile, ***restoring message sequences*** are collected for resetting the IoT device to a predefined status.

To establish an effective fuzzing, as depicted in Figure 2, SNIPUZZ first conducts a snippet determination process. Concretely, SNIPUZZ selects a message m in a seed $S \subset C$, from which a ***probe message*** p_i and a corresponding sequence M_i will be generated. Each message in M_i will trigger a ***response message*** r_i (response for short) containing the information about the execution output. SNIPUZZ assigns each message m a ***response pool*** R, which is utilized to determine if a new response r_i is unique. The uniqueness of a response indicates that it does not belong to any category of responses existed in the response pool. If r_i is unique, SNIPUZZ will add r_i into the pool R, and reserve the corresponding message sequence M_i as a new seed. SNIPUZZ then divides the message m into different snippets based on the responses (Section 4.2). Upon the snippets are obtained, SNIPUZZ performs mutation according to various strategies, *e.g.*, empty, bytes flip, data boundary, or havoc (detailed in Section 4.3). Throughout the fuzzing process, SNIPUZZ sets up a network monitor to detect crashes which may indicate vulnerabilities (Section 4.4).

4.1 Message Sequence Acquisition

The quality of initial seeds could influence the fuzzing campaigns significantly. Therefore, we consider to obtain high-quality initial seeds conforming to highly-structured formats required by IoT devices, as such inputs may exercise complex execution paths and enlarge the opportunity of exposing vulnerabilities at deep code. Generating seeds based on companion app reverse-engineering [9] or accessible specifications (as mentioned in Section 3.2) could be intuitive solutions. However, they either require heavy engineering

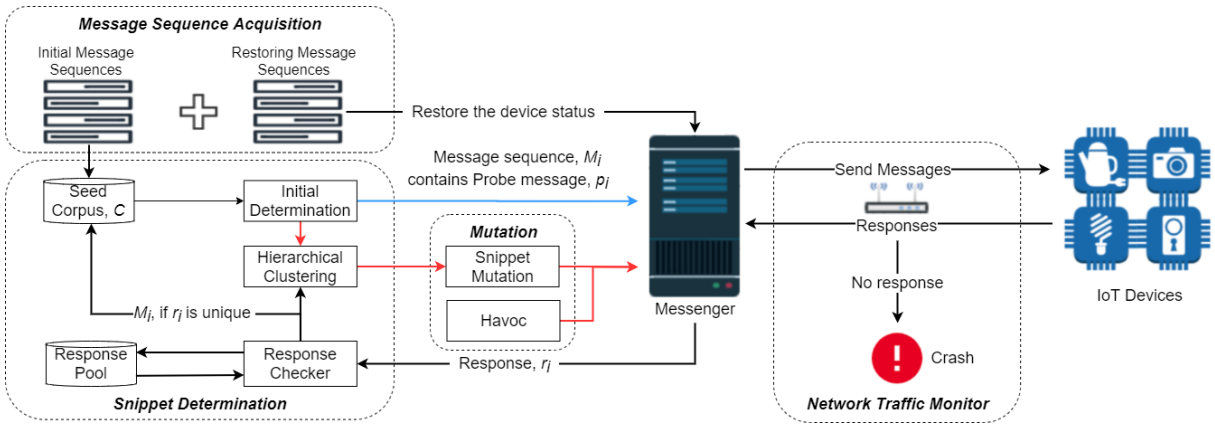

Figure 2: Workflow of SNIPUZZ. With the valid message sequences (seeds), SNIPUZZ performs snippet determination on each individual message. Then, SNIPUZZ mutates snippet(s) to generate new message sequences. By monitoring the network traffic, SNIPUZZ determines a crash when no responses are received.

efforts or could be error-prone (*e.g.*, seeds may violate the required formats or have the wrong order of messages).

Initial seed acquisition. SNIPUZZ proposes a lightweight solution to obtain initial valid seeds. Considering that many IoT devices have first- or third-party API documents as well as the test suites, the testing programs provided by both parties can effectively act as a client, sending control commands to IoT devices or remote servers. Most structural information (*e.g.*, header, message content) and protocols (*e.g.*, HTTP, HNAP, MQTT) of communication packets are defined in the API programs as message payloads. Therefore, SNIPUZZ leverages these test suites to communicate with the target devices, while at the same time, extracting the message sequences as initial seeds. For example, when using an API program to turn on a light bulb, the program first sends login information to the server or to the IoT device, then sends a message to locate a specific light bulb device, and finally sends a message to control the device to turn on the light. SNIPUZZ captures such a message sequence that triggers a functionality of IoT device as an initial seed.

Restoring message sequence acquisition. In order to replay a test case for the crash triage, SNIPUZZ ensures that the device under test has the same initial state in each round of testing. After sending any message sequence to the device, SNIPUZZ will send a restoring message sequence to reset the device to a predefined status.

Manual efforts. Although we try our best efforts to provide a lightweight fuzzer, SNIPUZZ still requires some manual efforts to obtain valid and usable initial seeds. First, we manually configure the programs from the test suites, such as setting up the IP address and the login information. Note that, we only need to configure these programs once per device. Second, to capture the message sequences dynamically, we need to manually define the specific format and protocol in the network traffic monitor. Finally, we filter out some message sequences that will mislead the fuzzing process. For instance, some API programs provide operations that can automatically update or restart the device. These operations will halt the device and thus no response will be sent back. This leads to false-positive crashes because we consider a no-response execution as a crash. The manual work costs roughly 5 man-hours per device and is only required during the message sequence acquisition phase of SNIPUZZ.

4.2 Snippet Determination

The key idea of SNIPUZZ is to optimize fuzzing process based on snippets determined by responses. Put differently, SNIPUZZ leverages snippet mutation to reduce the search space of inputs, while the snippets are automatically clustered via categorizing responses from IoT devices. The major challenge is to correctly understand the semantics of responses. For instance, due to the presence of timestamp, two semantically identical responses will be classified into different categories if utilizing a simple string comparison. Therefore, SNIPUZZ utilizes a heuristic algorithm and a hierarchical clustering approach to determine the snippets in each message.

4.2.1 Initial Determination. The essence of a message snippet is the consecutive bytes in a message that enables the firmware to execute a specific code segment. For experienced experts, it is not difficult to segment message snippets according to the semantic definition in official documents. However, for algorithms that lack such knowledge, it is essential to apply some automatic approaches to identify the meaning of each byte in the message.

SNIPUZZ first uses a heuristic algorithm to roughly divide each message into initial snippets. The core idea of the heuristic algorithm is to generate *probe messages* p_i by deleting a certain byte in the message m ($m \in$ *seed* S). By categorizing the responses r_i of each probe message, SNIPUZZ preliminarily determines the snippets in the message m.

For example, as shown in Table 2, to determine snippets in the message m = *{"on":true}*, SNIPUZZ generates *probe messages* by removing the bytes in m one by one. When the first byte '{' in m is deleted, the corresponding probe message p_1 is *"on":true}*. Similarly, when the second byte is deleted, the corresponding probe message p_2 is *{on":true}*. Therefore, the message m with 11 bytes can generate 11 different probe messages (p_1 to p_{11}). SNIPUZZ will send the 11 corresponding message sequences (M_1 to M_{11}) containing the probe messages to the device and collect responses.

Table 2: Examples of probe messages and corresponding response messages.

Messages	Content	Responses	Content	Category
Message m	{"on":true}	Response r_0	{"success":"/lights/1/state/on":true}	0
Probe message p_1	"on":true}	Response r_1	{"error":{"type":2,"address":"/lights/1/state","description":"body contains invalid json"}}	1
Probe message p_2	{on":true}	Response r_2	{"error":{"type":2,"address":"/lights/1/state","description":"body contains invalid json"}}	1
Probe message p_3	{"n":true}	Response r_3	{"error":{"type":6,"address":"/lights/1/state/n","description":"parameter, n, not available"}}	2
Probe message p_4	{"o":true}	Response r_4	{"error":{"type":6,"address":"/lights/1/state/o","description":"parameter, o, not available"}}	3
Probe message p_5	{"on:true}	Response r_5	{"error":{"type":2,"address":"/lights/1/state","description":"body contains invalid json"}}	1
Probe message p_{11}	{"on":true	Response r_{11}	{"error":{"type":2,"address":"/lights/1/state","description":"body contains invalid json"}}	1

Snipuzz then distinguishes the snippets in the message m by categorizing the responses. Specifically, the consecutive bytes with the same corresponding response type are merged into the same snippet. According to the examples illustrated in Table 2, the Response r_1, r_2, and r_5 are merged into one category that indicates an error in JSON syntax, while Response r_3 and r_4 are merged into another category which indicates an error of an invalid input parameter. Therefore, the consecutive bytes whose corresponding responses belong to the same category can form a message snippet. Through this heuristic approach, Snipuzz can determine all initial snippets in the message m.

A naive method to categorize responses is to utilize a string comparison, *i.e.*, comparing the content of responses byte by byte. However, due to the existence of randomness in responses (*e.g.*, timestamp and token), a simple string comparison may incorrectly distinguish the responses with same semantic meaning into different categories. Therefore, a more advanced solution, Edit Distance [42], is introduced to determine the category of responses. As shown in Equation (1), a similarity score, s_{kt}, between two responses r_k and r_t is calculated.

$$s_{kt} = 1 - \frac{edit\_distance(r_k, r_t)}{max\_len(r_k, r_t)}, \tag{1}$$

where the $max\_len()$ in the equation selects the longer string between the two responses and the $edit\_distance()$ counts the minimum number of operations, including insertion, deletion, and substitution, required to transform one string into the other. Therefore, the more similar two responses are, the larger the value of s_{kt} is.

Snipuzz first calculates a self-similarity score s_{ii} for each probe message p_i. Note that p_i is generated by mutating the i-th byte in the message m. Concretely, Snipuzz sends the same probe message p_i twice within an interval of one second. Two responses r_i, r_i' will be collected from the IoT device, correspondingly. The self-similarity score s_{ii} is then calculated based on the two responses r_i, r_i' according to Equation (1). Note that, due to the randomness in the responses, there could be differences between the two responses r_i, r_i', even though they are from the same probe message. Therefore, the self-similarity score could be smaller than 1.

To determine whether two responses belong to the same category, Snipuzz computes the similarity score of two responses and compares it with the self-similarity score. For example, for two responses r_i and r_j, Snipuzz uses the Equation (1) to compute the similarity score s_{ij}. After that, s_{ij} will be compared with the self-similarity. If $s_{ij} >= s_{ii}$ or $s_{ij} >= s_{jj}$ satisfies, responses r_i and r_j will be considered belonging to the same category; otherwise, responses r_i and r_j are then assigned to the different categories.

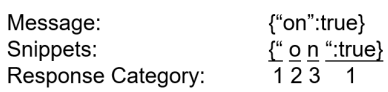

Figure 3: An example of snippet determination.

For a newly received response r_i, Snipuzz will compare it with all the responses in the corresponding response pool R based on the similarity score. If the new response r_i does not belong to any existing category, the response r_i as well as the corresponding probe message p_i will be added into the *Response Pool*.

With the response pool R, Snipuzz categories each byte in the message m. Specifically, the category of the i-th byte in message m is assigned according to the category of response r_i. Then the consecutive bytes with the same category will be merged into one snippet. Figure 3 shows an example of the initial snippet determination on the message m = *{"on":true}* according to the response categories in Table 2.

4.2.2 Hierarchical Clustering. Although Snipuzz utilizes similarity comparison to mitigate the mis-categorization caused by randomness in responses, two semantically identical responses may still be mis-categorized into different categories. This could occur when the responses contain contents extracted or copied from probe messages. For example, due to the quotation of specific error contents from probe messages, the heuristic algorithm will not assign them to one category. Specifically, the similarity score s_{34} of m = *{"on":true}* in Table 2 is 0.979, which is smaller than the self-similarity scores $s_{33} = 1.000$ and $s_{44} = 1.000$ (as there is no randomness in the responses). However, these two responses are semantically identical and should be identified into one category, *i.e.*, they are both error messages, indicating parameter syntax errors are located in the probe messages and the device is executing the same code block.

In order to solve the aforementioned problem, Snipuzz uses agglomerative hierarchical clusters to refine message snippets. The core idea of hierarchical clustering is to continuously merge the two most similar clusters until only one cluster remains.

As shown in Algorithm 1, Snipuzz will initialize the snippets according to Initial Snippets determined in Section 4.2.1 (line 1). After that, each response category in the response pool will be initialized as a cluster (line 2). Snipuzz will convert the responses into feature vectors (line 3, detailed in the later paragraph) which will be used to compute the distance between each pair of clusters (lines 5-7). Then the two closest clusters will be merged and the

Algorithm 1: Hierarchical Clustering for Snippets

```
Input: Initial Snippets F_0, Response Pool R
Result: Snippets F
1  F ← F_0;
2  C ← categorize(F_0);
3  V ← vectorize(R);
4  while size(C) > 1 do
5      for i ← size(C) to 2 do
6          for j ← size(C)-1 to 1 do
7              D ← distance_ij = ||v_i − v_j||;
           end
       end
8      i, j = argmin(D);
9      C ← merge_cluster(C, i, j);
10     V ← update_cluster_center(V, i, j);
11     F ← F + generate_snippets(C);
   end
```

cluster center will be updated accordingly (lines 8-10). After performing the cluster process, Snipuzz will generate new snippets according to the current cluster result and add the new snippets into the snippet segmentation result (line 11), which will be further used for mutation.

Concretely, Snipuzz first extracts features from responses, which vectorize responses into tuples of the self-similarity score, the length of the response, the number of alphabetic segments, the number of numeric segments, and the number of symbol segments. Each segment consists of consecutive bytes that have the same type. For instance, "123" is 1 numeric segment, and there are 2 alphabetic segments and 1 numeric segment in "*a1b*". More specifically, the r_1 in Table 2 will be vectorized to $v_1 = (1, 91, 10, 2, 10)$. Similarly, responses r_3 and r_4 will be converted to $v_2 = (1, 94, 11, 2, 13)$ and $v_3 = (1, 94, 11, 2, 13)$.

Figure 4 shows an example of clustering according to the message m = *{"on":true}* in Table 2. According to the Algorithm 1, in the preparation round (0th round) of clustering, each category in the response pool will be initialized a single cluster. In the 1st round, as clusters 2 and 3 are the two clusters with minimum distance ($\|v_2 - v_3\| = 0$), the two clusters are merged into a new cluster. Correspondingly, the message snippets 'o' and 'n' are merged into a new snippet, marked with index #4. Similarly, in the next round, the two closest clusters, the cluster 1 and the new cluster, are merged, and a new snippet will also be generated. Finally, all snippets in the message are merged into one new snippet, *i.e.*, the message itself. All the new generated snippets together with the initial snippets will be used in message mutation in the next stage.

4.3 Mutation Schemes

Snippet Mutation. In order to conduct an efficient fuzzing, Snipuzz mutates the snippets obtained in the stage of Snippet Determination. Note that the mutation schemes are performed on the entire snippet instead of a single byte in a message.

- **Empty.** The empty of a data domain may crash the firmware if the data domain is not properly checked. Therefore, Snipuzz deletes an entire snippet to empty the data domain.

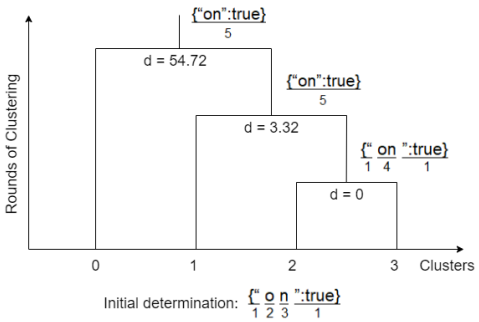

Figure 4: An example of hierarchical clustering.

- **Byte Flip.** To detect bugs in both the syntax parsers and the functional code, Snipuzz flips all bytes in a snippet. This changes the syntactic meaning of strings and will discover bugs when the parser does not properly check syntax. On the other hand, Byte Flip changes the values of data domains to examine firmware.
- **Data Boundary.** To detect the out-of-bound bugs that occur during assignment, Snipuzz modifies the values of numeric data to some boundary values (*e.g.*, 65536).
- **Dictionary.** For the scheme of Dictionary, Snipuzz replaces a snippet with a pre-defined string such as "true" and "false", which may directly explore more code coverage.
- **Repeat.** In order to detect bugs in syntax parsers, Snipuzz repeats a snippet for multiple times. Meanwhile, the repetition of data domain can detect defects caused by out-of-boundary problems.

Havoc. The conditions for triggering bugs may be complicated. For example, it may require modifying different data domains in the same message to trigger a bug. The aforementioned snippet mutation schemes only mutate one snippet at a time. However, the havoc mutation randomly selects some random snippets in a message, and performs the aforementioned mutation schemes on each of the selected snippets. Havoc mutation will not stop until finding a new response category or the target IoT device crashes.

4.4 Network Traffic Monitor

The network communication of the device is monitored and a timeout is set to determine whether the device has been crashed. In fact, the monitoring of device network communication is not a single step, and it occurs during the entire fuzzing process. In case of timeout, Snipuzz will continue to send the same message sequence for three times, as the cause of timeout could be network fluctuations instead of device crashes. If the timeout occurs for three times, Snipuzz will use the control command to physically restart the device and send the same sequence of messages to the device again. If the device still does not return the message on time, Snipuzz will record the crash and the corresponding message sequence.

4.5 Implementation

The design of Snipuzz consists of four steps: **Message Sequence Acquisition**, **Snippet Determination**, **Mutation**, and **Network Communication Monitoring**. In the **Message Sequence Acquisition** step, we use WireShark [45] in the program to detect and record the communication packets between the API and the IoT device, and manually cleaned these message sequences. The remaining core functional steps are packaged in a prototype implemented

with 4,000 lines of C# code. The network monitor will record every message sent to the device, and send the information to the device again when the device does not reply. A smart plug was used to implement the physical restart function of the target device. When Snipuzz needs to physically restart the device under test, it will send control messages to the smart plug, and the plug will be closed and then opened. In this way, the device under test will be powered off briefly and restarted.

5 EXPERIMENTAL EVALUATION

5.1 Experiment Setup

Environment setup. To initialize IoT devices, we use the applications provided by the manufacturers to complete the pairing. In order to better monitor the network communication, all devices under test are connected to a local router. Our automatic packet extractor and Snipuzz run on a Windows 10 desktop PC with Intel Core i7 six-core x 3.70 GHz CPU and 16 GB RAM, which is also connected to the router.

IoT Devices under test. We have selected 20 popular consumer IoT devices from both online and offline markets worldwide, covering various well-known brands, such as Philips, Xiaomi, TP-Link, and Netgear. The types of selected IoT devices include smart plugs, smart bulbs, routers, home bridge, IP camera, and fingerprint terminal. These devices are either recommended by Amazon or the best-selling products available in supermarkets. Table 1 details the information of the IoT devices under test.

Benchmark tools. In order to verify Snipuzz's performance in finding crashes and message segmentation, we used seven different fuzzing schemes as benchmarks.

- **IoTFuzzer [9].** The core idea of IoTFuzzer is to find the functions that send control commands to the IoT device by static analysis on companion apps, and to mutate the value of specific variables to perform fuzzing test without breaking the message format. Note that we try our best efforts to replicate IoTFuzzer since its source code is not publicly available, and we acknowledge that this could provide slightly different results with respect to the original version.
  We implement the IoTFuzzer by replacing the mutation algorithm in Snipuzz framework with the mutation strategies in IoTFuzzer. Considering that the purpose of companion apps analysis in IoTFuzzer is to ensure that only the data domain in the communication message is mutated, to make the benchmark as fair as possible, we use seeds same as the ones used in Snipuzz and manually segment the data domain of each seed message before feeding it to IoTFuzzer. We believe that such manual segmentation is sufficient to provide an upper bound performance of IoTFuzzer. Note that we remove the methods that are related to the feedback mechanism and snippet segmentation because these methods are not used in IoTFuzzer.
- **Nemesys [22].** Nemesys is a protocol reverse engineering tool for network message analysis. It utilizes the distribution of value changes in a single message to infer the boundaries of each data domain. Considering that Nemesys is a protocol inference method instead of an off-the-shelf fuzzing tool, we implement the method of Nemesys based on the Snipuzz framework to infer the snippet boundary, replacing corresponding snippet determination method (Section 4.2).
- **BooFuzz [31].** As a successor of Sulley [19], BooFuzz is an excellent network protocol fuzzer that has been involved in several recent fuzzing research [9, 37, 48]. Different from other automatic fuzzers, BooFuzz requires human-guided message segmentation strategies as inputs. In our research, we leverage this property and manually define more fuzzing strategies to enrich the benchmark evaluation.
  - **BooFuzz-Default**. In this strategy, we set each message in the input as a complete string, that is, BooFuzz will use the message as a string for mutation testing.
  - **BooFuzz-Byte**. Each byte of the message in the input will be used for a mutation test individually.
  - **BooFuzz-Reversal**. Contrary to the idea of IoTFuzzer, in this strategy, we focus on the mutation of non-data domain in the message, while keeping data domain unchanged.
- **Doona [44]**. Doona is a fork of the Bruterforce Exploit Detector (BED) [6], which is designed to detect potential vulnerabilities related to buffer and formats in network protocol. Different from other tools, Doona does not take network communication packets as seeds. The test cases of Doona are required to be pre-defined for each device or protocol under test.
- **Snipuzz-NoSnippet**. Snipuzz uses the segmentation of message snippets to enhance the fuzzing efficiency and the ability to find crashes. In order to verify whether the snippet determination indeed benefits fuzzing, we implement Snipuzz-NoSnippet based on Snipuzz. Snipuzz-NoSnippet does not have the snippet determination component, and blindly mutates bytes in messages without the knowledge of responses.

Except for Doona whose test cases are preset, all benchmark tools and Snipuzz are tested with same input sets. These input sets may be different in formats (*e.g.*, BooFuzz requires to manually set the input, and Numesys requires the input to in pcap file format), but the content is the same.

There are many other popular fuzzing tools which are able to test IoT devices via network communication, such as Peach [30] and AFLNET [33]. However, since they are grey-box fuzzers that requires to instrument firmware, it is infeasible and unfair to regard those tools as baselines for black-box schemes.

5.2 Vulnerability Identification

5.2.1 Snipuzz. After performing fuzz testing using Snipuzz on each of the 20 IoT devices for 24 hours, we detected 13 crashes in 5 devices as shown in Table 3. We further manually verified that the detected crashes include 7 null pointer dereferences, 1 denial of service, and 5 unknown crashes. The crashes found by Snipuzz are triggered by the malformed inputs. These malformed inputs break the message format in different ways. For example, deleting placeholders, emptying the data domain, or fortunately changing the type of data value.

Note that all the crashes identified by Snipuzz are on JSON-based devices, although we successfully conducted experiments on the 20 IoT devices with various communication formats, such as JSON, SOAP, and K-V pair. The experiments also show that Snipuzz

Table 3: Experiment Results. SNIPUZZ discovers the most number of categories and exposes the most number of bugs.

#	Devices	SNIPUZZ			IOTFUZZER			DOONA		BOOFUZZ-DEFAULT		BOOFUZZ-BYTE		BOOFUZZ-REVERSAL		NEMESYS		SNIPUZZ-NOSNIPPET	
		T	C	10/24	T	C	10/24	C	10/24	C	10/24	C	10/24	C	10/24	C	10/24	C	10/24
1	YLDP05YL	UC	3*	**46/71**	UC	1*	31/33	NA	NA/NA	0	11/17	0	11/41	0	11/22	0	26/61	0	21/69
2	YLDP13YL	UC	2*	**35/76**	UC	1*	20/24	NA	NA/NA	0	8/18	0	8/42	0	8/22	0	18/62	0	22/70
3	A60	DoS	1	**28/41**	/	0	18/22	0	5/16	0	7/13	0	8/33	0	5/21	0	22/36	0	20/39
4	Mini C	/	0	**46/72**	/	0	18/31	0	7/15	0	5/11	0	6/31	0	5/21	0	18/68	0	18/70
5	BR30	/	0	**28/51**	/	0	8/19	NA	NA/NA	0	4/11	0	4/31	0	4/20	0	13/40	0	13/48
6	Hue	/	0	**65/110**	/	0	29/36	0	4/11	0	7/11	0	9/31	0	7/25	0	34/110	0	22/99
7	Base Station	/	0	**34/51**	/	0	29/33	0	7/16	0	6/9	0	9/17	0	7/13	0	19/38	0	23/50
8	HS100	NPD	3	**24**/64	/	0	20/27	NA	NA/NA	0	6/13	0	6/31	0	6/22	0	20/64	0	19/**71**
9	HS110	NPD	4	**24/79**	/	0	17/22	NA	NA/NA	0	6/14	0	9/33	0	6/22	0	20/62	0	19/78
10	F7C027au	/	0	**13/21**	/	0	7/10	0	6/14	0	8/12	0	6/18	0	6/15	0	8/14	0	12/21
11	MSS310	/	0	**42/61**	/	0	15/17	0	8/16	0	5/11	0	8/45	0	8/21	0	30/59	0	20/61
12	B25AUS	/	0	**19/42**	/	0	8/13	0	7/19	0	7/14	0	11/17	0	7/11	0	16/36	0	9/41
13	Mini US	/	0	**25/61**	/	0	8/41	NA	NA/NA	0	7/16	0	7/35	0	7/22	0	9/55	0	8/49
14	SP4L-AU	/	0	**37/43**	/	0	18/32	0	5/11	0	5/17	0	7/32	0	5/23	0	23/40	0	17/40
15	R6400	/	0	11/37	/	0	**20**/24	0	4/13	0	3/12	0	4/24	0	4/18	0	6/30	0	6/**41**
16	WL100	/	0	**53/81**	/	0	38/44	NA	NA/NA	0	8/16	0	8/46	0	8/27	0	41/70	0	29/76
17	Alro Pro 2	/	0	**25**/36	/	0	16/22	0	10/14	0	8/13	0	14/22	0	10/17	0	18/22	0	13/**41**
18	F19821W	/	0	**39**/75	/	0	36/33	0	7/13	0	5/11	0	7/23	0	7/14	0	27/65	0	21/**76**
19	T-131P	/	0	**36**/80	/	0	9/22	0	7/16	0	7/20	0	9/42	0	7/35	0	21/65	0	20/**91**
20	RM mini 3	/	0	**14/36**	/	0	9/30	NA	NA/NA	0	10/17	0	14/31	0	10/23	0	6/30	0	5/35

UC: Unknown crash. **NPD**: Null pointer dereference. **DoS**: Denial of service. **T**: Vulnerability type. **C**: Number of crashes. **10/24**: Number of response categories (10 minutes/24 hours).
*: Remotely exploitable. **NA**: Since DOONA is only applicable to some network protocols, devices that cannot be tested are represented by 'NA'.

discovers a higher number of response categories compared to the other fuzzers (as detailed in Section 5.3).

Null pointer dereferences. As shown in Table 3, the 7 crashes triggered by SNIPUZZ in TP-Link HS110 and HS100 are all caused by null pointer dereferences. After sending the test cases to HS110 and HS100, the devices crashed, unable to reply to any interaction. However, after a few minutes, the devices automatically restarted and recovered to the initial state. Based on the analysis of test cases, we found that the vulnerabilities are all triggered by messages that mutated in JSON syntax. Concretely, when some important placeholders, such as curly braces and colons, or a part of the test message are mutated, the syntax structure and the semantic meaning of the message are broken. If the device cannot handle the mutated input message properly, it will crash the device. We reported the vulnerabilities to the device vendor, TP-Link, via email on June 13, 2020. They have confirmed the vulnerability and promised to fix it through a firmware update.

Denial of service. Another interesting finding is the denial of service vulnerability detected in Philips A60 smart bulb. After being tested by SNIPUZZ for 24 hours, Philips' official companion app could not manage the device normally. Specifically, the device cannot be found in the app and if any further messages are sent through the app, the response in the app will keep asking to bound the device to a device group and no further interaction is available. However, we observe that if the message packet is sent directly to the device, the device can work normally. This indicates that the device does not completely crash but its service via the companion app is denied.

Unknown crashes. SNIPUZZ found 5 crashes on Yeelight bulbs, YLDP05YL, and YLDP13YL. The devices crashed and then restarted by themselves within roughly 1 minute. By analyzing the test cases, we found that the crashes are due to the deletion of certain data domains, such as the nullify of parameters (marked as red in Table 4). As the firmware of the two devices is not publicly available, the root cause of the vulnerability cannot be determined; however, we can still deduce that the vulnerability is due to the device reading in null values during the parsing process, causing a crash during the assignment. We also find that its communication using a local network does not require any authentication, which means that the device can be crashed by any attackers in the local network. Therefore, we consider the vulnerabilities as 'remotely exploitable'.

Table 4: Mutated messages of SNIPUZZ & IOTFUZZER.

Contents of mutated messages	Generated by
{"{"id": 0, "method": "start_cf", "params": ["4, 4, "1000, 2, 2700,100,500 ,1,255,10,5000,7,0,0,500,2,5000,1"]}"	Original Message
{"{"id": 0, "method": "start_cf", "params": ["4, , "1000, 2, 2700,100,500 ,1,255,10,5000,7,0,0,500,2,5000,1"]}"	SNIPUZZ
{"{"id": 0, "method": "start_cf", "params": [", 4, "1000, 2, 270000,100,500 ,1,255,10,5000,7,0,0,500,2,5000,1"]}"	IOTFUZZER

5.2.2 Benchmark with state-of-the-art tools. As shown in Table 3, after 24 hours fuzz testing on each device, none of the benchmark tools found a crash, except IOTFUZZER. They did not find the crash due to various reasons. DONNA focuses more on the mutation of communication protocols. Further, DONNA cannot be applied on all devices, which limits its capacity. Since BOOFUZZ directly replaces the specified positions in the message with a preset string, it can only trigger limited types of vulnerabilities. NEMESYS offers a new idea of determining message snippets. However, since it determines message snippets by the distribution of values in messages, it is difficult for NEMESYS to accurately decide the boundary between data and non-data domains. Therefore, NEMESYS can hardly detect vulnerabilities that can only be triggered by mutating the data or non-data domains. SNIPUZZ-NOSNIPPET, which does not apply the snippet-based mutation method used in SNIPUZZ, is similar to the classic fuzzer AFL [24]. Since SNIPUZZ-NOSNIPPET does not infer the structure of the message but directly uses single or multiple consecutive bytes as the unit of mutation, most test cases generated by SNIPUZZ-NOSNIPPET destroy the structure of the messages. Such a method is difficult to work on devices that require highly-structured inputs.

IOTFUZZER detected 2 crashes in 2 smart bulb devices, *i.e.*, the YLDP05Y and YLDP013Y. Due to the mutation strategy of IOTFUZZER, the malformed input provided by IOTFUZZER is obtained by emptying the data domain. According to the examples of mutated

messages listed in Table 4, we can see that the messages mutated by IoTFuzzer resemble the ones generated by Snipuzz. The mutated domains of messages from Snipuzz and IoTFuzzer in Table 4 are all in the data domain. In terms of the mutation test effectiveness, Snipuzz and IoTFuzzer achieve the same goal on these two examples. However, Snipuzz can cover the mutation space of IoTFuzzer because IoTFuzzer only focuses on the data domain mutation, while Snipuzz can mutate both the data and non-data domains.

To further determine the root cause of the crash, we obtained the firmware source code of HS100 and HS110, two typical market consumer-grade smart plugs manufactured by TP-Link, and conducted a case study which reflected the differences between Snipuzz and IoTFuzzer. We found that one of the crashes triggered by Snipuzz on the two devices is caused by breaking the syntax structure and mutating both on data and non-data domains. More specifically, the mutated messages successfully bypassed the sanitizer and triggered the crash during function execution. We deduce that this could be caused by an error-prone third-party sanitizer (more details could be found in Section 5.5). On the other hand, due to the design of IoTFuzzer, the fuzzing is based on the grammatical rules as the IoTFuzzer tends to satisfy the grammar requirements with first-priority, in order not to be rejected by the sanitizer and ensure that each test case can reach the functional execution part in the firmware. Such strategy constraints the test range of fuzzing and its capacity to cover the sanitization part in comparison to Snipuzz. Therefore, we argue that, considering the complexity of IoT firmware testing, a lightweight and effective black-box vulnerability detection tool, such as Snipuzz, is a pressing need.

5.3 Runtime Performance

Figure 5 shows how Snipuzz and the other 7 benchmark fuzzers explored the device firmware during the first **10 minutes**. We repeated the fuzz testing for 10 times and recorded the median values of the numbers of response categories discovered by each method. We manually reviewed the response categories to remove the mis-categorization caused by randomness in responses or the response mechanism of devices.

As shown in Figure 5, Doona can only detect a small number of response categories. Doona is protocol-based fuzzing methods, and its tests are more biased towards protocol content. The mutation test on the communication protocol has a high probability of being directly rejected or ignored by the device unilaterally, resulting in few categories of responses that can be pass the sanitizer.

We implemented 3 fuzzing strategies based on Boofuzz, *i.e.*, mutating the whole message as a string, mutating each byte of the message, and mutating non-data domain. However, the testing results indicate that all of them explored very limited categories of responses on each device. The limitation of category coverage is due to the mutation strategy of Boofuzz, which replaces the target contents with a specific pre-defined string. For example, using strings, such as "`/./././././././.`", to replace the content of messages with different strategies (*e.g.*, replacing the entire strings, a single byte, or a non-data domain) may cause the violation of message format and could be easily rejected by the sanitizer. Therefore, most of the responses obtained by Boofuzz fall into the category of "error responses".

The number of response categories explored by IotFuzzer grows rapidly within a short period of time and then stagnates. In the mutation stage, IotFuzzer randomly selects a set of inputs from the original candidate inputs and randomly mutates the data domain for one or more message(s). It will continue to repeat this method until the device crashes or reaches the time limit. Such a method based on randomness helps IotFuzzer to mutate and test a large number of message data domains in the original input and collect response message categories quickly in the beginning. However, the number of response categories found by IotFuzzer will soon reaches the limitation as it focuses on data domain mutation only.

In most devices, Snipuzz has maintained a steady upward trend in most cases, and after a period of time, the number of response categories found by Snipuzz exceeds IotFuzzer. Unlike IotFuzzer, Snipuzz covers response categories through the **Snippet Determination** stage. Based on the message snippet exploration strategy, Snipuzz first explores all the response categories of a certain message as many as possible, then the snippets of a message are obtained and tested by **Snippet Mutation**. The next message will be processed in the same way until all messages in the initial message sequence have been tested. Followed by this method, Snipuzz may not get a large number of response categories at the very beginning. When Snipuzz detects a message snippet, every byte in the message content will be included in the test. Therefore, as shown by the bold numbers in Table 3, for 15 out of 20 devices, Snipuzz covers the most number of response categories after 24-hour fuzz testing, compared with other state-of-the-art IoT fuzzing tools.

On 5 devices, Snipuzz-NoSnippet collected more response categories than Snipuzz within 24 hours. The mutation method used by Snipuzz-NoSnippet is similar to the classic fuzzer AFL [24]. It directly performs mutation on a single byte or several consecutive bytes. However, Snipuzz-NoSnippet is difficult to cover response categories that are not obtained by breaking the grammatical format (*e.g.*, data out of bounds in the data domain). Theoretically, although the Snipuzz-NoSnippet mutation method is not so efficient, it still has the capability to explore the most categories of responses.

Nemesys explores more categories of responses than BooFuzz and Doona, but does not exceed Snipuzz. The Nemesys strategy performs deterministic mutations on each data domain of the messages in turn, which makes its trend of run-time performance similar to Snipuzz. However, the data domain determination strategy of Nemesys is not based on the responses from IoT device. Thus, the distribution of byte values in messages does not benefit in covering more response categories. Therefore, the number of response categories collected by Nemesys is limited.

It is interesting to observe that, in the case of R6400, Snipuzz also enters a stagnation after only finding a few response categories. We carefully checked the initial input message sequences and found that the average length of the message exceeds 400 bytes, forcing Snipuzz to generate and send a large number of probe messages to determine message snippets. As a result, in the first 10 minutes, Snipuzz was still exploring the response category of the first few messages, which limited its performance.

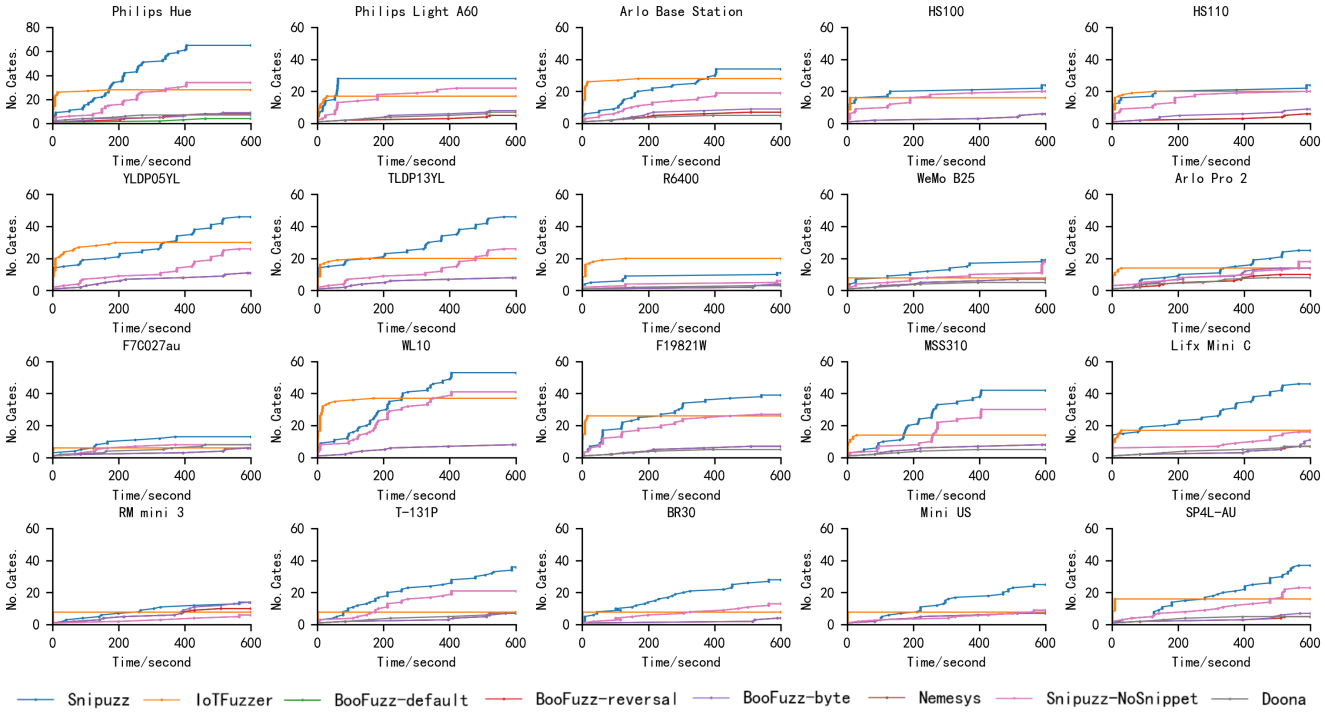

Figure 5: Runtime performance. The number of categories discovered in 10 minutes on all the 20 IoT devices. Snipuzz performed best on 19 devices.

5.4 Assessment on Message Snippet Inference

Among all strategies, Snipuzz and Nemesys utilize semantic segmentation, to assess their message snippet inference performance. We compare the snippets they produce during the fuzzing process with the grammar rules defined in API documents. Specifically, for some mature and popular languages, such as JSON, we establish the grammar rules as per their standard syntax; for custom formats, such as strings or custom bytes, we refer to the official API documents and define the grammar rules based on the instructions.

Equation (2) quantifies the quality of snippet inference, and *Similarity* indicates the percentage of correctly categorized bytes in a snippet-determined message, m, compared with the ground truth, g, manually extracted from the grammar rules.

$$Similarity(m) = 1 - \frac{count[cate(m) \oplus cate(g)]}{len(m)}, \qquad (2)$$

where $cate()$ returns the category of each message byte in a series of "0" and "1" bits, $count()$ counts the number of mis-categorized bytes, and $len()$ represents the length of a message. Note that in a ground truth message, "0" indicates the non-data domain (marked blue in Table 5), while "1" indicates the data domain (marked red in Table 5). Therefore, the $\oplus$ is the bitwise XOR operation.

In addition, followed by Equation (2), we compute the average similarity of the snippets (or data domain) determined by Snipuzz and Nemesys for all the 235 messages obtained from experiments. Note that during the calculation of the average similarity, for each message, if there are multiple snippet sets determined, we will select the snippet inference with the highest similarity value; therefore we present an upper-bound of performance.

Table 5: Inference results of Snipuzz and Nemesys.

Method	Ave. Similarity	Example
Snipuzz	87.1%	{"on":true,"sta":140,"bri":254}
Nemesys	64.5%	{"on":true,"sta":140,"bri:"254}
Ground Truth	100.0%	{"on":true,"sta":140,"bri":254}

The average similarity result of Snipuzz, 87.1%, indicates that, by applying snippet inference based on the hierarchical clustering approach, Snipuzz can effectively find the grammatical rules hidden in the message. Ideally, in Snipuzz, the merging of clusters removes the influence caused by the randomness in responses and by the replying message mechanism itself. Therefore, the message snippets will conform to the grammatical rules gradually, which leads Snipuzz to a higher similarity result.

However, we also found some differences between the snippet inference method and the grammatical rules in some results. For example, given the example shown in Table 5, the snippet inference method combines the strings belonging to the data domain in the grammatical rules (*i.e.*, 'true', '140' and '254') with some placeholders (such as double quotes and curly brackets). After analyzing the response messages, we found that the responses obtained after destroying these data domains and destroying placeholders are all about invalid format. This may due to the fact that in the firmware, when an error occurs in the parsing format, the response does

```
{"method":"passthrough", "params": {"deviceId": "800...12A",
 "requestData":
    "{\"schedule\":
        {\"edit_rule\":
            {\"stime_opt\":0,\"wday\":[0,1,1,1,1,1,0],
            \"smin\":1411,\"enable\":0,
            \"repeat\":1,\"etime_opt\":-1,
            \"id\":\"27D236000E8B21C641A59DCB5C93ECBB\",
            \"name\":\"\",\"eact\":-1,
            \"month\":0,\"sact\":0,\"year\":0,
            \"longitude\":0,\"day\":0,\"force\":0,\"latitude\":0,
            \"emin\":0},\"set_overall_enable\":{\"enable\":1
            }
        }
    }"
    }
}
```

Figure 6: An example of vulnerability triggering

not report a detailed description of the error but instead returns a general format error.

On the other hand, Nemesys uses the distribution of value changes in the protocol to determine the boundary of different data domains and achieve the semantic segmentation of a message. The advantage of this method is that it does not require any other additional information, such as grammar rules or a large number of training data sets in addition to the message itself.

The average similarity result of Nemesys, 64.5%, is lower than the Snipuzz result. Given the example shown in Table 5, when segmenting messages in a format requires restricted syntax, such as Json and XML, Nemesys can achieve a good semantic segmentation performance, because the placeholders usually use symbols unusually used in data domains. This distribution of byte value enables Nemesys to effectively find the boundaries between data domains. However, in IoT devices, customized formats are prevalent. For example, the smart bulb BR30 uses custom bytes as a means of communication, where each byte corresponds to a special meaning (*i.e.*, "`0x61`" represents "`CHANGE_MODE`" and "`0x0f`" represents "`TRUE`"). In such cases, the value distribution of characters can no longer be used as a guidance for the data domain determination, and thus the message segmentation determined by Nemesys is error-prone.

5.5 Mutation Effectiveness: A Case Study

The HS100 and HS110 manufactured by TP-Link are two classic market consumer-grade smart plugs. In the work by Chen *et al.* [9], they use HS110 with firmware version 1.3.1 to test IoTFuzzer. The results of their experiment show that IoTFuzzer triggered a vulnerability in the device by mutating the data domain in a message (changing **"light"** to **0**).

However, in the updated version of the firmware (version 1.5.2), IoTFuzzer did not find any vulnerabilities. Figure 6 shows an example of the original input message and the mutated snippets in a message that can trigger the vulnerability. In this case, Snipuzz triggered a vulnerability related to firmware input by breaking the JSON syntax structure in the message. The intention of the original message is to change some attributes (*e.g.*, "`stime_opt`" and "`wday`") in a rule (inferred by "`edit_rule`"). In the mutated message, Snipuzz randomly deleted some contents (inside the red frame), which break the JSON syntax. This may cause errors about parsing messages or passing parameters incorrectly handled by the firmware and, consequently, crashes the device.

```
1 la    $t9, cJSON_GetObjectItem
2 la    $a1, aSchedule
3 jalr  $t9; cJSON_GetObjectItem
4 lw    $gp, 0x40+var_30($sp)
5 beqz  $v0, loc_415E7C
6 lui   $a1, 0x47
7 move  $a0, $v0
```

Figure 7: A vulnerable code snippet from HS110 firmware.

To further determine the root cause of the crash, we obtained the firmware source code. Figure 7 shows a code snippet from the firmware, using cJSON,[3] a popular open-source lightweight JSON parser (5.4k stars in GitHub), to interpret input message fragments. The `jalr` instruction will save the result of `cJson_GetObjectItem` in `$t9` and jump to this address unconditionally (see line 3 in Figure 7), which means the firmware will pick the value corresponding to *"schedule"*. In the original message, the value corresponding to "schedule" is a JSON object headed by "`edit_rule`" (from line 4 to line 16 in Figure 6). Note that the aforementioned snippet-based mutation strategy implemented in Snipuzz is able to break the syntax structure and mutate on data and non-data domains at the same time. Interestingly, although the removing of two left curly braces breaks the JSON syntax, it is not recognized by cJSON parser, so the mutated message successfully bypasses the syntax validation and enters the functional code in firmware. When the firmware tries to access the successor JSON object in "`schedule`", *i.e.*, the object starts with "`edit_rule`", since the corresponding value is no more a JSON object, but an array, a null pointer exception is triggered.

Due to the design of IoTFuzzer, the fuzzing based on grammatical rules will offer priority to satisfying the grammar requirements in the mutation process in order not to be rejected by the firmware grammar detector. The advantage of this is to ensure that each test case can reach the functional execution part of the firmware. However, in this case, the test range of fuzzing based on grammatical rules cannot cover the firmware sanitising part.

To conclude, the root cause of the crash has two factors: 1) the validation of message syntax heavily relies on a third-party library; 2) the firmware does not correctly handle the null pointer exception caused by data type mismatch. Although it is not reasonable to require a vendor to develop products purely from scratch, we argue that thorough testing and validation on the open-source library are essential. Considering the complexity of IoT firmware testing, a lightweight and effective black-box vulnerability detection tool, such as Snipuzz, is a pressing need.

6 DISCUSSION AND LIMITATIONS

Snipuzz has successfully examined 20 different devices and exposed security vulnerabilities on five of them. However, there are still some factors limit the efficiency and scalability of Snipuzz.

Scalability and manual effort. In our prototype, we capture communication packets through API programs and monitoring network communication. Even in the absence of API programs or documents, the message formats can be determined from the official Apps of IoT devices through decompilation and taint analysis. Otherwise, we can solve this problem by intercepting the communication between APPs and IoT devices, and then recovering message formats

[3] https://github.com/DaveGamble/cJSON

from the captured packets. However, both methods could introduce overhead and involve manual effort. Recall that Snipuzz requires 5 man-hours per device to collect the initial seeds (Section 4.1. The manual efforts mainly focus on cleaning the packets from the API programs that are obtained from publicly available first- and third-party resources. To mitigate, when applying Snipuzz to IoT devices, techniques such as crawlers could be used to automatically gather API programs. Moreover, the process of cleaning the packets could also be improved by pre-processing keywords through scripts to achieve automatic collection of communication packages.

Threats to validity. As Snipuzz collects initial message sequences via API programs and network sniffers, the first threat comes from the absence of API programs. In this case, we can recover message formats based on the companion apps of IoT devices (similar to IoTFuzzer), but may lead to more manual efforts. Second, the encryption in messages decreases the effectiveness of snippet determination because the semantic information could be corrupted. A potential solution to the encryption issue is to integrate decryption modules into Snipuzz. Finally, the code coverage of firmware could be subject to the accessibility of API programs, since Snipuzz can only examine the functionalities that are covered in API programs. Recombining the message snippets from different seeds to generate new valid inputs could mitigate this limitation.

Code coverage. The code coverage of firmware explored by Snipuzz depends on the API programs. For example, if the API programs of a bulb only support the functionality of turning on power, it is almost impossible to explore the functionality of adjusting the brightness via mutating the messages captured during the power turned on. In the future work, without the support of grammar, we will consider recombining the message snippets to try to generate new valid inputs. This method can help explore more firmware execution coverage in addition to the original inputs provided.

Requirements on detailed responses. The detection effectiveness of Snipuzz depends on the quality of message snippets which is contingent on how much information could be obtained from the responses of IoT devices. To put differently, if the IoT device does not provide responses that are detailed enough, for example reporting all the errors with a uniform message, it could be hard for Snipuzz to determine the message snippets. Fortunately, in many IoT devices, advanced error descriptions could be obtained in debug mode which will significantly improve the determination process of message snippets in Snipuzz.

7 RELATED WORK

Our Snipuzz performs in a black-box manner for detecting vulnerabilities in IoT devices. Unlike existing black-box fuzzing for IoT devices, which blindly mutates messages, Snipuzz optimizes the mutation process of black-box fuzzing via utilizing responses. This feedback mechanism improves the effectiveness of bug discovery. For instance, IoTFuzzer [9] obtains the data domain, on which IoTFuzzer performs blind mutation. Thus, IoTFuzzer lacks the knowledge of the quality of the generated inputs, resulting in a waste of resource on the low-quality inputs. There are also several dynamic analysis approaches focusing on the networking modules of IoT devices. For example, SPFuzz defines a new language for describing protocol specifications, protocol state transitions, and their correlations [37]. SPFuzz can ensure the correctness of the message format in the conversation state and the dependence of the protocol. IoTHunter is a grey-box approach to fuzz the state protocol of IoT firmware [47]. IoTHunter can constantly switch the protocol state to perform a feedback-based exploration of IoT devices. In a recent example, AFLNet acts as a client and continuously replays the variation of the original message sequence sent to target (*i.e.*, server or device) [33]. AFLNet uses response codes, which are the numbers indicating the execution states, to identify the execution status of targets and explore more regions of their networking modules.

Another research line for dynamic analysis of IoT devices is the usage of emulators. The disadvantages of emulation are the heavy engineering efforts and the requisite of firmware, although the emulation of IoT firmware can analyze more thoroughly than black-box fuzzing. Two major challenges for emulation of IoT firmware are the scalability and throughput. Therefore, the efforts in improving the performance of emulation include full-system emulation [8, 27], improvement of emulation success rates [21], hardware-independent emulation [17, 38], and combination of user- and system-mode emulation [51]. Based on the emulation, fuzzing can be integrated into those frameworks and can hunter defects in firmware [38, 51].

Static analysis of firmware is the complementary approach of dynamic analysis. Semantic similarity is one of the major techniques that make static analysis successful. Researchers analyze semantic similarity via comparison of files and modules [13], Control Flow Graphs (CFGs) [14], parser and complex processing logic [11], and multi-binary interactions [35]. There are also many similarity-based approaches that can detect vulnerabilities across different firmware architectures. They usually extract various architecture-independent features from firmware for each node in a CFG to represent a function, and then check whether two functions' CFG representations are similar [15, 32].

8 CONCLUSION

In this paper we have presented a black-box fuzzing framework Snipuzz designed for detecting vulnerabilities hiding in IoT devices. Different from other black-box network fuzz testing, Snipuzz uses the response messages returned by the device to establish a feedback mechanism for guiding the fuzzing mutation process. In addition, Snipuzz infers the grammatical role of each byte in the messages based on the responses from the device, so that Snipuzz can generate test cases that meet the device's grammar without the guidance of grammatical rules. We have used 20 consumer-grade IoT devices from the market to test Snipuzz, and it has successfully found 5 zero-day vulnerabilities on 5 different devices.

ACKNOWLEDGEMENTS

We thank all the anonymous reviewers for their valuable feedback. Minhui Xue was, in part, supported by the Australian Research Council (ARC) Discovery Project (DP210102670) and the Research Center for Cyber Security at Tel Aviv University established by the State of Israel, the Prime Minister's Office and Tel Aviv University. In addition, this research is partially supported by the Australian Research Council projects DP200100886 and LP180100170.

REFERENCES

[1] 2020. The Three Software Stacks Required for IoT Architectures. *IoT Eclipse (White Paper)* (2020).

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *The Network and Distributed System Security Symposium (NDSS)*.

[3] I. Ashraf, X. Ma, B. Jiang, and W. K. Chan. 2020. GasFuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. *IEEE Access* (2020).

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[6] Kali Bot. 2019. bed. https://gitlab.com/kalilinux/packages/bed.

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[8] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for Linux-based embedded firmware. In *The Network and Distributed System Security Symposium (NDSS)*.

[9] 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 *The Network and Distributed System Security Symposium (NDSS)*.

[10] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX '20)*.

[11] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. 2015. PIE: Parser identification in embedded systems.

[12] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[13] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*.

[14] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (english version). *Journal of Computer Virology and Hacking Techniques* (2005).

[15] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Network and Distributed Systems Security (NDSS)*.

[16] Pwnie Express. 2020. *What makes IoT so vulnerable to attack?* Technical Report. Outpost24.

[17] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[18] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[19] Fitblip. 2019. Sulley. https://github.com/OpenRCE/sulley.

[20] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2019. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*.

[21] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis. In *Annual Computer Security Applications Conference*.

[22] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*.

[23] Karla Lant. 2017. By 2020, there will be 4 devices for every human on earth. *Futurism* (2017).

[24] lcamtuf. 2017. AFL. https://lcamtuf.coredump.cx/afl/.

[25] Trend Micro. 2020. *Mirai botnet exploit weaponized to attack IoT devices via CVE-2020-5902.* Technical Report. Security Intelligence Blog.

[26] Trend Micro. 2020. *Smart yet flawed: IoT device vulnerabilities explained.* Technical Report. Security News.

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium*.

[28] Lindsey O'Donnell. 2020. *More than half of IoT devices vulnerable to severe attacks.* Technical Report. ThreatPost.

[29] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*.

[30] Peachtech. 2021. PEACH: The PEACH fuzzer platform. https://www.peach.tech/products/peach-fuzzer/ Accessed: 2021-01.

[31] Joshua Pereyda. 2017. boofuzz: Network protocol fuzzing for humans. https://boofuzz.readthedocs.io/en/stable/.

[32] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy (SP)*.

[33] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A greybox fuzzer for network protocols. In *IEEE International Conference on Software Testing, Verification and Validation (ICST) 2020*.

[34] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* (2019).

[35] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[36] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted feedback fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*.

[37] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. 2019. SPFuzz: a hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access* (2019).

[38] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. FirmFuzz: automated IoT firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*.

[39] Liam Tung. 2017. *IoT devices will outnumber the world's population this year for the first time.* Technical Report. ZDNet.

[40] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[41] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *The Network and Distributed System Security Symposium (NDSS)*.

[42] Wikipedia. 2021. Edit distance. https://en.wikipedia.org/wiki/Edit_distance.

[43] Wikipedia. 2021. Hierarchical clustering. https://en.wikipedia.org/wiki/Hierarchical_clustering.

[44] wireghoul. 2019. Doona. https://github.com/wireghoul/doona.

[45] wireshark. 2020. About wireshark. https://www.wireshark.org/about.html.

[46] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.

[47] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. 2019. Poster: Fuzzing iot firmware via multi-stage message generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[48] Y. Yu, Z. Chen, S. Gan, and X. Wang. 2020. SGPFuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access* (2020).

[49] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*.

[50] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *The Network and Distributed System Security Symposium (NDSS)*.

[51] 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 *28th USENIX Security Symposium (USENIX Security 19)*.