Cybersecurity

RESEARCH

Open Access

MSLFuzzer: black-box fuzzing of SOHO router devices via message segment list inference

Yixuan Cheng[1,2], Wenqing Fan[1,2], Wei Huang[1,2], Jingyu Yang[1,2], Gaoqing Yu[1,2] and Wen Liu[1,2*]

Abstract

The popularity of small office and home office routers has brought convenience, but it also caused many security issues due to vulnerabilities. Black-box fuzzing through network protocols to discover vulnerabilities becomes a viable option. The main drawbacks of state-of-the-art black-box fuzzers can be summarized as follows. First, the feedback process neglects to discover the missing fields in the raw message. Secondly, the guidance of the raw message content in the mutation process is aimless. Finally, the randomized validity of the test case structure can cause most fuzzing tests to end up with an invalid response of the tested device. To address these challenges, we propose a novel black-box fuzzing framework called MSLFuzzer. MSLFuzzer infers the raw message structure according to the response from a tested device and generates a message segment list. Furthermore, MSLFuzzer performs semantic, sequence, and stability analyses on each message segment to enhance the complementation of missing fields in the raw message and guide the mutation process. We construct a dataset of 35 real-world vulnerabilities and evaluate MSLFuzzer. The evaluation results show that MSLFuzzer can find more vulnerabilities and elicit more types of responses from fuzzing targets. Additionally, MSLFuzzer successfully discovered 10 previously unknown vulnerabilities.

Keywords Vulnerability discovery, Black-box fuzzing, SOHO routers, Feedback mechanism

Introduction

Internet of Things (IoT) technologies have snowballed in recent years, many of which have seen widespread adoption. According to a recent report, total semiconductor consumption for IoT endpoints is $130.2 billion in 2021 and will grow to $243.2 billion by 2026 (Yamaji 2022). This booming IoT ecosystem inevitably attracts cybercriminals. Among the 1.2 million IoT smart devices analyzed by Palo Alto Networks, 57% are vulnerable to medium or high-severity attacks (Unit 42 2020). Small office and home office (SOHO) routers are typical representatives of IoT smart devices. They are widely used to provide network services for various IoT smart devices, so the security of SOHO routers is crucial. This is especially true in the current climate where SOHO routers are being used more widely due to the increase in home office scenarios during the COVID-19 pandemic. Trend Micro reports that introducing vulnerable devices into the home will expose employees to malware and attacks that could sneak into the corporate network (Micro 2020).

There is thus an urgent need for techniques that can effectively and efficiently discover security vulnerabilities in SOHO routers. Security vulnerabilities in SOHO routers are usually implementation flaws in device firmware (Cheng et al. 2022), which is software that provides hardware support for upper-level users. Since SOHO routers are essentially computing devices with networking

*Correspondence:
Wen Liu
lw8206@cuc.edu.cn
[1] State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing, China
[2] School of Computer and Cyber Sciences, Communication University of China, Beijing, China

capabilities, they are not immune to attacks from the Internet as long as they can be accessed remotely through specific network protocols (Shu and Yan 2022). This has motivated researchers to explore how to perform fuzzing tests for device firmware over network protocols (Chen et al. 2018; Feng et al. 2021; Yu et al. 2022; Zhang et al. 2019). Unfortunately, device vendors usually do not provide source code, documentation, and firmware of these devices in public (Cheng et al. 2022; Redini et al. 2021), and the hardware debug interface is usually disabled (Chen et al. 2018). Therefore, black-box fuzzing is usually the only practical way to discover device vulnerabilities because of its independence of source code, firmware, and debugging capabilities (Shu and Yan 2022).

Black-box fuzzing is a software testing technique that does not have the source code of the target program nor the internal state of each execution (Zhu et al. 2022). Black-box fuzzers randomly mutate program inputs to generate a large corpus and feed each input to the program. Taking the built-in remote connection capability of SOHO routers into account, black-box fuzzers are usually designed to send mutated communication messages to the target device over the network to detect if it shows any symptoms of malfunction. Potential flaws or vulnerabilities may be found if a crash is triggered during execution or if the device sends back an exception message. In practice, SOHO routers usually verify the format and parameters of the input message, and most vulnerabilities usually exist in the function code after the sanitization code (Feng et al. 2021). Therefore, the test cases generated by black-box fuzzers based on simple random mutation usually fail to pass the sanitization. These test cases are even less able to reach deeper defective code, which makes the efficiency of vulnerability discovery extremely low.

State-of-the-art smart device black-box fuzzing methods such as SNIPUZZ (Feng et al. 2021) use the feedback response messages of smart devices to optimize the mutation process of seed messages to make the format of test cases as legal as possible. SNIPUZZ builds several probe messages by removing byte by byte from the seed message. The response messages from these probe messages are associated with the deleted bytes. Adjacent bytes with the same response message class are combined into a snippet mutation unit. These snippets are used in the subsequent mutation process. This method utilizes the vital feedback response information to constrain the mutation process and reduces the size of the input space to be searched.

Unfortunately, SNIPUZZ still has some unresolved challenges. First, the mutation process of SNIPUZZ is unguided. After SNIPUZ completed the construction of the snippet, it chose to perform havoc mutation, that is, randomly select some random snippets in the message and execute a set of mutation schemes for each selected snippet. This random selection of snippets is straightforward but blind, which causes SNIPUZZ to be unguided. The content of the raw message needs to be better exploited to guide the mutation process. Second, SNIPUZZ cannot discover potentially missing fields in the raw message. Some parameter fields in the raw message are empty by default. Since the default fields are logical fields and do not occupy actual bytes, SNIPUZZ cannot construct snippets corresponding to these default fields. This caused SNIPUZZ to fail to find such vulnerabilities when the vulnerability parameter was placed in an empty field. Finally, SNIPUZZ cannot guarantee the validity of its test case structure. Each snippet constructed by SNIPUZZ is equally weighted in the mutation process. This makes some critical snippets used to ensure the validity of the test case structure may be changed during the mutation process. When these vital snippets are changed, the entire test case may fail to pass the initial validity check of the IoT device and be discarded. This shows that the test case generation process of SNIPUZZ lacks constraints. For these reasons, SNIPUZZ does not consistently produce constrained, well-structured fuzzing inputs that reach deeper code locations and thus uncover more vulnerabilities.

Our approach

This paper proposes a novel black-box fuzzing framework, MSLFuzzer, to detect potential flaws and vulnerabilities in SOHO routers. We first define a message segment list, a data structure that normalizes the description of the message content. This data structure can better describe the attributes and sequence relationships of message segments than snippets do. To overcome the limitation of SNIPUZZ, MSLFuzzer improves the correlation process of response messages and completes the construction of initial message segments. Further, MSLFuzzer analyzes the message content of the initial message segment list to supplement the attribute information of the corresponding segment. Then, MSLFuzzer mines the sequence relationship between segments based on the n-gram algorithm to find those missing segments that are not reflected in the message, which we call invisible segments. Next, MSLFuzzer conducts hierarchical stability analysis on each message segment and determines the mutation priority between each segment. This makes the mutated message as far as possible through the early sanitization of the firmware to the deep-level code defect location. Compared with SNIPUZZ, MSLFuzzer mines deeper message content, making the fuzzing process more constrained and oriented and reducing the input search space size.

Contributions

We summarize the contributions of the paper as follows:

- *New technique* We develop a message content analysis method based on a list of message segments to improve the efficiency of black-box fuzzing for SOHO routers.
- *New framework* We propose a response feedback-based black-box fuzzing framework MSLFuzzer to find implementation flaws in SOHO routers. Based on the device feedback message and fine-grained message content analysis method, MSLFuzzer enriches the attribute and sequence information of the message segment list, making the fuzzing process more constrained and oriented.
- *Implementation and findings* We implemented a full-featured prototype of MSLFuzzer and constructed a dataset of 20 devices with 35 real-world vulnerabilities to evaluate the vulnerability discovery capabilities of MSLFuzzer. In total, 10 zero-day vulnerabilities were discovered by MSLFuzzer.

Roadmap

In the remainder of this article, Sect. "Background and related work" reviews the background and related work of IoT smart device communication architecture and SOHO routers fuzzing. Section "Methodology" presents a detailed design of the MSLFuzzer. The implementation details and evaluation results are summarized in Sect. "Implementation and evaluation". Section "Discussion and future work" discusses some limitations of the current design and points out future work. Finally, Sect. "Conclusion" concludes the paper.

Background and related work

SOHO routers are typical representatives of IoT smart devices. This section introduces common communication architectures for IoT smart devices, fuzzing for SOHO routers, and fuzzing response feedback mechanisms.

Common IoT smart device communication architecture

In a typical IoT ecosystem (such as a smart home environment), users deploy multiple smart devices for specific purposes, equipped with many sensors for external information collection and wireless connectivity modules for data transmission (Chen et al. 2018). These smart devices usually transmit data or receive control commands remotely through network protocols. To react to external raw input messages, most IoT smart devices implement a similar high-level communication architecture inside their firmware, including 1) Sanitizer, 2) Function Switch, 3) Function Definitions, and 4) Replier (Feng et al. 2021). The Sanitizer checks and parses the input when the IoT smart device receives external input. Suppose the input violates the protocol specification or syntax requirements. In that case, Sanitizer directly sends a response message describing the input error to notify the Replier and terminates the processing of the input. If the input passes the check, the Function Switch extracts the parameter name and value pairs from the input and transfers control to the corresponding functions. The process is terminated if no input parameter is found and a response message is returned. Different functions specifically implement the processing of parameter values passed by the user.

In the method of fuzzing the target smart device through a network protocol, the mutation operation on the message mainly focuses on the message structure and user input parameters (Zhu et al. 2022). Mutations to the message structure may trigger bugs in the target device's protocol stack or structure parsing components. It is usually located at the Sanitizer. Mutations to user input parameters may trigger the target device to receive and process user input parameters, usually in different functions. Due to the large number of functions and the frequent handling of external user data, there is a higher possibility of bugs and vulnerabilities hidden in functions. The mutated message needs to have a legal message structure as much as possible to pass the Sanitizer and Function Switch smoothly and reach the functions. Simple random mutations often cause the message structure to be corrupted and discarded before reaching the location of the defective code.

SOHO routers fuzzing

According to the amount of information observed during execution, fuzzing methods can be divided into black-box, grey-box, and white-box fuzzing (Eceiza et al. 2021). Black-box fuzzing does not know the internal state of each execution (Chen et al. 2018; Han et al. 2019; Lee et al. 2020). The object under test is a black box to fuzzers, which usually optimize the fuzzing process by exploiting input formats or different output states (Dinh et al. 2021; Han et al. 2019). White-box fuzzing usually needs to have all the source code of the target object and can obtain all the execution information of the target object during the fuzzing process (Huang et al. 2020). Grey-box fuzzing acquires knowledge of the execution state between black-box and white-box fuzzers. Grey-box fuzzers do not need to acquire the source code of the target object and usually use edge coverage as the internal execution state (Aschermann et al. 2019; Gan et al. 2018).

For SOHO routers, since device vendors usually do not provide the source code and documentation of device

firmware in public, white-box fuzzers that rely on source code are not suitable for SOHO routers fuzzing (Cheng et al. 2022). Grey-box fuzzers need to obtain target runtime context information, and when applied to SOHO routers, they usually need to be combined with an emulator (Eceiza et al. 2021). Emulators can execute programs originally running on IoT firmware without corresponding hardware. Typical emulators include Avatar (Zaddach et al. 2014), Avatar2 (Muench et al. 2018), Firmadyne (Chen et al. 2016), and FirmAE (Kim et al. 2020). An emulator can provide a test object for both a grey-box fuzzing method and a black-box fuzzing method, which can reduce the dependence of the fuzzing method on actual equipment and the economic cost. Black box fuzzing only requires external access to the emulated device as if it was an actual device. The grey-box fuzzing requires the further acquisition of contextual information during the emulation process and firmware execution, especially the edge coverage. The higher the edge coverage, the more branch conditions are triggered. If a test case triggers a new branch, it will be treated as an exciting seed (Zhu et al. 2022).

Typical grey-box fuzzers for SOHO routers include Firm-AFL (Zheng et al. 2019), FirmHunter (Yin et al. 2021), CGFuzzer (Yu et al. 2022), and IoTHunter (Khandait et al. 2021). When firmware can be emulated successfully, and the edge coverage information can be obtained, the grey-box fuzzer can effectively find flaws and vulnerabilities in the firmware. Unfortunately, not all device firmware is publicly available (Redini et al. 2021). Meanwhile, firmware unpacking and analyzing firmware is a challenging task since firmware may have multiple formats and can run on different architectures (Wang et al. 2019). Furthermore, emulators have a relatively low success rate for emulating firmware and the range of applicable vendor device firmware (Kim et al. 2020). Grey-box fuzzing methods that rely on emulation and debugging capabilities do not apply to devices that cannot be successfully emulated or debugged.

Benefiting from the ability of SOHO routers to communicate with the network, security researchers usually perform black-box fuzzing on target SOHO routers based on network communication. Some black-box fuzzing methods try to start from the mobile app side that communicates with SOHO routers (Chen et al. 2018; Redini et al. 2021). IoTFuzzer (Chen et al. 2018) analyzes the UI elements of the app and then reversely identifies the relevant program elements that send messages to the device from the control events through data flow analysis. Finally, the mutation operation of the corresponding field is completed. DIANE (Redini et al. 2021) views the execution of an app as a series of functions that transform user-introduced data into network data. Based on IoTFuzzer, DIANE converts the input position of mutation data from the first function to the last function so that the generated mutation data is not subject to application-side validation. These methods can effectively find vulnerabilities in the code that communicates with the application on the target device. However, not all devices have corresponding mobile apps. At the same time, this approach also fails to find vulnerabilities in components of the target device that do not communicate with the mobile app.

Other black-box fuzzing methods directly analyze the captured raw communication messages of SOHO routers. Boofuzz (Pereyda 2022) is an excellent successor to the classic Sulley (Amini et al. 2019) fuzzing framework. Boofuzz uses human knowledge guidance to solve input problems. Before fuzzing, Boofuzz requires users to define a set of highly customized messages and write corresponding independent scripts for each captured raw communication message. Unlike Boofuzz, SRFuzzer (Zhang et al. 2019) first captures many web requests from physical devices and then models user input semantics to generate test cases. SRFuzzer builds a Key-Value model for each message content in the request to distinguish the data type of the value and assign different mutation rules. Based on SRFuzzer, ESRFuzzer (Zhang et al. 2021) adds a new D-CONF mode that can detect some issues that SRFuzzer misses, including memory corruption, command injection, and stored cross-site scripting. SRFuzzer and ESRFuzzer mainly consider the Key-Value model, but in practice, there are various message formats. In addition to key-value pairs, there are JSON, XML, soap, and custom message formats (Feng et al. 2021). To be applied to various devices, the new solution should be able to infer the format from the raw message.

Response feedback mechanism

Since most existing SOHO routers network black-box fuzzers (Chen et al. 2018; Pereyda 2022; Redini et al. 2021) do not have an excellent mechanism to constrain the format of test cases, some researchers try to use the response messages of SOHO routers to guide the mutation process.

SNIPUZZ (Feng et al. 2021) is the first method that proposed using the response messages of a device as feedback to guide the fuzzing strategy. Specifically, SNIPUZZ first collects the response message corresponding to the deleted byte by deleting the raw message byte and sending the deleted message to the target device. SNIPUZZ judges whether different test cases cover different code execution paths in the device firmware through the content of the response message. Based on this mechanism, SNIPUZZ uses a novel heuristic algorithm to detect the role of each byte in the message. Adjacent bytes with

similar response messages have the same role in the initial message fragment and can be packed together and linked into a basic mutation unit. The feedback mechanism of SNIPUZZ is novel, and its packaged fundamental mutation unit narrows the search range of test cases to a certain extent.

SNIPUZZ did not further explore the constructed mutated units. Specifically, first, SNIPUZZ did not perform content analysis on the mutated units themselves. The mutation unit constructed by SNIPUZZ is only a byte fragment, and its data type and semantic meaning (such as IP, domain name, and MAC address) have not been analyzed. Content analysis of mutated units can effectively guide the mutation process. Second, SNIPUZZ does not recognize the default variable field in the message. Some fields are usually allowed to default in the communication message of the target device, and the fields corresponding to the default fields in the raw message are usually empty. Therefore, the mutation unit sequence constructed by SNIPUZZ by deleting the raw message byte by byte cannot identify and represent the empty default field. These default fields corresponding to variable fields are often essential entry points for triggering potential vulnerabilities in firmware. Finally, there is an equivalence among mutated units constructed by SNIPUZZ. In other words, SNIPUZZ adopts the Havoc method in mutation. That is, some random mutation units in the message are randomly selected, and a random mutation scheme is performed on each selected mutation unit. These mutation units represent the format and structure of the raw message, and randomly selecting mutation units to mutate will destroy the structure of the raw message. Although this can find the vulnerabilities of those format parsing components in the device firmware to a certain extent, it also prevents mutation messages from reaching deep functional code, which may contain more defects and vulnerabilities. Therefore, the new fuzzing scheme based on the feedback mechanism should further consider the analysis and exploration of the content and structure of the mutation unit.

Methodology

To describe the structure and content of the message in a normalized manner, we first define a data structure called message segment list, short for MSL, as in (1). *MSL* consists of n message segments with attributes, where $n \in N$. The definition of a message segment, in short for ms, is shown in (2), each message segment has m attributes, where $m \in N$.

$$MSL = [ms_1, ms_2, \ldots, ms_i, \ldots, ms_n] \tag{1}$$

$$ms = [a_1, a_2, \ldots, a_j, \ldots, a_m] \tag{2}$$

Typical segment attributes include value, data type, encoding type, data semantics, segment stability level, data nesting type, etc. The value of the message segment corresponds to the contiguous bytes of the raw message. The concatenation result of all message segment values in the message segment list is the raw message. Since the message segment list carries all the bytes of the raw message and contains the sequence relationship, any original message can be represented by MSL, and MSL can be directly serialized into the raw message. In the fuzzing process, the serialized mutated message can be directly obtained by mutating each message segment in MSL.

Based on MSL, we present the detailed design of MSLFuzzer as illustrated in Fig. 1. MSLFuzzer receives a raw message as input, obtains the corresponding feedback response message by constructing the probe message, and stores it in the response pool. These feedback response messages group the bytes in the raw message

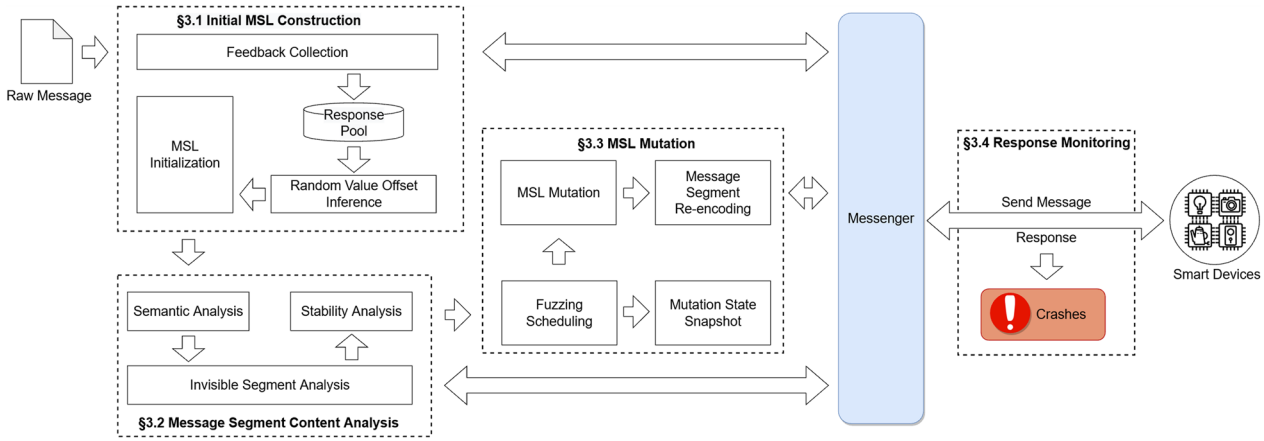

Fig. 1 Overview of MSLFuzzer

and build the initial MSL (Sect. Initial message segment list construction). MSLFuzzer performs semantic analysis, invisible segment analysis, and stability analysis on each segment in MSL. The result of the analysis is added to the corresponding segment as an attribute of each message segment (Sect. Message segment content analysis). Then, the constructed list of message segments with attributes will be used as a normalized mutation seed for fuzzing. MSLFuzzer mutates the list of message segments with attributes, serializes the result of each mutation into a mutated message, and sends it to the target SOHO router through the messenger (Sect. Message segment list mutation). Finally, MSLFuzzer will output crash exceptions and mutated messages that trigger crashes (Sect. Response monitoring).

Initial message segment list construction

For black-box fuzzing, since the internal execution information cannot be obtained from inside the device, the feedback response messages of the SOHO routers are regarded as a valuable source of device state information at runtime. Different bytes in the message can be associated with the corresponding response category, obtained by sending a probe message constructed after the corresponding bytes are deleted to the target device. This association relationship can provide support for the construction of message segments. In this process, a challenge is to correctly identify the randomness in the response message, such as timestamps or tokens. These contents will significantly interfere with the classification of response messages. We improved the heuristic approach of SNIPUZZ to correlate the bytes in the raw message and the response message class. At the same time, we propose a difference-based response format random value offset inference method to avoid the interference of random values on response classification to the greatest extent. Specifically, the entire initial segment list construction process is divided into three steps: 1) feedback response message collection, 2) random value offset inference, and 3) message segment list construction.

Feedback response message collection

MSLFuzzer first constructs several probe messages by deleting the corresponding bytes in the raw message byte by byte. These probe messages will be sent to the target device. The response message of each probe message will be associated with the deleted bytes in the probe message.

Random value offset inference

After completing the association of the bytes in the raw message with the corresponding response message, the response message needs to be classified to realize the clustering of adjacent bytes in the raw message. However, due to the influence of random values in the response, the categories of response messages with the same semantics may need to be correctly merged. Therefore, it is necessary to find where random values may be located and eliminate their interference when calculating the response message.

Considering that in most public or private protocols, although the protocol usually has extension parts, the relative offset of some random values in the protocol is usually fixed. Therefore, we make two assumptions: 1) The offset of the random value in the response is fixed relative to the starting position of the response header; 2) Two response messages with the same semantic meaning differ only in random values and the same in other positions. We propose a difference-based response random value offset inference method based on these two assumptions, as shown in Algorithm 1.

Algorithm 1: Random Value Offset Inference

Input: The associated set of bytes and response messages R
Output: Random value offset set O'

1 Initialize parameters of inference function.
2 $P = \emptyset$
3 **for** $R_i \in R$ **do**
4 $\quad P_t = \emptyset$
5 $\quad$ **for** $R_j \in R[i:]$ **do**
6 $\quad\quad ES_{ij} = editing\_similarity(R_i, R_j)$
7 $\quad\quad$ **if** $ES_{ij} > p$ **then**
8 $\quad\quad\quad P_t \leftarrow (R_i, R_j)$
9 $\quad P \leftarrow P_t$
10 **for** $Q \in P$ **do**
11 $\quad$ **for** $Q_i \in Q$ **do**
12 $\quad\quad$ **for** $Q_j \in Q[i:]$ **do**
13 $\quad\quad\quad O_k, V_k, N_k = diff(Q_i, Q_j)$
14 $\quad\quad\quad S_{ovn} \leftarrow O_k, V_k, N_k$
15 $NP = calc\_num(R)$
16 **for** $O_i, V_i, N_i \in S_{ovn}$ **do**
17 $\quad$ **if** $N_i/NP > q$ **then**
18 $\quad\quad O, V, N \leftarrow O_i, V_i, N_i$
19 $O, V, N = consecutive(O, V, N)$
20 $O' \leftarrow merge(O, V, N)$

The input to the algorithm is the set of bytes in the raw message and the associated response message R.First, in the set R of all response messages, the editing similarity ES_{ij} between response messages R_i and R_j is calculated according to (3) (lines 1–6). When the similarity ES_{ij} of the two response messages is greater than the threshold p, the two messages are considered to belong to the same category P_t (lines 7–8). The threshold p is the average initial self-similarity of each probe message and is calculated according to (4), where NP is the total number of responses. The threshold p is used to distinguish whether different messages belong to the same category. Its value

ranges from 0 to 1. When the similarity between message A and message B is greater than the average self-similarity threshold p, message A and message B are considered to belong to the same category. The sample mean is often used to distinguish whether two classes of targets belong to the same category. For example, in the field of intrusion detection, the sample mean is used as a boundary to distinguish between normal and abnormal behaviour (Jones and Sielken 2000). MSLFuzzer computes self-similarity ES_i for each probe message pm_i. The probe message pm_i corresponding to the i th byte is constructed by removing the i th byte in the raw message. MSLFuzzer sends the same probe message pm_i twice within a second interval. The two responses r_i, r_j are collected from the target device accordingly. The self-similarity ES_i is calculated according to (3). All categories P_t together form the response category pool P (line 9).

$$editing\_similarity(R_i, R_j) = 1 - \frac{edit\_distance(R_i, R_j)}{\max\_len(R_i, R_j)} \tag{3}$$

$$p = \frac{\sum_i^{N_p} ES_i}{NP} \tag{4}$$

Second, for a response class Q in the response pool P, pairwise combine the response messages Q_i and Q_j in Q (lines 10–12). Next, make a difference between Q_i and Q_j for each combination and record all inconsistent byte positions O_k, the value V_k of the position, and the number of responses N_k that have had differences in the byte position (line 13). O_k, V_k, N_k are recorded in set S_{ovn} (line 14). The total number of responses NP is statistically obtained for subsequent calculations (line 15). Then, all inconsistent byte offset in set S_{ovn} are filtered. For each set of O_i, V_i, N_i in S_{ovn}, when the proportion of N_i in the total number of responses NP is greater than threshold q, the corresponding inconsistent byte offset O_i is reserved, and O_k, V_k, N_k are recorded in set O, V, N respectively (lines 16–18). Among them, O is the set of all inconsistent byte positions that meet the requirements, V is the set of values corresponding to all inconsistent byte positions in set O, and N is the number of responses with differences in all inconsistent byte positions in set O. The threshold q is calculated according to (5). This threshold q must meet three conditions: 1) The value range is from 0.5 to 1. That is, the proportion should be at least greater than half of the total number of responses, but at the same time, this threshold must be less than 1 because the byte offsets found may not appear in all responses at the same time and there needs to be slight fault tolerance. 2) The value of this threshold is related to the total number of responses NP. The larger NP is, the larger q is. This is to prevent significant differences in threshold settings in various situations due to changes in NP size. The larger NP is, the higher the proportion should be. 3) When NP approaches positive infinity, the limit of this threshold is 1. In formula (5), the range of NP is all positive integers greater than zero. And q increases positively with NP. The larger NP is, the larger q is. Satisfying condition 2. When n equals 1, q is 0.5. When n approaches positive infinity, the limit of q is 1. Satisfying conditions 1 and 3). Therefore, this formula meets all conditions for setting threshold q.

Further, all consecutive inconsistent byte offsets in set O are preserved, discrete inconsistent byte offsets are removed, and the sets V and N should also be modified accordingly (line 19). Finally, the intersection $O\prime$ of consecutive inconsistent bytes in all response classes is obtained. If these categories only intersect part of the bytes, the maximum consecutive inconsistent bytes are taken (line 20). Potential response random number byte position $O\prime$ is output.

$$q = \frac{NP}{NP + 1} \tag{5}$$

MSL initialization

After obtaining the potential response random number byte position, the response message can be classified. Different from the calculation method of SNIPUZZ, MSLFuzzer first corrects the response message according to the potential byte offset of the random value. Specifically, for the response messages r_i and r_j, the byte segment r_o corresponding to the potential random value offset O is deleted to obtain $r_i^{'}$, $r_j^{'}$, as shown in (6). The corrected response message is then used to calculate the similarity. As shown in (7), the corrected self-similarity score $ES_i^{'}$ of the probe message pm_i is obtained by calculating the similarity of its two corrected response messages $r_i^{'}$ and $r_j^{'}$.

$$r_i' = r_i - r_o,\ r_j' = r_j - r_o \tag{6}$$

$$ES_i' = 1 - \frac{edit\_distance\left(r_i', r_j'\right)}{\max\_len\left(r_i', r_j'\right)} \tag{7}$$

Next, the response category needs further determined to be the basis for merging the bytes in the raw message. It is important to note that the method for combining response categories in this step differs from the method for combining response categories during random value offset inference. The response categories obtained during random value offset inference are just a coarse-grained classification algorithm to aid in finding random value

offsets. Under the control of a threshold p, this coarse-grained classification helps determine random value offsets but is not suitable for fine-grained classification for this step. MSLFuzzer performs fine-grained similarity calculation for each pair of response messages of each probe message based on (7). When the corrected similarity ES'_{ij} of the two response messages r_i and r_j is greater than the corrected self-similarity ES'_i or ES'_j of each response message, the two response messages are considered to belong to the same category. Finally, the response category corresponding to each response message can be obtained.

When all response classes are successfully classified, several adjacent bytes with the same response class in the raw message will be merged into the same message segment. Adjacent bytes will be merged into the byte stream as the initial value of the new message segment. All constructed message segments are assembled in the byte order corresponding to the raw message, and the initial MSL is completed.

Table 1 Message content analysis items

Analysis category	Analysis item
Nested structure analysis	XML
	JSON
	Key-Value Pair
Data type analysis	Alphabet
	Number
	Symbol
	String
	Non-printable Bytes
Semantic analysis	IP Address
	MAC Address
	Protocol Name
	Domain Name
	Logical Keywords
Field encoding analysis	Base32
	Base64
	URL

Message segment content analysis

After obtaining the initial MSL, MSLFuzzer analyzes each initial message segment, including 1) message semantic analysis, 2) invisible segment analysis, and 3) message stability analysis.

Message semantic analysis

The raw message usually contains control instructions, configuration parameters, or data payloads, which the user usually sends to the SOHO router. Although the raw message may be a printable string or a custom byte stream, the field data carried in it are printable string information in many cases. Therefore, semantic analysis of each message segment can better help the fuzzer understand the characteristics of the raw message. MSLFuzzer uses a heuristic-based message semantic analysis method to perform semantic analysis on the message segment list. The core idea of the heuristic message semantic analysis method is to analyze and arbitrate the target message segment by category. MSLFuzzer divides analysis categories into four categories: nested structure analysis, data type analysis, semantic analysis, and field encoding analysis. The analysis items supported by these analysis methods are shown in Table 1.

Nested structure analysis refers to analyzing the potentially nested structure (such as JSON and XML) in the message segment and further dismantling it according to its structural characteristics to form a more fine-grained message segment. The previous MSL construction process has disassembled the underlying nested structure in the raw message. However, in practice, when faced with some devices whose response messages are not detailed, there may still be nested structures in the message segment. For example, some devices use a unified message to report all errors. Further splitting of these message segments in the nested format can effectively increase the fuzzing capability of devices whose response messages need to be more detailed. Data type analysis refers to the analysis and identification of the data type of the value of the target message segment, such as letters, symbols, numbers, non-printable byte streams, or a combination of the above types. Semantic analysis refers to analyzing and identifying the semantic meaning of the value of the target message segment, such as IP address, MAC address, and domain name. Field encoding analysis refers to analyzing the encoding format used to identify the value of the target message segment, such as base64 encoding and URL encoding.

MSLFuzzer performs heuristic recognition for different semantic categories. We mainly adopted the heuristic method's idea of trial and error and the rule of thumb. Specifically, for nested format analysis and field encoding analysis, MSLFuzzer leverage standard nested format parsing tools and encoding parsing tools to try to parse the target message segment. If the parsing is wrong, continue using the following parsing tool. Until one of the parsing tools resolves correctly or all parsing tools fail to parse. If there is a correct parsing result, this field's nested type or encoding type is the corresponding parsing tool type. Message segments with nested types are broken down into finer-grained message segments. A message segment with an encoding type is supplemented with an encoding type attribute. For data type analysis and semantic analysis, MSLFuzzer mainly matches based on the data type rule table and semantic meaning rule table. Since the communication packets of SOHO routers usually contain information related to network configuration, the rule

table contains matching rules for common data types and semantics related to network configuration. When the value in the message segment matches the corresponding item in the table, MSLFuzzer considers the semantic or data type of the field to be the corresponding type in the rule table. The corresponding data types and semantic meanings are supplemented by the attributes of the message segment, providing essential information for the subsequent analysis process.

Invisible segment analysis

A single raw message usually carries multiple fields, but some fields are empty by default if not configured. In this case, these empty segments will not be included in the constructed message segment list. We call these unconfigured empty segments: invisible segments. When the variables that trigger the vulnerability are in these invisible segments, since these invisible segments do not occupy an independent position in the constructed message segment list, it is usually challenging to trigger these vulnerabilities by directly mutating the message segment list. These invisible segments need to be found and added to the message segment list to improve the effectiveness of fuzzing.

Black-box fuzzing cannot obtain the internal execution code and memory contents of the device, so we cannot directly obtain the specific invisible segment location. Fortunately, MSLFuzzer has constructed a complete message segment list in the previous steps. The basic properties of segments have also been supplemented, so we can use the sequence properties of the message segments themselves to determine the location of the invisible segment. MSLFuzzer adopts an n-gram-based invisible segment expansion algorithm, as shown in Algorithm 2.

Algorithm 2: Invisible Segment Expansion Algorithm

Input: The message segment list with attributes MSL

Output: The message segment list with invisible segment MSL_I

```
1  Initialize parameters of expansion function.
2  GN = n_gram(MSL)
3  GN1 = n1_gram(MSL)
4  for GN_i ∈ GN do
5      for GN1_i ∈ GN1 do
6          TS_n = type_sequence(GN_i)
7          TS_n1 = type_sequence(GN1_i)
8          TD, m = edit_distance(TS_n, TS_n1)
9          if TD == 1 and m is insert then
10             i_o = get_insert_offset(TS_n, TS_n1)
11             if GN_i[i_o − 1] == GN1_i[i_o − 1] then
12                 if GN_i[i_o + 1] == GN1_i[i_o + 1] then
13                     T ← creat_template(GN_i, GN1_i)
14 for T_i ∈ T do
15     ISP ← search_invisible_segment(MSL, T_i)
16 for ISP_i ∈ ISP do
17     MSL_I ← insert_invisible_segment(MSL, ISP_i)
```

The input to the algorithm is MSL with attributes constructed in the previous step. First, MSLFuzzer uses the n-gram and (n + 1)-gram algorithms in sequence on MSL to obtain the n-gram sequence set GN and the (n + 1)-gram sequence set $GN1$(line 1–3). Based on practical experience and experimental results, the value of the parameter n in the n-gram algorithm is set to 4. In Sect. "Invisible segment identification", we provide details of the experiments conducted. Next, compare each gram GN_i in GN with each gram $GN1_i$ in $GN1$ in turn to obtain the respective type attribute sequences TS_n and TS_{n1} (lines 4–7). Based on the edit distance, the attribute edit distance TD and edit distance operation method m of TS_n and TS_{n1} are calculated (line 8). When TD is equal to one and the operation mode is inserted, the inserted position in TS_n is considered as a candidate invisible segment (lines 9–10). Then, MSLFuzzer performs regression judgment on the candidate invisible segments. If the value of the segment before and after the potential invisible segment in the n-gram is the same as the value of the corresponding (n + 1)-gram, the candidate invisible segment is considered as an invisible segment, and an invisible segment sequence template T is created (lines 11–13). Finally, MSLFuzzer searches for the positions of all invisible segments in the raw message according to the invisible segment sequence template and inserts invisible segments in the corresponding positions to obtain MSL_I (lines 14–17).

The message segment list is used for the subsequent analysis process after the completion of the invisible segment expansion. Since the default value of inserted invisible segments is empty, only when these invisible segments are selected to mutate their values will be filled with mutated data. Therefore, when other segments are mutated, invisible segments will not affect the mutation results of other segments. At the same time, the issue that the variable that triggers the vulnerability appears in the invisible segment can also be solved.

Message stability analysis

Most of the defect code generated by vulnerabilities is located in various functional codes after the protocol analysis component. According to software development practice, before the data in the raw message reaches the actual defect code, the protocol format and data format of the raw message need to pass sanitization. When these sanitization constraints are reflected in a fuzzer, the fuzzer needs to generate mutated data that conforms to the specification as much as possible. More vividly, building a message segment list is similar to building a block castle. The process of mutating the message segment list

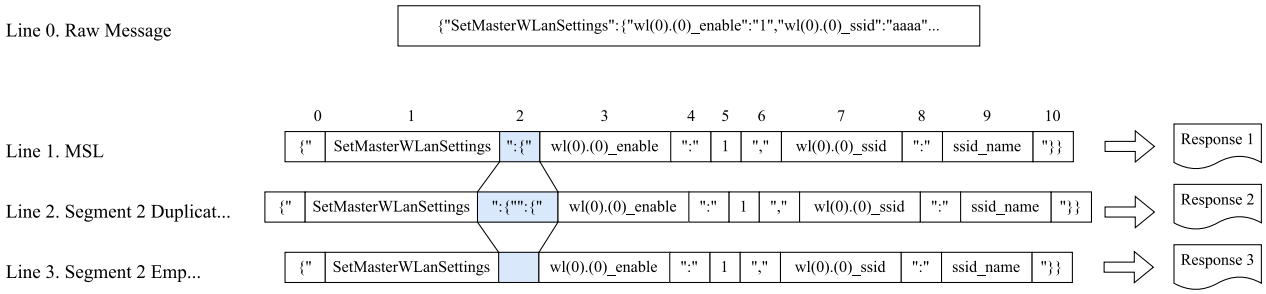

Fig. 2 Example of message segment stability judgment process

to generate data that the firmware can verify is similar to removing blocks from a block castle one by one and ensuring that the block castle does not collapse. Once the blocks that play a key supporting role in the structure are pulled away, the entire block castle will collapse. Similarly, once the message segments checked by the firmware are changed, the entire mutated message may fail to pass the check.

To better describe the stability of each message segment, we introduce the concept of message stability, which is used to describe the satisfaction degree of mutated messages to message legitimacy. Because the raw message is an essential seed for mutation, we usually assume that the structure and content of the original message seed are valid, and its corresponding initial response is the correct response. This correct response can be used to infer the stability of each message segment. MSLFuzzer first copies and blanks the value of the target message segment and constructs two new messages based on the copied and blanked message segment. MSLFuzzer sends a new message to the target device and receives a response message corresponding to the message. Finally, MSLFuzzer determines the degree of stability of a message segment by judging the difference between the response of the constructed message and the response of the seed message.

Figure 2 shows an example of this process. Among them, Line 0 is the raw message, and Line 1 is the constructed MSL. Response 1 is the response obtained by serializing the MSL in Line 1 into a message and sending it to the target device. Assuming that we need to make a stability judgment on the message segment numbered 2 in Line 1, MSLFuzzer first copies the value of message segment 2 and constructs a new message segment, as shown in Line 2. The MSL in Line 2 completes the serialization and sends it to the target device to get Response 2. Similarly, the MSL in Line 3 and the corresponding Response 3 can be obtained. Later, MSLFuzzer judges the stability of the message segment numbered 2 based on Response 1, Response 2, and Response 3.

Table 2 Message stability level

Level	Segment value duplication	Segment value empty
0	Changed	Changed
1	Unchanged / changed	Changed / unchanged
2	Unchanged	Unchanged

Stability is classified into 3 levels, as shown in Table 2. Level 0 stability is also called *root stability level*. For a message segment, when both Response 1 and Response 2 are changed, no matter whether the value of the segment is copied or blanked, the stability level of the message segment is the root stable level. In this case, the raw message bytes corresponding to the message segment are usually some message identification fields or key structure fields to be checked. Level 1 stability is also called *structural stability level*. When the segment value is copied or blanked, only one of Response 1 and Response 2 will change. In this case, the raw message bytes corresponding to the message segment are usually specially checked data values or edge structures. Level 2 stability is also called *value stability level*. Neither Response 1 nor Response 2 changes when segment values are inserted outside or inside the segment. In this case, the corresponding raw message are usually the variable values carried by the message.

A message segment with a higher stability level indicates a higher probability of passing the sanitization during the mutation process, and the exception-handling mechanism of the SOHO router does not easily capture the mutation message. Message segments of higher priority should be mutated first. A message segment with a lower stability level is more likely to be used as the validation field in the mutation process. Mutating the data of these message segments makes it easy for the mutated message to be caught by the exception-handling mechanism of the device, which means that the probability of the mutated message reaching the deep defect code

location is lower. Therefore, the mutation priority of a message segment with lower stability levels should be lowered. The stability level value of the segment is added as a property of MSL after it is determined. The enriched message segment list will be used for the subsequent mutation process.

Message segment list mutation

After the previous step is finished, the list of message segments complemented by semantic information will be used as input to the mutation phase of fuzzing.

Message segment mutation

MSLFuzzer mutates the message segments list obtained in the previous stage. It is worth noting that the mutation strategy is performed on a single message segment rather than a single byte in the raw message. For effective fuzzing, MSLFuzzer combines the attribute information of each message segment in the mutation process. MSLFuzzer uses the following strategies for mutation:

- *Byte Fragment Length Manipulation* MSLFuzzer changes the length of the value of the message segment to trigger buffer overflow vulnerabilities or out-of-bounds access vulnerabilities. MSLFuzzer generates random or repeated strings or byte fragments of varying lengths.
- *Numeric Bounds Break* MSLFuzzer alters the value of an integer, double, or float value to cause an integer overflow or out-of-bounds access vulnerability. MSLFuzzer generates tremendous, negative, zero values or data boundary values.
- *Empty Fill* An empty data field may crash the firmware if the data field is not properly checked. Therefore, MSLFuzzer deletes the entire fragment to clear the data field.
- *Byte Flip* MSLFuzzer flips all the bytes in the message segment to detect potential errors in the parsing code.
- *Specification Check* In order to detect more device execution states, MSLFuzzer replaces some potential judgment characters, such as "on" and "off", "true" and "false", according to a predefined knowledge base. At the same time, MSLFuzzer also identifies message segments if their attributes contain semantic information (such as IP address and domain name) according to the knowledge base. The values of these message segments are replaced with values that violate the semantic specification to detect flaws in the semantic format check components.
- *Command Inject.* Due to many command injection vulnerabilities in SOHO routers, MSLFuzzer inserts a set of predefined random command injection payloads into the segment value of stability level 1 or 2. The execution commands included in these payloads usually change the network status of the target device (for example, a reboot command) so that the subsequent network monitoring process can identify an abnormal network connection status.

Message segment encoding

Since some SOHO routers encode fields in the message, directly fuzzing the encoded fields will cause the device to fail to decode and interrupt the message-processing workflow. MSLFuzzer uses the corresponding decoder to decode the value of the message segment according to the encoding attribute in the message segment. The decoded message segment value is then subjected to the above mutation operation. MSLFuzzer finally uses the corresponding encoder to re-encode the mutated result. The encoded mutated value is used as the new message segment value.

Fuzzing scheduling

The conditions that trigger a crash can be complex. For example, modifying different data fields in the same message may be necessary to trigger an error. Nevertheless, at the same time, it is easy to be rejected by the device due to the mutation of all fields. So, we want to randomly select one or more subsets of fields to mutate while maintaining the stability of the message structure rather than mutating all fields simultaneously. MSLFuzzer randomly selects the variant message segments in the message segment list based on the stability level of each message segment, and the message segment with a higher stability level has a higher probability of being selected. After selecting mutated message segments, MSLFuzzer executes the above mutation scheme for each segment, serializes the mutated message segment list into a complete message, and sends it to the target device. A new message segment combination will be selected for the next fuzzing round only when the entire mutation process lasts for a user-defined time or completes a predefined number of mutations. After fuzzing scheduling, MSLFuzzer will record and store the combination of protocol message segments selected in the schedule into a JSON mutation state snapshot file. Whenever MSLFuzzer re-runs, it will first load the mutation state snapshot and then perform the fuzzing process to avoid the loss of the previous fuzzing state information.

Response monitoring

Since a black-box fuzzer does not have intrusive memory access to SOHO routers, MSLFuzzer identifies whether there are potential anomalies or vulnerabilities triggered by monitoring the network activity of the target device. For the potential command injection vulnerability in the device, since the default injection payload used by MSLFuzzer is the command to change the network state of the target device, the detection of the injection result can thus be implemented by checking the network activity state of the device. Therefore, when fuzzing a device, MSLFuzzer analyzes the response messages to detect crashes.

Specifically, MSLFuzzer sends a raw message and monitors regular device activity response results before mutation data is sent to the device. When the device works correctly, MSLFuzzer performs the subsequent fuzzing process and receives a response for each mutated message. Suppose the response message times out, or the connection is interrupted during this process. In that case, MSLFuzzer will repeatedly send the mutated message to check whether the exception is a false positive caused by the slow processing speed of the device. When there is no response to the mutated messages sent repeatedly, MSLFuzzer will send the standard probe message again. If the normal probe message has a response, it is considered that the message processing component has crashed. Otherwise, the entire device service is considered to have crashed. Because in some crash scenarios (such as some command injection vulnerabilities), the device will send a response message at first and no response after, which will cause the following unrelated mutation message to be recorded instead of the previous probe message that triggers the crash. Therefore, when a crash is found, MSLFuzzer will log the context mutation message that triggered the crash for subsequent further analysis.

Implementation and evaluation

This section introduces a prototype implementation of MSLFuzzer and analyzes the evaluation results. Specifically, Sect. "Framework implementation" provides implementation details, and Sect. "Experiment setup" presents the experimental setup. Sect. "Features of MSLFuzzer" compares the features of MSLFuzzer with state-of-the-art fuzzers. Sect. "Efficiency" discusses the effectiveness of MSLFuzzer based on the evaluation results. Sects. "Response category trigger" and "Invisible segment identification" evaluate the ability of MSLFuzzer to trigger response categories and discover invisible segments, respectively. Finally, Sect. "Unknown vulnerability identification" explores the ability of MSLFuzzer to discover unknown vulnerabilities.

Framework implementation

We have implemented a fully functional MSLFuzzer prototype with about 5000 lines of Python code. The design of MSLFuzzer includes four stages: initial message segment list construction, message segment content analysis, message segment list mutation, and response monitoring. These core functions are packaged in this prototype. Since the input to MSLFuzzer is raw messages, we use Wireshark (2022) to capture the communication packets of SOHO routers and manually sanitize these message sequences as input to MSLFuzzer. Specifically, we first use Wireshark to capture standard communication packets of SOHO routers and store all communication traffic as packet capture (PCAP) file. Secondly, we used Wireshark to track and analyze each TCP or UDP flow in the data packet and manually selected some typical seed packets as candidate seeds. Next, we analyzed these candidate seed messages and extracted some critical bytes from them as filtering conditions for filtering. We manually code Wireshark's filtering rules based on the characteristic vital bytes. We applied them to the entire data packet to filter out all data packets that meet the filtering rules. Finally, we do a regression check on all the filtered data packets to determine whether they are our expected packets. We use the expected message as the input of MSLFuzzer.

To make MSLFuzzer easier to be driven by external programs, we wrote a fuzzer wrapper to drive MSLFuzzer to fuzz the target device. This wrapper takes the raw messages from the packets and passes them to MSLFuzzer as input. At the same time, when a raw message contains authentication credentials, the wrapper will dynamically update the authentication credentials in the raw message before the fuzzing process according to different types of devices to ensure the validity of the raw message in the fuzzing process.

Experiment setup

Dataset

Existing IoT black-box fuzzing work is usually evaluated on physical devices. The physical devices selected for each job are different, and many physical devices have been sold out and cannot be purchased. At the same time, since the firmware in the physical device may have been updated, many historical vulnerabilities cannot be exploited. Therefore, it is challenging to reproduce the evaluation experiments of each work fully. This makes it difficult for comparative experiments of different works to compare fairly on a relatively deterministic dataset. With this in mind, we constructed a deterministic, easy-to-use dataset that would make it easier and affordable for other researchers to reproduce our work and compare it more easily with other methods.

Firmware emulation technology can meet the needs of low-cost and rapid construction of IoT test environments. Although the success rate of firmware emulation technology for different manufacturers and device models varies greatly, and the overall emulation success rate is low, from the perspective of building a dataset, we can still get some candidates by increasing the base of available firmware. A successfully emulated firmware is finally used for an experimental evaluation. It is worth noting that our method and the other methods compared in this paper are all black-box fuzzing methods and do not depend on the success rate of firmware emulation. The emulation-capable firmware is only used to build the test environment. For experimental evaluation, whether the test environment is, an emulated device or a physical device is transparent to the evaluated method.

To better evaluate the efficiency of MSLFuzzer, we compare MSLFuzzer with other methods on a particular dataset to evaluate its effectiveness in discovering vulnerabilities. We need to build a dataset of known vulnerabilities of SOHO routers. We mainly considered the following factors when constructing this dataset: 1) Real-world firmware image. Every firmware image in the datasets should be from the real world. These firmware images should cover mainstream architectures such as MIPS and ARM. 2) Real World Vulnerabilities. Every firmware image in the datasets should contain the real-world vulnerability. Selecting real-world vulnerabilities is more effective for verifying the performance of fuzzers in practice than artificially implanted vulnerabilities through the forward porting method (Hazimeh et al. 2020). 3) Typical vulnerability types. The types of vulnerabilities in the firmware should cover typical SOHO router device vulnerabilities. The typical types of vulnerabilities mainly studied in the existing SOHO router device black-box fuzzing research work include memory corruption, command injection, and denial of service (Feng et al. 2021; Shu and Yan 2022; Zhang et al. 2021). Therefore, the vulnerabilities contained in the baseline firmware image should contain the above vulnerability types. 4) Emulation and fidelity of firmware images. Every benchmark firmware should be able to be emulated successfully and efficiently used. The fidelity of the firmware image also needs to be verified. For example, many emulated firmware crashes after deep interactions. Such as, clicking on a web page to set properties affects the validity of fuzzer evaluations. Therefore, the fidelity of the firmware image also needs to be verified.

Environment setup

We implemented an emulation device control tool to facilitate the MSLFuzzer to control the emulated SOHO routers to achieve all of the device states or to restart the device after the emulation crash. The tool provides a RESTful API, and all emulated devices can be started or stopped through an API call. When a crash of the target service is detected, the target device needs to be restarted in order for the next round of fuzzing to proceed. The fuzzer wrapper automatically reboots the target device and restores the target service by calling the API. Since the fuzzing process is random, we repeat the experiment 5 times and count the average data. Like existing evaluation methods, we set the maximum testing time for each fuzzing experiment to be 24 h. We deployed MSLFuzzer on an Ubuntu 20.04 desktop PC with Intel Core i7 8-core X 3.70 GHz CPU and 16 GB RAM.

Benchmark tools

To verify the performance of MSLFuzzer in finding crashes, we used four different fuzzing schemes as benchmarks.

- *SNIPUZZ* The initial SNIPUZZ code published by its authors is written in C# for Windows machines. As a demo version, the code lacks a user manual, has complex dependencies, and is not packaged as an independent tool. Its authors try to reimplement SNIPUZZ in python to enhance its generality. Unfortunately, the python version of SNIPUZZ was not released when we started our experiment. Therefore, we reimplemented SNIPUZZ based on our understanding of the method presented in the paper (Feng et al. 2021) and its C# implementation code. Considering that SNIPUZZ needs to infer message content based on feedback response messages, to make the benchmark as fair as possible, we use the same raw message as seed for SNIPUZZ and MSLFuzzer.
- *Boofuzz* Unlike other black-box fuzzers, Boofuzz requires human knowledge to guide. Specifically, Boofuzz needs to manually code a corresponding script for each raw message, which defines the format of the raw message, the fields that need to be mutated, and the mutation strategy. We refer to the evaluation methods of Boofuzz from other works (Feng et al. 2021), exploit this property of Boofuzz, and manually define more fuzzing strategies to enrich the benchmark evaluation.
  - *Boofuzz-Default* Each message in the input is set to a full string. Boofuzz will mutate that message as a string.
  - *Boofuzz-Byte* Each byte of the message in the input will be used for mutation individually.
- *MSLFuzzer-NoAnalysis* MSLFuzzer further analyzes the message content to improve the fuzzing efficiency

and the ability to find crashes. To verify whether fine-grained message analysis is beneficial for fuzzing, we removed the message content analysis code used in MSLFuzzer and implemented MSLFuzzer-NoAnalysis. MSLFuzzer-NoAnalysis does not further analyze the constructed message segment list and directly completes the subsequent mutation operations.

All benchmarking tools and MSLFuzzer are tested on the same vulnerability dataset to make the benchmark as fair as possible. These inputs may have different formats (e.g., Boofuzz requires manual input settings, and SNIPUZZ requires raw messages), but the content is the same.

Features of MSLFuzzer

This section evaluates MSLFuzzer and existing black-box fuzzer tools for SOHO routers. We compared MSLFuzzer with state-of-the-art network fuzzers Boofuzz (Pereyda 2022), SNIPUZZ (Feng et al. 2021) and UCRF (Qin et al. 2023) regarding the firmware dependent, analysis object, analysis method, seed generation, and field completion. Since UCRF is not open source, we mainly compare its functions in this section.

As shown in Table 4, MSLFuzzer, SNIPUZZ and Boofuzz do not depend on the firmware image, and the analysis object is the message communicated with the target device. UCRF relies on firmware images, which require static analysis, and new seeds can be generated. Therefore, the use scenarios of MSLFuzzer and UCRF are different, and when the router device firmware image is unavailable, the application scenarios of MSLFuzzer are more comprehensive. Unlike SNIPUZZ, MSLFuzzer adds MSL inference based on response feedback technology and can complete the default message fields. Therefore, the analysis of MSLFuzzer for response messages is more comprehensive.

Efficiency

MSLFuzzer is evaluated on the dataset containing 35 real-world vulnerabilities in 20 emulated SOHO routers. Our process of judging different fuzzers triggering vulnerabilities is as follows. First, before the fuzzing experiment corresponding to each vulnerability, we provided a unified seed for all tested fuzzers. The network interface contained in the seed corresponds to a specific service component on the device under test, and this component has a known vulnerability that we have verified. Second, we captured all traffic packets during fuzzing. After the fuzzing is over, we code Wireshark filter rules, manually locate the specific packet that triggers the exception, and analyze whether the specific mutation position of the mutated packet is consistent with the trigger field of the target known vulnerability in the component. Finally, we replayed the message, observed the context information given by the emulator when the device reported an error, and finally judged whether the target vulnerability was triggered.

In evaluating the performance of each fuzzer, we prioritize using the number of vulnerabilities found by each fuzzer as the primary evaluation metric. When each fuzzer can or cannot find a specific vulnerability, the response category triggered by them will be used as an additional evaluation metric. We choose such an evaluation metric mainly following the suggestion of Magma (Hazimeh et al. 2020), a state-of-the-art fuzzer evaluation work. The final metric for evaluating two fuzzers is to compare the number of bugs found by each fuzzer. If fuzzer A finds more bugs than fuzzer B, then A is better than B. However, considering that some fuzzers can or cannot find the same vulnerability, we use the number of response categories as an additional metric to supplement the evaluation. The results of the experiment are shown in Table 3. MSLFuzzer found 31 out of 35 vulnerabilities, which is higher than the other four involved benchmark tools. We conducted a detailed analysis of the fuzzing process for these vulnerabilities.

We first analyzed the fuzzing process for four vulnerabilities that MSLFuzzer did not successfully trigger. We found that CVE-2019-10891, CVE-2019-20215 and CVE-2020-15893 were not successfully triggered because the number of responses triggered by these two vulnerabilities was small, and the message format did not have a specific nested structure. Therefore, MSLFuzzer fails to construct detailed message segments, and the entire fuzzing process is close to random mutation. CVE-2019-6258 was not successfully triggered because the component affected by the vulnerability differs from the component receiving the mutated message. MSLFuzzer observes the state of the component that receives mutated messages. However, the component did not behave abnormally, so the vulnerability was not detected by MSLFuzzer. These three vulnerabilities were also not discovered by SNIPUZZ.

Then, we analyzed the vulnerabilities that both MSLFuzzer and SNIPUZZ could trigger successfully. SNIPUZZ found more responses on CVE-2017-13772 than MSLFuzzer did, and MSLFuzzer found more responses on CVE-2019-17510 than SNIPUZZ did. Of the remaining vulnerabilities, MSLFuzzer found the same number of responses as SNIPUZZ. This shows that MSLFuzzer and SNIPUZZ have relatively comparable performances on these vulnerabilities.

Next, we focus on the vulnerabilities that MSLFuzzer triggers successfully, but SNIPUZZ does not trigger successfully. An essential reason that MSLFuzzer

Table 3 Experiment Results. MSLFuzzer discovers the greatest number of vulnerabilities

#	ID	Device	Protocol	MSLFuzzer		SNIPUZZ		Boofuzz-Default		Boofuzz-Byte		MSLFuzzer NoAnalysis	
				C	N	C	N	C	N	C	N	C	N
1	CVE-2017-13772	TP-Link WR940N	HTTP	+	15	+	**25**	–	1	+	1	–	2
2	CVE-2017-17215	Huawei HG532	SOAP	+	**2**	–	**2**	–	1	–	1	–	1
3	CVE-2018-14558	Tenda AC7	HTTP	+	2	+	2	–	2	–	2	+	**7**
4	CVE-2018-16334	Tenda AC9	HTTP	+	2	+	2	–	1	–	1	–	**5**
5	CVE-2018-18728	Tenda AC9	HTTP	+	5	–	5	–	2	–	2	–	**10**
6	CVE-2018-19987	D-Link DIR-822	HNAP	+	**3**	–	1	–	2	–	1	–	2
7	CVE-2018-19989	D-Link DIR-822	HNAP	+	**3**	–	1	–	2	–	1	–	1
8	CVE-2019-10891	D-Link DIR-806	HNAP	–	**4**	–	1	–	1	–	1	–	1
9	CVE-2019-17510	D-Link DIR-846	HNAP	+	**6**	+	3	–	4	–	3	+	4
10	CVE-2019-17621	D-Link DIR-859	UPNP	+	**3**	–	2	–	2	–	2	–	1
11	CVE-2019-20215	D-Link DIR-859	UPNP	–	**2**	–	**2**	–	1	–	1	–	1
12	CVE-2019-20760	NETGEAR R9000	HTTP	+	**4**	–	**4**	–	2	–	3	–	3
13	CVE-2019-6258	D-Link DIR822	SOAP	–	**2**	–	**2**	–	**2**	–	**2**	–	**2**
14	CVE-2019-6989	TP-Link WR940N	HTTP	+	**4**	–	2	–	1	–	1	–	1
15	CVE-2019-7297	D-Link DIR-823G	SOAP	+	6	–	**39**	–	23	–	29	–	1
16	CVE-2020-10215	D-Link DIR-825	HTTP	+	37	–	**209**	–	3	–	79	–	3
17	CVE-2020-10216	D-Link DIR-825	HTTP	+	2	–	1	–	1	–	**5**	–	3
18	CVE-2020-13392	Tenda AC6	HTTP	+	8	–	2	–	1	–	1	–	**9**
19	CVE-2020-13394	Tenda AC18	HTTP	+	**2**	+	**2**	–	1	–	1	+	**2**
20	CVE-2020-13782	D-Link DIR-865L	HTTP	+	**3**	–	2	–	1	–	1	–	2
21	CVE-2020-15893	D-Link DIR-816L	UPNP	–	**2**	–	**2**	–	**2**	–	1	–	**2**
22	CVE-2020-25367	D-Link DIR-823G	SOAP	+	**3**	+	**3**	–	1	+	1	+	1
23	CVE-2020-27600	D-Link DIR-846	HNAP	+	3	–	3	–	**4**	–	1	–	1
24	CVE-2020-8423	TP-Link WR841N	HTTP	+	5	–	**29**	–	12	–	12	–	17
25	CVE-2021-43474	D-Link DIR-823G	SOAP	+	**3**	–	2	–	2	–	2	–	2
26	CVE-2021-46314	D-Link DIR-846	HNAP	+	**7**	–	**7**	–	3	–	3	–	3
27	CVE-2022-24355	TP-Link WR940N	HTTP	+	2	+	2	–	**5**	–	1	+	2
28	CVE-2022-25079	TOTOLINK A810R	HTTP	+	**5**	–	4	–	2	–	2	–	3
29	CVE-2022-25439	Tenda AC9	HTTP	+	**3**	–	**3**	–	2	–	2	–	2
30	CVE-2022-29638	TOTOLINK A3100R	HTTP	+	**3**	+	**3**	–	2	–	2	–	2
31	CVE-2022-29643	TOTOLINK A3100R	HTTP	+	**5**	+	**5**	–	2	–	2	–	2
32	CVE-2022-35619	D-Link DIR-818L	SOAP	+	2	–	**3**	–	1	–	1	–	2
33	CVE-2022-35620	D-Link DIR-818L	UPNP	+	**2**	–	1	–	1	–	1	–	1
34	Disclosed but Unassigned-1	NETGEAR WNDR3700	HTTP	+	**4**	–	1	–	**4**	+	1	–	5
35	Disclosed but Unassigned-2	NETGEAR WNDR3700	HTTP	+	4	–	2	+	1	+	**12**	–	9

The bold numbers indicate that the corresponding fuzzer has the best fuzzing performance for that vulnerability corresponding to that row among all fuzzers, i.e., it triggered the most responses

C Crashed, + implies at least one crash, − means no crash, *N* Number of response categories

found these vulnerabilities while SNIPUZZ did not is that the responses of many SOHO routers could be more detailed, and many devices respond with a suitable error response type. Therefore, when the number of responses is too small, the primary segment snippet constructed by SNIPUZZ cannot sufficiently describe the original structure of the message, which makes the fuzzing process close to random mutation and cannot successfully find a vulnerability. MSLFuzzer performs nested structure analysis after the initial message segment list is constructed. Therefore, when the device adopts a standard protocol format, its message structure can be effectively disassembled. This allows MSL to better describe the message structure, and MSLFuzzer to obtain more diverse response types. As a result, MSLFuzzer can discover CVE-2017-17215,

CVE-2019-6989, CVE-2020-10216, CVE-2020-13392, Unassigned-1, and Unassigned-2. Although MSLFuzzer cannot handle some undefined message structures, such as CVE-2019-20215 and CVE-2020-15893, it can handle most cases correctly, which mitigates the problems encountered by SNIPUZZ. MSLFuzzer can also discover CVE-2022-25079 by performing random value offset inference. The response message corresponding to this vulnerability carries random value information. MSLFuzzer successfully locates the position of the random value, which makes the message structure more accurately described.

Another important reason we found that MSLFuzzer can find more vulnerabilities than the benchmarked fuzz tools is because MSLFuzzer performs content analysis on the mutated MSL. CVE-2019-20760 and CVE-2020-8423 can be found because MSLFuzzer performs the encoding analysis. The fields that trigger these two vulnerabilities must be encoded to reach the flawed code location. CVE-2018-19987, CVE-2018-19989, CVE-2019-7297, CVE-2020-13392, CVE-2020-13782, CVE-2020-27600, and CVE-2021-46314 were discovered because MSLFuzzer performed invisible segment analysis. There are many parameters in the raw seed message corresponding to these vulnerabilities, and the value of the field that triggers the vulnerability is empty. MSLFuzzer can locate the location of these invisible segments, so in the process of mutation, these invisible segments can be successfully selected for mutation. CVE-2019-7297 and CVE-2020-10215 produced more responses than other vulnerabilities. This is because the response message carries the field value in the mutated message. During the mutation process of SNIPUZZ, the probability of each snippet being selected for mutation is the same, so the test case generated by SNIPUZZ destroys the message structure required for device sanitization. MSLFuzzer conducted a stability analysis, and the possibility of each message segment being selected is different. The constructed mutated message structure is more stable, and the mutated message is easier to reach the defect code position. Therefore, these two vulnerabilities were successfully discovered by MSLFuzzer.

In addition, we observed that SNIPUZZ triggered significantly higher response categories on CVE-2019-7297, CVE-2020-10215, and CVE-2020-8423. We analyzed the complete fuzzing process and response categories for these vulnerabilities and found that this is mainly due to two reasons. First, these devices will feed back part of the field information in the test case to the user as part of the response. Second, we adopt the idea of Fail-fast in the experiment. The fuzzing process is stopped after the vulnerability is triggered, so the response category stops growing after the vulnerability is triggered. Among these vulnerabilities, since MSLFuzzer successfully triggered the vulnerability, but SNIPUZZ did not, although SNIPUZZ triggered more types of responses, MSLFuzzer performed better.

Boofuzz-Byte found four vulnerabilities, and Boofuzz-Default found only one. Boofuzz directly replaces the specified position in the message with a pre-set string or byte and does not split and analyze the content of the message, so it finds far fewer than MSLFuzzer. MSLFuzzer-NoAnalysis is a fuzzing tool that does not use the message segment content analysis method of MSLFuzzer. It discovered 6 vulnerabilities, outperforming Boofuzz but slightly inferior to SNIPUZZ. The reason is that MSLFuzzer-NoAnalysis does not analyze the message content in various ways, but directly mutates it after constructing the initial MSL. Therefore, most of the test cases generated by MSLFuzzer-NoAnalysis usually fail to pass the sanitization phase. This method may be more effective in finding vulnerabilities on devices that require highly structured input.

The experimental results show that MSLFuzzer has the most vital ability to find vulnerabilities among all benchmark tools. Its ability to trigger different numbers of responses is close to the state-of-the-art black-box feedback fuzzer SNIPUZZ. This shows that MSLFuzzer can effectively discover potential vulnerabilities in SOHO routers.

Response category trigger

In this section, we evaluate the ability of MSLFuzzer to trigger response categories. Response categories are used as additional metrics for evaluating fuzzers when multiple fuzzers can or cannot find vulnerabilities. We used two metrics to measure the ability of each fuzzer to trigger more response categories. The first metric is the cumulative number of valid response categories triggered by each fuzzer across all experiments. This metric measures the combined performance of each fuzzer across all experiments. It should be noted that some vulnerabilities may carry part of the test case content in their responses, resulting in an abnormal number of responses. We removed these vulnerabilities when counting this metric to avoid interference with the experimental results. The second metric is the number of experimental groups in which each fuzzer triggered the most responses among 35 groups of vulnerability experiments. The second metric measures the average performance of each fuzzer in each set of experiments.

Figure 3 displays the experimental results of the first metric as a histogram. MSLFuzzer found the most significant number of response categories among all fuzzers and performed the best. The second metric displays the experimental results as a line graph. Among the 35 groups of vulnerability experiments, MSLFuzzer

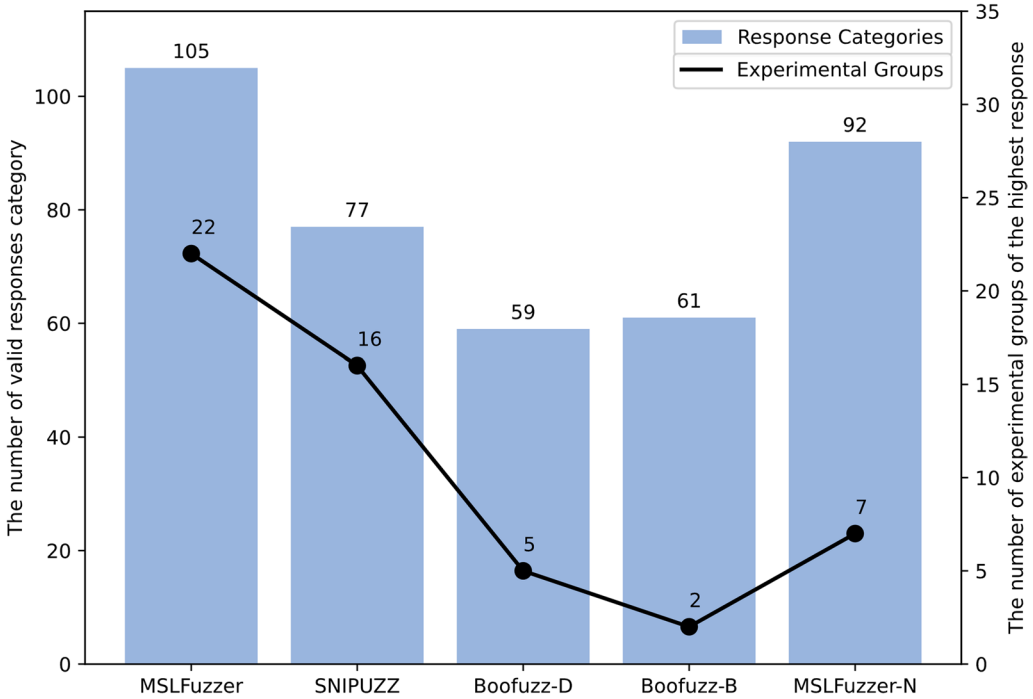

Fig. 3 Results for each fuzzer triggering response categories. Boofuzz-D is an abbreviation for Boofuzz-Default, Boofuzz-B is an abbreviation for Boofuzz-Byte, and MSLFuzzer-N is an abbreviation for MSLFuzzer-NoAnalysis

achieved the best performance in 22 groups of experiments, outperforming all other fuzzers. This shows that MSLFuzzer can effectively trigger more response categories and perform more stably.

Invisible segment identification

In this section, we evaluate the ability of MSLFuzzer to discover invisible segments. Firstly, we removed some valid fields from the seed messages of each vulnerability. Then, the messages with invisible segments were used as new seeds and input to MSLFuzzer. Finally, we calculated the proportion of successfully identified invisible segments among all seeds carrying invisible segments when the parameter n in the n-gram of Algorithm 2 ranged from 3 to 10.

As shown in Fig. 4, when the parameter n equals 4, Algorithm 2 identifies the maximum number of invisible segments. As n increases, the proportion of identified invisible segments gradually decreases. When n equals 7 and 10, the proportion is at its lowest, 62.2%. Therefore, MSLFuzzer uses a value of n equal to 4. At this value, MSLFuzzer identifies 94.6% of invisible segments. This demonstrates that MSLFuzzer can effectively infer potential invisible segments in message packets.

Unknown vulnerability identification

MSLFuzzer has demonstrated superior performance in previous experiments, but we still want to verify further its ability to discover unknown vulnerabilities.

We first collected firmware that can be emulated in building the known vulnerability dataset. Because a fuzzer requires primitive seeds, we use the control interface provided by a successfully emulated firmware and artificially access these interfaces to generate as many raw messages as possible. At the same time, we captured the network communication traffic in the process and screened the raw messages containing the data submitted by the user as seeds. MSLFuzzer uses these seeds as input.

By evaluating these emulated devices from five vendors, MSLFuzzer found 15 crashes. We manually verified crashes one by one to determine if they were previously unknown vulnerabilities. In this process, we first locate the specific message that triggers the exception. We replayed the message that triggered the exception and observed the context information given by the emulator when the emulated device reported an error. Secondly, we searched the NVD and CNVD databases using device manufacturers, models, and interfaces as keywords. For all the known vulnerabilities retrieved,

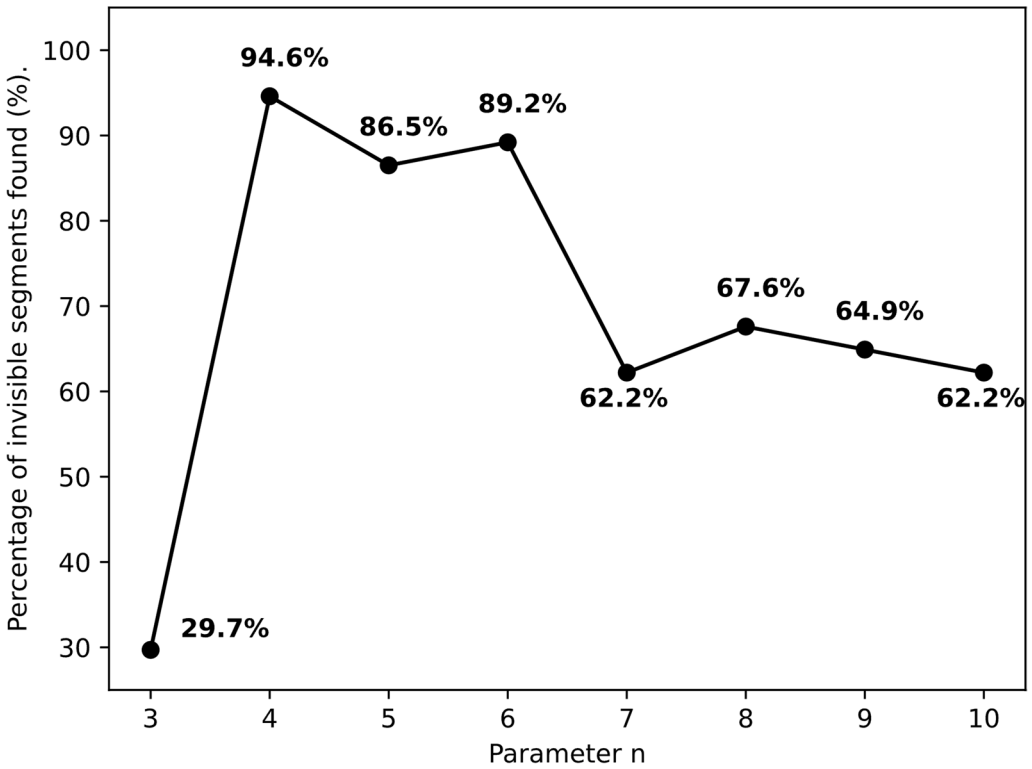

Fig. 4 Invisible segment identification variation curve diagram

Table 4 Comparison of state-of-the-art fuzzers

Fuzzer	Firmware dependent	Analysis object	Analysis method	Field completion
MSLFuzzer	No	Message	Response Feedback + MSL Inference	Yes
SNIPUZZ	No	Message	Response Feedback	No
Boofuzz	No	Message	Manual	No
UCRF	Yes	Firmware	Static Analysis	No

we analyzed their vulnerability descriptions, interface names, parameter information, PoC, and other information individually. We compared them with the triggered crash to complete the judgment. In the end, MSLFuzzer found 10 zero-day vulnerabilities. Four were confirmed by the CVE, and one was confirmed by the CNVD, as shown in Table 5. Types of these vulnerabilities include stack overflow and command injection. We have reported all these vulnerabilities to CNCERT/CC (National computer network emergency response technical team 2021) in pursuit of helping vendors fix them. We also tried using SNIPUZZ to find these vulnerabilities and extended the fuzzing time to 48 h. Unfortunately, SNIPUZZ was not able to find these vulnerabilities.

While validating these crashes as unknown vulnerabilities, we discovered some interesting things. Before we applied for the CVE number, there were more than 50 historical vulnerabilities in the Tenda ac9 firmware included in the NVD. Such a large number of vulnerabilities in one firmware shows that researchers have thoroughly analyzed the security of this firmware, and it is difficult to find new vulnerabilities in it. Nevertheless, even in this case, MSLFuzzer could still find some missed vulnerabilities, which shows that MSLFuzzer can comprehensively discover vulnerabilities in SOHO routers. In addition, NVD does not contain any vulnerabilities in the NETGEAR WNAP 320 firmware. MSLFuzzer successfully found the first vulnerability of this device. The experimental results show that MSLFuzzer can be applied to SOHO routers from different manufacturers and can effectively discover unknown vulnerabilities.

Discussion and future work

The evaluation results show that our framework can effectively discover memory corruption and command injection vulnerabilities in SOHO routers, but there are still some directions for future improvement. In this section, we discuss the limitations that exist in the

Table 5 Summary of Discovered Unknown Vulnerabilities

#	Vulnerability	Device	Vulnerability type	Severity
1	CVE-2022-36568	Tenda ac9	Stack overflow	High
2	CVE-2022-36569	Tenda ac9	Stack overflow	High
3	CVE-2022-36570	Tenda ac9	Stack OVERFLOW	High
4	CVE-2022-36571	Tenda ac9	Stack Overflow	High
5	CVE-2022-46641	D-Link DIR-846	Command injection	High
6	CVE-2022-46642	D-Link DIR-846	Command injection	High
7	CNVD-2022-62390	D-Link DIR-823G	Command injection	Medium
8	Reported but Not Disclosed	TOTOLINK A3100R	Stack overflow	Medium
9	Reported but Not Disclosed	NETGEAR WNAP320	Command injection	High
10	Reported but Not Disclosed	NETGEAR WNAP320	Command injection	High

current design and explore how these limitations can be addressed in the future.

Initial seed acquisition

The initial input to MSLFuzzer is a raw message. The raw message is used as an initial seed for mutation, and its quality significantly impacts the final effect of fuzzing. The initial seed construction method is to manually complete the communication process with SOHO routers and monitor network communication to capture all communication packets. Corresponding filtering rules are set to extract high-quality raw messages in packets. In the above process, an automatic generation of valid raw messages is challenging. A promising solution is to automatically analyze the user input interface of the device console program and generate corresponding request traffic. For example, for a SOHO router that supports control through a web page, a compelling seed message can be generated by analyzing the dependencies between parameters and input rules from the front-end page. Automated initial seed generation will be our next step.

More semantic categories

In the message segment content analysis stage, MSLFuzzer adopts a rule-based heuristic method to analyze four semantic categories. These four semantic categories have covered some common semantic situations in device messages. However, from the perspective of completeness, more categories and fine-grained categories can increase the understanding of the message content, which in turn can impose more substantial constraints on mutated messages and improve fuzzing efficiency. This is a process of continuous improvement. Given the recent excellent performance of deep learning techniques in the field of natural language processing, in future work, we consider introducing deep learning techniques to enhance the understanding of message content further.

Inter-message dependencies

In the current implementation of MSLFuzzer, we only focus on messages that can accomplish the corresponding function with a single request. However, some functions still require multiple request messages to work together. A possible solution is to perform correlation analysis on multiple messages in a message sequence, tracking and modeling the entire conversation flow during the initial analysis of the messages. At the same time, the corresponding protocol state machine is constructed, and the customized fuzzing is carried out based on a protocol state machine. This approach may have advantages in finding vulnerabilities triggered by multiple message combinations and vulnerabilities due to violations in the implementation of the protocol specification.

Details of the response message

The details of the response message affect the quality of the initial construction of the message segment list. The more details the response message contains, the more accurately the segment list segments the raw message. MSLFuzzer further disassembles the nested message segment in analyzing the content of the message segment, which can make up for the lack of detail in some response messages. When applied to a binary protocol, since the standard string nesting structure may not exist in the binary protocol, the division of the binary protocol structure largely depends on the detail level of the response provided by the target device. However, the response details provided by the SOHO routers are too vague (e.g., using a unified message to report all errors), and the nested format cannot be appropriately recognized. In that case, the number of message segments may be low, which makes fuzzing inefficient. Fortunately, in practice, this problem can be mitigated in two ways. The first way, for some SOHO routers that support debug mode, we can get advanced error descriptions in debug

mode, which will significantly improve the process of MSLFuzzer building the message segment list. In a second way, researchers can code plug-ins that parse the protocol and build a message segment list when prior knowledge of the target message format or binary protocol specification is obtained. Since the message segment list can describe the message content in a normalized manner, the plug-in code for message parsing can be easily integrated with MSLFuzzer, and the subsequent analysis process of MSLFuzzer can still be used commonly. Both of these can alleviate the complete failure of fuzzing when encountering insufficiently detailed device responses.

Validation of effective fields

Judging how many effective fields in the message segment completed by MSLFuzzer are consistent with the fields in the firmware processing logic can guide the further improvement of MSLFuzzer. However, due to the customization of the control protocol by different firmware models, the valid fields cannot be directly confirmed by observation. For example, some byte segments may be the segmentation identifier of the protocol and have no specific meaning. Some byte segments are data submitted by the user, but the components that process the interface in the firmware do not process these data. Some byte segments The segment is a format customized by the firmware, and it is impossible to directly judge which bytes in it are specific valid fields through observation. Manual reverse analysis assistance is possible, but it consumes many human resources and time. A possible solution is to comb all the parameters received and processed by each interface of the firmware based on reverse analysis of the firmware, combined with data flow analysis, and associate them with the fuzzing process. This will be one of the directions to improve MSLFuzzer.

Encrypted traffic

While preparing the raw message as a fuzzing seed, we noticed that some devices use encryption to protect communication. Since the encryption algorithm destroys the original format of the message, modifications made to the raw message are often challenging to decrypt to legitimate messages. Moreover, since the feedback response message is also encrypted, it is impossible to directly judge the category based on the content of the message. Relieving the constraints of encryption from outside the target device based solely on network traffic is challenging, and solutions are often difficult to generalize. Nevertheless, when the encryption and decryption algorithms are known, it is feasible to apply MSLFuzzer to this scenario. It can be solved by integrating the encryption and decryption algorithm with MSLFuzzer. The mutated message is encrypted each time before it is sent to the target device, and the response message is decrypted after it is received. The rest of the message segment construction and fuzzing processes can be directly reused with MSLFuzzer. In this way, the fuzzing process of the target device can be completed in this scenario.

Conclusion

This paper proposes a black-box fuzzing framework, MSLFuzzer, to discover vulnerabilities in SOHO routers. Unlike other black-box network fuzzers, MSLFuzzer uses device feedback response messages to conduct preliminary structural division of raw messages and performs semantic analysis, invisible segment analysis, and stability analysis on each message segment. This mutation strategy based on the message segment list has better constraints and orientation, narrows the search space, and can ensure that well-structured test cases are generated to test more functions of the tested devices other than the sanitizer. We construct a dataset of 35 real-world vulnerabilities and 20 consumer-grade SOHO routers that can be emulated and evaluate MSLFuzzer on this dataset. MSLFuzzer found more known vulnerabilities than other state-of-the-art benchmark tools and successfully discovered 10 zero-day vulnerabilities with four CVEs and one CNVD.

Acknowledgements
Not applicable.

Author contributions
YXC, WQF, and WH designed this research. YXC and JYY built this framework and performed experiments. YXC, WL wrote this paper. YXC, WQF, and WL reviewed and edited the manuscript. All authors read and approved the manuscript.

Funding
This work was supported by the major project of Science and Technology Innovation 2030, "The next generation of Artificial Intelligence" under Grant Number 2021ZD0111400, the Open project of the State Key Laboratory of Computer Architecture, Neural Network Enhanced Symbolic Execution Algorithm Research under Grant Number CARCH201910, and the Fundamental Research Funds for the Central Universities under Grant Number 3132018XNG1814 and 3132018XNG1815.

Availability of data and materials
The data used to support the findings of this study are available from the corresponding author upon request. The disclosed security vulnerabilities used and found in this paper can be accessed in the CVE (https://cve.mitre.org/) and CNVD (http://www.cnvd.org.cn).

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 1 May 2023 Accepted: 29 August 2023
Published online: 07 November 2023

References

Amini P, Portnoy A, Sears R (2019) Sulley: a pure-python fully automated and unattended fuzzing framework. Available https://github.com/OpenRCE/sulley. Accessed 9 Nov 2022

Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T, Bochum R (2019) REDQUEEN: fuzzing with input-to-state correspondence. In: 2019 network and distributed system security symposium

Chen D, Woo M, Brumley D (2016) Towards automated dynamic analysis for linux-based embedded firmware. In: Network and distributed system security symposium

Chen J, Diao W, Zhao Q et al. (2018) IoTFuzzer: discovering memory corruptions in IoT through app-based fuzzing. In: Network and distributed system security symposium

Cheng Y et al (2022) PDFuzzerGen: policy-driven black-box fuzzer generation for smart devices. Secur Commun Netw. https://doi.org/10.1155/2022/9788219

Dinh S et al. (2021) Favocado: fuzzing the binding code of JavaScript engines using semantically correct test cases. In: Network and distributed system security symposium

Eceiza M, Flores JL, Iturbe M (2021) Fuzzing the Internet of Things: a review on the techniques and challenges for efficient vulnerability discovery in embedded systems. IEEE Internet Things J 8(13):10390–10411

Feng X et al. (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, pp. 337–350.

Gan S et al. (2018) CollAFL: path sensitive fuzzing. In: 2018 IEEE symposium on security and privacy, pp. 679–696

Han H, Oh D, Cha S (2019) CodeAlchemist: semantics-aware code generation to find vulnerabilities in JavaScript engines. In: Network and distributed system security symposium

Hazimeh A, Herrera A, Payer M (2020) Magma: a ground-truth fuzzing benchmark. Proc ACM Meas Anal Comput Syst 4(3):1–29

Huang H, Yao P, Wu R, Shi Q, Zhang C (2020) Pangolin: incremental hybrid fuzzing with polyhedral path abstraction. In: 2020 IEEE symposium on security and privacy, pp. 1613–1627

Jones AK, Sielken RS (2000) Computer system intrusion detection: a survey. Comput Sci Tech Rep, pp. 1–25.

Khandait P, Hubballi N, Mazumdar B (2021) IoTHunter: IoT network traffic classification using device specific keywords. IET Netw 10(2):59–75

Kim M, Kim D, Kim E, Kim S, Jang Y, Kim Y (2020) Firmae: towards large-scale emulation of iot firmware for dynamic analysis. In: Annual computer security applications conference, pp. 733–745.

Lee S, Han H, Cha S, Son S (2020) Montage: a neural network language {Model-Guided}{JavaScript} engine fuzzer. In: 29th USENIX security symposium, pp. 2613–2630.

Micro T (2020) Smart yet flawed: IoT device vulnerabilities explained. Security News, Tech. Rep.

Muench M, Nisi D, Francillon A, Balzarotti D (2018) Avatar 2: a multi-target orchestration platform. In: Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.), vol. 18, pp. 1–11.

National computer network emergency response technical team/coordination center of China, 2021. Available: https://www.cert.org.cn/publish/english/index.html. Accessed 9 Nov 2022

Pereyda J (2022) boofuzz: network protocol fuzzing for humans. Available https://boofuzz.readthedocs.io/en/latest/. Accessed 9 Nov 2022

Qin C, Peng J, Liu P et al (2023) UCRF: static analyzing firmware to generate under-constrained seed for fuzzing SOHO router. Comput Secur 128:103157

Redini N et al. (2021) Diane: identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In: 2021 IEEE symposium on security and privacy, pp. 484–500.

Shu Z, Yan G (2022) IoTInfer: automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference. IEEE Internet Things J 9(22):22737–22751

Unit 42, "2020 Unit 42 IoT Threat Report", Palo Alto Networks, Inc., 2020. Available https://unit42.paloaltonetworks.com/iot-threat-report-2020/. Accessed 9 Nov 2022

Wang D, Zhang X, Chen T, Li J (2019) Discovering vulnerabilities in COTS IoT devices through blackbox fuzzing web management interface. Secur Commun Netw. https://doi.org/10.1155/2019/5076324

Wireshark. Available https://www.wireshark.org/. Accessed 9 Nov 2022

Yamaji M (2022) Forecast: IoT semiconductors, worldwide, 2Q22 update. Gartner, Inc.. Available https://www.gartner.com/en/documents/4016543/. Accessed 9 Nov 2022

Yin Q, Zhou X, Zhang H (2021) FirmHunter: state-aware and introspection-driven grey-box fuzzing towards IoT firmware. Appl Sci 11(19):9094

Yu Z, Wang H, Wang D, Li Z, Song H (2022) CGFuzzer: a fuzzing approach based on coverage-guided generative adversarial networks for industrial IoT protocols. IEEE Internet Things J 9(21):21607–21619

Zaddach J, Bruno L, Francillon A, Balzarotti D (2014) AVATAR: a framework to support dynamic security analysis of embedded systems' firmwares. In: 2014 network and distributed system security symposium

Zhang Y et al (2021) ESRFuzzer: an enhanced fuzzing framework for physical SOHO router devices to discover multi-Type vulnerabilities. Cybersecurity 4(1):1–22

Zhang Y et al. (2019) SRFuzzer: an automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities. In: Proceedings of the 35th annual computer security applications conference, pp. 544–556

Zheng Y, Davanian A, Yin H, Song C, Zhu H, Sun L (2019) {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In: 28th USENIX security symposium, pp. 1099–1114

Zhu X, Wen S, Camtepe S, Xiang Y (2022) Fuzzing: a survey for roadmap. ACM Comput Surv 54(11s):1–36

Publisher's Note