LABRADOR: Response Guided Directed Fuzzing for Black-box IoT Devices

Hangtian Liu[1,2,4], Shuitao Gan[2,4✉], Chao Zhang[2,3✉], Zicong Gao[1,2,4],
Hongqi Zhang[1,5], Xiangzhi Wang[6], Guangming Gao[4]

[1]*Information Engineering University,*
[2]*Tsinghua University,* [3]*Zhongguancun Laboratory,*
[4]*Laboratory for Advanced Computing and Intelligence Engineering,*
[5]*Henan Key Laboratory of Information Security,*
[6]*University of Electronic Science and Technology of China*

***Abstract*—Fuzzing is a popular solution to finding vulnerabilities in software including IoT firmware. However, due to the challenges of emulating or rehosting firmware, some IoT devices (e.g., enterprise-level devices) can only be fuzzed in a black-box manner, which makes fuzzers blind and inefficient due to missing feedbacks (e.g., code coverage or distance). In this paper, we present a novel response guided directed fuzzing solution LABRADOR, able to test black-box IoT devices efficiently. Specifically, we leverage the network response to infer the execution trace of firmware and deduce the code coverage of testing. Second, we leverage the test case (i.e., request) and its response to estimate the distance to the target sensitive code (i.e., sink). Lastly, we further leverage the distance to guide test case mutation, which efficiently drives directed fuzzing toward candidate vulnerable code. We have implemented a prototype of LABRADOR and evaluated it on 14 different enterprise-level IoT devices. Results showed that LABRADOR significantly outperforms state-of-the-art (SOTA) solutions. It finds 44X more vulnerabilities than SNIPUZZ, BOOFUZZ and FIRM-AFL and 8.57X more vulnerabilities than SaTC. In total, it discovered 79 unknown vulnerabilities, of which 61 were assigned with CVEs.**

1. Introduction

In the era of the "Internet of Everything", billions of IoT devices seamlessly connect humans, machines, and objects through network [37], providing various online services for daily convenience and benefits [17]. The exponentially increasing use of IoT devices also comes with growing threats. As reported in [1, 21], the number of vulnerabilities and attacks against IoT devices has grown significantly in recent years. To mitigate such threats, it is crucial to proactively discover vulnerabilities and fix them in advance [5].

The first type of vulnerability discovery solution for IoT firmware is static analysis, e.g., SaTC [11], KARONTE [35]. Such solutions generally focus on taint-style vulnerabilities, which first recognize sensitive code (i.e., sinks) via static analysis and then infer potentially vulnerable paths reachable to sinks via taint analysis or symbolic execution. Such solutions often have high false positives and low efficiency.

Another type of popular vulnerability discovery solution for IoT firmware is fuzzing. Since IoT devices often require peripherals or configurations to run, many fuzzing solutions (e.g., Firm-AFL [52], Firmadyne [8]) focus on emulating or rehosting IoT firmware, to improve efficiency and scalability of fuzzing and capability of bug detection. However, for some IoT devices, especially those enterprise-level ones, it is incredibly challenging to emulate or rehost their firmware [44]. Rehosting firmware, even for Linux/BSD-based images, still faces critical problems like insufficient fidelity and low success rate [42]. As a result, only black-box fuzzing applies to such devices.

But, black-box fuzzing solutions are generally less effective than gray-box fuzzing. Specifically, modern gray-box fuzzers often rely on code instrumentation [7, 22, 34] or hardware-based tracing [2, 27] to collect feedback (e.g., code coverage, distance) during testing, and use the feedback to guide the fuzzing process, including seed preservation, seed selection, and seed mutation, etc., to either explore more code or trigger target code with higher efficiency. Such feedback would make the fuzzer efficiently evolve towards its goal and avoid blindness.

Therefore, black-box fuzzing for IoT devices is demanded but challenging. SNIPUZZ [18] is the SOTA black-box fuzzing solution for IoT devices. Specifically, it leverages the response to infer message snippets for test case mutation. However, no solutions have been yet proposed to collect the traditional feedback for black-box fuzzing, including (1) code coverage feedback for exploring more code and (2) distance feedback for exploring target-sensitive code (sinks).

In this paper, we present a novel solution LABRADOR to collect feedback for black-box IoT devices. Service attributes are crucial for IoT devices, with web interfaces enabling various applications for most of them [3]. Exploiting vulnerabilities in these interfaces can raise unau-

✉*Corresponding authors:ganshuitao@gmail.com, chaoz@tsinghua.edu.cn.*

thorized access, data theft, or RCE attacks. Web interface vulnerabilities in IoT devices often have high or critical severity levels in the CVE database, emphasizing the importance of vulnerability discovery and fuzzing in this area of research. For web interface, it's inevitable to introduce many meaningful strings to the devices spanning the request, response, and back-end binaries. Moreover, these strings are often partially shared among the response, request, and binaries. *As a result, we could infer which code blocks have been executed by examining strings in the response and estimate the distance towards target code blocks.* Based on this observation, LABRADOR could get the feedback of code coverage and distance, therefore performing a response guided directed fuzzing for black-box IoT devices. Notably, in our 'black-box' setting, we cannot monitor or alter the device's internal execution, which is significantly different from traditional gray-box solutions depending on instrumentation and hardware. We have the firmware for static analysis, which is common in real-world scenarios. Our core insight is to generate a gray-box model from this black-box setting, addressing the challenges of black-box fuzzing. Specifically, LABRADOR adopts three core techniques to improve the fuzzing efficiency significantly.

Response-based Execution Trace Inference (RTI). Given that some code blocks reference and manipulate strings and output them to the response, we could examine the strings in response to infer which code blocks have been executed. Therefore, we present RTI to infer the execution trace from responses. It extracts strings from back-end binaries and performs string similarity analysis with responses to obtain shared strings, denoted as *explicit string*. For a given response, RTI parses explicit strings and locates code blocks referencing them, denoted as *explicit block* deduced on the execution trace. Note that this deduction is a rough estimation rather than an exact calculation. However, it is helpful for fuzzing. As shown in our evaluation (Table 2) , the explicit blocks are reachable to 62.2% sensitive functions, providing valuable guidance for fuzzing.

IO Oriented Distance Measurement (IODM). Given a test case (i.e., request), we will evaluate its distance to a set of target codes (sinks). Note that it consists of string tokens, and different tokens have different effects on reaching sinks if they get mutated. Therefore, we first evaluate the potential gains (w.r.t reaching sinks) of all strings in the request, then use the *reverse* of the accumulated string gains to measure the request's distance to sinks. In other words, *the higher the accumulated string gains, the closer the request to sinks*. The response's distance is computed similarly and is further used to adjust the measurement. For an explicit string, its potential gain is the accumulation of weights of all sinks reachable from explicit blocks referencing the string. For other non-explicit strings, its potential gain is a small constant.

Distance guided mutation. Given the above code coverage and distance feedbacks, we further leverage them to guide mutating a seed (i.e., test case), including how many times to mutate (i.e., energy assignment), where to mutate, and how to mutate. First, we evaluate the selected seed's distance to sinks and learn from the testing history to evaluate its historical performance, i.e., its potential to yield better test cases. If the distance is lower and the historical performance is better, more energy will be assigned to the seed. Second, we parse the seed into a tree representation and prioritize tree nodes that are more likely to reach sinks to mutate based on node type, node depth, node's potential gain, as well as the historical performance of the seed. Third, we design two mutation operators: one is to generate the node's value according to its properties, including sink and data type for detecting specified vulnerability types, and the other is to mutate the tree structure for larger path exploration.

We have implemented a prototype of LABRADOR and evaluated it on 14 different enterprise-level IoT devices. LABRADOR outperforms four state-of-the-art (SOTA) vulnerability detection tools, including one static analysis tool SaTC [11], one gray-box IoT fuzzer FIRM-AFL [52], one black-box IoT fuzzer SNIPUZZ [18] and one generation-based fuzzer BOOFUZZ [33], in terms of vulnerability discovery. On these IoT device subjects, LABRADOR finds 44X more vulnerabilities than SNIPUZZ, BOOFUZZ and FIRM-AFL, and finds 8.57X more vulnerabilities than SaTC. In total, LABRADOR discovered 79 unknown vulnerabilities on these 14 IoT devices and reported them to the vendors, of which 61 have been assigned with CVEs.

In summary, this paper makes the following contributions.

- We present a novel solution RTI to collect code coverage feedback for black-box IoT devices and present a novel distance measurement scheme accordingly.
- We present the first response guided directed fuzzing solution for black-box IoT devices, which collects the code coverage and distance feedbacks and leverages them to guide efficient test case mutation.
- We evaluated LABRADOR on 14 *enterprise-level* IoT devices. Results showed that it significantly outperforms baselines on vulnerability discovery. It finds 79 unknown vulnerabilities, and 61 are assigned with CVEs.

2. Background and Motivation

2.1. Web Interface of IoT Device

The web serves as the primary communication interface for IoT devices, consisting of front-end and back-end, which adopts human-machine interaction mode (details in Appendix A), and massive natural language properties have to be introduced to represent parameters in request or response, that benefits for the human to understand.

Therefore, the processing logic of back-end binaries will inevitably involve many functions for receiving, parsing, and outputting various strings. Intuitively, a high proportion of basic blocks may reference those strings, which drives us to conduct an empirical study (Table 2) on *explicit block* quantization. Some back-end binaries even have an

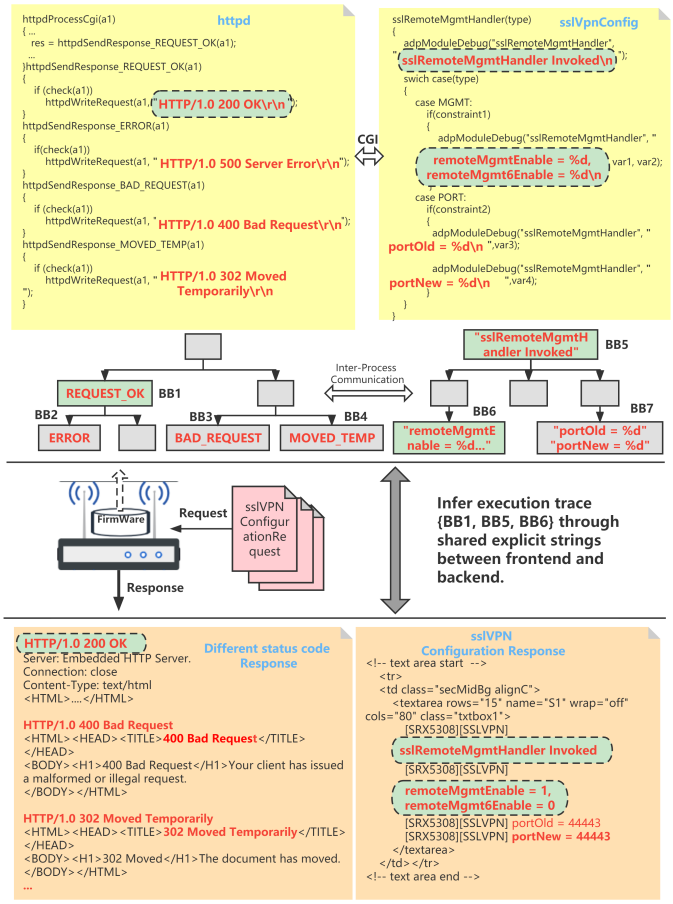

Figure 1: IoT web's response contains many strings used in back-end binaries.

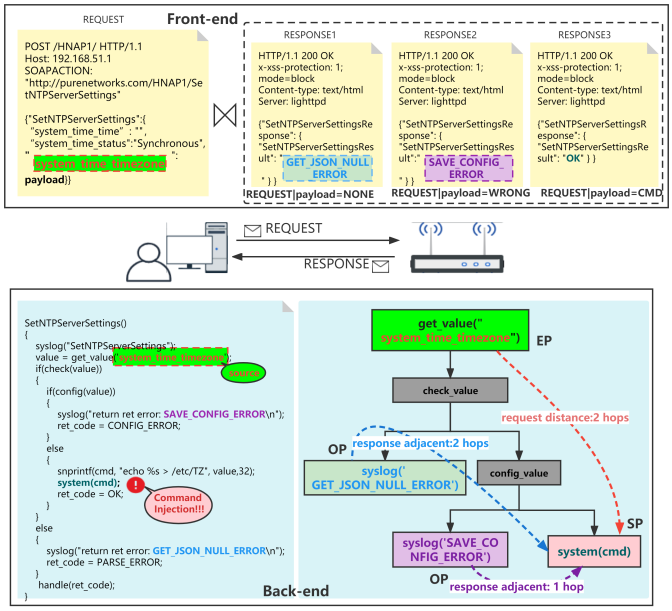

Figure 2: Distance between executed trace and sinks can be inferred from explicit strings in web IO, which can be further used to guide mutation.

explicit block proportion of over 50% (e.g., uci2dat) and an explicit function (i.e., function containing explicit block) proportion approaching 70% (e.g., tcpdump). It reflects that the execution trace inference from the response can generate good utility in obtaining basic code blocks.

2.2. Execution Trace Inference

As previously mentioned, the properties of the web interface can result in various *explicit string* appearing in the response, which is referenced by code blocks in back-end binaries. To demonstrate the process of RTI, Fig. 1 supports a specific case of Netgear devices. When the front-end sends an HTTP request to configure SSL VPN, the *httpd* server parses the parameters and conveys them to the $sslVpnConfig$ application. The processed configuration result is returned to the server and output to the response.

By performing static analysis for *httpd* and $sslVpnConfig$ binaries, we can extract explicit strings and conduct similarity analysis to easily detect those strings that appear in the response, such as "HTTP/1.0 200 OK", "sslRemoteMgmtHandler Invoked" and "remoteMgmtEnable = 1". With reference analysis, {BB1, BB5, BB6} would be inferred as the execution trace.

2.3. IO Oriented Distance Measurement

Previous studies [6, 28] have shown that distance measurement is crucial for directed fuzzing on source code programs. However, there was no distance measurement related work for IoT devices.

We proposed IO (Request & Response) oriented distance measurement for IoT devices firmware as Fig. 2, which demonstrates a specific case in Motorola CX2L router, where the user sends NTP request (upper left of Fig. 2) from the front-end, its three mutated packets are constructed based on the differences in the *payload* field. Corresponding responses (upper right of Fig. 2) exhibited different explicit strings.

The explicit blocks inferred from the request are EP (Entry Point), and the explicit blocks inferred from the response are defined as OP (Outputting Point). In addition, the blocks containing dangerous functions are defined as SP (Sink Point). After reduction, the $SCFG$ (Simplified Control Flow Graph) only contains EP, OP, SP, and other connected blocks.

In this example, it can be determined that all three request packets are associated with the EP referencing the "system_time_timezone" string, which is two hops away from the SP block (lower right of Fig. 2). The OP associated with the "GET_JSON_NULL_ERROR" string in the first response packet is two hops away from the SP. Another OP associated with the "SAVE_CONFIG_ERROR" string in the second response packet is one hop away from the SP.

The third response packet can be inferred to have executed SP successfully, which may trigger a command injection with a distance of just 0 hops from the SP.

3. Design of LABRADOR

3.1. Overview

To avoid the traditional feedback mechanism bottleneck, LABRADOR builds a novel trace and distance feedback model for black-box IoT devices. This supports an efficient mutation policy to cover more code blocks and discover more vulnerabilities. Fig. 3 detailed the workflow of LABRADOR.

Firstly, we designed a response-based execution trace inference mechanism (Section 3.2) to detect which blocks may be executed in the back-end by observing *explicit strings* emerging in the response through a lightweight static analysis.

Then, we designed an IO oriented distance measurement (Section 3.3) for further defining the directed model. LABRADOR characterizes a control flow distance between the execution trace of request and sinks (i.e., dangerous functions). Specifically, LABRADOR first computes a static distance for the request, then updates the distance with the help of inferred execution trace from the response.

Lastly, we designed a distance guided mutation policy to efficiently tune the directed fuzzing(Section 3.4). LABRADOR transfers the request to tree-based representation, estimates the probabilities of tree nodes participating in mutation, and takes type analysis and energy schedule to determine how to mutate.

3.2. Response-based Execution Trace Inference

As shown in FIRM-AFL [52], code coverage feedback is proven to help improve vulnerability discovery in IoT devices. However, it depends on the firmware emulation environment. In reality, many IoT devices lack firmware emulators [45]. More strictly, *enterprise-level* IoT devices tend to enhance security to prevent instrumentation operation [43]. Black-box testing has become the general situation for most industrial-level devices [30].

Benefiting from the frequent human-machine interactive mode of the web interface, we observe that rich strings referenced by the back-end binaries are output to the response. Our empirical study on over 40 web back-end binaries reveals *explicit block* with an average proportion of 14.3%, *explicit function* with an average proportion of 35.8%, and that reached dangerous calls with an average proportion of 62.2% (Table 2). Those statistics imply the potential to infer executed *explicit blocks* in the back-end by identifying their referenced strings emerging in the response.

3.2.1. Execution Trace Inference. An execution trace is a set of basic code blocks. The key to infer execution trace is to perform similarity analysis between strings separately emerging in the response and code blocks. As an explicit string typically shares a similar format between the front-end and back-end, the word shape similarity model is more suitable for this scenario.

There are two situations for measuring similarity, one is suitable for inferring similarity by capturing differences, and the other is suitable for computing similarity directly. Given two strings s_1 and s_2 with close lengths, it is the typical former situation, and Levenshtein distance[1] $EDIT(s_1, s_2)$ is fit to this measurement. For the other situation, as two strings(s_1, s_2) include the same explicit string but with significantly different lengths, that frequently occur in the real world. For example, in Netgear devices, the explicit string "pptpEnable" of the front-end is flowed to the back-end and manifested as "pptpServerIPRange.pptpEnable" after extending with another string. Instead of using Levenshtein distance measurement, the longest common sub-string computing function $LCS(s_1, s_2)$ is more suitable to the latter situation.

Overall the above analysis, we formula the string similarity measurement $SIM(s_1, s_2)$ as Eq. (1).

$$SIM(s_1, s_2) = MAX(1 - \frac{EDIT(s_1, s_2)}{MAX(L(s_1), L(s_2))}, \frac{LCS(s_1, s_2)}{MIN(L(s_1), L(s_2))}) \quad (1)$$

It takes the maximal value as the final similarity measurement to minimize false negatives as much as possible.

In addition to the similarity measurement definition, how to get explicit strings and response are also the necessary steps of RTI. Algorithm 1 shows the main workflow of RTI. The Line 1 generates potential explicit strings by calling $StaticExtract$, which performs static analysis based on back-end binary parsing and disassembling operations. It obtains the response by sending the request ($seed$) through the web interface (Line 2). RTI takes each line of response as a string (str), which compared with each potential explicit string (es) in binary one by one. Once the similarity between these two strings, $SIM(str, es)$, is computed to be not less than one threshold (ξ), all explicit blocks referencing this true explicit string ($EB(es)$) will be merged to the trace. If the set of code blocks ($EB(es)$) triggers new block coverage, all new blocks will be added to the global block coverage cov.(Lines 4 to 9 of Algorithm 1).

3.2.2. Capability Analysis. What we are concerned best with is whether RTI has a better advantage in performance and applicability compared to the traditional instrumented trace feedback scheme.

Performance Advantage. RTI is more lightweight than intrusive instrumentation. As shown in our evaluation (Table 10), the overhead of execution once under the black-box environment is approximately hundreds of milliseconds on average. RTI brings a small overhead (ten milliseconds) due to the lightweight static analysis. Notably, RTI leverages a line cache mechanism to avoid repeated similarity analysis, accelerating the process by only focusing on those

1. https://en.wikipedia.org/wiki/Levenshtein_distance

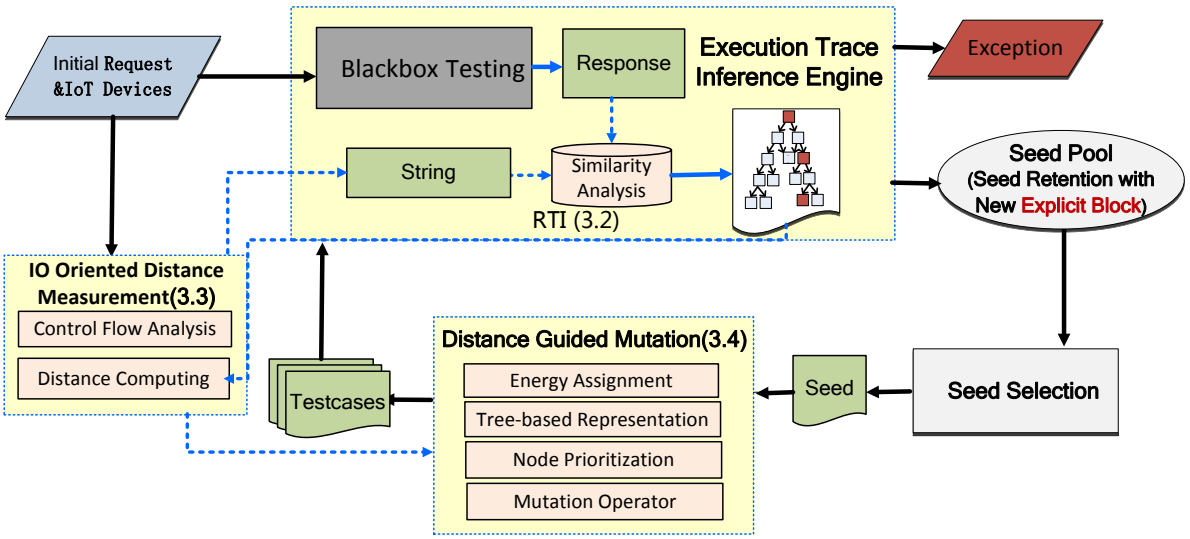

Figure 3: LABRADOR workflow.

Algorithm 1: Execution Trace Inference.

Input: Tested device $\mathcal{P}$ with black-box environment, request as $seed$
Output: $Trace$ of $seed$, updated coverage Cov

```
/* Get response by sending request seed to device P */
PotentialExplicitStrings ← StaticExtract(P)
response ← REQUEST{P, seed}
Trace ← ∅
foreach str ∈ response do /* Take each line as string */
    foreach es ∈ PotentialExplicitStrings do /* es is one unique potential explicit string in binary */
        if SIM(str, es) ≥ ξ then
            Trace ← Trace ∪ {EB(es)}
            if EB(es) ∉ Cov then
                Cov ← Cov ∪ {EB(es)}
```

new sentences in the response. For intrusive instrumentation schemes, dynamic trace tracking introduces at least ten times more overhead [22].

Applicability Advantage. Unlike traditional trace tracking schemes, RTI has no limitation on the testing environment; it can adapt to physical devices or emulated machines. More importantly, many industrial-level devices lack an emulated environment and prevent intrusive instrumentation. In reality, RTI could apply to any IoT device subject with a web interface.

3.3. IO Oriented Distance Measurement

Distance measurement is critical to directed gray-box fuzzing. Previous work focuses on how to model control flow [6] and data flow [28] related distance to improve the directed efficiency. Some other work, such as unreachable paths filtering [25, 54], is also proven to be an effective way to accelerate the directed process.

LABRADOR firstly performs static control flow analysis (Section 3.3.1), which generates a simplified control flow graph ($SCFG$) by removing blocks irrelevant to the request, response, and sinks.

With the assistance of $SCFG$, LABRADOR further performs IO oriented distance measurement (Section 3.3.2), which computes the distance between IO (i.e., request & response) and sinks, being used to tune the efficiency of directed fuzzing.

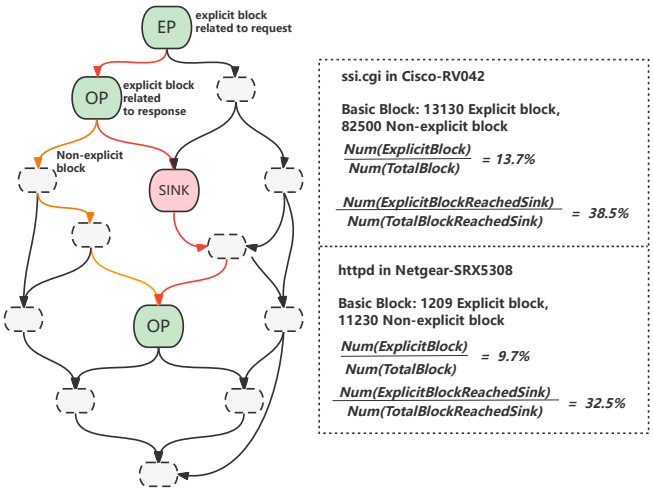

Figure 4: Simplified Control flow Graph.

3.3.1. Control Flow Analysis. Given a whole inter-process control flow graph ($ICFG$) generated by initial static analysis, LABRADOR further tags attribute for each basic block by similarity analysis. As shown in Fig. 4, there are four

basic blocks in the $SCFG$: EP related to the request, OP related to the response, SP (i.e., sink), and normal block. LABRADOR takes reachable analysis between sink blocks and other blocks, which aims to filter nodes unreachable to sinks when computing distance. LABRADOR identifies insecure functions as sinks (SP) in $ICFG$. Because insecure functions lack a unified naming mechanism, LABRADOR builds a database saving conventional insecure function symbols and employs similarity analysis to locate more potential insecure functions. In addition, LABRADOR opens an interface allowing for the introduction of expert knowledge. Insecure code snippets analyzed by human experiences, such as patches or program loops, could be defined in the open interface and treated as SP. As evaluated (Table 2), the scale of original $ICFG$ will be reduced to about 10% after similarity analysis.

3.3.2. Distance Computing . Given one seed, LABRADOR measures the distance between it and sinks from two perspectives. One is from request (i.e., input) to sinks, which calculates a distance between the two before execution. The other is from response (i.e., output) to sinks, which gets a distance after execution trace inference. The request's perspective is to assess the potential value of the current seed, and the response's distance is to get a more accurate deal and help LABRADOR update the assessment in time. These two distance metrics will be used to build effective mutation and energy schedule policies.

For any explicit string es, the gain is assessed through its distance to sink, defined as Eq. (2). The closer the distance, the bigger the gain.

$$GAIN(es) = \sum_{eb \in EB(es)} \sum_{sink \in REACH(eb)} \frac{W(sink)}{DIS_{dij}(eb, sink)} \tag{2}$$

which computes the distance between the referenced explicit blocks $EB(es)$ and reachable-sinks $REACH(eb)$ (i.e., the set of sinks that can be reached by any block eb in $EB(es)$). Using the harmonic mean [2] to evaluate the distance between multiple points is suitable. Moreover, sinks may introduce different levels of security risk. For example, function $strcpy$ is more dangerous than homogeneous function $strncpy$, as it does not perform length checking and is more likely to trigger buffer overflow. To reflect the level of security risk introduced by sinks, Eq. (2) assigns danger factor $W(sink)$ to each sink. The bigger weight of the sink, the more dangerous it is. Sinks such as $system$ and $popen$ are set highest weight as they are dangerous and easily exploited, while some sink like $strcpy$ and $memcpy$ are set higher weight, which can easily result in buffer overflow. Other sinks like $strncpy$ and $snprintf$ are set lower weight. In summary, we use a weighted harmonic mean to calculate the distance between an explicit string and multiple sinks, where a smaller distance (corresponding to a bigger gain) indicates a closer execution path to the sink.

2. https://en.wikipedia.org/wiki/Harmonic_mean

Moreover, there are many non-explicit strings in the request. We further compute the gain of a general string as Eq. (3).

$$GAIN'(str) = \begin{cases} GAIN(str), & str \in ES \\ k \cdot \frac{|ES|}{\sum_{str' \in ES} 1/GAIN(str')}, & str \notin ES \end{cases} \tag{3}$$

If str is out of the set of explicit strings ES, we compute $GAIN'(str)$ as the gain that is a positive correlation with the average gain of all explicit strings, where k is a tiny constant. The purpose of $GAIN'(str)$ is to illustrate that those non-explicit strings in the request shouldn't be ignored.

Finally, we define the distance between IO (i.e., request & response) and sinks as Eq. (4).

$$DIST(IO) = \begin{cases} \frac{1}{\sum_{para \in IO} GAIN'(para)}, & IO = Request \\ \frac{1}{\sum_{str \in ES \cap IO} GAIN'(str)}, & IO = Response \end{cases} \tag{4}$$

Note that when measuring the distance of the request, we need to consider parameters belonging to non-explicit strings due to the false negative brought by static analysis.

3.4. Distance Guided Mutation

When mutating a seed, the fuzzer has to decide how many test cases to yield (i.e., energy assignment), where to mutate, and how to mutate. LABRADOR adopts dynamic strategies to address these questions based on the above distances. The overall workflow of distance-guided mutation is shown as Algorithm 2.

3.4.1. Energy Assignment. For a seed s, its mutation power (i.e., energy) is computed statically according to the request's distance and adjusted dynamically according to the response's distance, computed as Eq. (6).

$$HIST\_Value(s) = DIST(RES(s)) - \underset{s' \in S'}{MIN}\, DIST(RES(s')) \tag{5}$$

$$ENERGY(s) = A \times \ln\left(DIST(s)^{-1} + 1\right) \times \delta^{HIST\_Value(s)} \tag{6}$$

where A is the power unit, deciding the overall energy power scale, and recommended as 500. $RES(s)$ is the response that output by executing seed s, S' is the set of seeds mutated by s in the latest fixed times. δ is a constant slightly larger than 1.0, which controls the power adjustment step.

3.4.2. Tree-based Representation. For IoT web interfaces, its request format is mainly based on structural grammar. Therefore, LABRADOR transfers the request to tree-based representation before mutation, shown as Fig. 5, where the node of the tree is the minor mutation granularity.

The tree is divided into domains including $header$, $parameter$, and $attribute$. $parameter$ is the primary target for mutation, as it contains the request's non-trivial parameters. $header$ includes standard information such as content type, cookie, and user agent, mutated at lower probabilities to ensure the server can parse most mutated seeds. $attribute$

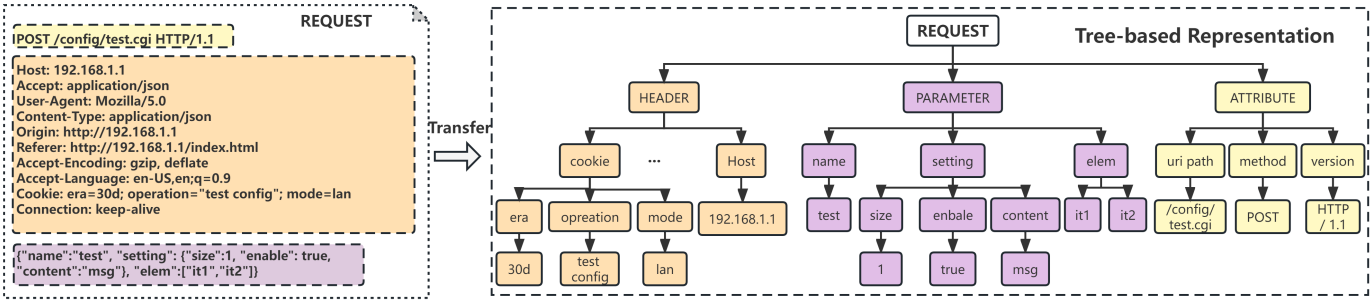

Figure 5: Transfer request seed to tree-based representation.

Algorithm 2: Mutation Policy of LABRADOR.

Input: seed s as request, $response$ after device executing seed
Output: S' as set of seeds that mutated by s

```
/* Transfer request to tree-based
representation as Fig. 5 */
tree ← TreeGen(s)
power ← ENERGY(s) /* Eq. (6) */
repeat/* mutation */
    tree_tmp ← tree
    /* Selected probability according
    to Eq. (7) */
    Nodes ← SelectNode(tree)
    foreach node ∈ Nodes do
        /* Get type of reached sinks */
        {type_sink, type_data} ← GetAttributes(node)
        /* Select mutation operator randomly
        */
        mut_op ← SelectOperator(node)
        tree' ←
          MUTATE(tree_tmp, node, type_data, type_sink, mut_op)
        tree_tmp ← tree'
    seed' ← SeedRecover(tree')/* Transfer tree to
      request */
    S' ← S' ∪ seed'
    power ← power − 1
until power = 0
```

includes method, URI path, and HTTP version, mutated at lower probabilities as well, as they have strict value requirements and affect the parsing of requests. *header* and *attribute* are usually parsed by the web server (e.g., *httpd*) according to the HTTP protocol, while *parameter* is generally processed by multiple applications in the back-end (e.g., *ssi.cgi*).

Every node in the tree represents the corresponding token string in the request, having two properties: sink type and data type. The sink type is the vulnerability type that reachable sinks may introduce, and it is used to select appropriate strategies to trigger vulnerability at those sinks. Meanwhile, data type represents the data associated with the given node, such as numeric, string, or boolean. This information determines how to manipulate the value when conducting mutation on the node.

3.4.3. Node Prioritization. Different nodes have a varying impact on the effect of mutation. To represent this variation, LABRADOR gives priority to nodes which can accelerate the directed process. Specifically, each node is given a mutation probability according to its potentiality and performance. We assess nodes' mutation probability according to the following factors: (1) Gain of node. The bigger the gain, the higher the priority. (2) Depth in the tree. The deeper the depth, the higher the priority. Deeper nodes usually indicate longer parsing paths in the back-end and should be given more opportunities for mutation. (3) Domain in the tree. The domain reflects the region of code where the node is parsed in the back-end, which should be given different opportunities. (4) Dynamic distance of response. The above three factors are used to give a static assessment for the seed before execution, which is then adjusted dynamically during test by the response's distance, following the rule that the closer the response, the higher the priority.

Specifically, the gain of node defines Eq. (12), the depth of node defines Eq. (11), and the domain of node defines Eq. (10). They together define Eq. (9). Response's distance defines Eq. (8). The overall prioritization is defined as Eq. (7).

$$DYN\_PROB_{mut}(n) = PROB_{mut}(n) \times L(Response, Response') \tag{7}$$

where $PROB_{mut}(n)$ is the static probability of node n, computed as Eq. (9), and $L(Response, Response')$ is the dynamical adjustment factor, computed as Eq. (8).

$$L(Response, Response') = \theta^{(DIST(Response')^{-1} - DIST(Response)^{-1})} \tag{8}$$

where $Response$ is the output of execution before mutation and $Response'$ is the output with the closest distance among the seeds generated by mutating this seed in the latest fixed times. θ is slightly more significant than 1.0, which controls the probability adjustment step.

$$PROB_{mut}(n) = DOMAIN(n) \times DEPTH_{nor}(n) \times GAIN_{nor}(n) \tag{9}$$

As shown in Eq. (10), it gives different constant probabilities to node n according to its domain. Specifically, LABRADOR set α to 0.05 and β to 0.95, which are statistical ratios of explicit blocks distributed in different back-end

binaries [3]. Meanwhile, as the node n goes deeper in the tree, the more significant normalized probability would be given to it, detailed as Eq. (11). Furthermore, the node n will get a more significant normalized probability if it's closer to sinks, as shown in Eq. (12).

$$DOMAIN(n) = \begin{cases} \alpha, & n \in header \bigcup attribute \\ \beta, & n \in parameter \end{cases} \tag{10}$$

$$DEPTH_{nor}(n) = \frac{e^{DEPTH(n)} - e^{\underset{n \in Tree}{MIN}\,(DEPTH(n))}}{e^{\underset{n \in Tree}{MAX}\,(DEPTH(n))} - e^{\underset{n \in Tree}{MIN}\,(DEPTH(n))}} \tag{11}$$

$$GAIN_{nor}(n) = \frac{GAIN'(n) - \underset{n \in Tree}{MIN}\, GAIN'(n)}{\underset{n \in Tree}{MAX}\, GAIN'(n) - \underset{n \in Tree}{MIN}\, GAIN'(n)} \tag{12}$$

3.4.4. How to Mutate. Two mutation operators are implemented on the type and structure level separately.

The type operator modifies the node's value according to its properties, including sink and data type. For example, if the sink type is command injection (CI), LABRADOR favors embedding CI-related payload into value; If the data type is numeric, LABRADOR tends to change it to extreme values such as 0, -1, or a large number. The type operator also includes a havoc strategy that randomly perturbs the node's value, such as randomly selecting a byte to add or subtract a number.

The structure operator modifies the tree's structure by copying, adding, deleting, or replacing sub-trees, corresponding to the same modification on the request's parameters.

4. Implementation

We have developed a prototype of LABRADOR, which consists of over 7100 lines of Python code, over 400 lines of C code, and over 120 lines of Bash shell code.

The whole framework is divided into static analysis and fuzz campaign. The static analysis component extracts essential information from the firmware to support the fuzz campaign. The main components are as follows.

Precedent Static Analyzer. LABRADOR first generates ICFG for back-end binaries, then extracts all strings from the back-end binaries, excluding symbols obviously not present in the front-end, as potential explicit strings. Then LABRADOR marks code blocks referencing explicit strings in the ICFG as explicit blocks, marks EP in ICFG for request through similarity analysis and marks insecure functions and actions as SP. LABRADOR calculates the shortest distance between any explicit block and SP in the ICFG. Meanwhile, it computes a static distance for every initial request in advance according to Eq. (4).

Additionally, we designed an open interface that allowed for introducing expert knowledge. Insecure code snippets analyzed by human experiences, such as patches or program loops, could be defined and treated as sinks. We classified sink types based on the type of insecure function, indicating the potential vulnerability the sink could trigger.

Tree-Based Mutator. LABRADOR employs tree-based representation to mutate the request. The mutator implements the distance-guided mutation policy(Algorithm 2).

Dynamic Feedback Monitor. The monitor utilizes RTI to infer execution traces in the back-end. Additionally, it leverages the response to assist in directed fuzzing. By searching explicit strings in the response, it calculates the response's distance according to Eq. (4). This distance metric is then used to adjust tree nodes' mutation probability (Eq. (7)) and seed's mutation energy (Eq. (6)) dynamically during the test.

Multiple Exceptions Detector. LABRADOR can detect multiple exceptions, including command injection, memory corruption, and cross-site scripting (XSS). Specific detection policies have been designed for each type. In the case of CI, LABRADOR attempts to execute a specific command on the back-end and detects whether a specific request can be received from the network. The detector checks the network delay to determine if disconnection occurred, portending the memory corruption in the back-end. For XSS, the detector examines the response's content for the presence of a magic string pattern.

Token-based Session Keeper. Many web interfaces of IoT devices have the login check for authentication. Once login, a session is started. LABRADOR needs login to the web and keeps the session alive during the test. As illustrated in Appendix A, the key to maintaining state lies in the session token. Thus LABRADOR continuously checks whether the session token has expired and updates it if needed, ensuring that the session remains active all the time.

5. Evaluation

In this section, we evaluated the efficiency of LABRADOR and showed improvements compared to other SOTA tools.

5.1. Experiment setup

IoT Devices Testing Set. As SOTA works [11, 35, 40, 52], we chose one of the main branches of IoT, networking-based devices for testing, whose security concerns have been in the spotlight in recent years.

We selected device subjects based on manufacturer popularity, testing frequency, firmware updating frequency, and functional diversity of web service. Fourteen enterprise-level subjects were selected as the testing set, all from well-known companies (Cisco, Netgear, and TPLink, etc.) and focusing on network and security services such as routing, firewall, and VPN. The detailed model of devices and the latest version of firmware are shown as Table 1.

Comparison Tools. We compared LABRADOR with other popular fuzzers, including SNIPUZZ, BOOFUZZ, and FIRM-AFL. Our choice was based on specific factors.

3. That is explicit blocks in back-end binaries that parse $parameter$ domain account for 95% of all explicit blocks. It shows that the code size for processing $parameter$ is much larger than the other two and should have a higher mutation opportunity.

TABLE 1: Benchmark and Environment.

Vendor	Device	Version	Environment
Cisco	RV-042	4.2.3.14	Physical Machine
Netgear	SRX5308	4.3.5-3	Physical Machine
	FVS318G	3.1.1-18	Physical Machine
	R7000	1.0.11.136	Physical Machine
	WNDR3700v2	1.0.1.14	Emulated Machine
TrendNet	TEW-811DRU	1.0.10.0	Physical Machine
	TEW-652BRP	3.04B01	Emulated Machine
Linksys	WRT54GL	4.30.18.006	Emulated Machine
	E1000	2.1.03.005	Emulated Machine
TPLink	Archer C50	US_V2_160801	Emulated Machine
	Archer C20	V1_150707	Emulated Machine
	ARcher C7	US_V2_180114	Emulated Machine
Ubiquiti	EdgeRouterX	2.0.9	Physical Machine
Motorola	CX2L	1.0.1	Physical Machine

Firstly, we chose SNIPUZZ because it also runs on the black-box testing environment and utilizes response-based feedback, which can help to highlight certain advantages of LABRADOR's feedback mechanism. Secondly, BOOFUZZ is a well-known black-box fuzzer for network protocols, which has excellent stability and is frequently used as a benchmark fuzzer for assessing fuzzing performance. Thirdly, FIRM-AFL is a gray-box IoT fuzzer relying on emulation to provide code coverage feedback, which is beneficial to express challenges in fuzzing IoT. Lastly, we evaluated the capability of LABRADOR to discover vulnerability by comparing it with SaTC, a static analysis tool that performs well in detecting firmware vulnerabilities.

Performance Measurement. We have selected vulnerability discovery and explicit block coverage as the primary metrics to compare the efficiency of LABRADOR with each fuzzer. To calculate explicit block coverage for SNIPUZZ, we have implemented RTI and filtered its duplicated seeds based on unique explicit block identification. We were tracking the total number of vulnerabilities detected by different tools to evaluate the ability for vulnerability discovery. Moreover, we assessed the effectiveness of directedness by eliminating various optimization strategies for LABRADOR, mainly through tracking the effect on the growth trend of explicit block coverage and the number of exceptions.

Experiment environment. We ran each tool on the Kali 6.1.0 system, equipped with an Intel Core i7 processor at 2.6 GHz and 8 GB RAM.

Additionally, we prioritized emulated environments for IoT devices, and LABRADOR would turn to the physical device when lacking a emulated environment, as shown in Table 1.

5.2. Empirical Static Analysis

The motivation for RTI originates from empirical static analysis, which aims to ascertain whether there exists a sufficient proportion of explicit blocks and explicit functions. Moreover, we account for the proportion of explicit functions which can reach sinks, to show the proportion of effective information leakage of CFG for directed fuzzing.

As shown in Table 2 , the empirical study on 42 web programs reveals an average of 14.3% explicit blocks, that are even exceeding 40% in several web applications including $single\_cgi$, $hnap$ and $uci2dat$. On the explicit function (function containing explicit block), its average proportion is increased to 35.8%. This provides a basis for constructing an effective gray-box feedback mechanism.

Specifically, after conducting reachability analysis on sinks (i.e., dangerous functions), we discovered that explicit functions reachable to sinks occupy as high as 62.2% of all functions that can reach. This provided us with good insight into modeling directed fuzzing.

It is worth emphasizing that the average of 86.6% explicit functions could reach sinks, which further illustrates that most web interfaces directly expose serious attack faces.

5.3. Effect of Directedness

We further used the four randomly selected devices in the empirical study to evaluate the effect of directedness. As the fuzzer went, more and more seeds hitting new explicit blocks were added to the seed pool, showing as Fig. 9, which exhibited the evolution capability of LABRADOR. The explicit block coverage showed a similar growth trend as the seed pool, demonstrated in Fig. 6.

We developed three variants of LABRADOR to assess the impact of two models: Response-based Execution Trace Inference (RTI) and IO oriented Distance Assistance (IODA). LABRADOR-NoIODA excludes IODA from LABRADOR, restricting it to RTI for seed preservation. This implies that the mutation is conducted randomly based on the tree without any distance assistance. LABRADOR-NoRTI removes RTI, meaning that explicit strings search in the response is impossible and thus cannot supplement new seeds. As a result, LABRADOR-NoRTI can solely employ IODA in the request, providing only static distance assistance for mutation as Eq. (9) depicts. LABRADOR-NoRTI-NoIODA eliminates both RTI and IODA, utilizing only tree-based mutation with fixed energy allocation and without seed preservation.

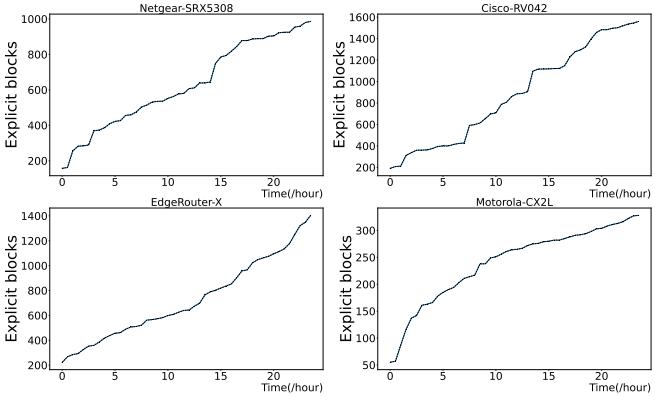

Figure 6: Growth trend of explicit block coverage.

Impact on the growth trend of exception generation. The number of exceptions discovered is strongly relevant to the ability of vulnerability discovery. Fig. 7 illustrates the 24-hour exceptions reported by LABRADOR

TABLE 2: Occurrence of explicit blocks and functions in back-end binaries. #Blocks =total blocks in binary, #ExB = total blocks that reference explicit strings, ExB.ratio = $\frac{\#ExB}{\#Blocks}$, #Fun =total functions in binary, #ExF = total functions that reference explicit strings, ExF.ratio = $\frac{\#ExF}{\#Fun}$,#RF = total functions that reach sinks, #RExF = total explicit functions that reach sinks, R.ratio = $\frac{\#RExF}{\#RF}$.

Firmware	Back-end Binary	Blocks			Functions			Functions for Reaching Sink			$\frac{\#RExF}{\#ExF}$
		#Blocks	#ExB	ExB.ratio	#Fun	#ExF	ExF.ratio	#RF	#RExF	R.ratio	
Netgear-5308	httpd	12439	1209	9.7%	302	88	29.1%	115	83	72.2%	94.3%
	pptpdConfig	1886	264	14%	47	6	12.8%	10	6	60.0%	100.0%
	sslvpnConfig	11964	1839	15.4%	141	43	30.5%	50	42	84.0%	97.7%
	firewallPolicy	58137	10004	17.2%	402	263	65.4%	234	180	76.9%	68.4%
	upnpd	2426	359	14.8%	153	29	19.0%	48	29	60.4%	100.0%
	single_cgi	6031	2937	48.7%	219	63	28.8%	64	53	82.8%	84.1%
	dhcpConfig	3591	322	9.0%	56	8	14.3%	14	8	57.1%	100.0%
	dimclient	53657	3692	6.9%	2035	484	23.8%	1257	443	35.2%	91.5%
	vipsecureConfig	23719	1515	6.4%	255	92	36.1%	96	73	76.0%	79.3%
	tcpdump	73359	13944	19.0%	761	406	53.4%	375	286	76.3%	70.4%
	firewalld	58171	10329	17.8%	405	267	65.9%	234	182	77.8%	68.2%
EdgeRouterX	lighttpd	49929	2257	4.5%	682	206	30.2%	244	129	52.9%	62.6%
	miniupnpd	16442	1144	7.0%	259	118	45.6%	147	113	76.9%	95.8%
	upnpc	8927	453	5.1%	136	49	36.0%	63	46	73.0%	93.9%
	upnpd	3149	375	11.9%	145	28	19.3%	49	28	57.1%	100.0%
	ubnt-util	229597	8281	3.6%	2814	765	27.2%	1038	573	55.2%	74.9%
	squid	543316	70678	13.0%	9320	4405	47.3%	6500	4169	64.1%	94.6%
	ubnt-udapi-server	395076	5704	1.4%	2349	536	22.8%	856	451	52.7%	84.1%
	udapi-bridge	121505	11001	9.1%	2804	996	35.5%	1299	832	64.0%	83.5%
	tcpdump	119922	15988	13.3%	732	509	69.5%	434	377	86.9%	74.1%
Cisco-RV042	ssi.cgi	95630	13130	13.7%	1977	954	48.3%	1141	858	75.2%	89.9%
	httpd	29049	3080	10.6%	1038	442	42.6%	593	366	61.7%	82.8%
	pluto	49244	5538	11.2%	1302	553	42.5%	721	494	68.5%	89.3%
	smbclient	69859	9338	13.4%	2490	696	28.0%	1357	659	48.6%	94.7%
	pppd	29653	3182	10.7%	903	276	30.6%	439	249	56.7%	90.2%
	dhcpd	71761	7802	10.9%	1575	620	39.4%	974	590	60.6%	95.2%
	snmpd	112402	11059	9.8%	3068	1618	52.7%	2231	1579	70.8%	97.6%
	main_bin	42358	5603	13.2%	1373	682	49.7%	921	509	55.3%	74.6%
	import_config.cgi	23073	2237	9.7%	847	376	44.4%	544	301	55.3%	80.1%
	upnpd	28545	1286	4.50%	863	119	13.8%	370	105	28.4%	88.2%
	userLogin.cgi	66893	3831	5.7%	1875	450	24.0%	857	365	42.6%	81.1%
	webBoot	33231	7102	21.4%	721	348	48.3%	397	324	81.6%	93.1%
Motorola-CX2L	hnap	4892	2134	43.6%	255	104	40.8%	141	101	71.6%	97.1%
	lighttpd	9494	1580	16.6%	512	108	21.1%	152	72	47.4%	66.7%
	miniupnpd	5752	886	15.4%	236	103	43.6%	138	100	72.5%	79.1%
	netifd	7638	663	8.7%	572	122	21.3%	236	95	40.3%	77.9%
	pppd	14472	1726	11.9%	593	202	34.1%	353	185	52.4%	91.6%
	prog.cgi	18409	5574	30.3%	822	469	57.1%	574	452	78.7%	96.4%
	scopd	9988	1134	11.4%	430	162	37.7%	225	154	68.4%	95.1%
	uci2dat	1758	912	51.9%	52	8	15.4%	19	6	31.6%	75.0%
	upload_firmware	2198	318	14.5%	151	41	27.2%	76	38	50.0%	92.7%
	upload_settings	2053	314	15.3%	136	41	30.1%	74	38	51.4%	92.7%
Average				14.3%			35.8%			62.2%	86.6%

and its variants. Among them, LABRADOR outperformed others by continuously discovering exceptions during the test. On the other hand, LABRADOR-NoRTI-NoIODA exhibited slower exception growth. LABRADOR-NoRTI and LABRADOR-NoIODA both showed significant improvement compared to LABRADOR-NoRTI-NoIODA, highlighting the effectiveness of RTI and IODA. Overall, LABRADOR discovered 2X more exceptions than any other variants. Worthy noting, LABRADOR-NoRTI was more efficient at triggering exceptions than LABRADOR-NoIODA because IODA's directedness could accelerate the process to trigger dangerous calls.

Impact on the total number of vulnerabilities. Fig. 8 displays the overall count of vulnerabilities detected by LABRADOR and its different versions. Consistent with exception evaluation, LABRADOR surpassed all the others regarding the number of identified vulnerabilities. Both LABRADOR-NoRTI and LABRADOR-NoIODA exposed a substantially larger quantity of vulnerabilities compared to LABRADOR-NoRTI-NoIODA. Despite not utilizing code coverage feedback to direct the fuzzer, LABRADOR-NoRTI discovered many vulnerabilities, slightly more than LABRADOR-NoIODA. This superior performance can be attributed to IODA's excellent direction capability.

Impact on vulnerability discovery speed. To quantitatively evaluate the directedness of LABRADOR, we employed the evaluation criteria as previous work [6], which includes Time-To-Trigger (TTT), factor and $\hat{A}12$, where TTT is the primary metric that measures the time spent to trigger a specific type of vulnerability for the first time. Each run was repeated five times, and the average results were recorded, effectively demonstrating LABRADOR's ability to expose multiple vulnerabilities quickly.

As shown in Table 7, LABRADOR outperformed others in all cases regarding TTT. LABRADOR-NoRTI behaved better directedness, finding vulnerabilities faster than

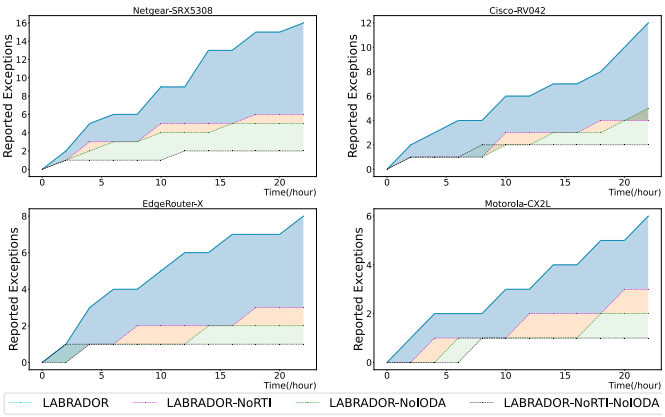

Figure 7: Improvements brought by RTI and IODA, regarding exceptions reported for four devices.

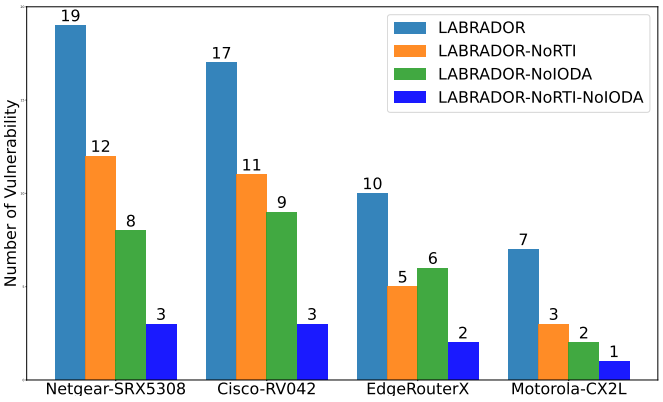

Figure 8: Overall count of vulnerabilities detected by LABRADOR and its different versions.

LABRADOR-NoIODA and LABRADOR-NoRTI-NoIODA in most cases, except one memory corruption in EdgeRouterX, indicating that the distance feedback is more sensitive to vulnerability than the code coverage for the directed model.

Especially, LABRADOR-NoIODA was even worse than LABRADOR-NoRTI-NoIODA in some cases. For instance, it failed to discover the command injection in CISCO-RV042 within 24 hours, while LABRADOR-NoRTI-NoIODA took over 18 hours to detect the vulnerability. It shows LABRADOR-NoIODA may waste more time mutating those seeds far from sinks without assistance from IO oriented distance.

5.4. Comparison with target tools

As aforementioned, we chose SNIPUZZ, BOOFUZZ, FIRM-AFL, and SaTC as compared tools and set the head-to-head experiment for LABRADOR and these 4 tools.

LABRADOR VS. SNIPUZZ. Both LABRADOR and SNIPUZZ leverage the response of the IoT device to construct a feedback mechanism. The most significant difference is that LABRADOR combines binary static analysis and response to infer executed explicit blocks. In contrast,

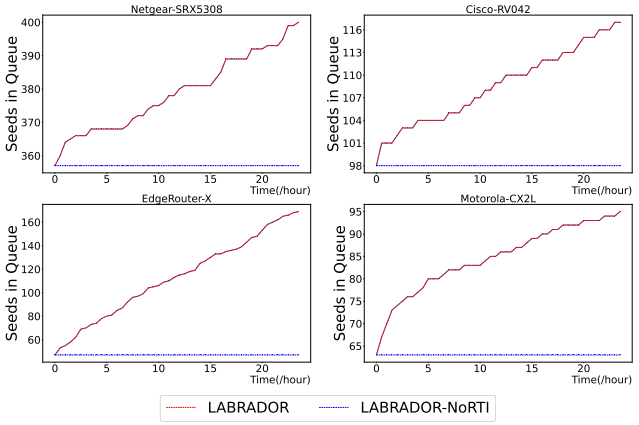

Figure 9: Growth trend of total seeds in seed pool.

SNIPUZZ directly uses the difference of response to infer message snippets to improve mutation policy. The feedback mechanism is the basis for seed preservation, and the fact is that different responses may be from the same execution path as shown in Fig. 10, which demonstrates most seeds kept by SNIPUZZ are duplicated in terms of explicit blocks. It brought the serious result as shown in Table 3, among the 8 device subjects randomly selected from the testing set, SNIPUZZ only discovered 1 vulnerability in TPLink-C7 due to many vital seeds being discarded, whereas LABRADOR discovered 45 vulnerabilities. It shows LABRADOR outperformed SNIPUZZ in vulnerability discovery.

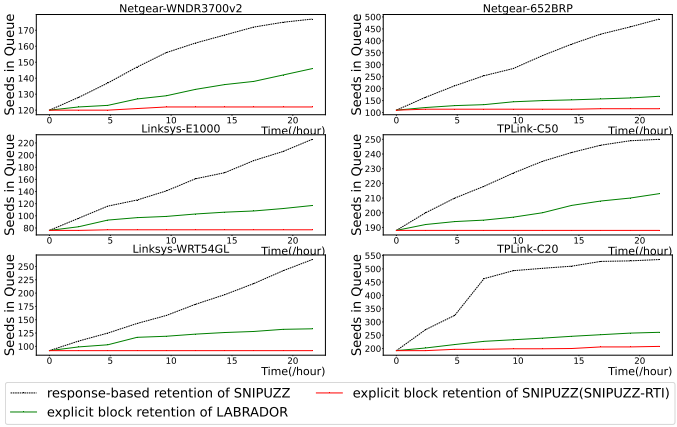

Figure 10: Compare response feedback with SNIPUZZ.

LABRADOR VS. BOOFUZZ. Before comparing with BOOFUZZ, we tried our best to implement the IODA strategy on BOOFUZZ and generated an enhanced version BOOFUZZ-IODA, which dynamically adjusts mutation templates with the help of IODA to prioritize different string tokens in the request. As shown in Table 3, BOOFUZZ and BOOFUZZ-IODA failed to discover any vulnerability on the same benchmark as SNIPUZZ. This is because BOOFUZZ is a traditional black-box fuzzer, which lacks any feedback mechanism and depends on template-based seed generation. This limits BOOFUZZ's mutation space, and it isn't easy to leverage the advantage of the IODA strategy.

TABLE 3: Compared to SNIPUZZ/BOOFUZZ in vulnerability discovery. #BOFUZZ=Both BOOFUZZ and BOOFUZZ-IODA.

Vendor	Device	SNIPUZZ	#BOOFUZZ	LABRADOR
TPLink	C50	0	0	1
	C7	1	0	1
Netgear	WNDR3700v2	0	0	3
	SRX5308	0	0	19
Linksys	WRT54GL	0	0	3
	E1000	0	0	3
Ubiquiti	EdgeRouterX	0	0	10
Trendnet	TEW-652BRP	0	0	4
Total	-	1	0	44

TABLE 4: Further enhance BOOFUZZ with RTI. B-IODA=BOOFUZZ-IODA, B-ENHANCE=BOOFUZZ-ENHANCE, #VUL=discoverd vulnerabilities.

Vendor	**Device**	**Fuzzer**	**ExplicitBlock**	**#VUL**
Linksys	WRT54GL	B-IODA	137	0
		B-ENHANCE	425 (+210%)	1
Netgear	WNDR3700v2	B-IODA	384	0
		B-ENHANCE	765 (+99%)	0
TrendNet	TEW-652BRP	B-IODA	189	0
		B-ENHANCE	412 (+118%)	1
Ubiquiti	EdgeRouterX	B-IODA	261	0
		B-ENHANCE	638 (144%)	0

To evaluate the significance of explicit blocks, we further implemented BOOFUZZ-ENHANCE, which added the RTI mechanism to BOOFUZZ-IODA. We conducted comparison experiments on four devices, each continuously tested for 24 hours. Result Table 4 shows that BOOFUZZ-ENHANCE triggered 1.4 times more explicit blocks on average, and successfully discovered 2 unknown crashes on 2 devices.

LABRADOR VS. FIRM-AFL. FIRM-AFL depends on the emulation environment to implement instrumentation. Thus, we selected emulated machines in the testing set for comparison. As shown in Table 5, FIRM-AFL failed to discover any vulnerability. Digging deeper, we believe its mutation policy may be the main reason. Despite code coverage feedback, FIRM-AFL directly uses AFL [49] to mutate seed on bit-level and entirely randomly, rather than leveraging the feedback further, which is difficult to generate high-quality test cases.

LABRADOR VS. SaTC. Besides comparing LABRADOR to SOTA fuzzers, we also compared it to SaTC, a static analysis tool. SaTC extracts common keywords from the front-end and the back-end, then locates their referenced points in the back-end binary to promote taint analysis. SaTC is labor-intensive, spending more than 40 human days on analyzing its results for randomly selected 8 devices in Table 6. At last, 7 vulnerabilities were confirmed on 4 devices, whereas LABRADOR has successfully discovered 67 vulnerabilities from all 8 devices and only spent 24 hours.

We analyzed the leading root causes. Firstly, SaTC misses many potential source points because some shared keywords are not formatted identically. In comparison, LABRADOR uses a similarity-based matching method to identify more keywords and comprehensively utilizes shared information between the front-end and the back-end. Secondly, SaTC lacks implicit control flow information due to the open problem of static taint analysis, which leads to many false positives of vulnerable path exposure.

TABLE 5: Compared to FIRM-AFL in vulnerability discovery.

Vendor	Device	FIRM-AFL	LABRADOR
Netgear	WNDR3700v2	0	3
Trendnet	TEW-652BRP	0	4
Linksys	WRT54GL	0	3
	E1000	0	3
TPLink	C50	0	1
	C20	0	1
	C7	0	1
Total	-	0	16

TABLE 6: Compared to SaTC in vulnerability discovery.

Vendor	Device	SaTC	LABRADOR
Cisco	RV042	0	17
Motorola	CX2L	3	7
TrendNet	TEW-811DRU	1	5
Netgear	WNDR3700v2	0	3
	SRX5308	0	19
	R7000	2	3
Ubiquiti	EdgeRouterX	1	10
Linksys	WRT54GL	0	3
Total	-	7	67

TABLE 7: The effect of directed fuzzing on the specific type of vulnerability. #NoIODA=LABRADOR-NoIODA, #NoRTI=LABRADOR-NoRTI, #NoRTI-NoIODA=LABRADOR-NoRTI-NoIODA,#Type.VUL =the type of vulnerability

Device	**#Type.VUL**	**Tool**	**Runs**	**μ TTT**	**Factor**	**$\hat{A}$ 12**
Netgear-SRX5308	Stored-XSS	LABRADOR	5	18min	3.83	0.93
		#NoIODA	5	202min	0.34	0.18
		#NoRTI	5	34min	2.03	0.76
		#NoRTI-NoIODA	5	69min	-	-
	Corruption	Labrador	5	103min	-	-
		#NoIODA	5	None	None	None
		#NoRTI	5	329min	None	None
		#NoRTI-NoIODA	5	None	-	-
Cisco-RV042	CI	Labrador	5	672min	1.67	0.97
		#NoIODA	5	None	None	None
		#NoRTI	5	783min	1.44	0.81
		#NoRTI-NoIODA	5	1125min	-	-
	Stored-XSS	Labrador	5	39min	2.44	0.96
		#NoIODA	5	127min	0.75	0.32
		#NoRTI	5	102min	0.93	0.29
		#NoRTI-NoIODA	5	95min	-	-
EdgeRouter-X	CI	Labrador	5	351min	1.17	0.68
		#NoIODA	5	374min	1.1	0.59
		#NoRTI	5	393min	1.05	0.5
		#NoRTI-NoIODA	5	411min	-	-
	Corruption	Labrador	5	209min	2	0.67
		#NoIODA	5	396min	1.05	0.55
		#NoRTI	5	426min	0.98	0.48
		#NoRTI-NoIODA	5	417min	-	-
Motorola-CX2L	CI	Labrador	5	567min	1.53	0.89
		#NoIODA	5	653min	1.33	0.76
		#NoRTI	5	572min	1.52	0.88
		#NoRTI-NoIODA	5	868min	-	-

TABLE 8: Vulnerabilities discovered by LABRADOR. #Type.VUL =the type of vulnerability, #VUL = the total number of vulnerabilities discovered by LABRADOR, #CVE = the total number of CVE ID LABRADOR obtained, #ID = detailed CVE ID LABRADOR obtained

Vendor	Device	#Type.VUL	#VUL	#CVE	#ID
Cisco	RV-042	CI	1	15	CVE-2023-20137 ∼CVE-2023-20151
		Stored-XSS	14		
		Reflected-XSS	2		
Netgear	SRX5308	Corruption	1	17	CVE-2023-2380 ∼CVE-2023-2396
		Stored-XSS	16		
		Reflected-XSS	2		
	FVS318G	Stored-XSS	1	-	-
		Corrupton	1		
	R7000	Corruption	2	-	-
		Reflected-XSS	1		
	WNDR3700v2	CI	1	3	CVE-2023-0848, ∼ CVE-2023-0850
		Corruption	2		
TrendNet	TEW-811DRU	CI	1	5	CVE-2023-0638, CVE-2023-0612, CVE-2023-0613, CVE-2023-0637, CVE-2023-0617
		Corruption	4		
	TEW-652BRP	CI	2	4	CVE-2023-0640, CVE-2023-0611, CVE-2023-0618, CVE-2023-0639
		Corruption	1		
		Reflected-XSS	1		
Linksys	WRT54GL	Corruption	3	-	-
	E1000	Corruption	3	-	
TPLink	Archer C50	Corruption	1	3	CVE-2023-0936
	Archer C20	Corruption	1		CVE-2023-30383
	ARcher C7	Corruption	1		CVE-2023-2646
Ubiquiti	EdgeRouterX	CI	9	10	CVE-2023-1456, ∼CVE-2023-1458, CVE-2023-2373 ∼CVE-2023-2379
		Corruption	1		
Motorola	CX2L	CI	6	4	CVE-2023-31528 ∼CVE-2023-31531
		Corruption	1		
Total	-	-	79	61	-

5.5. Vulnerability Discovery

As shown in Table 8, LABRADOR has successfully discovered 79 previously unknown vulnerabilities on 14 IoT devices, all of which have been reported to their respective device vendors. The vendors have confirmed all vulnerabilities, and 61 have been assigned CVE numbers so far.

Those vulnerabilities discovered by LABRADOR cover several types, including command injection (CI), memory corruption, and cross-site scripting (XSS). Especially, CI is one type of logical style vulnerability exposing colossal risk, which may introduce remote arbitrary code execution attack by the accessible exploit mode [41]. In total, LABRADOR has discovered 20 CI vulnerabilities on 6 IoT devices of Cisco, Netgear, TrendNet, Ubiquiti, and Motorola, 6 of which were classified as critical level severity by CVSS[4].

IoT devices are usually designed with weak defense mechanisms on stack and heap overflow exploitation to keep fast speed performance, letting memory corruption becomes one of the most prevalent types of vulnerability in IoT devices. In addition, even the dos attack introduced by memory corruption harms IoT devices server, such as CVE-2023-0618, classified as high level severity by CVSS due to its profound effect on the web service in TEW-652BRP of TrendNet. In total, LABRADOR has discovered 22 memory corruption vulnerabilities on 13 IoT devices, which shows memory corruption is also the most common vulnerability in IoT devices as the software on personal computers.

XSS vulnerability allows attackers to inject malicious code into HTML executed by the user's browser and commonly exists in the IoT device web [23]. In total, LABRADOR discovered 37 XSS vulnerabilities on 5 IoT devices and obtained 32 CVEs, which enable attackers to steal sensitive information (e.g., cookie), manipulate user sessions (e.g., hijack), or spread malware to other users (e.g., worm). More specially, reflected XSS and stored XSS can be detected by LABRADOR, where the former type is exploited through the malicious code reflected to the user via a submit or in a URL (e.g., CVE-2023-0639). The latter type is exploited through the malicious code stored on the server and executed when a user revisits the affected page (e.g., CVE-2023-20137).

Overall, the statistics indicate that out of the 61 vulnerabilities confirmed by the CVE organization, 58.3% were categorized as medium level severity, 31.7% as high level, and 10.0% as critical level.

5.6. Overhead Analysis

Speed is critical for the fuzzer to determine whether it can effectively perform the test. We conducted experimental statistics separately for the static analysis and fuzzing phase to present the detailed performance overhead introduced by different stages.

Static Analysis Phase. Table 9 shows that different devices have varying numbers of back-end binaries and sizes, listed in the "#Num.bin" and "Size(KB)" columns. Each device has about seven back-end binaries with an average size of 4.8 MB. Our methodology identifies three key areas overhead concentrated – explicit string extraction, graph construction, and distance measurement. Time consumed by these three areas averaged 4.9, 32, and 96.4 minutes. Due to the independence between the static analysis stage and the fuzzing stage, the statistics indicate that the overhead introduced by static analysis is acceptable for LABRADOR's practicability, and the static analysis provides abundant valuable information for fuzzing.

Fuzzing Phase. We recorded the overhead for various stages of the fuzz campaign, including tree-based mutation, trace inference, distance measurement, and network interaction. As shown in Table 10, the network interaction is the heaviest stage consuming 349 ms on average, which refers to the duration between when the fuzzer sends a request and when it receives a response from the device. The statistic demonstrates significant differences in interaction time across devices, ranging from 100 ms to 700 ms. Relatively, all other stages introduce much less overhead, where the average overhead for tree-based mutation, trace inference, and distance measure were 21.7 ms, 42.8 ms, and 0.4 ms, respectively.

Similarly, other black-box fuzzers also suffer from heavy overhead from network interaction. Therefore, LABRADOR keeps the performance advantage in native execution.

4. e.g., https://nvd.nist.gov/vuln/detail/CVE-2023-0640

TABLE 9: Detailed overhead of static analysis. #Num.bin= the number of binaries, #Str.Time= the overhead of string extraction in binary, #Graph.Time=the overhead of control flow graph construction, #Dis.time=the overhead of distance measurement, #Total.time= the total overhead of static analysis phase.

Vendor	Device	Version	#Num.bin	Size(kB)	#Str.Time	#Graph.Time	#Dis.time	#Total.time
Cisco	RV-042	4.2.3.14	12	9284	19 min	81 min	253 min	353 min
Netgear	SRX5308	4.3.5-3	11	2650	4 min	18 min	55 min	77 min
	FVS318G	3.1.1-18	2	2704	2 min	42 min	113 min	157 min
	R7000	1.0.11.136	4	4347	5 min	69 min	186 min	260 min
	WNDR3700v2	1.0.1.14	3	3595	1 min	8 min	27 min	36 min
TrendNet	TEW-811DRU	1.0.10.0	10	4075	7 min	32 min	98 min	137 min
	TEW-652BRP	3.04B01	6	2243	1 min	9 min	31 min	41 min
Linksys	WRT54GL	4.30.18.006	4	976	1 min	4 min	7 min	12 min
	E1000	2.1.03.005	9	2967	2 min	5 min	9 min	16 min
TPLink	Archer C50	US_V2_160801	8	1639	1 min	6 min	13 min	20 min
	Archer C20	V1_150707	8	2560	1 min	6 min	15 min	22 min
	ARcher C7	US_V2_180114	8	16179	9 min	73 min	188 min	270 min
Ubiquiti	EdgeRouterX	2.0.9	9	14132	13 min	84 min	327 min	424 min
Motorola	CX2L	1.0.1	10	1372	3 min	11 min	28 min	42 min
Average	-	-	7.4	4908.8	4.9 min	32 min	96.4 min	133.4 min

TABLE 10: Detailed overhead of fuzzing in LABRADOR. #Mutation= the overhead of tree-based mutation, #Trace= the overhead of trace inference, #Dis=the overhead of distance measurement, #Inter=the overhead of Interaction, #Overall= the total overhead of test case from generation to execution.

Vendor	Device	#Mutation	#Trace	#Dis	#Inter	#Overall
Cisco	RV-042	16 ms	48 ms	0.4 ms	376.7 ms	441.1 ms
Netgear	SRX5308	29 ms	87 ms	0.5 ms	504.2 ms	620.7 ms
	FVS318G	17 ms	61 ms	0.4 ms	586.8 ms	665.2 ms
	R7000	26 ms	66 ms	0.4 ms	671.9 ms	764.3 ms
	WNDR3700v2	21 ms	23 ms	0.2 ms	113.4 ms	157.6 ms
TrendNet	TEW-811DRU	28 ms	35 ms	0.4 ms	353.6 ms	417 ms
	TEW-652BRP	26 ms	37 ms	0.3 ms	196.3 ms	259.6 ms
Linksys	WRT54GL	18 ms	42 ms	0.4 ms	111.5 ms	171.9 ms
	E1000	22 ms	46 ms	0.4 ms	396.3 ms	464.7 ms
TPLink	Archer C50	19 ms	30 ms	0.3 ms	188.7 ms	238 ms
	Archer C20	17 ms	31 ms	0.3 ms	169.6 ms	217.9 ms
	ARcher C7	23 ms	33 ms	0.4 ms	172.3 ms	228.7 ms
Ubiquiti	EdgeRouterX	26 ms	28 ms	0.3 ms	400.5 ms	454.8 ms
Motorola	CX2L	16 ms	32 ms	0.4 ms	648.1 ms	696.5 ms
Average	-	21.7 ms	42.8 ms	0.36 ms	349.3 ms	414.1 ms

6. Discussion

Here we discuss LABRADOR's limitations and how to address them in the future.

Accuracy of RTI. The probability feature of similarity analysis may introduce false positives and negatives to RTI. Although we consider multiple differences between strings, imprecision cannot be avoided because some strings may undergo profound word type conversion before being output to the response. In the future, we need to learn more string changes through data flow propagation analysis from the back-end to the front-end to improve the accuracy of RTI.

Imprecision caused by indirect call. Indirect call recovery is an open problem and is not the focus of the paper. So, the recovered SCFG is incomplete. In future, we will try to recognize indirect call to improve the precision.

Application scenario. We focus on the web interface of IoT devices in the paper, fuzzing web services through HTTP, HTTPS, and WEB-SOCKET protocols. Ideally, LABRADOR could apply to any application with natural language processing attributes (serving humans), which generally contains rich explicit strings in programs. However, many other potential attack faces exist, such as private services or mobile apps, which may bring new challenges to LABRADOR. First, LABRADOR requires the availability of back-end binaries. Second, some applications may obfuscate or encode programs to prevent binary reverse analysis. Once failing to perform static analysis, RTI will lose its advantage. Third, private protocols may contain various encryption mechanisms causing labor-intensive analysis. In future, we will extend LABRADOR to more application scenarios.

7. Related Work

7.1. IoT Devices Fuzzing

Black-box Fuzzing. Black-box is the dominant method for fuzzing IoT devices. IoTFuzzer [10] uses mobile apps to send malformed payloads to devices in a black-box manner, but its mutation enormously suffers limitations from the app-side sanitizer. DIANE [36] further leverages code snippets in apps after validation but before transformation to expand mutation space compared to IoTFuzzer. However, it tends to destroy data format, which will affect mutation efficiency.

SNIPUZZ [18] leverages response difference to infer the snippet message format to improve mutation efficiency. IoTScope [46] focused on potential hidden interfaces in the IoT web. SRFuzzer [51] utilized heuristic methods to enhance mutation policy to fuzz the IoT web. Overall, Black-box fuzzing is limited to exploring more code space due to lacking code coverage feedback. LABRADOR proposes RTI to fill this gap.

Gray-box Fuzzing. Code coverage is essential for the evolution capability of the gray-box fuzzer. Due to the closure of IoT devices, some SOTA works try to turn to emulation, hardware support, etc, to get code coverage.

FIRM-AFL [52] achieved a high-throughput CGF (Coverage-Guided Fuzz) based on QEMU [4] emulator, which combines performance in user mode and compatibility in system mode, limited to instrument one single binary. EM-Fuzz [19] and IoTHunter [47] both implanted real-time checkers in the emulation environment to assist the fuzzer. TriforceAFL [24] is a CGF for kernel through a full-system emulator, and some work [53] adapted it to IoT programs, which all suffer from serious execution overhead.

Emulation for IoT. Implementing emulation for IoT devices of diverse types is nontrivial. Firmadyne [8] set up an automated and scalable framework for Linux-based devices, using QEMU as a foundation. FrimAE [26] further tried to alleviate the challenge of high-level failures in Firmadyne by appropriate interventions. Avatar [48] combined real hardware and emulation to mitigate the high failure rate resulting from specific hardware.

However, the success rate for IoT emulation is still low. Some works [14, 32, 38] tried to employ binary rewriting to obtain the program's internal state. However, binary rewriting also faces unresolved problems like the recovery of control flow. μAFL [29] utilized ETM [2], a hardware debug interface in some ARM processors, providing real-time instruction and data tracing to reconstruct execution. However, such hardware-level debug support in IoT is impractical; many industry-level products are protected and remove debugging supports.

LABRADOR's RTI mechanism can infer the program's run-time state in a black-box manner, which avoids intruding in firmware and has weak limitations.

7.2. Static Taint Analysis

IoT devices are susceptible to taint-style vulnerabilities. Thus static taint analysis is another mainstream technical means. The SaTC [11] identifies sources in back-end programs by performing keyword matching between the front-end and back-end, which helps to cut the path of taint propagation effectively. KARONTE [35] concentrated on insecure interactions between multiple binaries, which is common in IoT web interfaces, by propagating taint information between them. Saluki [20] analyzed data dependence facts in binary with path-sensitive, context-sensitive recovery, which were used to express security properties. EmTaint [13] tried to mitigate false negatives introduced by the indirect call through alias analysis based on the symbolic expression. DTaint [12] tried to discover data dependency in complex binary using pointer aliasing, inter-procedural data flow, and the similarity of the data structure layout. The CRYPTOREX [50] tried to identify crypto misuse in IoT devices across different architectures, dynamically updating its API list during the taint analysis. Besides false positives, static analysis lacks PoC (Proof of Concept), suffering from heavy labor to validate the results.

7.3. Directed Fuzzing

MUTAFLOW [31] mutates the seed and observes impacts on data flow by instrumentation. Although observation may tell if mutation takes effect, it cannot guide the mutation to the directed points and depends on instrumentation. Taint inference for web interface [39] was designed and implemented under black-box, which infers taint propagation by observing the front-end. Similarly, it also lacks appropriate guidance for the fuzzer. KameleonFuzz [16] detects XSS in the web interface by taint inference, which guides mutation. However, it only analyzes the front-end files, which limits its ability to find multiple types of vulnerability.

Other SOTA directed gray-box fuzzers [6, 9, 15, 28, 54] rely heavily on compile-instrumentation to access run-time program internals, which is impractical in IoT scenarios. LABRADOR implemented a code coverage tracking and distance-guided mutation to perform efficiently directed fuzzing by keeping the advantage of the black-box manner.

8. Conclusions

This paper proposes a novel response guided directed fuzzing solution LABRADOR for black-box IoT devices. It designs a novel response-based execution trace inference (RTI) mechanism to collect code coverage by inferring which code blocks are executed via examining strings in the response. It further designs IO oriented distance measurement (IODM) to drive the fuzzer to perform an efficient distance-guided mutation. Through evaluation of multiple enterprise-level IoT devices, it significantly outperforms 4 SOTA tools (SNIPUZZ, BOOFUZZ, FIRM-AFL, and SaTC) on vulnerability discovery, and finds dozens of unknown vulnerabilities of command injection, memory corruption, and XSS.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful suggestions on our paper. This work is supported in part by the National Key Research and Development Program of China (2021YFB2701000) and the National Natural Science Foundation of China (61972224).

References

[1] Rasheed Ahmad and Izzat Alsmadi. Machine learning approaches to iot security: A systematic literature review. *Internet of Things*, 14:100365, 2021. 1

[2] ARM. Embedded trace macrocell architecture specification. Website. https://developer.arm.com/documentation/ihi0014/q/?lang=en. 1, 15

[3] Vinícius A Barros, Sérgio AB Junior, Sarita M Bruschi, Francisco J Monaco, and Júlio C Estrella. An iot multi-protocol strategy for the interoperability of distinct communication protocols applied to web of things. In *Proceedings of the 25th Brazillian Symposium on Multimedia and the Web*, 2019. 1

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46, 2005. 15

[5] Mohammad Beyrouti, Ahmed Lounis, Benjamin Lussier, Abdelmadjid Bouadallah, and Abed Ellatif Samhat. Vulnerability and threat assessment framework for internet of things systems. In *2023 6th Conference on Cloud and Internet of Things (CIoT)*, pages 62–69. IEEE, 2023. 1

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Computer and Communications Security*, CCS, pages 2329–2344. ACM, 2017. 3, 5, 10, 15

[7] Derek Bruening and Timothy Garnett. Building dynamic instrumentation tools with dynamorio. In *Proc. Int. Conf. IEEE/ACM Code Generation and Optimization (CGO), Shen Zhen, China*, 2013. 1

[8] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Network and Distributed System Security Symposium*, 2016. 1, 15

[9] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. CCS '18, page 2095–2108, 2018. 15

[10] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, Xiaofeng Wang, W. Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Network and Distributed System Security Symposium*, 2018. 14

[11] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *USENIX Security Symposium*, pages 303–319, 2021. 1, 2, 8, 15

[12] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: Detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441, 2018. 15

[13] Kai Cheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, and Limin Sun. Finding taint-style vulnerabilities in linux-based embedded firmware with sse-based alias analysis, 2021. 15

[14] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. Armore: Pushing love back into binaries. In *USENIX Security*, SEC. USENIX, 2023. 15

[15] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022. 15

[16] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, page 37–48. Association for Computing Machinery, 2014. 15

[17] MA Ezechina, KK Okwara, and CAU Ugboaja. The internet of things (iot): a scalable approach to connecting everything. *The International Journal of Engineering and Science*, 4(1):09–12, 2015. 1

[18] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, S. Wen, Dongxi Liu, S. Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021. 1, 2, 15

[19] Jian Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. Em-fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39:3420–3432, 2020. 15

[20] Ivan Gotovchits, R. V. Tonder, and David Brumley. Saluki: Finding taint-style vulnerabilities with static property checking. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018. 15

[21] Tianbo Gu, Allaukik Abhishek, Hao Fu, Huanle Zhang, Debraj Basu, and Prasant Mohapatra. Towards learning-automation iot attack detection through reinforcement learning. In *IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 88–97, 2020. 1

[22] Christophe Guillon. Program instrumentation with qemu. In *1st International QEMU Users' Forum*, volume 1, pages 15–18. Citeseer, 2011. 1, 5

[23] Shashank Gupta and Brij Bhooshan Gupta. Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8: 512–530, 2017. 13

[24] Jesse Hertz and Tim Newsham. Project triforce: Run afl on everything. *NCC Group, Tech. Rep.*, 2016. 15

[25] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022. 5

[26] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. *Annual Computer Security Applications Conference*, 2020. 15

[27] Andi Kleen and Beeman Strong. Intel processor trace on linux. *Tracing Summit*, 2015. 1

[28] Gwangmu Lee, Woo-Jae Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *USENIX Security Symposium*, 2021. 3, 5, 15

[29] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. μafl: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, 2022. 15

[30] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019. 4

[31] Björn Mathis, Vitalii Avdiienko, Ezekiel O. Soremekun, Marcel Böhme, and Andreas Zeller. Detecting information flow by mutating input data. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 263–273, 2017. 15

[32] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *USENIX Security Symposium*, 2021. 15

[33] Joshua Pereyda. Boofuzz documentation. Last updated 2022. https://github.com/jtpereyda/boofuzz. 2

[34] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education*, 2004. 1

[35] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, C. Kruegel, and G. Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *IEEE Symposium on Security and Privacy*, 2020. 1, 8, 15

[36] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, C. Kruegel, and G. Vigna. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In *IEEE Symposium on Security and Privacy*, 2021. 14

[37] Karen Rose, Scott Eldridge, and Lyman Chapin. The internet of things: An overview. *The internet society (ISOC)*, 80:1–50, 2015. 1

[38] Majid Salehi, Danny Hughes, and Bruno Crispo. μsbs: Static binary sanitization of bare-metal embedded devices for fault observability. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020. 15

[39] R Sekar. An efficient black-box technique for defeating web application attacks. In *Network and Distributed Systems Security*, 2009. 15

[40] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated iot firmware introspection and analysis. Association for Computing Machinery, 2019. 8

[41] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *Acm Sigplan Notices*, 41(1):372–382, 2006. 13

[42] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, Adam Doupé, Tiffany Bao, Yan Shoshitaishvili, and Ruoyu Wang. Greenhouse: Single-Service rehosting of Linux-Based firmware binaries in User-Space emulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023. 1

[43] Jinzhu Wang, Ruijie Cai, and Shengli Liu. Research on the protection mechanism of cisco ios exploit. In *Journal of Physics: Conference Series*, volume 1584, page 012045. IOP Publishing, 2020. 4

[44] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 2021. 1

[45] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 2021. 4

[46] Wei Xie, Jiongyi Chen, Zhenhua Wang, Chao Feng, Enze Wang, Yifei Gao, Baosheng Wang, and Kai Lu. Game of hide-and-seek: Exposing hidden interfaces in embedded web applications of iot devices. In *Proceedings of the ACM Web Conference*, 2022. 15

[47] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. Poster: Fuzzing iot firmware via multi-stage message generation. *ACM SIGSAC Conference on Computer and Communications Security*, 2019. 15

[48] Jonas Zaddach, L. Bruno, Aurélien Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium*, 2014. 15

[49] Michal Zalewski. AFL: American Fuzzy Loop. Technical report, Google, 2013. URL https://lcamtuf.coredump.cx/afl. 12

[50] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, J. Weng, and Kehuan Zhang. Cryptorex: Large-scale analysis of cryptographic misuse in iot devices. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019. 15

[51] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. Srfuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. In *the 35th Annual Computer Security Applications Conference*, 2019. 15

[52] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium*, pages 1099–1114, 2019. 1, 2, 4, 8, 15

[53] Yaowen Zheng, Zhanwei Song, Yuyan Sun, Kai Cheng, Hongsong Zhu, and Limin Sun. An efficient greybox fuzzing scheme for linux-based iot programs through

binary static analysis. In *International Performance Computing and Communications Conference*, 2019. 15

[54] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Conference on Security Symposium*, 2020. 5, 15

Appendix A. Communication Mechanism of IoT Web

Web architecture consists of two parts shown as Fig. 11: the front-end and back-end. The front-end is the web part that users can see and mainly includes requests and responses. The front-end usually interacts with the user through the browser. The request carries front-end parameters submitted to the back-end, while the response contains back-end data returned to the front-end. Meanwhile, the back-end interprets requests sent by the front-end and sends out responses to the front-end. The back-end usually comprises a server, applications, and a database, generally binaries in IoT for speed performance. The front-end and the back-end use HTTP, a text-based protocol, to communicate. When the back-end receives a request from the front-end, it is first parsed by a web server according to HTTP protocol, and parameters are extracted. The web server then transfers these parameters to relevant applications. Finally, web applications generate data returned to the web server, packaged into a response, and sent back to the front-end. The web server operates as a daemon process, while applications handle specific business functions. They use interprocess communication (IPC), such as common gateway interface(CGI) and PIPE, to communicate. Web applications typically need to query information like device status and configurations, often stored in a database such as NVRAM. (Description for Section 2.1.)

According to the HTTP protocol, all requests are independent, with no relationship or state transition. Thus HTTP is a stateless protocol that does not remember interaction history. However, many web services running on HTTP are stateful. For instance, most web implementations incorporate security mechanisms such as authentication, authorization, and audit (3A). Before using web services, users must complete login operations to verify their identity and authorization. Some webs also audit users' operation activities. The session is added to give HTTP state, which associates a series of independent requests, and state transitions are achieved by managing the session. The session mechanism usually uses a token, a series of parameters identifying the session. The session token is usually located in the cookie, header, and body and marks all the messages within the same session. (Description for Section 4.)

Appendix B. Meta-Review

B.1. Summary

This paper proposes a black-box fuzzer LABRADOR targeting IoT device firmware. LABRADOR exploits the connection between IoT device firmware and their companied web server implementations, and uses the distance to the key strings from the responses of IoT devices as the feedback to guide the fuzzing. The evaluation of several router-based firmware demonstrates the effectiveness and efficiency of LABRADOR.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Addresses a Long-Known Issue.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

1) A new feedback mechanism using distance to web server related key strings from the response.
2) New vulnerabilities found in router-based firmware.

B.4. Noteworthy Concerns

1) Incomplete and unfair comparison with FIRM-AFL. One reviewer and shepherd feel surprised that a black-box fuzzer could beat a gray-box fuzzer. They think the potential explanation of why FIRM-AFL could not find anything is incorrect. A correct setup and a fair comparison should be to check out the code coverage as well for both fuzzers since some firmware can be emulated.
2) Dependency on web server implementations within firmware. One reviewer and shepherd expect us to show a concrete study to support the argument of "the generality of web implementations in IoT devices", explicitly including the statistics that indicate the proportion of web implementation in IoT devices.
3) Routers are not typical IoT devices. One reviewer and shepherd think it does not make sense to say IoT in the title while the evaluation is only about routers.

Appendix C. Response to the Meta-Review

We sincerely appreciate the insightful feedback from the reviewers and express our gratitude for Shepherd's invaluable guidance. In response to the concerns in the meta-review, we aim to provide a clear explanation as follows:

1) Our work aims to enhance black-box fuzzing capabilities to match those of gray-box fuzzing in IoT

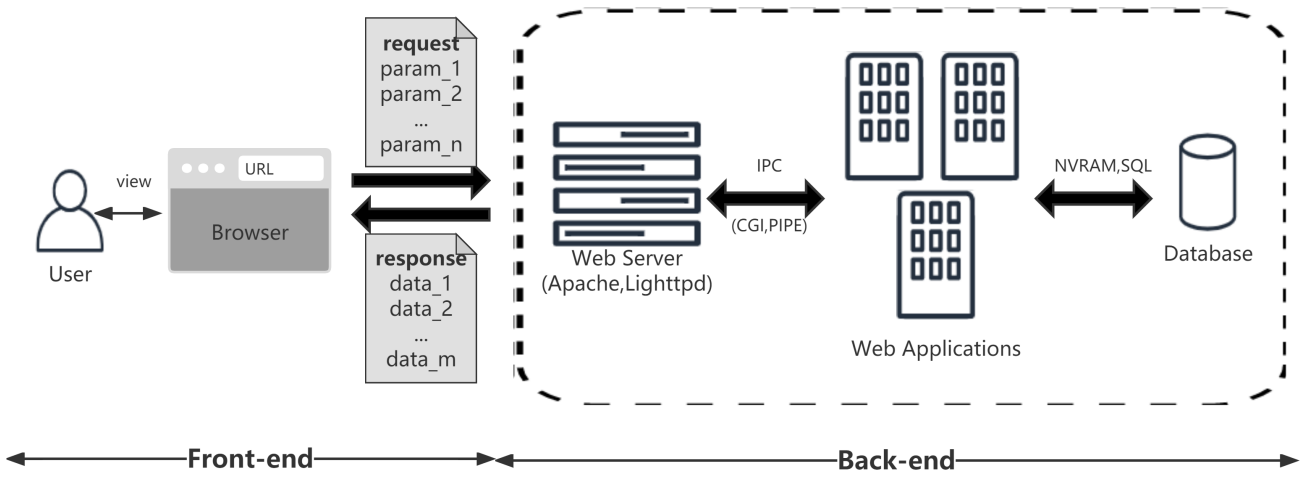

Figure 11: Communication mechanism in IoT web.

devices. The comparison between black-box and gray-box fuzzing is not to prove one is better than the other, but to reflect the practical testing abilities of current fuzzers for IoT devices. Pure gray-box firmware fuzzers still face challenges. For example, FIRM-AFL lacks guidance in its random mutation strategy, making it relatively inefficient. It can only extract code coverage for a single binary and has limited feedback due to the handling of requests by multiple processes, resulting in slow growth in code coverage.

2) We have made concerted efforts to obtain precise statistics about the prevalence of web implementation in IoT devices. Despite our thorough search, accurate statistics are currently lacking. However, drawing from existing research and our observations, we are confident in asserting that the web interface in IoT devices represents a focal point for SOTA works and has garnered significant attention due to its associated security concerns.
3) It is evident that our dataset does not encompass a wide range of IoT devices. This limitation is attributed to the rapidly evolving nature of IoT and its continuously evolving concept. Furthermore, the absence of clear and standardized datasets for evaluating IoT devices, all of which are subject to SOTA works, presents a significant challenge. In our evaluation, we specifically focused on networking-based devices (routers, firewalls, and VPNs) due to their heightened security implications and their prominence in SOTA works related to IoT devices such as FRIM-AFL, SaTC, FirmFuzz, etc.