iFIZZ: Deep-State and Efficient Fault-Scenario Generation to Test IoT Firmware

Peiyu Liu[†], Shouling Ji[†,‡,(✉),*], Xuhong Zhang[†,‡,§], Qinming Dai[†], Kangjie Lu[¶], Lirong Fu[†],
Wenzhi Chen[†,(✉)], Peng Cheng[†], Wenhai Wang[†], Raheem Beyah[‖]
[†]Zhejiang University, Hangzhou, China,
{liupeiyu,sji,fulirong007,chenwz,zdzzlab}@zju.edu.cn, {xuhongnever,saodiseng}@gmail.com
[‡]Binjiang Institute of Zhejiang University, Hangzhou, China
[§]Zhejiang University NGICS Platform, Hangzhou, China
[¶]University of Minnesota Twin Cities, Minneapolis, USA, kjlu@umn.edu
[‖]Georgia Institute of Technology, Atlanta, USA, rbeyah@ece.gatech.edu

***Abstract*—IoT devices are abnormally prone to diverse errors due to harsh environments and limited computational capabilities. As a result, correct error handling is critical in IoT. Implementing correct error handling is non-trivial, thus requiring extensive testing such as fuzzing. However, existing fuzzing cannot effectively test IoT error-handling code. First, errors typically represent corner cases, thus are hard to trigger. Second, testing error-handling code would frequently crash the execution, which prevents fuzzing from testing following deep error paths.**

In this paper, we propose iFIZZ, a new bug detection system specifically designed for testing error-handling code in Linux-based IoT firmware. iFIZZ first employs an automated binary-based approach to identify realistic runtime errors by analyzing errors and error conditions in closed-source IoT firmware. Then, iFIZZ employs state-aware and bounded error generation to reach deep error paths effectively. We implement and evaluate iFIZZ on 10 popular IoT firmware. The results show that iFIZZ can find many bugs hidden in deep error paths. Specifically, iFIZZ finds 109 critical bugs, 63 of which are even in widely used IoT libraries. iFIZZ also features high code coverage and efficiency, and covers 67.3% more error paths than normal execution. Meanwhile, the depth of error handling covered by iFIZZ is 7.3 times deeper than that covered by the state-of-the-art method. Furthermore, iFIZZ has been practically adopted and deployed in a worldwide leading IoT company. We will open-source iFIZZ to facilitate further research in this area.

I. Introduction

Widely adopted IoT devices that interact with physical environments are safety-critical, making it a high priority to maintain high reliability for practical deployment. However, previous works already show that IoT devices are abnormally prone to diverse errors due to constraints such as complex hardware dependence, limited hardware and system resources, and disruptive environmental conditions [1]–[7]. Once an error is not handled appropriately, it may cause a device to become unresponsive or enter an incorrect state and finally lead to severe consequences, such as leaving a valve open, flooding a factory, or leaving a window unlocked. Thus, correct error-handling code – which is intended to deal with erroneous situations where security or reliability issues may potentially occur – is important in IoT firmware.

[*] Shouling Ji and Wenzhi Chen are co-corresponding authors.

TABLE I
Comparison of state-of-the-art fuzzing systems.

System	Tailored for IoT	Fault Injection	Input-independent Error Identification	Deep Error Path Test
AVATAR [9]	●	○	○	○
P^2IM [10]	●	○	○	○
HALucinator [11]	●	○	○	○
IoTFuzzer [12]	●	○	○	○
FIRM-AFL [13]	●	○	○	○
LFI [14]	○	●	◐	○
EH-Test [15]	○	●	◐	○
FIFUZZ [8]	○	●	◐	●
iFIZZ	●	●	●	●

● = well-supported; ◐ = partially-supported; ○ = unsupported.

Unfortunately, the error-handling code itself tends to be error-prone because it is hard to be tested. Therefore, bugs are quite common in error-handling code [8]. Additionally, although error-handling code is infrequently triggered in normal executions, error-handling bugs can lead to severe problems, e.g., device crashes or data loss. Besides, bugs in error-handling code may exist for a long time because it is more difficult to detect them. Thus, it is critical to comprehensively and effectively test the error-handling code of IoT firmware to detect hidden bugs. Although dynamic detection methods, such as fuzzing, have been shown to be promising in finding bugs in IoT devices, they still suffer from an important limitation with effectively testing error-handling code in IoT, especially the ones triggered by input-independent errors, such as hardware failures and memory allocation failures.

To cover more error paths, researchers have adopted fault (or error) injection [8], [14]–[22] which intentionally and deterministically generates runtime errors to force the execution of error paths. However, designing an effective fault-injection solution to test the error-handling code in IoT firmware faces several key challenges. (1) *Automatically Identifying input-independent errors.* First, due to complex hardware dependence and execution environments, a large number of diverse input-independent errors occur in different IoT devices. It is impractical to identify all these errors manually. Second, the source code of IoT firmware is often not available to third-party researchers, making the identification of potential errors even harder. (2) *Testing deep error paths.* The execution path of a tested IoT firmware may contain various error sites, and error-handling code commonly terminates the execution; if an early error stops the execution, the testing will not reach deep error paths. As shown in Table I, existing tools cannot simultaneously solve all the above challenges. For example, FIFUZZ [8] can find deep bugs hidden in error-handling code. However, it cannot identify and produce errors without the source code of the tested program. Thus, it is not suitable for testing IoT firmware.

In this paper, we propose iFIZZ, a new fuzzing framework, to efficiently test deep error-handling code in Linux-based IoT firmware. The core of iFIZZ is a fault-scenario generation system, which overcomes the aforementioned challenges by the following new techniques.

Automated identification of input-independent errors. Before generating errors in a tested IoT firmware, we have to identify the errors caused by input-independent events (such as hardware failures and insufficient memory) in this firmware as fault-injection targets. Previous works, such as [23], usually identify error-functions by a simple heuristic that finds nullptrs or negative values. However, we must identify the self-defined error code in different closed-source firmware. To this end, we propose an *automated binary-based* approach to statically identify the functions that may have input-independent errors. This approach is based on two observations of errors and error-handling code in IoT firmware. (1) *Error code as the return value.* Following the programming convention, an erroneous function would return error code (e.g., -1) to represent potential errors, which are to be further checked (e.g., line 3 in Listing 1) in callers. (2) *Input-independent error conditions.* Runtime errors in IoT firmware can be triggered by various input-independent conditions (described in §II-A). With the observations, our technique (1) infers error code by examining whether a value is often checked when used as a return value in disassembled code, (2) leverages error code to infer errors, and (3) analyzes the error conditions to infer if the error is input-independent. In this way, our technique can automatically identify input-independent errors in closed-source IoT firmware.

```
1 FILE *open_memstream( ... ) {
2     register __oms_cookie *cookie;
3     if ((cookie->buf = malloc(...)) == NULL) {
4         goto EXIT_cookie;
5     }
6 EXIT_cookie:
7     free(cookie);
8     return NULL;
```

Listing 1. An example of error check and error-handling code.

Testing of deep error paths. Intuitively, knowing which functions may return errors, the following step is to generate errors at runtime. However, this strategy is hard to reach deep error paths—if we always generate an error early, the execution will crash before reaching deep error paths. Thus, to reach deep error paths, we propose a *state-aware and bounded error generation* method to generate fault scenarios that can guide the fuzzing to test deep error paths effectively. (1) *State-aware error producing*. We observe that if an error at a specific call stack leads to a crash, the error at the same call stack may trigger the same (redundant) crash in other fault scenarios. Based on this observation, we propose to reduce redundant fault scenarios by leveraging the state (defined as the runtime context of an error site, i.e., its call stack and the sequence of the previous injected errors in this fault scenario) of error sites. In particular, we first examine the state of an error site to determine whether an error should be produced. If the historical tests indicate that an error site has led to a crash in a specific state, we do not re-produce errors on this error site in the same state in subsequent tests. (2) *Bounded faults*. Although the state-aware technique can mitigate early crashes, we still face the problem of exponential fault-scenario explosion when producing multiple errors. For example, if there are N error sites in a runtime trace, we can generate 2^N - 1 fault scenarios. It would take unaffordable time to test all the scenarios. On the other hand, our empirical study reveals that most crashes are triggered by *only* a small number of errors in a fault scenario. Thus, we propose a bounded faults approach to seek a proper number of faults to reduce redundant fault scenarios while covering deep error paths.

We have implemented a full-featured prototype of iFIZZ and deployed it on 10 Linux-based IoT devices. iFIZZ successfully discovers 109 serious bugs. We also compare iFIZZ to existing fuzzing tools and find that iFIZZ discovers many bugs that are otherwise missed by existing tools. Furthermore, we conduct evaluations to measure the error-handling code coverage. The results show that in 24 hours, iFIZZ can cover 67.3% more error-handling code. Meanwhile, the depth of error-handling code covered by iFIZZ is 7.3 times deeper than that covered by existing software fault injection (SFI) methods on average. By collaborating with a worldwide leading IoT company, we have deployed iFIZZ in practical adoption.

Overall, we make the following technical contributions:

- *New open-source framework.* We propose iFIZZ, a new framework specifically designed for testing IoT error-handling code by generating fault scenarios. We will open-source iFIZZ to facilitate further research in this area [24].
- *New techniques.* We propose multiple new techniques in iFIZZ. (1) Our automated binary-based approach can identify potential errors in closed-source IoT firmware. (2) The state-aware and bounded fault-scenario generation approach effectively reaches deep error paths.
- *New findings.* We evaluate iFIZZ on 10 widely-used Linux-based IoT firmware images from leading vendors. It in total finds 109 bugs. These bugs can lead to critical security issues such as DoS and memory leakage.
- *Practical adoption.* iFIZZ has been adopted in a world-wide leading IoT company to analyze a large scale of commodity IoT devices. Extensive experiments on the real-world platform show that iFIZZ can efficiently find bugs in commodity IoT devices.

II. Problem Statement and Motivation

A. *Errors in IoT Firmware*

Various reasons can trigger errors in IoT devices. Typically, errors in IoT devices can be divided into two categories according to the source of errors. (1) Input-dependent errors are caused by invalid inputs given in the command line, such as damaged files and invalid parameters. For example, as shown in Listing 2, an input-dependent error (line 3) may be triggered by an invalid input file. Compared to traditional PC programs, IoT programs may have fewer input-dependent errors since most IoT is not designed to process standard inputs. (2) By contrast, input-independent errors caused by occasional runtime events, such as exhausted memory, hardware failure, and network unreachability, might be more common on IoT devices than traditional PCs due to the limited resources and complex hardware dependency of IoT devices. For example, as shown in Listing 2, an input-independent error (line 6) may occur due to the lack of memory. If standard inputs trigger an error, existing fuzzing can already cover it by mutating inputs. Therefore, in IFIZZ, we focus on input-independent errors.

```
1 int main(int argc, const char *argv[]){
2     int *a;
3     FILE *f = fopen (argv[1], "r");
4     if(check_header(f) < 0) return -1;
5     ...
6     if(a = malloc(10) == NULL) goto ERR;
```

Listing 2. Examples of input-dependent and input-independent errors.

However, triggering input-independent errors is much harder than triggering input-dependent errors. Input-independent errors occur only when occasional events happen, such as memory exhaustion and physical hardware damage, rare during normal execution. Moreover, physically producing these occasional events is also inefficient. Thus, we use software fault-scenario generation to produce input-independent errors in this paper.

B. *Impact of Error-handling Bugs in IoT*

Bugs in error-handling code can cause critical issues because the intended protection is void. Taking Listing 3 as an example, the error-handling code (lines 3 - 7) in the firmware of a smart lock is used to handle a hardware failure. The error-handling code will check if there is any motion in front of the door. If so, it will take a photo and send it to the user (to help the user determine whether an attacker is prying the lock). Then, it will send a message to notify the user the lock is broken. However, pointer *m* at line 4 might be NULL when the previous function `get_motion()` fails. In this case, a null-pointer dereference bug will lead to a crash. Subsequently, the user cannot get any notification about the broken lock.

```
1 int main(){
2     if (check_hardware_failure()) {
3         Motion *m = get_motion()
4         if(m->state){
5             take_and_send_photo();
6         }
7         send_msg("The door is broken!", email);
8     }
9 }
```

Listing 3. A bug in the error-handling code of a smart lock.

TABLE II
Study result of IoT firmware patches.

Program	OpenWRT		DD-WRT	
	Patches	error-handling	Patches	error-handling
busybox	43	8(18.6%)	148	38(25.7%)
dnsmasq	66	27(40.9%)	81	29(35.8%)
dropbear	27	5(18.5%)	68	23(33.8%)
iptables	35	8(22.9%)	52	16(30.8%)
Total	171	48(28.1%)	349	106(30.4%)

```
char *c = NULL;
char d[10];
switch(error){
    case '1':
        free(a); break;
    case '2':
        free(b); break;
    case '3':
        //bug1: null-pointer dereference
        memcpy(c,"error",6); break;
    case '4':
        puts(d[12]); break; //bug2: buffer-over-flow
}
free(a); //bug3: double free
b->func(); //bug4: use-after-free
```

Listing 4. Examples of bugs in error-handling code.

Worse, in addition to voiding the intended protection, bugs in error-handling code can lead to various impacts. For example, as shown in Listing 4, common impacts of bugs in error-handling code include DoS (bug1), out-of-bounds memory access (bug2), information leakage (bug3), arbitrary code execution (bug4), etc. Therefore, it is important to detect and patch bugs in error-handling code.

C. *Error-handling Code in IoT Firmware*

To understand the reliability of error-handling code in IoT firmware, we perform a preliminary study to identify IoT firmware patches that add or modify error-handling code. We manually look into patches from 4 open-source IoT programs, namely `busybox`, `dnsmasq`, `dropbear`, and `iptables`, in `OpenWRT` [25] and `DD-WRT` [26]. As shown in Table II, we analyze 520 patches published between 2013 and 2020. Finally, we find that more than 28% of patches fix bugs in the error-handling code, confirming that error handling is buggy. Further analysis shows that common bugs in error-handling code include null pointer dereference, memory overflow, etc., leading to severe problems such as system crashes and information leakage.

Further, we believe that the patched bug is just the tip of the iceberg, and there are many more hidden bugs in IoT firmware for the following reasons. First, IoT firmware needs to handle a large number of nested runtime errors that may occur at runtime due to the complex hardware dependency and limited resources. Thus, it is challenging to implement correct error-handling code, i.e., developers may make mistakes when handling complex nested errors. Second, it is difficult to find the bugs hidden in error-handling code of IoT firmware since such code is hard to test by nature. For example, the IoT firmware is often closed-source, making it impractical to manually identify the runtime errors or produce errors through

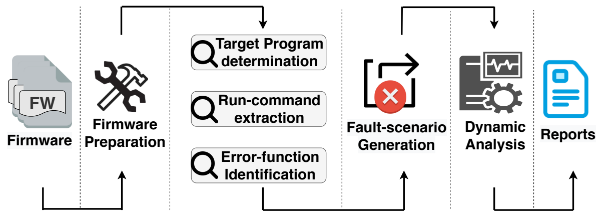

FW
Firmware
Firmware Preparation
Target Program determination
Run-command extraction
Error-function Identification
Fault-scenario Generation
Dynamic Analysis
Reports

Fig. 1. The workflow of our framework.

compile-time instrumentation. Besides, facing a large number of runtime errors in IoT firmware, covering a certain error path is hard due to the complex error context. For example, the crash caused by an early error may prevent the testing of subsequent error paths. Therefore, considering that the error-handling code in IoT firmware is critical yet buggy, and to the best of our knowledge, there still exist no practical approaches for IoT error-handling code analysis, we believe that testing error-handling code in IoT firmware is important and necessary for securing IoT applications.

III. Design of iFIZZ

A. The Framework

We develop iFIZZ as an easy-to-use system, whose workflow is shown in Figure 1. At a high level, there are 4 phases in iFIZZ. 1) *Firmware preparation*. iFIZZ conducts several preparations in this phase. For example, we enable the debug interfaces and facilities of the tested firmware. (detailed in §IV). 2) *Error-function identification*. iFIZZ automatically identifies the error-functions that can result in runtime errors by analyzing their possible return values and the corresponding conditions for triggering the error return value. 3) *Fault-scenario generation*. iFIZZ generates useful test cases by utilizing our state-aware and bounded fault-scenario generation approach. 4) *Dynamic analysis*. iFIZZ produces errors according to our fault-scenarios and executes the tested code.

B. Preliminary Definitions

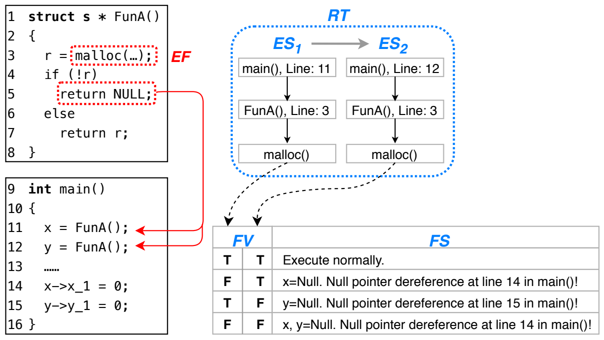
Definition 1: Error-function (EF). EF is a library (either standard or customized) function which can result in an input-independent error in IoT firmware.

Definition 2: Error Stack (ES). ES = $(Fun_1,\ Loc_1) \rightarrow (Fun_2,\ Loc_2) \rightarrow, ..., \rightarrow (EF,\ Loc_{EF})$ includes a sequence of function calls at the call site of an error-function EF (in the order from caller to callee), including the locations of function calls and called functions.

Definition 3: Runtime Trace (RT). $RT = ES_1 \rightarrow ES_2 \rightarrow, ..., \rightarrow ES_n$ is a sequence of ESs. A RT records all ESs during a software life cycle.

Definition 4: Fault Value (FV). FV = V_1, V_2, ..., V_n is a sequence of boolean variables. Each boolean variable V (T or F) is used to indicate generating an error or not.

Definition 5: Fault Scenario (FS). FS = $< RT,\ FV >$ is a pair of RT and FV. A fault scenario is used to guide an instance of runtime fault injection.



```
1  struct s * FunA()
2  {
3    r = malloc(…);
4    if (!r)
5      return NULL;
6    else
7      return r;
8  }
```

```
9  int main()
10 {
11   x = FunA();
12   y = FunA();
13   ……
14   x->x_1 = 0;
15   y->y_1 = 0;
16 }
```

FV		FS
T	T	Execute normally.
F	T	x=Null. Null pointer dereference at line 14 in main()!
T	F	y=Null. Null pointer dereference at line 15 in main()!
F	F	x, y=Null. Null pointer dereference at line 14 in main()!

Fig. 2. Examples Fault scenarios.

Figure 2 shows a simple example of these definitions. In the function `main`, the objects x and y are allocated by calling function `FunA`. In `FunA`, the object r is allocated by calling `malloc` and is returned to `main`. The `malloc` in `FunA` is an EF. There are two RTs of `malloc`, and four FSs can be generated to trigger null pointer dereferences of the objects x and y at runtime.

C. Automated Binary-based Runtime Error Identification

To dynamically test the error-handling code in IoT firmware, the first problem we need to solve is to identify functions that may introduce runtime errors. Due to the complex hardware dependency and execution environments, various runtime errors may occur in different IoT devices. Thus, it is impractical to identify runtime errors in each tested firmware manually. Meanwhile, since most IoT firmware is closed-source, identifying runtime errors by source code analysis is also impractical. Therefore, to effectively identify runtime errors in IoT firmware, we aim to identify EFs automatically. To this end, we first conduct an empirical analysis on `uClibc` (an open-source C library widely used in Linux-based IoT devices [27]). Specifically, we manually identify 20 EFs in `uClibc` and then analyze these functions and their uses. From our analysis, we observe two characteristics of runtime errors in IoT firmware.

1) *Error code as the return value*. Following programming conventions, an EF often returns error code to its caller to represent the occurrence of errors. Meanwhile, the caller often checks the return value to find out whether an error occurs. For example, function `malloc` returns NULL to represent an error. Then, its caller, function `open_memstream` in Listing 1, checks this return value.
2) *Input-independent error conditions*. For an input-independent error, the error condition is an occasional runtime event, such as lack of memory or hardware failure. Such error conditions are input-independent. For example, in function `open_memstream` of Listing 1, the error condition at line 3 is used to check the lack of memory but not standard inputs.

Based on our observations, we propose an automated binary-based runtime error identification approach to identify EFs effectively. First, we leverage error code to infer errors. Then, we identify input-independent errors by analyzing error conditions. We show the approach of error-function identification in Algorithm 1. For a function f, iFIZZ first scans its assembly code to collect all its return values (line 2). For each return value, iFIZZ examines whether or not there is a caller of f who checks it (line 6), and if so, infers that the return value is an error code (ec) (line 7). From each ec, iFIZZ searches backward in f to find its check condition (cc) (i.e., ec is control-dependent on cc) (line 8). Then, iFIZZ performs a backward inter-procedural dataflow analysis to collect the sources on which cc is flow-dependent (line 9). Finally, iFIZZ examines whether or not there is a source of cc that is not a program parameter (line 10). If so, iFIZZ infers that *"f returns ec when an input-independent error occurs"*, and thus, f is an EF (line 11).

Algorithm 1: Error-function Identification

```
Input: $F$: set of functions in IoT firmware.
Output: $EF$: set of functions that can result in
        input-independent errors in IoT firmware.
1  foreach $f$ in $F$ do
2      $R$ = GetReturnValues($f$);
3      $C$ = GetCallers($f$);
4      foreach $c$ in $C$ do
5          foreach $r$ in $R$ do
6              if IsChecked(r, c) == True then
7                  $ec = r$;
8                  $cc$ = GetCondition($ec$, $f$);
9                  $s$ = GetSource($cc$);
10                 if IsInputDenp(s) == False then
11                     Append($f$, $EF$);
12                     break;
13         if $f$ in $EF$ then
14             break;
```

Performing a completely accurate data flow analysis will encounter two common problems: indirect call and data flow explosion. Fortunately, iFIZZ does not rely on completely accurate data flow analysis since it aims to find an input-independent source instead of finding all sources. First, even if indirect calls influence some data flows, we can still complete the analysis through other data flows. Thus, in iFIZZ, indirect calls are ignored at present. This is an inherent limitation of this technique. However, the evaluation in §V-B indicates that iFIZZ can still dramatically reduce the manual work of identifying realistic EFs. Second, once a data flow proves that a source is input-independent, the analysis completes instead of constantly analyzing other sources.

D. State-aware and Bounded Fault-scenario Generation

The fault-scenario generation aims to 1) cover as many error paths as possible and 2) reach deep error paths within a limited time. We first conduct a study to reveal obstacles in achieving these goals. Specifically, we implement a simple prototype that injects error randomly on every error site and evaluate it on a randomly select firmware. Then, we analyze the experimental results and summarize two main obstacles in generating efficient fault-scenario.

1) *Early crashes.* Error testing would frequently crash the tested program, which prevents us from testing the following deeper error paths. For example, suppose that $RT = ES_1 \rightarrow ES_2 \rightarrow ES_3$ and the tested program always crashes when an error occurs on ES_1. In this case, the FSs that produce the error on ES_1 are profitless and redundant, since they always lead to the same crash and never test the error-handling code of ES_2 and ES_3.
2) *FS explosion.* A naive approach to generate FSs is to traverse all the combinations of EFs. If there are n EFs along RT, the number of generated FSs can be 2^n - 1. Obviously, generating all FSs is infeasible if the tested program contains a large number of EFs. It can take unaffordable time if we apply all these FSs.

To overcome these obstacles, we propose a state-aware and bounded approach to effectively generate the fault scenarios covering different error paths in modest time and reduce redundancy.

Algorithm 2: State-aware and Bounded Error Producing

```
Input: $L$: crash log. $FS$: current fault scenario. $ME$: bound
       of the number of errors. $MBE$: bound of the distance
       between the first and the last error.
1  $n$ = GetErrNum($FS$);
2  if $n < ME$ then
3      $d$ = GetErrDist($FS$);
4      if $d < MBE$ then
5          foreach $e$ in $FS$ do
6              if SiteInLog(e, L) == True then
7                  $s$ = GetState($e$);
8                  if StateInLog(s, L) == True then
9                      NotInjectErr();
10                     break;
11             InjectErr();
```

State-aware error producing. We define *the state of an error site as (1) its ES and (2) the sequence of the previous injected errors in this FS.* Suppose an error site in a certain state leads to a crash. In that case, we avoid producing an error on the same error site with the same state in subsequent tests since such tests are redundant and prevent testing deep error paths. Specifically, we record the state of the latest error site in a crash log when a crash occurs. Meanwhile, we record the state of the current error site dynamically. Thus, as shown in Algorithm 2, before we produce a runtime error on an error site, we first check whether this error site is in the crash logs (line 6); if so, we further check whether the state of the current error site is the same as the one in crash logs (line 8). When the current error site and its state appear in the crash logs, we skip the error producing on this error site (line 9). In this way, we prevent redundant crashes but still cover subsequent error-handling code.

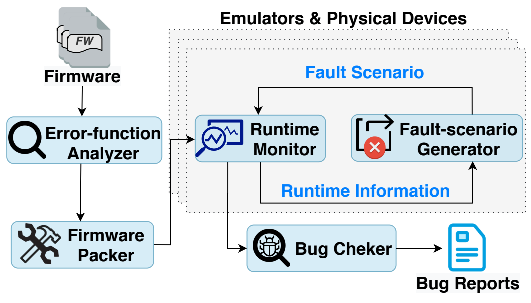


Fig. 3. Overall architecture of iFIZZ.

Bounded faults. Our state-aware approach supports us in covering deep error paths by mitigating early crashes. However, we still face FS explosion when producing a large number of errors. Such a problem exists in all of the tested IoT firmware, making it impractical to test them effectively. To solve this problem, we propose a bounded approach to produce a suitable number of errors in a fault scenario. The design is based on two observations obtained from the evaluation of the simple prototype. First, most crashes are caused by only a small number of errors, and generating fault scenarios with a large number of errors is often unnecessary. Thus, we propose the **first rule**: the maximum number of errors (ME) in a fault scenario should be bounded (line 2 in Algorithm 2). Second, we also observe that most crashes are caused by neighboring errors. Hence, we propose the **second rule**: the maximum distance (i.e., the number of error sites) between the first and the last error (MBE) in a fault scenario should also be bounded (line 4 in Algorithm 2).

IV. Implementation

We implement a full-featured prototype of iFIZZ. Figure 3 shows its architecture, consisting of the following five parts.

Error-function analyzer. It unpacks firmware images and analyzes their assembly-code by leveraging our automated binary-based approach (§III-C) to identify EFs. We develop a customized unpacker based on `FIRMADYNE` [28] to unpack a firmware image. Additionally, we implement an IDA script [29] to perform assembly-code analysis on the obtained IoT programs. The assembly-code analysis can also be achieved by utilizing other widely-used tools, such as RADARE2 [30] and ODA [31]. We use IDA because it has the highest precision value when disassembling binary code [32].

Firmware packer. It repacks the tested programs and other necessary tools, e.g., `telnet`, to obtain the IoT firmware with the necessary capabilities to perform our test. First of all, we need to enable the debug interfaces of the tested firmware. Unlike PCs that offer complete interaction and debugging interfaces to users, manufacturing best-practices of IoT devices dictate stripping out or disabling these interfaces. Thus, researchers cannot analyze IoT firmware in the same way as operating a PC, such as simply connecting a keyboard and a monitor to the tested devices. Some firmware contains a connectivity tool for remote login, such as `ssh` and `telnet`. However, most vendors keep their authentication keys secret for security concerns. It is inefficient to brute force keys on all the tested firmware. Moreover, vendors utilize a set of technologies to mitigate key leakage after firmware unpacking [33]. Thus, to obtain a debug interface, we insert a `telnet` into the extracted file system. Meanwhile, we modify the auto-start scripts in the file system to make the `telnet` service automatically start with a customized authentication key when the firmware starts. After that, we can operate the firmware through the inserted tool and the authentication key later. Additionally, we also insert a customized library loader into the extracted file system to support debugging facilities, such as `LD_PRELOAD` to support library functions hijacking. We also put the fault-scenario generator and the runtime monitor into the extracted file system. After obtaining the new packed firmware, we first run it in multiple emulators and physical devices to build the test environments. Then, the tested programs are assigned to these environments for concurrent tests.

```
1 int hijacking_error_function(){
2     collect_state();
3     if(!state_in_crash_log())
4         return ERROR;
5     return original_error_function();
6 }
```

Listing 5. An example of hijacking function.

Fault-scenario generator. It creates test cases according to our state-aware and bounded fault-scenario generation approach (§III-D). iFIZZ leverages library function hijacking to record the state of error sites and produce errors. We implement the fault scenario generator into a dynamically linked library. For each EF, a hijacking function is implemented in this library. By utilizing the `LD_PRELOAD` facility we add to the tested firmware, our hijacking functions are executed when the corresponding original EFs are called. The hijacking functions record the state of error sites and produce errors. To enable a better understanding, we present an example of a hijacking function in Listing 5. Specifically, we record the state of the latest error site in a crash log when a crash occurs. Meanwhile, we record the state of the current error site dynamically (line 2). Before we produce a runtime error on an error site, we first check whether its state is in the crash logs (line 3). If so, we skip the error producing on this error site and return the original EF (line 5); otherwise, we produce an error (i.e., return the error code of the original EF directly) at this error-site (line 4).

Runtime monitor. First, it performs dynamic analysis before starting fuzz-testing to obtain the target IoT programs and their corresponding run-commands. An IoT firmware may contain abundant software. Intuitively, analysts can analyze the whole firmware by separately testing each contained software. However, it can be extraordinarily time-consuming to do so. Additionally, many programs in firmware are poorly documented. Thus, even though analysts can locate a program, they may still miss the corresponding parameters to execute those programs accurately. Thus, our runtime monitor runs the tested firmware and traces the runtime process information to obtain the target IoT programs and their corresponding

TABLE III
BASIC INFORMATION OF THE TESTED FIRMWARE. AP IS ACCESS POINT; (E) INDICATES EMULATED DEVICES.

Model	Vendor	Version	Device	Arch
DIR-850L	DLink	1.00B05	Router (E)	Mipseb
DGS-1210-48	DLink	2.03.001	Switch (E)	Armel
FW_TV-IP121WN	Trendnet	V2_1.2.1.17	Camera (E)	Mipseb
K2	Phicomm	v163	Router	Mipsel
K2	OpenWRT	17.01.0	Router	Mipsel
TYCAM110	Tuya	V2	Camera	Armel
WAP200	Cisco	2.0.4.0	AP (E)	Mipseb
WAP4410N	Cisco	2.0.7.8	AP (E)	Mipseb
WNAP320	Netgear	v3.0.5.0	AP (E)	Mipseb
WG103	Netgear	V2.2.5	AP (E)	Mipseb

run-commands. Then, the runtime monitor repeatedly runs the tested IoT programs to perform bug testing.

Bug checker. It analyzes the crash log to generate crash reports. We implement the bug checker as an IDA script. Based on static analysis techniques, the error-function extractor and the bug checker can analyze cross-platform firmware. The fault scenario generator and the runtime monitor can easily be cross-compiled to different architectures, and then directly run both in emulators and devices. Therefore, IFIZZ is suitable for testing cross-platform IoT firmware.

V. EVALUATION

In this section, we first describe the experimental setup (§V-A). Then, we evaluate the effectiveness of EF identification (§V-B) and the variation caused by bounded-fault (§V-C). The result of error-handling testing and further analysis are given in §V-D and §V-E, respectively. Finally, we present the comparison with existing tools §V-F and the practical adoption of IFIZZ (§V-G).

A. Experimental Setup

IFIZZ is designed and implemented to be applicable to different types of devices with different operating systems, processors, and runtime libraries. This section evaluates IFIZZ with popular routers, IP cameras, access points, and switches from different leading vendors - DLink, Cisco, Netgear, etc. We choose these devices and vendors because they are representative, and they have a large market share [34]. As shown in Table III, the 10 IoT firmware produced by 7 vendors are used for evaluation, in which 7 firmware images are tested on emulators, and 3 are tested on physical devices. Before testing firmware images, IFIZZ first performs dynamic analysis of the tested firmware to obtain the target programs. In total, IFIZZ obtains 112 vendor-specific programs and 509 run-commands to run these programs.

B. EF Identification

Table IV shows the result of EF identification, including the number of all the library functions and the number of EFs identified by IFIZZ. In total, IFIZZ identifies 140 EFs out of 3,349 functions. To measure the accuracy of our method, we further conduct manual analysis on the 140 EFs. We finally confirm 129 EFs related to occasional errors, such as memory allocation failures and peripheral access failures, that can indeed occur and trigger error-handling code at runtime. We also analyze the 11 false positives in the identified EFs. The reason is that some identified functions never trigger input-independent errors. For example, function `strcpy` meets all the requirements that we used to identify EFs, i.e., its callers often check its return value, and the condition of its return value is not related to standard inputs. However, this function never triggers runtime errors. On the other hand, intuitively, our method may have false negatives. If there exists an EF whose return value is never checked by its caller, our method will miss it. However, we did not find an example in this case. In summary, the accuracy of IFIZZ for identifying EFs is 92.1%, which indicates that IFIZZ can dramatically reduce the manual work of identifying realistic EFs.

TABLE IV
RESULTS OF EF IDENTIFICATION.

Library	Function	Error-function
libuClibc-0.9.29.so	937	82
libuClibc-0.9.30.so	1090	11
libuClibc-0.9.30.3.so	1138	44
libcrypt-0.9.29.so	3	1
libcrypt-0.9.30.3.so	4	1
libxtables.so.2.0.0	40	2
Total	**3349**	**140**

C. ME and MBE

As described in §III-D, we propose two bounds, i.e., ME (the maximum number of errors in a FS) and MBE (the maximum distance between the first and the last error in a FS), to improve the efficiency of generating useful test cases. To understand the variation caused by different bounds, we evaluate IFIZZ with different MEs and MBEs on 10 randomly sampled popular programs, namely `sed`, `find`, `restore-configuration`, `killall`, `logger`, `md5sum`, `pidof`, `syslogd`, `lighttpd`, and `configd` from the WNAP320 firmware. For each group of ME and MBE, we use IFIZZ to conduct tests for 24 hours and record the number of program executions, crashes, and unique crashes. We count unique crashes by analyzing runtime traces.

As shown in Figure 4, when ME and MBE become larger, despite the throughput increases, the number of crashes and unique crashes does not always increase. For example, when ME = 7 and MBE = 14, the throughput is larger than that when ME = 6 and MBE = 12. There are two main reasons for the increase of throughput. First, smaller ME and MBE give up a large number of test cases due to the bound limitation. Thus, it takes more time to generate another suitable test case after each finished test. Second, the test cases generated in terms of large bounds contain more faults at the beginning of the testing. Thus, these test cases are highly possible to trigger a crash within a short time of execution and make the new test start earlier.

From Figure 4, we also find a trade-off between the value of the bounds and the number of crashes. The number of crashes and unique crashes when ME = 7 and MBE = 14 are smaller than that when ME = 6 and MBE = 12. On the one hand, if

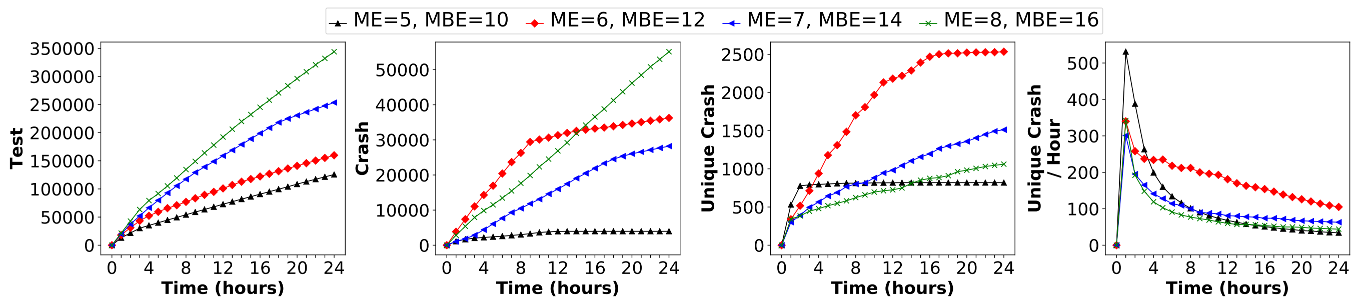


Fig. 4. Variation of results with respect to different ME and MBE.

the bounds are too small, we have to give up too many test cases, which leads to the miss of crashes. On the other hand, if the bounds are too large, we may generate too many redundant test cases, which leads to low efficiency when finding crashes. In summary, the results indicate that in a certain testing time (24 hours in our test), a set of moderate bounds (ME = 6 and MBE = 12) can improve the efficiency of discovering unique crashes. Based on this evaluation, we set ME = 6 and MBE = 12 by default in the remaining tests.

D. Results of Error-handling Testing

Detected bugs. Leveraging the 129 confirmed EFs in §V-B, we perform bug detection on the 10 firmware listed in Table III. We evaluate each firmware image for 24 hours. To uniquely count bugs, we identify their root causes by manually checking the assembly code and count bugs based on root causes. Table V shows the unique bugs detected by IFIZZ. Specifically, IFIZZ finds 109 bugs (including 46 program bugs and 63 library bugs) in the tested firmware images.

```
1  FILE *open_memstream(...) {
2      register __oms_cookie *cookie;
3      if ((cookie = malloc(...)) != NULL) {
4          if ((cookie->buf = malloc(...)) == NULL) {
5              goto EXIT_cookie;
6          }
7          ...
8      }
9      free(cookie->buf);
10 EXIT_cookie:
11     free(cookie);
12     return NULL;
13 }
```

Listing 6. Null pointer dereference in `uClibc`.

For example, IFIZZ finds a bug in `uClibc`, which is a C library for embedded Linux systems and is widely used in IoT devices [27]. In this library, there is a null pointer dereference that exists in the `open_memstream()` function. It is worth noting that this bug exists in nested error handling paths. At least two failures are needed to trigger this bug. For example, in `busybox`, if a memory allocation fails, it executes the error-handling code that invokes `vasprintf()` to show an alert message. Then, `vasprintf()` calls `open_memstream()`, as shown in Listing 6. If another memory allocation in `open_memstream()` fails, i.e., `cookie` in Line 3 becomes a null pointer, it calls another error-handling code in Lines 9-12. However, the error-handling code is not implemented correctly, which will result in a dereference to a null pointer in Line

TABLE V
DETECTED BUGS IN THE TESTED FIRMWARE. BP REPRESENTS BUGS IN IOT PROGRAM. BL REPRESENTS BUGS IN IOT LIBRARY.

Firmware	Unique Crash	Confirmed Bug	BP	BL
DIR-8505	167	9	2	7
DGS-1210-48	6	4	2	2
FW_TV-IP121WN	21	2	0	2
K2	127	4	2	2
OpenWRT	45	5	3	2
TYCAM110	227	32	17	15
WAP200	190	11	2	9
WAP4410N	3079	7	0	7
WNAP320	2112	23	13	10
WG103	2270	12	5	7
Total	**8244**	**109**	**46**	**63**

9. This bug indicates that 1) it is necessary to generate fault scenarios that contain more than one error for finding deep bugs in the nested error-handling code; 2) even though sometimes developers have implemented error-handling code, they may make mistakes in the code due to the complex contexts of nested errors, and thus necessary testings are desired. We find that this bug exists in `uClibc` before version v1.0.31. Before developers noticed and patched this bug in September 2019, this bug has existed in uClibc for more than 15 years. In these years, developers modified this source file many times and even patched the error-handling code that contains this error. However, this bug is ignored for an extraordinarily long time. IFIZZ can find this bug in a few seconds. Thus, we believe that with IFIZZ, analysts and developers can effectively and efficiently improve the security of their code.

Bug features. Reviewing the bugs found by IFIZZ, we find three interesting features. Firstly, some bugs are triggered by more than one failure in different EFs, which indicates that it is necessary to generate FSs with multiple errors covering multiple EFs. Secondly, the lack of error-handling code in nested error-paths causes many bugs, which indicates that it is necessary to test deep error-paths. Third, as shown in Table V, different unique crashes could be caused by the same bug. For example, IFIZZ discovered more than 2,112 unique crashes in WNAP320. However, after further analysis, we find that 10 bugs in IoT libraries lead to 2,021 unique crashes. These buggy libraries are frequently used in IoT programs. Thus, they lead to a large number of crashes under different execution paths. This discovery gives us two insights. (1) The bugs

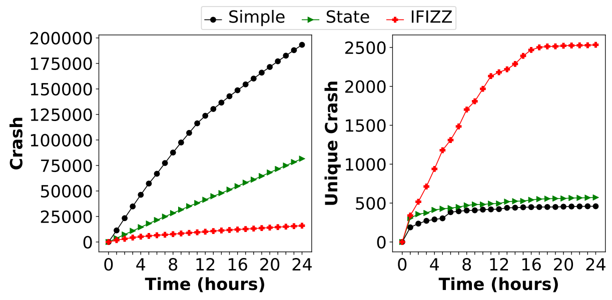


Fig. 5. Crashes discovered by different fault-scenario generation approaches.

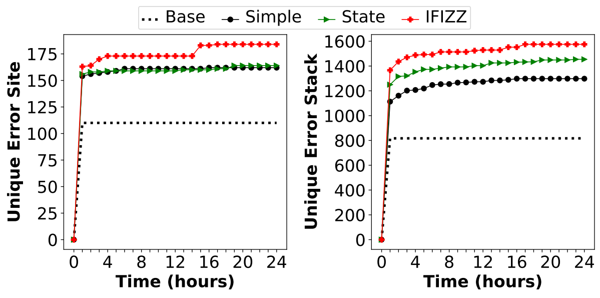


Fig. 6. Code coverage of different fault-scenario generation approaches.

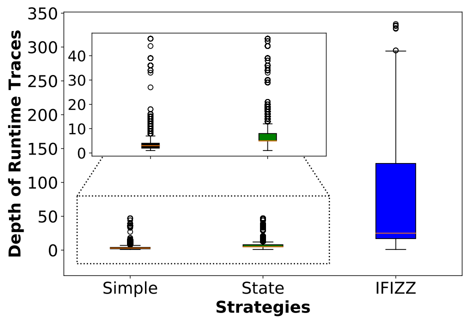


Fig. 7. Depth of runtime traces covered by different fault-scenario generation approaches.

in IoT libraries are very harmful because they will affect a large number of programs. (2) Although unique crashes, i.e., crashes under different execution paths, are widely used to evaluate the effectiveness of fuzzing systems, it is not always fair and objective to evaluate fuzzers only by this metric. Building up a reasonable evaluation criteria system for fuzzers is an interesting yet challenging problem by itself, which is a promising future research direction.

E. Ablation Study

In IFIZZ, our state-aware and bounded fault-scenario generation are important techniques for generating effective FSs. To evaluate the benefits of them, we develop other tools by modifying IFIZZ via removing different strategies. We implement and compare three tools in this evaluation. 1) `Simple` is implemented with none strategy. It performs fault injection to every error site. 2) `State` is implemented with the state-aware approach. 3) IFIZZ is implemented with all approaches. We evaluate the resulting tools on the 10 representative programs described in §V-C for 24 hours.

Unique crashes. We first investigate the performance of IFIZZ on finding crashes and unique crashes. We count unique crashes by identifying unique runtime traces of all crashes. As shown in Figure 5, despite that IFIZZ triggers fewer crashes within a given time, it can still find most unique crashes. For example, `Simple` triggered 193,259 crashes in 24 hours. However, there are only 459 (0.2%) unique crashes. `State` discovers 572 (0.7%) unique crashes. By contrast, 2,534 (15.9%) out of the 15,986 crashes identified by IFIZZ are unique. The results indicate that the state-aware and bounded fault-scenario generation approach leveraged by IFIZZ effectively discovers unique crashes in IoT firmware. Besides, we also find that IFIZZ can discover all the unique crashes found by `Simple` and `State`, which reveals that IFIZZ does not miss unique crashes when reducing redundant crashes.

Error-path coverage. A good detection approach should generate effective FSs that cover more unique error sites and error stacks intending to trigger more error-handling code. Thus, we then evaluate the error-path coverage of IFIZZ. We implement another tool, `Base`, as a baseline in this experiment. `Base` does not produce errors. It runs the tested program repeatedly. As shown in Figure 6, each of our approaches improves the error-path coverage, and the more strategies we included, the more error-paths we cover. For instance, `Base` covers only 110 unique error sites and 817 unique error stacks. When we produce errors, `Simple` discovers 162 unique error sites and 1,298 unique error stacks. After we add our state-aware approach, `State` can find 164 unique error sites and 1,454 unique error stacks. Finally, IFIZZ can cover most unique error sites (184) and error stacks (1,575). There are several reasons for this increment. First, producing errors can improve error-path coverage by forcing the execution of these paths. Second, the state-aware and bounded approach can further improve the error-path coverage by efficiently covering deep error-paths.

Error-path depth. Besides, we investigate the depth of error-paths covered by different tools. Specifically, we evaluate error-path depth from two aspects. (1) The depth of runtime traces, i.e., the number of error stacks in a runtime trace. (2) The depth of error stacks, i.e., the number of function calls in an error stack. Figure 7 shows the depth distribution of runtime traces. IFIZZ can trigger deeper runtime traces than other tools. In particular, when using `Simple`, the depth median of the runtime traces is 3. In comparison, the depth median of the runtime traces of IFIZZ is 25 (7.3 times deeper). Meanwhile, each approach used by IFIZZ helps trigger deep runtime traces. The reason is that they can help IFIZZ reach deeper error stack by automatically skipping the production of errors on duplicate error stacks. Similarly, as shown in Figure 8, IFIZZ can also trigger deeper error stacks than other tools. For example, the depth median of error stacks tested by IFIZZ (13) is 44.4% deeper than that tested by `Simple` (9).

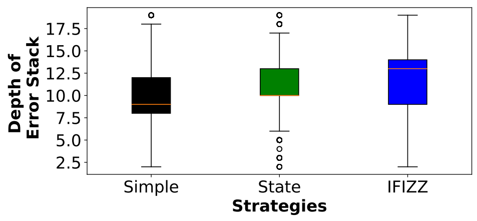


Fig. 8. Depth of error stacks covered by different fault-scenario generation approaches.

TABLE VI
RESULTS OF IFIZZ AND FIRMAFL.

Program/Lib	IFIZZ		FirmAFL	
	Crash	Unique Crash	Crash	Unique Crash
bzcat	28	**12**	5.07M	1
cmp	53	**13**	0	0
wc	56	**21**	182	19
uniq	89	**23**	0	0
Total	226	**69**	>5M	20

F. Comparison with Existing Tools

Recently, several tools have been developed to fuzz IoT firmware. Among them, we select the state-of-the-art and open-source fuzzing tool, FirmAFL [13], to make a detailed comparison with IFIZZ (most of the other tools are closed-source). Meanwhile, to validate the generality of IFIZZ, we further select `busybox`, an open-source program widely used in IoT firmware, as the tested program. Note that such a selection is mainly for conveniently comparing IFIZZ and FirmAFL. Further, considering that FirmAFL can only test the programs with standard inputs, we turn to test four representative applets of `busybox` including `bzcat`, `cmp`, `wc`, and `uniq`, which have standard inputs. As FirmAFL can only work on an IoT emulator, we conduct this evaluation on the IoT emulator used in [13] for 24 hours per applet. We show the results in Table VI, from which we have the following observations. (1) IFIZZ can find much more unique crashes than FirmAFL. For instance, FirmAFL does not find any crash on `cmp` and `uniq`. By contrast, IFIZZ can find 36 unique crashes on these two applets. (2) IFIZZ can report unique crashes more efficiently. For example, on `bzcat`, IFIZZ discovers 12 unique crashes out of 28 found crashes, while FirmAFL only finds 1 unique crash out of 5.07 million crashes, which indicates that FirmAFL wastes much time in triggering the same bug. The reason is that compared to FirmAFL, IFIZZ can cover input-independent error paths using state-aware and bounded fault-scenario generation. Therefore, IFIZZ can effectively find more bugs in these deep error paths missed by FirmAFL.

G. Practical Adoption

To verify the availability of IFIZZ in the practical production environment, we deploy IFIZZ in a large scale of commodity IoT devices by collaborating with a world-wide leading IoT company. Our cooperative company provides 500+ types of IoT products and services to users in 220+ countries and regions. By the time of submission, the company have confirmed 32 previously-unknown bugs detected by IFIZZ. IFIZZ is constantly discovering new bugs. Thus, we are continually working with the company to confirm bugs and develop patches to fix them. Extensive evaluation in real-world adoption shows that IFIZZ is more than a laboratory tool - it can efficiently find bugs in commodity IoT devices and help companies improve the security of their products.

VI. DISCUSSION

Error-function identification. IFIZZ identifies error-functions that can actually fail and trigger error-handling code by leveraging the approach proposed in §III-C. However, as described in §V-B, there are false positives in the identified error-functions. The main reason for false positives is that our approach treats a function as an error-function as long as it returns error code and its return condition is input-independent. However, some functions, such as `strcmp`, that meet these requirements are not real error-functions. Thus, it is interesting to develop an automated and effective method for more accurate error-function identification.

On the other hand, IFIZZ cannot identify error-functions whose return values have never been checked in any IoT firmware. Therefore, IFIZZ may have false negatives. However, the possibility of this case is minimal, and we have not found any such instance during our manual analysis. We will conduct more analysis on the possible false negatives in the future.

Bug detection. Similar to existing works, IFIZZ may miss bugs in error-handling code. There are many reasons for these false negatives: 1) IFIZZ may miss bugs caused by static error-functions. Even though IFIZZ can effectively and comprehensively identify the extern error-functions in libraries, the functions implemented in IoT programs may also cause occasional errors. However, IFIZZ cannot test the error-handling code triggered by such functions at this moment. 2) IFIZZ relies on observable crashes to detect bugs. However, previous works [13], [35] have proved that the effects of memory corruption are often less visible. As a result, IFIZZ may miss the bugs that never cause crashes. To solve this problem, we can use advanced checkers, such as the heuristics proposed in [35], to detect the missed bugs. 3) IFIZZ cannot cover all the code in the tested IoT firmware. For example, some code only executes with specific inputs. Thus, IFIZZ may miss the error-handling code in such input-related paths. To solve this problem, we plan to combine input-mutation fuzzers with IFIZZ to cover more paths.

VII. RELATED WORK

A. Analysis of Error-handling Code

Many static methods detect bugs in error-handling code by analyzing source code [36]–[41]. For instance, EPEX [36] identifies error paths based on error specifications and explores different error paths to find bugs. Static analysis can conveniently analyze the target program without actually executing it. However, it often reports many unreal bugs due to the lack of the exact runtime information. Moreover, existing approaches need the source code of the tested programs, which is rarely the case for IoT firmware.

In terms of dynamic analysis, T-FUZZ [42] tests deep paths by removing sanity checks. Intuitively, T-FUZZ can cover error-handling code. However, it does not inject any fault (such as set a NULL pointer). It thus misses many bugs in fault scenarios, such as null pointer dereference bugs. Many SFI-based methods can test error-handling code and have shown promising performance on PC programs [8], [14]–[22]. Some approaches inject single [15], [16], [21] or random [14], [19], [20] faults in each test case to trigger error-handling code. However, these methods can only cover a limited number of error-handling code and report many unreal bugs [43]–[45]. Several approaches [8], [17], [18], [22] can cover more error-handling code to detect more real bugs. For instance, FIFUZZ [8] can find deep error-handling bugs by utilizing a context-sensitive SFI approach. However, the existing SFI-based approaches are mainly designed for testing PC programs. They do not provide an efficient/proper solution for testing IoT firmware. For example, they fail to identify the target functions in IoT firmware binaries and to generate efficient fault scenarios. IFIZZ solves these problems by leveraging multiple strategies described in §III-C and §III-D.

B. Vulnerable IoT Device Analysis

Without the source code of IoT firmware, many approaches perform static analysis on the binary image [46]–[52]. For instance, Gemini [50] utilizes a neural network–based approach to detect known vulnerable functions. However, these methods suffer from high false positives due to the lack of runtime information. Moreover, these static methods are limited in discovering known bugs.

To mitigate these problems, several approaches support dynamic analysis on IoT firmware [9]–[13], [28], [35], [53], [54]. However, running a full fuzzing operation inside the device is impractical, because IoT devices are typically designed to be as low-cost or low-power as possible. Thus, previous works [9]–[11], [13], [28], [53], [54] emulate embedded firmware based on QEMU [55]. However, the main goal of these works is to provide a suitable environment for running and testing IoT firmware. Although some of them used existing tools, such as AFL, to perform bug detection in their environment, most of them are not dedicatedly designed for bug detection. Recently, IoTFuzzer [12] directly performs fuzzing on physical IoT devices. However, the efficiency of IoTFuzzer is low due to the slow throughput. To the best of our knowledge, IFIZZ is the first work to perform dynamic bug analysis on not only IoT emulators but also physical devices. Meanwhile, IFIZZ is the first work that tests the error-handling code in IoT firmware.

VIII. Conclusion

Error-handling code in IoT devices is prevalent but highly buggy. Testing error-handling code in IoT devices faces extra challenges due to their complex running environments and limited computation power. In this paper, we presented a novel framework named IFIZZ to effectively test the deep error-handling code of IoT firmware. IFIZZ can efficiently and effectively test error handling because it automatically identifies potential errors and constructs effective fault scenarios. It can also test deep error paths since it performs state-aware and bounded fault-scenario generation. We develop a full-featured prototype of IFIZZ and evaluate it on 10 real-world IoT firmware images. IFIZZ finally found 109 critical bugs. It also features high code coverage. Notably, IFIZZ covers 67.3% more error paths than normal execution, and the depth of error-handling code covered by IFIZZ is 7.3 times deeper than that covered by traditional fault injection on average. The promising results benefit from IFIZZ's strengths in effectively and efficiently exploring error-handling code. Finally, we compared IFIZZ with a state-of-the-art tool, `FirmAFL`, and the results show that IFIZZ can find many bugs that are missed by `FirmAFL`. We have deployed IFIZZ in practical adoption, and we will open-source IFIZZ for facilitating future IoT security research. Our study may shed new light on designing practical IoT vulnerability detection approaches for the research community and IoT industry.

IX. Acknowledgment

This work was partly supported by NSFC under No. U1936215, the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under No. LR19F020003, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), the National Key Research and Development Program of China under Grant No. 2020AAA0140004, and Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies.

References

[1] A. Sengupta, T. Leesatapornwongsa, M. S. Ardekani, and C. A. Stuardo, "Transactuations: where transactions meet the physical world," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 91–106.

[2] J. Choi, H. Jeoung, J. Kim, Y. Ko, W. Jung, H. Kim, and J. Kim, "Detecting and identifying faulty iot devices in smart home with context extraction," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 610–621.

[3] M. S. Ardekani, R. P. Singh, N. Agrawal, D. B. Terry, and R. O. Suminto, "Rivulet: a fault-tolerant platform for smart-home applications," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 41–54.

[4] P. A. Kodeswaran, R. Kokku, S. Sen, and M. Srivatsa, "Idea: A system for efficient failure management in smart iot environments," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 43–56.

[5] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu, and R. Beyah, "Mpinspector: A systematic and automatic approach for evaluating the security of iot messaging protocols," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[6] K. Kapitanova, E. Hoque, J. A. Stankovic, K. Whitehouse, and S. H. Son, "Being smart about failures: assessing repairs in smart homes," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, 2012, pp. 51–60.

[7] A. K. Sikder, H. Aksu, and A. S. Uluagac, "6thsense: A context-aware sensor-based attack detector for smart devices," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 397–414.

[8] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[9] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, vol. 14, 2014, pp. 1–16.

[10] B. Feng, A. Mera, and L. Lu, "P^{2}im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[11] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1–18.

[12] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[13] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1099–1114.

[14] P. D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 379–388.

[15] J.-J. Bai, Y.-P. Wang, J. Yin, and S.-M. Hu, "Testing error handling code in device drivers using characteristic fault injection," in *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, 2016, pp. 635–647.

[16] J.-J. Bai, Y.-P. Wang, H.-Q. Liu, and S.-M. Hu, "Mining and checking paired functions in device drivers using characteristic fault injection," *Information and Software Technology*, vol. 73, pp. 122–133, 2016.

[17] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 281–294.

[18] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 361–372.

[19] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott, "Testing of java web services for robustness," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004, pp. 23–34.

[20] M. Mendonca and N. Neves, "Robustness testing of the windows ddk," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 2007, pp. 554–564.

[21] M. Susskraut and C. Fetzer, "Automatically finding and patching bad error handling," in *2006 Sixth European Dependable Computing Conference*. IEEE, 2006, pp. 13–22.

[22] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 595–605.

[23] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a reactive immune system for software services," 2005.

[24] *iFIZZ*, April 2021, https://github.com/decentL/iFIZZ-ASE21.

[25] *OpenWRT*, April 2021, https://openwrt.org.

[26] *DD-WRT*, April 2021, https://dd-wrt.com.

[27] *uClibc*, April 2021, https://uclibc.org.

[28] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, vol. 16, 2016, pp. 1–16.

[29] *IDA*, April 2021, https://www.hex-rays.com/products/ida.

[30] *RADARE2*, April 2021, https://www.radare.org/n/.

[31] *ODA*, April 2021, https://onlinedisassembler.com/.

[32] J. Muhui, Z. Yajin, L. Xiapu, W. Ruoyu, L. Yang, and R. Kui, "An empirical study on arm disassembly tools," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[33] X. Wang, Y. Sun, S. Nanda, and X. Wang, "Looking from the mirror: evaluating iot device security through mobile companion apps," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1151–1167.

[34] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All things considered: an analysis of iot devices on home networks," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1169–1185.

[35] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." in *NDSS*, 2018.

[36] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 345–362.

[37] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 472–482.

[38] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller, "Finding error handling bugs in openssl using coccinelle," in *2010 European Dependable Computing Conference*. IEEE, 2010, pp. 191–196.

[39] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.

[40] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 496–506.

[41] Z. Jia, S. Li, T. Yu, X. Liao, J. Wang, X. Liu, and Y. Liu, "Detecting error-handling bugs without error specification input," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 213–225.

[42] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[43] N. Kikuchi, T. Yoshimura, R. Sakuma, and K. Kono, "Do injected faults cause real failures? a case study of linux," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 2014, pp. 174–179.

[44] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira, "Representativeness analysis of injected software faults in complex software," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2010, pp. 437–446.

[45] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2012.

[46] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "Pie: Parser identification in embedded systems," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 251–260.

[47] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 95–110.

[48] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[49] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.

[50] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[51] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 896–899.

[52] B. Zhao, S. Ji, W.-H. Lee, C. Lin, H. Weng, J. Wu, P. Zhou, L. Fang, and R. Beyah, "A large-scale empirical study on thevulnerability of deployed iot devices," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[53] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 437–448.

[54] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: peripheral proxying supported embedded code testing," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 329–340.

[55] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.