SFuzz: Slice-based Fuzzing for Real-Time Operating Systems

Libo Chen
Shandong University, SJTU
chenlibo147@mail.sdu.edu.cn

Quanpu Cai
SJTU, Alibaba Group
cpeggsjtu@sjtu.edu.cn

Zhenbang Ma
QI-ANXIN
mazhenbang@qianxin.com

Yanhao Wang*
QI-ANXIN
wangyanhao136@gmail.com

Hong Hu
Pennsylvania State University
honghu@psu.edu

Minghang Shen
Tencent Security Xuanwu Lab
hadreys1007@gmail.com

Yue Liu
Southeast University, QI-ANXIN
230209311@seu.edu.cn

Shanqing Guo*
Shandong University
guoshanqing@sdu.edu.cn

Haixin Duan
Tsinghua University
duanhx@tsinghua.edu.cn

Kaida Jiang
SJTU
kaida@sjtu.edu.cn

Zhi Xue
SJTU
zxue@sjtu.edu.cn

ABSTRACT

Real-Time Operating System (RTOS) has become the main category of embedded systems. It is widely used to support tasks requiring real-time response such as printers and switches. The security of RTOS has been long overlooked as it was running in special environments isolated from attackers. However, with the rapid development of IoT devices, tremendous RTOS devices are connected to the public network. Due to the lack of security mechanisms, these devices are extremely vulnerable to a wide spectrum of attacks. Even worse, the monolithic design of RTOS combines various tasks and services into a single binary, which hinders the current program testing and analysis techniques working on RTOS.

In this paper, we propose SFuzz, a novel slice-based fuzzer, to detect security vulnerabilities in RTOS. Our insight is that RTOS usually divides a complicated binary into many separated but single-minded tasks. Each task accomplishes a particular event in a deterministic way and its control flow is usually straightforward and independent. Therefore, we identify such code from the monolithic RTOS binary and synthesize a slice for effective testing. Specifically, SFuzz first identifies functions that handle user input, constructs call graphs that start from callers of these functions, and leverages forward slicing to build the execution tree based on the call graphs and pruning the paths independent of external inputs. Then, it detects and handles roadblocks within the coarse-grain scope that hinder effective fuzzing, such as instructions unrelated to the user input. And then, it conducts coverage-guided fuzzing on these code snippets. Finally, SFuzz leverages forward and backward slicing to track and verify each path constraint and determine whether a bug discovered in the fuzzer is a real vulnerability. SFuzz successfully discovered 77 zero-day bugs on 35 RTOS samples, and 67 of them have been assigned CVE or CNVD IDs. Our empirical evaluation shows that SFuzz outperforms the state-of-the-art tools (e.g., UnicornAFL) on testing RTOS.

CCS CONCEPTS

• **Security and privacy** → **Systems security**;

KEYWORDS

RTOS; Slice-based Fuzzing; Taint analysis; Concolic execution

ACM Reference Format:
Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang*, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo*, Haixin Duan, Kaida Jiang, and Zhi Xue. 2022. SFuzz: Slice-based Fuzzing for Real-Time Operating Systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3559367

1 INTRODUCTION

A real-time operating system (RTOS) is designed to serve real-time applications. It has been widely deployed on top of embedded microcontrollers and CPUs, with even more installations than fully-fledged operating systems. For example, VxWorks, the industry-leading RTOS [38], runs on over two Billion devices [33]. Many industry scenarios of RTOS not only require real-time, deterministic performance, but also expect safety and security certifications such as NASA's InSight Spacecraft [24].

However, it is challenging to apply traditional security mechanisms into RTOS due to various development constraints. For instance, to support immediate responses to real-time tasks, RTOS abandons the isolation between kernel and user spaces [20] and runs all tasks in a flat mode to avoid frequent context switching. In this case, all software modules have unrestricted access to all data and instructions in memory space [11], which brings more potential threats to RTOS. This monolithic design was acceptable

*Corresponding authors.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559367

since RTOS mainly runs in local networks and is isolated from external threats.

These days, the Internet of things (IoT) connects more RTOS devices directly to the Internet, which opens the door of RTOS devices to external attackers. It is urgent to discover vulnerabilities in RTOS systems before attackers compromise these weak devices. Although many researchers proposed various bug detection mechanisms on embedded devices [7, 9, 21, 31, 44], few of them can be directly applied to RTOSes. The main reason is that RTOS is usually presented in a blob-firmware format and barely runs on microcontrollers and CPUs in a single monolithic execution, including kernel module, schedule module, and other task modules. This feature brings difficulty to traditional bug-detection mechanisms.

For static analysis [9, 31], considering the large size of the monolithic RTOS binary, classic static methods (e.g., symbolic execution [6, 10, 34]) suffer from path explosion issues. Further, due to the lack of explicit function symbols and the complexity of RTOSes, it is hard to reveal the semantics of functions on binary-level. Therefore, we cannot easily identify modules that are related to sensitive data, nor conduct any analysis on these modules. Dynamic solutions [7, 21, 37, 44], such as fuzzing, either require actual devices or rely on correct and steady emulations to test target firmware and essential services. Since RTOSes from different vendors [15, 16, 38] employ various peripherals with diverse interfaces, it is challenging to emulate all real-world scenarios with manageable efforts.

As RTOSes get more attention, several works develop customized tools for detecting bugs from RTOSes. Zhu et al. [45] introduce a debugging method to detect vulnerabilities from VxWorks-based IoT devices. Wen et al. [37] propose to check configuration errors from bare-metal firmware on BLE devices. However, these methods either only work on specific devices, rely on real devices [45], or detect limited bug types [37]. Salehi et al. [32] instrument bare-metal firmware binaries to make memory corruptions observable. Clements et al. [12] extend HALucinator [13] to work with VxWorks. However, these tools require manual analysis and domain knowledge, and thus can hardly be applied to diverse RTOS systems. Many of these works even need the source code of RTOSes to get more details of the hardware [12, 13] or need much extension development for efficient fuzz testing [32]. Hence, all of them are limited by scalability. Overall, there lacks a flexible and general way to discover vulnerabilities effectively in RTOS.

Despite the numerous difficulties, we notice that specific RTOS features provide unique opportunities to bypass the testing barriers, like the multi-tasking mechanism. Specifically, RTOS usually divides a complicated application into many separate but single-minded tasks. Each task accomplishes a particular event in a deterministic way. The control flow of each atomic task is usually straightforward and independent. More importantly, if these tasks belong to the same category, their data flow may have similar patterns. Therefore, we search for the data flow that starts from diverse external data entries to potential sink functions (e.g., `memcpy`), and slice the corresponding code snippets among tasks of RTOS. These slices are small enough to be tested using existing fuzzing logic. Furthermore, they present much smaller but more critical control-flow scopes. It can sharply alleviate emulation difficulty and analysis complexity, which will allow us to perform more effective and efficient testing, like greybox fuzzing and symbolic execution.

Based on our insight, we propose SFuzz, a novel fuzzing method that leverages forward slicing to construct a tailored code space that drives greybox fuzzing on the emulator. Then, SFuzz incorporates backward slicing to perform concolic execution to verify crash inputs from fuzzing. We design SFuzz with four main components:

Forward Slicer. Our code slicer starts by identifying functions that handle user inputs. Since RTOS binaries do not have function names, we define a set of heuristics to locate such functions. After that, we construct a call graph that starts from one caller of these functions. Through coarse-grained propagation, we perform forward slicing in this call graph by pruning paths independent of external inputs. We also modify conditional branches irrelevant to inputs to ensure the control flow reaches potential sink functions. Because of covering the data-sharing paradigm with direct physical memory access (frequently used in RTOS), our forward slicer can flexibly extend across different tasks and dynamically stitch paths in execution.

Control Flow Node Handler. The fuzzing engine will lack the full context and runtime state of the RTOS if it directly executes the code snippets. Hence, Control Flow Node Handler is used to guide the fuzzer to determine how to handle the function calls and conditional branches unrelated to the user inputs, helping the fuzzing engine increase the efficiency and stability of path exploration.

Micro Fuzzing. Our fuzzing engine focuses on instructions in the pruned execution tree. Starting from the input source, it updates the execution context through instruction-level emulation. The fuzzing engine will execute input-related code snippets and ignore massive unnecessary paths, including other input handlers. To inspect dangerous behaviors, it monitors the contexts of sink function calls and reports potential bugs when the contexts violate pre-defined safety policies.

Concolic Analyzer. To fetch missed context information in the pruned call graph, we recover modified conditional branches and symbolize the context of ignored functions. Then, we perform concolic execution based on the forward slice and use the bug-triggering input to provide the concrete value. We execute backward slicing that starts from the sink function to the written object allocation position and other input reference sites. Then, we perform symbolic execution starting from the ending of backward slices to get constraints of object size and iterate every predicate of other input to check whether the corresponding condition is necessary or not. At last, we can realize a complete and accurate path condition to evaluate a vulnerability.

We implement our prototype of SFuzz based on Ghidra [17] and UnicornAFL [25] with around 6,200 lines of Python code, 4,300 lines of C code, and 5,100 lines of Java code. To understand the efficacy of SFuzz in detecting security vulnerabilities in RTOS, we apply our tool to 35 firmware samples from 11 vendors. SFuzz successfully discovered 77 unknown vulnerabilities in these latest-version firmware samples. We also compare SFuzz with the state-of-the-art tools, and SFuzz outperforms all compared tools.

In summary, we make the following contributions:

- We propose a slice-based method to test RTOS, which utilizes forward slicing to prune control flow for efficient fuzz testing, and incorporates backward slicing to validate alerts from fuzzing.

- We design and implement SFuzz[1], which performs slice-based fuzzing through cross-platform CPU emulation to effectively detect vulnerabilities in RTOS firmware.
- We evaluated SFuzz on 35 real-world RTOS firmware samples from 11 vendors and discovered 77 unknown bugs. 68 bugs have been assigned CVE/CNVD IDs.

2 PROBLEM AND APPROACH OVERVIEW

In this section, we first provide the background of vulnerabilities in RTOS. Then, we present the overview of our approach and discuss the associated challenges.

2.1 RTOS and Embedded Devices

The Real-Time Operating System (RTOS) is designed to provide a deterministic execution pattern. It focuses on timely task execution and works for embedded devices with real-time requirements, such as printers, switches, and routers. Because of the limitation of memory space and the requirements for fast task scheduling and quick response (i.e., performance constraints) [20, 31], some vendors compile their RTOS into a single binary with all functionalities. Meanwhile, they also strip the system symbols to reduce the size of binary files. These factors bring challenges for researchers to emulate the whole system or analyze the security of the embedded devices that use RTOS as their operating system.

However, these devices (i.e., printers, routers, and so on) usually receive data packages from the outside (e.g., network, Bluetooth, and so on) and parse them to finish various tasks, providing attackers with ways to hijack them. Meanwhile, these devices are usually the critical points of the home network or Local Area Network (LAN). Without state-of-the-art defense mechanisms, such as Executable Space Protection [23] and stack canary [14], attackers pay a lot of attention to them and like to hack them through their data entries. Hence, it is necessary for security researchers to overcome the challenges and propose a proper method to detect vulnerabilities in the embedded devices that are easy to touch through their data entries.

2.2 Motivation Example

Listing 1 shows a simplified snippet that contains a buffer overflow error in Line 40. We found this bug in the RTOS of TP-Link WDR7660 using our tool, SFuzz, and reported it to its vendor. The bug has been fixed and assigned a CVE number `CVE-2020-28877`. The functionality of this code snippet is to receive data from outside (Line 7) and call the corresponding function set to handle the data package (Line 9-44). In detail, it first steps into function `protocol_handler` (Line 9) to check the basic format and size of the package header (Line 15), match the magic bytes (Line 16), and check the integrity of the whole package (Line 17). If the input package satisfies all constraints, the execution will step into the function `msg_handler` (Line 18) to call the corresponding handler (Line 24) based on the version of the package (Line 23). Function `parse_advertisement` resolves and extracts the header of the structure `element` (Line 35-36), which is the basic data unit in the package's payload, and copies the `element` to a memory space (Line 40). Because the length of the data section in the structure `element` is recorded in its `len` field, an attacker could trigger a stack overflow vulnerability here via accurately constructing a package and setting the value of `len` larger than the length of the buffer used to store the data section.

```
1  void devDiscoverHandle(int sockfd) {
2    int len, ret;
3    struct sockaddr_in src_addr;
4    int addrlen = sizeof(struct sockaddr_in);
5    memset((uint8 *)&src_addr, 0, 0x10);
6    memset(Global_addr, 0, 0x5C0);
7    len = recvfrom(sockfd, Global_addr+0x1c, 0x5a4, 0, (struct
         sockaddr *)&src_addr, (socklen_t *)&addrlen);
8    if ( len != ERROR )
9      ret = protocol_handler((packet *)(Global_addr+0x1c));
10     if (ret == ERROR)
11       logOutput("devDiscoverHandle Error!");
12 }
13 int protocol_handler(packet *data) {
14   bytes[4] = {0xe1, 0x2b, 0x83, 0xc7};
15   if (header_check(data))
16     if (magic_check(data->magic_bytes, bytes, 4))
17       if (checksum(data))
18         return msg_handler(data);
19   return ERROR;
20 }
21 int msg_handler(packet *data) {
22   int ret = ERROR;
23   if (data->version == 0x01)
24     ret=parse_advertisement(data->payload, data->payloadLen);
25   return ret;
26 }
27 int parse_advertisement(uint8 *payload, int payloadLen) {
28   char* dst;
29   char* var_addr;
30   char buffer[64];
31   int index;
32   var_addr = DAT_404d33a8;
33   msg_element *element;
34   msg_element_header *element_header;
35   element = parse_msg_element(payload, payloadLen);
36   element_header = element->header;
37   if (element_header) {
38     index = (int)*(var_addr+4));
39     dst = buffer+index;
40     if (copy_msg_element((char *)element->data, dst,
         element_header->len)) == 0) //Stack Overflow
41       return SUCCESS;
42   }
43   return ERROR;
44 }
```

Listing 1: Pseudocode of the simplified motivation example.

Unfortunately, current bug-finding techniques for embedded systems cannot detect this vulnerability effectively. Dynamic solutions, like fuzzing and emulation, cannot guarantee to emulate a whole RTOS with closed hardware features and cover all program states, especially for the specific code parsing the self-defined data format. For example, suppose we have a device and want to use the recent work SRFuzzer [43] to identify this bug. In that case, we have to leverage reverse engineering to analyze all data formats the device can handle and then generate and send requests to trigger the handling logic code. It requires a high labor cost, and the requirements for analysts are very high. Static approaches such as KARONTE [31] and SaTC [9] cannot effectively locate the data entries in a monolithic RTOS binary. For example, KARONTE only focuses on the data related to the inter-process communication and SaTC takes the keywords used to mark the user input to find

[1]We will release the source code at https://github.com/NSSL-SJTU/SFuzz.

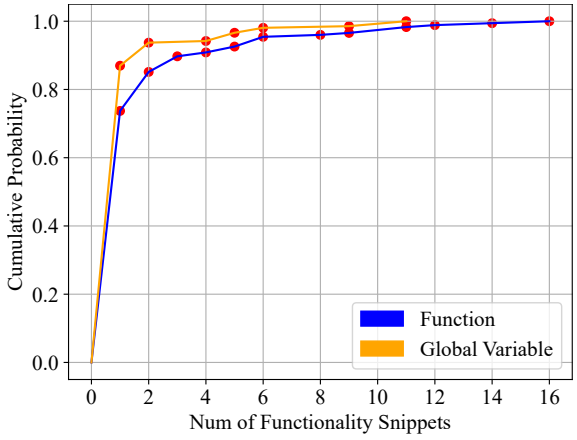


Figure 1: (1) Distribution of functions w.r.t. the number of functionality snippets they belong to. 75% of functions in the functionality snippets we identified only belong to one functionality snippet, and less than 5% of functions are shared by more than six functionality snippets. **(2) Distribution of global variables w.r.t. the number of functionality snippets they are used in.** 87% of global variables in these functionality snippets are only used in one functionality snippet. These results show that most functionality snippets are independent of each other.

the data entry. Moreover, the complex code logic (we simplify the control flow graph of the snippet in the motivation sample) in the snippet is hard for them to conduct static taint analysis or symbolic execution to efficiently find the sensitive path from the data entry point (e.g., `recvfrom`) to the sink function (e.g., `memcpy`).

2.3 Necessity and Reasonability of SFuzz

According to the above analysis and comparison, the dynamic method is more applicable in detecting RTOS bugs than the static analysis method. However, without much manual analysis and a robust full-system emulation method, how can we effectively detect these vulnerabilities in the real-time operating systems of various embedded devices via a dynamic method?

Our intuition is that *conducting fuzzing on functionally independent snippets is an effective way to discover bugs in RTOSes*, and we name it **slice-based fuzzing**. Suppose we collect a program slice containing the complete function set that receives and handles the corresponding data package. We can effectively and reasonably detect the vulnerabilities in the piece of code via hybrid fuzzing and instruction-level emulation. Moreover, we can ignore the difficulties of emulating various hardware and service features.

Consider the sample in Listing 1, after identifying the data-receiving function (Line 7) and all the functions related to the package handling (e.g., `copy` in Line 40), we could construct the code snippet and generate input to trigger the stack overflow vulnerability through coverage-guided hybrid fuzzing. Due to lacking complete context while executing the code snippet, we need to handle the control flow nodes unrelated to the external package to make fuzzing efficient and stable. Once the vulnerability is found, we do a concolic analysis based on the input that triggers the vulnerability, and obtain a final result, including the complete constraints related to the control flow nodes we handle, to help us determine whether the vulnerability is a false positive. As we see, the exploration scope of our method is limited to risky code snippets by static analysis. Meanwhile, we only use symbolic execution to help us create new test cases while the fuzzing gets stuck, and verify the crash result. These make our method, compared with the traditional static analysis and symbolic execution methods, could have better performance and mitigate the path explosion problem.

To prove the reasonability of the slice-based fuzzing method on RTOS, we selected systems of four embedded devices from two vendors (i.e., TP-Link and MERCURY) to check whether their functions can be divided into task-specific, independent snippets. We choose these four devices because they all contain symbol files or log functions, which facilitate our judgment on the functionality of each function and the manual verification of the result. We search all types of data read-in points in the system, such as the `recvfrom` function, and take their caller functions (e.g., `devDiscoverHandle`) as the root points of the data handling trees and the specific task modules. Then we recursively search the function calls existing in a root point function and its children functions and form a function set. We use the name of a root point function to name its function set (as listed in the UpSet figure). In theory, these function sets should correspond to different data read-in points, data processing tasks, and functionalities. And only in this way can prove our slice-based fuzzing method is reasonable. The less intersection of functions between different function sets, the higher the functional independence of each set.

Because the verification results of the four devices are similar, we take TP-Link WDR7660 as an example to illustrate their results. We provide the CDF diagram to simplify the presentation of the distribution of the function intersection between different functionality snippets. As Figure 1 shows, few of the functions (less than 25%) belong to more than one set. According to our analysis, the intersection mainly comes from two aspects. One is the functions (e.g., `recvfrom`, `memcpy`, and so on) of the standard library, and different functionalities will use them. The second is the similarity between different function sets. Although these function sets have different data entry points, they handle similar data packages, such as `bindRequestHandle` and `registerRequestHandle`. We also provide an UpSet figure (Due to the page limitation, we put it on Github[2]), a visualization technique for the quantitative analysis of sets, to show the details of the intersection between different function sets collected from the RTOS.

To further study the coupling metrics of modules in RTOS, we inspect the shared data among tasks in the above four embedded devices. The result shows that the ratio of global variables employed to share data among functionality snippets is low. We identified 14,592 global variables, 207 of which are used in the functionality snippets we identified, and only 27 global variables are shared by different snippets (less than 13%, listed in Figure 1). Furthermore, we check the data-sharing paradigm through shared keywords operated by several API couples (e.g., `set_env` and `get_env`), as presented in KARONTE [31] and SaTC [9]. This data-sharing paradigm is prevalent in embedded systems due to its convenience. We find no data sharing among functionality snippets in these four samples that utilize corresponding API couples.

[2] https://github.com/NSSL-SJTU/SFuzz/blob/main/motivation_upset.md

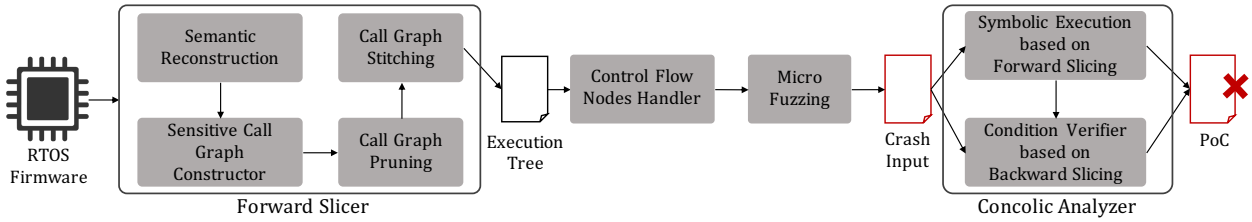


Figure 2: Overview of SFuzz. SFuzz takes the firmware of the real-time embedded devices as input and outputs their bug reports. It first recovers the semantic of the functions in RTOSes and uses the forward slicer and Control Flow Nodes handler to extract the code snippet related to the outside inputs. And then, it uses the emulation execution and coverage-based fuzzing to explore the paths in the sliced execution tree. Finally, it constructs the proof-of-concept based on the concolic analyzer.

The above analysis shows that these function sets collected from RTOS meet the characteristics of functional independence concerning control flow and data flow. Hence, the experiment result verifies the reasonability of the slice-based fuzzing method.

2.4 Challenges of Slice-based Fuzzing

To apply our method to RTOS of various embedded devices, we need to address three main challenges.

C1. How to determine the scope of the snippets? Firstly, we need methods to identify the data read-in functions no matter whether the target RTOS contains symbol files. Secondly, we need an approach to determine the scope of the snippet corresponding to a data receiving point. Constructing the functionality set via function calls will include some paths and functions unrelated to the processing logic for the input data. Meanwhile, some paths that cannot touch the sink function will also be included in the scope. Both of them will affect the efficiency of slice-based fuzzing.

C2. How to handle the points related to control flow in the snippet? Some function calls and conditional branches will affect the reachability of the execution path[3] and the efficiency of fuzzing. For example, some functions that the emulator cannot emulate will affect the path reachability. Another example, some compare instructions will use global variables or the variables unrelated to the input we are mutating. Because we cannot mutate the values of these variables via seed mutation, we cannot control the jump direction of the following branches of these compare instructions.

C3. How to effectively conduct slice-based fuzzing and verify the PoC? On the one hand, the pure fuzzing technique is hard to generate valid seeds to pass various kinds of condition guards without any prior knowledge of the application or input format [30, 35]. Similar to Driller [35], we combine fuzzing with symbolic execution to make path exploration more effective. On the other hand, because we conduct fuzzing on code snippets and preprocess some instructions related to control flow, we need to design a method to judge whether a proof of concept (PoC) results in a crash is a real vulnerability in the original RTOS.

3 DESIGN

Approach Overview. In this paper, we design SFuzz to address the above challenges and leverage the slice-based fuzzing technique to detect vulnerabilities in code snippets of RTOS of embedded devices. As Figure 2 shows, SFuzz takes the firmware of the real-time embedded devices as input and outputs their bug reports. It first recovers the semantic of the functions in RTOSes and uses the forward slicer and Control Flow Nodes handler to extract the code snippets related to the outside inputs. And then, it uses the slice-based fuzzing technique to explore the execution tree[4] of the code snippet. Finally, it constructs the proof-of-concept based on the concolic analyzer.

In detail, the forward slicer combines the call graph analysis with the forward taint analysis to determine the exploration space of each task for slice-based fuzzing; Control Flow Nodes Handler is used to help the following fuzzing part skip the unnecessary path exploration and those nodes which will make the fuzzing phase get stuck; the micro fuzzing engine is a hybrid greybox fuzzer, which combines some low-level techniques, such as error detection policies, to make the fuzzer could run smoothly and find errors; the Concolic Analyzer is mainly to help us filter false positives due to exploration pruning and context missing.

3.1 Forward Slicer

To conduct the slice-based fuzzing on functionality snippets of the target RTOS, we first recover the semantic of the critical functions in the firmware (details listed in §4) to locate the external data entry points (e.g., `recvfrom`), global data sharing functions (e.g., `nvram_get`) and vulnerable functions (e.g., `memcpy`), and then leverage the forward slicer module to output the execution tree related to handling the external input and global data.

Forward Slicer contains three parts, and its workflow is shown in Figure 2. The *Sensitive Call Graph Constructor* detects input obtaining functions (e.g., `recvfrom` in Listing 1) and global data read points, and then takes the callers of these functions as root nodes to build call graphs[5]. To make the fuzzing test focus on the vulnerable path, we prune the branches that cannot touch potential sink functions (i.e., `memcpy`, `strcpy`, `sprintf`, and so on) in the graphs. The *Call Graph Pruning* component further prunes the subgraphs or paths that are independent of external input. Finally, the *Call Graph Stitching* component splices some edges between

[3]Execution Path is a sequence of instructions executed by the target firmware from a data receiving point with a specific test case.

[4]Execution Tree merges all execution paths from one data receiving point taken as a root node, and every node in the tree represents an instruction forking a new path from the current path, such as branch and function call instructions.

[5]A call graph is a control-flow graph, which represents calling relationships between functions in a program. Each node represents a function and each edge (f, g) indicates that function f calls function g.

```
/*date set point*/
char *var = WebGetsVar(a1, "wanPPPoEUser");
nvram_set("wan0_pppoe_username", var);
/*data get point*/
sprintf(usename, "wan%d_pppoe_username", var);
char *var1 = nvram_get(usename);
```

Listing 2: Code Samples for Call Graph Stitching (dynamic method).

the nodes of different call graphs. These edges are missing while building these call graphs due to the lack of direct correlation.

Call Graph Pruning. To judge whether an external input or global data could affect the parameters of the potential sink functions, SFuzz leverages lightweight (coarse-grained) taint analysis technology to track each path (from the root node to the leaf node) in the call graph, determines the possible scope of the influence of the external input and global data, and filters the paths independent of them. Here we describe the high-level design of the taint engine. For each call path, the taint engine steps into the function body of each node in the path. It marks the parameters or return values of the input reception and parsing functions as taint sources based on the functions' semantic, such as the parameter `Global_addr+0x1c` of `recvfrom` in Listing 1, the memory space which points to is used to store the input data. While conducting the taint analysis on each instruction, the taint engine firstly translates instructions into intermediate instructions whose semantic are simpler than the assembly instructions of various disparate architectures. And then, for each intermediate instruction, it includes the output operand(s) of the instruction into the tainted operands set if the input operands are affected by the external input. For function call instruction whose callee does not belong to the call path, the taint engine will propagate the taint attribute from its tainted parameters to the return value. If the risky parameter(s) of the sink function (e.g., `count` for function `memcpy`(*dest, *src, count)) are affected by the input, SFuzz retains the corresponding call path.

Call Graph Stitching. As KARONTE [31] and SaTC [9] proposed, some data-flow of external input could be interrupted by data sharing paradigms (e.g., `set_env` and `get_env`). Unfortunately, RTOS also faces the same challenge. Unlike the previous approach, in addition to using static analysis to splice the deterministic associated nodes, we also use dynamic technique to detect the non-deterministic correlation between the data set and use point(s). For the data sharing paradigms (i.e., set and use point) labeled with constant strings, we search and match them based on the constant strings. Then connect the two call paths and use a virtual node (i.e., two-tuples, such as <`nvram_set`, `nvram_get`>) to represent the paradigm in the merged call graph. For the paradigms marked by dynamically created variables, such as "wan%d_pppoe_username" in Listing 2, we get these paradigms based on the approximate string matching method and create a virtual condition node to connect the potential data sharing paradigms. And then determine whether jump to the global data read point based on the actual value of the variable during emulation execution. For the set points that have many corresponding get points, we also build a virtual condition node and determine the jump direction based on random probability.

3.2 Control Flow Nodes Handler

After the handling of the Forward Slicer module, we could build an execution tree of the target code snippet based on the call graph. However, to make the fuzzing test on the execution tree work smoothly and avoid unnecessary path exploration, we still need to handle several types of instructions related to control flow (i.e., (i) function call, (ii) conditional branch), which will affect the reachability of the execution path and the efficiency of the test. In other words, because of lacking full context and runtime state of the RTOS, we need strategies to guide the fuzzer to determine how to handle the function call in the snippet and choose which branch of the conditional statement to jump.

Call Instruction. On the one hand, we add the address of the function call instruction, whose callee's parameters are not affected by the external input, into `PatchedFunc` set and guide the fuzzer to skip the function call. We do this mainly because its arguments are unrelated to the input, and its return value and parameters will not be changed via mutating the seed input. Hence, patching this kind of function will help the fuzzer ignore its complexity and increase the fuzzing efficiency. On the other hand, for all the function calls that belong to the sensitive call graphs or whose arguments are affected by the input, such as `protocol_handler` and `header_check` in Listing 1, we keep them. Only by stepping into these functions can we ensure the reachability of the sensitive path.

Conditional Branch. A conditional statement directs the control flow to a target address only if the specified branch constraint is satisfied. However, while conducting fuzzing on code snippets, we have to face a problem determining the direction of the conditional jump if the condition has no relationship with the input data, which means we cannot change the jump direction by mutating the input. To make the fuzzing test more effective and sound, we propose several methods for handling various conditional branches.

- Assuming only one branch of a conditional jump instruction is reachable to the sink function. If its condition is affected by the input, we insert the target address of the unreachable branch to the `PatchedJMP` set, which guides the fuzzer to avoid exploring this branch. Otherwise, we add the address of the jump instruction into `PatchedJMP` set and guide the fuzzer to replace the conditional jump with the fixed jump to the reachable branch.
- Suppose both branches of a conditional statement are reachable to the sink functions. If its constraint is not related to the input data, we add the address of the instruction into the `PatchedJMP` set, which guides the fuzzer to replace the instruction with a random jump statement. Otherwise, we do not change the code. It is mainly to help us explore as many paths that are not determined by the input as possible.
- If no branch of a conditional jump instruction is reachable to the sink function, we add the target addresses of the two branches to the `PatchedJMP` set, which will guide the fuzzer to exit the current path exploration when encountering these addresses.

3.3 Micro Fuzzing

As the core of our fuzzing engine, we name the slice-based fuzzing technique as *Micro Fuzzing*. It takes code snippets as input, explores paths in the execution tree, and ignores irrelevant call sites and other input data handlers. The engine simultaneously inspects the

context of sink function call sites and exports crash input when memory access against pre-defined policies.

Image Loader. After loading the RTOS firmware, the image loader will preprocess the tailored code snippets marked by the previous module. For the call instruction should be skipped (in `PatchedFuncset`), the image loader replaces the call instruction with NOP-like instruction. For the branch that should avoid being explored (in `PatchedJMPset`), SFuzz adds the `AvoidExplore` statements to its target address. When the program executes to the corresponding address, the program will exit the current path exploration directly. For the branch which should be replaced with a fixed jump or random jump, SFuzz handles it with the corresponding operation.

Fuzzing Engine. When the core engine is invoked, it loads the RTOS system and repeatedly executes the target code snippets from its beginning (the root node of the execution tree). This engine will generate random data in input entry points. Based on UnicornAFL, this engine can perform coverage-guided fuzzing and emulate instructions execution on this tailored execution tree. When the core fuzzing engine gets stuck, the hybrid fuzzer invokes its concolic execution component. This component randomly selects an input from the seed queue and symbolizes each byte in the input. After tracing the execution path corresponding to the input, the concolic execution component utilizes its constraint-solving engine to identify inputs that would force execution down previously unexplored paths. The fuzzing engine will exit if no new path is found within a threshold time.

By pruning unnecessary paths, Micro Fuzzing ignores input-irrelevant function calls (in `PatchedFuncset`) and executes NOP-like instruction instead. On the other hand, it directly skips emulation-hard instructions, avoids marked condition branches (in `PatchedJMPset`), and allocates concrete data when pointer reference to uninitialized memory. The emulation-hard instructions are usually related to interrupts interacting with hardware or HAL (hardware abstraction layer) modules (e.g., send a signal to the CPU scheduler) and do not affect the data-flow starting from input sources. Thus, skipping these instructions in emulation brings fewer side effects. Finally, these two ways make the emulation more stable and focus on exploring the code snippets handling the target input data.

Memory Safety Policies. Because bare-metal [32] and RTOS devices often lack memory sanitizer mechanisms due to cost sensitivity and resource constraints. Thus, SFuzz needs to provide lightweight memory safety check policies for the fuzzing engine, which define the violation of memory access in call sites of sink function. Here we focus on detecting the bugs that occur in memory buffer operation. We classify memory buffer into two categories: the buffer that can be statically analyzed to determine the size (including the buffer on the stack, the buffer created via a `malloc`-like function, the global variable that can be identified its size based on adjacent variables, etc.), and the buffer that cannot determine its size statically. For the buffer whose size can be statically identified (SFuzz conducts the analysis in the forward slicer module), we determine whether an overflow has occurred by detecting whether the buffer boundary data has been modified after executing the sink function. For the buffer whose size cannot be determined, we directly output an alarm and further identify the buffer size in the subsequent concolic analyzer module.

Algorithm 1 The workflow of Concolic Analyzer.

```
1:  function ConcolicAnalyzer(CrashInput, RTOSProject)
2:      Trace ← Tracer(CrashInput, RTOSProject)
3:      TargetSink ← GetSinkPoint(Trace)
4:      CompletePoC ← ∅
5:      State ← SimulationStart(RTOSProject)
6:      State.AddConcreteConstraints(CrashInput)
7:      while State.active do                          ▷ State still satisfies all constraints
8:          if isTargetSinkFunc(State, TargetSink) then
9:              StateConstraints ← BackwardSlicing(State).Constraints()
10:             for constraint ∈ StateConstraints do
11:                 if ReSymexec(RTOSProject, constraint.invert(), CrashInput, TargetSink) then
12:                     StateConstraints.remove(constraint)
13:                 end if
14:             end for
15:             if SinkBufferDeterminable(StateConstraints) then
16:                 OutputCompletePoC(StateConstraints, CrashInput, RTOSProject)
17:                 return
18:             end if
19:         else
20:             if State ∈ PatchedFuncset then
21:                 SymValues ← SymRetValue(State) ⋃ SymArgsValue(State)
22:                 State.AddNewSymbols(SymValues)
23:             else if State ∈ PatchedJMPset then
24:                 State.SetJumpDirection(Trace)
25:             end if
26:         end if
27:         State.Step()                               ▷ Step to next concolic state
28:     end while
29:     OutputFailedInfo(CrashInput, RTOSProject)      ▷ State cannot reach sink
30: end function
```

3.4 Concolic Analyzer

When the *Micro Fuzzing* engine exits, *Concolic Analyzer* will check all crash inputs that have triggered violations. It takes these cases as concrete inputs and conducts concolic execution in the corresponding execution paths for constraint solving.

As the pruned function call instructions (in `PatchedFuncset`) and conditional branches (in `PatchedJMPset`) might be misleading control flow in the original execution tree, thus we need to check whether a crash input could trigger a real vulnerability in the original RTOS. To conduct this check, Concolic Analyzer first recovers these branches and symbolizes parameters and return values in these patched functions call sites, and then performs concolic testing based on forward and backward slicing.

The workflow is shown in Algorithm 1. In detail, the *forward slicing-based concolic testing* part takes a crash input as the concrete value to perform concolic execution on the path triggered by the input. If a function call site in the path belongs to `PatchedFuncset`, it will apply new symbols to the function's arguments and return value. It collects all constraints along with the execution path until reaching the sink function. For constraints related to the other input data entry points or patched functions, the *backward slicing-based condition verifier* flips each constraint once at a time and reruns the symbolic engine from the beginning with the given inverted constraint. If the symbolic engine can still reach the sink function, the corresponding constraint is judged as non-required for the current PoC. For constraints related to sink buffer size that has not been determined in former modules, the verifier calculates its value and only alerts when sink buffer size can be determined and is overflowed by input data.

We use a real-world sample (CVE-2021-32186) to present how the concolic analyzer works. As Listing 3 shows, `nvram_set` and

```
1  void vulnSet(webRequest* a1, webRequestData* a2)
2  {
3    char *ledStatus;
4    char *ledClsTime;
5    char *ledTime;
6    char argbuf[0x100];
7    int ledCtlType;
8    ledClsTime = webVar(a1, "LEDCloseTime");// Input
9    ledStatus = webVar(a1, "LEDStatus");    // Other input #1
10   ledCtlType = nvram_get("led_ctl_type"); // Other input #2
11   if (strcmp(ledCtlType, ledStatus))
12     nvram_set("led_ctl_type", ledStatus);
13   if (!strcmp("2", ledStatus) ) {
14     ledTime = nvram_get("led_time");      // Other input #3
15     sub_800D487C(a2, argbuf);
16     if (strcmp(ledTime, ledClsTime))
17       nvram_set("led_time", ledClsTime);  // Global data set
18   }
19 }
20 void vulnGet()
21 {
22   char v8[64];
23   memset(v8,0,sizeof(v8));                // Written object
24   ledTime = nvram_get("led_time");        // Global data get
25   strcpy(v8, ledTime);                    // Sink
26 }
```

Listing 3: Code Sample for Concolic Analyzer

`nvram_get` construct a data sharing paradigm. The data source labeled by `"LEDCloseTime"` is passed to NVRAM (Non-volatile random-access memory) when another data labeled by `"LEDStatus"` is set to `"2"`, and then can trigger a stack buffer overflow in `vulnGet`.

Forward slicing-based concolic testing. After getting a crash input, Micro Fuzzing also outputs the corresponding execution path, including the input entry point and the sink function call site. As Listing 3 shows, the entry point is in Line 8, and due to this slice being stitched by data sharing paradigms, the sink point in current function `vulnSet` is `nvram_set` (Line 17). Thus, we perform concolic testing from Line 8 and check execution path conditions in Line 17. In this process, the symbolic execution engine collects path constraints, including the symbolic expression of parameters and return values from other input data reading functions in Line 9, 10, and 14. At last, the path conditions (Line 17) contain all these other input data because they respectively constitute branch conditions in Line 11, 13, and 16.

Backward slicing-based condition verifier. Although an execution path constraint could present a specific constraint set that every input data should satisfy, it still lacks critical information in two parts: first, whether the constraints on other input data are necessary or not, which will bring false negative in bug detection; second, the size constraint of the object written by the sink function, which will bring false positive and false negative. Accordingly, we solve problems in two aspects:

- As `vulnSet` in Listing 3 shows, backward slicing starts from the sink function in Line 17 and backward tracks the execution path. By inspecting the execution path condition in the sink function, we extract constraints that include other input data and locate corresponding source positions as endpoints of backward slicing in Line 9, 10, and 14. We invert these constraints and rerun the symbolic execution individually, checking satisfiability by symbolic execution from the endpoint of corresponding backward slicing. If the state could still reach the sink function, the constraints are proven to be unnecessary for sink reaching. At last, we can remove non-essential constraints (Line 11) and contain necessary conditions (Line 13 and 16).
- As `vulnGet` in Listing 3 shows, backward slicing start from the sink function in Line 25 and backtracks the execution tree. In this process, backtracking focuses on the current function scope, captures any memory handler function related to this written object, and sets the farthest call site as the ending (Line 23). At last, symbolic execution starts from the endpoint and solves constraints of allocation size by leveraging function semantics of relevant handlers.

4 IMPLEMENTATION

We implement the prototype system of SFuzz with around 6,200 lines of Python code, 4,300 lines of C code, and 5,100 lines of Java code. The taint analysis module and semantic recovery part are implemented based on Ghidra [17]. The fuzzing engine is built based on UnicornAFL and Driller [35], and the concolic analyzer is implemented based on Angr [36]. We extended Driller to make it work for RTOS images, including re-implementing its trace logger based on our self-designed RTOS loader to track the execution trace of the target code snippets. Our system is based on several basic procedures as follows:

Image Extraction. We leverage strings embedding in the firmware to identify the type of RTOS (e.g., VxWorks 5.5.1) and leverage BinWalk to extract the content of the RTOS image. Meanwhile, for disassembling the content, we use the feature of the machine code in the image to determine the type of CPU architecture (e.g., MIPS).

Base Address Recognition. Because many data reference or function call operations in RTOS systems are dependent on the base address, and wrong addresses will result in incorrect data references or control flow jumps. We implemented this part based on the core idea that *only the correct base address can link the most data reference pointers with the intended targets*. The method is proposed in Vxhunter [45] and used in some related works, such as FirmXRay [37]. This module contains two steps to recognize the base address. It first identifies and extracts the data reference pointers from the system; secondly, it matches the absolute address of the data pointers with the intended targets. It should be claimed that (i) we only use string pointers to help recognize the based address, which is good enough (as the result in §5.4 shows); (ii) we implement this method based on PCode, which is Ghidra's intermediate representation for assembly language instructions, instead of the instructions of a specific architecture. Therefore, it could support more architecture.

Function Semantic Reconstruction. As explained in §3.1, we need function semantic to guide the taint analysis and locates the sensitive snippets. SFuzz mainly recovers the semantic and functionalities of three types of functions: (i) the functions that receive, parse or share the external input data (i.e., user input); (ii) the sink functions (e.g., memcpy); (iii) the functions that set or get global data. We implement four methods to automatically recover the function semantics of functions and identify sensitive functions.

- **Symbol File & Log Function**. According to our analysis, some vendors (e.g., TP-Link and MERCURY) will release the symbol files that label the name of the functions; meanwhile, the log

Table 1: Dataset of device samples. We selected 35 device samples from 11 vendors, including router, firewall, printer, switch and BCI on four architectures. Size represents the sum size of the samples collected from the corresponding vendor before unpacking.

Vendor	Type	Series	OS	#	Size	Architecture
Sonicwall	Firewall	TZ/SOHO	VxWorks	2	153M	MIPS(BE)
RIOCH	Printer	SP/AficioSP	VxWorks	4	41M	ARM(LE)
Xerox	Printer	WC/Phaser	VxWorks	4	66M	ARM(LE/BE)/MIPS(BE)
CISCO	Switch	SG	VxWorks	1	7M	ARM(LE)
Linksys	Switch	LG	VxWorks	1	7M	ARM(LE)
Tenda	Router	AC	eCos	7	14M	MIPS(LE/BE)
FAST	Router	FAC/FW	eCos	3	4M	MIPS(LE)
MERCURY	Router	MW/M/D	VxWorks	3	6M	ARM/MIPS(LE)
TP-Link	Router	WDR	VxWorks	6	11M	ARM(LE)/MIPS(BE)
D-Link	Router	DIR	eCos	3	3M	MIPS(LE/BE)
Vendor*	BCI	BCI_V1	FreeRTOS	1	2M	ARM LE
Total	**5**	**17**	**3**	**35**	**314**	**4**

functions used to output the runtime error are also can be used to recover the function names. According to our analysis, some vendors (e.g., TP-Link and MERCURY) retain symbol files that can label the names of the functions in the firmware. Meanwhile, some vendors use log functions to output the runtime error, which can also help recover the function names, such as `devDiscover` in the statement `logOutput`(ostream, "devDiscover: error, ret = %d", retcode).

- **Virtual Execution**. Firstly, the method compares the number of the target function's arguments and return value with the standard library functions to find the potential matching functions, such as strcpy. Secondly, it allocates memory space, initializes the state of registers, and sets the initial values of the arguments for the function. At last, it simulates code in the function body and determines the matching function by analyzing the value of output and the memory space the simulation affects. We leverage this method to recognize the standard library functions, such as `memcpy` and `printf`.
- **Web Service Semantic**. We leverage the shared strings used to mark the user input both in the front-end files (e.g., HTML, PHP, and JavaScript) and back-end files to recover the semantic of some functions related to the web services. We implemented this approach based on the method proposed in SaTC [9].
- **Open source firmware**. Some vendors select open-source RTOS projects to build their systems, such as eCos and FreeRTOS. For these types of systems, after identifying their version while preprocessing the firmware, we can leverage B2SFinder [42] and some other tools [41] to match the functions in the firmware with the functions in the open-source projects based on strings, immediate, and other explicit features embedded in the code.

5 EVALUATION

For evaluation of our approach, we should answer the following research questions:

- **RQ1.** Can SFuzz discover real-world vulnerabilities in RTOS of embedded devices? (§5.1)
- **RQ2.** Whether each part of SFuzz is necessary for effectively discovering bugs in RTOS? Compared with state-of-the-art tools, how does our tool perform? (§5.2 and §5.3)
- **RQ3.** Is SFuzz accurate and efficient for vulnerability discovery in each step? (§5.4)

Table 2: Experiment configurations. ✔ indicates feature enabled; ✗ means feature disabled. `FuncCall` represents function call Instruction, `CBranch` represents conditional branch.

Experiment	Coverage-Guided Fuzzing	Symbolic Execution	Handler for FuncCall	Handler for CBranch
SFuzz	✔	✔	✔	✔
SFuzz-Handler	✔	✔	✗	✗
SFuzz-FHandler	✔	✔	✗	✔
SFuzz-CHandler	✔	✔	✔	✗
UnicornAFL	✔	✗	✗	✗

Dataset. As shown in Table 1, we collected 35 firmware samples from 17 series in 11 vendors. These devices cover three RTOS types and supply various services, including 23 routers, seven printers, two firewalls, two switches, and one BCI (Brain-Computer Interface). Among these samples, seven devices adopt the MIPS-BE architecture, 13 adopt the MIPS-LE architecture, and one adopts the ARM-BE architecture, while the other 14 use the ARM-LE architecture. On average, each firmware is 9 megabytes, and SFuzz processed up to 314 megabytes in total.

Environment Setup. Our experiments run on a Ubuntu 18.04 host with a RAM of 256 GB and a 32-core Intel Xeon Processor of 2.4 GHz. Especially, we set the time limit for the fuzzing part of each experiment to be six hours when handling one data entry point and operated each experiment with one CPU core. According to our observation, none of the experiments could find new paths or crashes after this timeline.

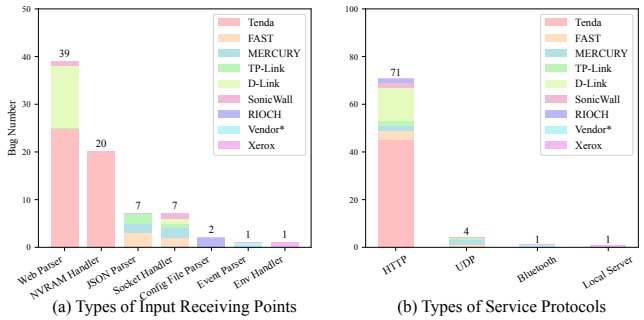
Experiments Design. To answer the `RQ2`, we design five experiments that test the selected RTOS with different configurations, as shown in Table 2. SFuzz is a full-featured fuzzer, which utilizes the handler for function calls and conditional jump instructions, and the symbolic execution engine to boost the fuzzer. SFuzz-Handler does not use the *control flow node handler* (§3.2) to process the critical nodes related to control flow. SFuzz-FHandler only handles the conditional branch statements, while SFuzz-CHandler merely handles the function call instructions. The last experiment is conducted using the existing state-of-the-art fuzzer for code fragments, `UnicornAFL` [25], and we improved its program loader to make it work for various RTOS firmware in our dataset. Note that the vanilla greybox fuzzer (e.g., UnicornAFL) and other methods (e.g., SFuzz) are deployed on the same original execution tree[6] from the beginning point. Then, each method applies tailored strategies according to their respective principles.

Bug Confirmation. Each alert produced by SFuzz contains a unique crash input from the source point and symbolic expressions for the path constraint, which may include other data sources or global variables. We manually verified each alert, and only it can result in a real bug we consider it is a vulnerability.

5.1 Real-world Vulnerabilities

SFuzz found 77 new bugs[7] in 20 firmware samples of different devices, including router, printer, firewall, and BCI. By the time of submission, 68 of them have been confirmed by the vendors, and

[6] All paths that begin from an external data entry point (i.e., an instruction) in the target RTOS form one original execution tree.
[7] All bugs are listed at https://github.com/NSSL-SJTU/SFuzz/blob/main/rw_vuls.md.



(a) Types of Input Receiving Points (b) Types of Service Protocols

Figure 3: Statistics of Input Receiving Points types and Service Protocols types corresponding to the real-world vulnerabilities.

67 have been assigned CVE or CNVD IDs (46 CVE and 21 CNVD, 64 are high severity); 9 bugs are still waiting for responses from the vendors. Figure 3 shows the types of data receiving points and tasks of the snippets corresponding to these bugs. In detail, these bugs exist in many different tasks, such as HTTP, UDP, and Bluetooth. Moreover, the input data comes from several data sources, such as Web parser, NVRAM handler, and Socket handler. Additionally, we list detailed case studies of revealed bugs in our dataset[8].

5.2 Comparison with Existing Methods

To understand the contribution of SFuzz's every component for fuzzing result and performance, and analyze the actual performance of our tool compared to other "similar" tools, we design this experiment. Because all these works cannot be directly applied to RTOSes and are difficult to migrate to RTOSes, we simulate these works, as experiment setup part shows. The strategies used in SFuzz-FHandler are a superset of the strategies used in T-Fuzz [26], and T-Fuzz is not entirely open-source and cannot be used on code snippets of RTOS. Hence, here we use SFuzz-FHandler to represent T-Fuzz. Meanwhile, SFuzz-Handler equals to a version of Driller [35] for code fragment execution. We feed all these tools with the same execution trees collected from five devices and start them from the same source points that read the data package. According to the experiments on five tools, we compare their performance in three aspects: effectiveness, stability, and efficiency.

Effectiveness. As shown in Table 3, all 5 tools can find vulnerabilities in real devices to a certain extent, ranging from 4 to 34. Both UnicornAFL and SFuzz-Handler can only find bugs from 4 execution trees, and they only explored 1,390 and 1,366 execution paths. When we applied `Cbranch` patch on the basis of SFuzz-Handler (i.e., SFuzz-FHandler or T-Fuzz), we can see that although the number of explored paths has increased by 149% to 3,397, the actual number of bugs that can be found is basically the same. When we applied `FuncCall` patch on the basis of SFuzz-Handler (i.e., SFuzz-CHandler), although path exploration only increased by 23%, 17 bugs were found on 19 execution trees. Our complete method combines the above modes and can finally trigger up to 134 unique crashes on 105 execution trees on 5 models and discover 34 bugs. And SFuzz outperforms all compared tools.

[8] https://github.com/NSSL-SJTU/SFuzz/blob/main/cases

Table 3: Compared with other tools. `#Path` represents the number of paths that can be discovered by each tool. `#ExecTree` represents the number of the execution trees that contain crashes. `#CrashInput` represents the number of inputs resulting in unique crashes. `#Bug` represents the number of real-world bugs that can be discovered.

Mode	Device	#Path	#ExecTree	#CrashInput	#Bug
UnicornAFL	Tenda AC11	269	3	3	3
	TP-Link WDR7660	206	0	0	0
	RICOH SP221	26	0	0	0
	FAST_FAC1200R_Q	838	0	0	0
	MERCURY_M6G	51	1	2	1
	Total	**1,390**	**4**	**5**	**4**
SFuzz-Handler	Tenda AC11	269	3	3	3
	TP-Link WDR7660	211	0	0	0
	RICOH SP221	26	0	0	0
	FAST_FAC1200R_Q	814	0	0	0
	MERCURY_M6G	46	1	3	1
	Total	**1,366**	**4**	**6**	**4**
SFuzz-FHandler	Tenda AC11	1,629	4	5	4
	TP-Link WDR7660	462	0	0	0
	RICOH SP221	120	1	6	1
	FAST_FAC1200R_Q	1,149	0	0	0
	MERCURY_M6G	37	1	4	1
	Total	**3,397**	**6**	**15**	**6**
SFuzz-CHandler	Tenda AC11	409	15	18	13
	TP-Link WDR7660	234	2	5	2
	RICOH SP221	25	0	0	0
	FAST_FAC1200R_Q	947	1	1	1
	MERCURY_M6G	64	1	3	1
	Total	**1,679**	**19**	**27**	**17**
SFuzz	Tenda AC11	1,841	34	46	25
	TP-Link WDR7660	754	19	19	2
	RICOH SP221	141	1	6	2
	FAST_FAC1200R_Q	1,364	46	57	3
	MERCURY_M6G	79	5	6	2
	Total	**4,179**	**105**	**134**	**34**

Stability. To compare the stability among these tools, we inspect the successful simulation ratio among all fuzzing executions on five devices. As shown in the results[9], different devices have different stability variations among tools, but we still find that FHandler (handles the function call instructions) improves the stability greatly. In detail, the stability of Tenda AC11 has increased from 6.02% (SFuzz-Handler) to 97.69% (SFuzz-CHandler), and TP-Link WDR7660 has increased from 27.98% (SFuzz-Handler) to 64.62% (SFuzz-CHandler). On the other hand, CHandler (handles the conditional branches) brings some lightly adverse effects. Thus, SFuzz applies both FHandler and CHandler and can maintain satisfactory stability.

Efficiency. When checking time consumption in these five tools, we find that the more complicated methods are applied, the more time is spent on testing. The experiment result[10] demonstrates the average fuzzing time different tools spend on each device. In detail, UnicornAFL, SFuzz-Handler, SFuzz-FHandler, SFuzz-CHandler, and SFuzz take 293s, 723s, 909s, 1,006s, and 1,049s on average in fuzz testing on one execution tree of one device, respectively. It shows that SFuzz spends more time but maintains an acceptable range.

[9] https://github.com/NSSL-SJTU/SFuzz/blob/main/success_rate.md.
[10] https://github.com/NSSL-SJTU/SFuzz/blob/main/avg_time.md.

Table 4: Compared with traditional symbolic execution. SE means symbolic execution tool, SE+Handler represents the combination of symbolic execution tool and control flow node handler.

Device	Bug IDs	Time-to-Exposure (TTE)		
		SFuzz	SE	SE+Handler
Tenda AC11	CVE-2021-31755	7min42s	42min15s	18min44s
	CVE-2021-31756	44min08s	✗	✗
	CVE-2021-31757	1min42s	8min35s	6min09s
	CVE-2021-31758	40min42s	✗	87min21s
	CVE-2021-32180	4min21s	✗	19min19s
	CVE-2021-32181	45min25s	✗	✗
	CVE-2021-32186	16min13s	243min14s	130min12s
	CVE-2021-32187	45min28s	✗	✗
	CVE-2021-32188	54min56s	✗	✗
	CVE-2021-32189	40min05s	✗	141min51s
	CVE-2021-32190	44min30s	✗	✗
	CVE-2021-32191	3min18s	28min31s	11min07s
	CVE-2021-32192	32min57s	✗	81min05s
	CVE-2021-34100	33min03s	✗	62min04s
	CVE-2021-34102	59min41s	✗	✗
	CNVD-2021-33391	45min28s	✗	✗
	Reported Issue 1	45min34s	✗	✗
	Reported Issue 2	55min40s	✗	✗
	Reported Issue 3	8min19s	✗	29min28s
	Reported Issue 4	42min12s	✗	✗
	Reported Issue 5	7min37s	38min14s	19min55s
	Reported Issue 6	27min33s	✗	50min51s
	Reported Issue 7	7min48s	✗	12min20s
	Reported Issue 8	56min29s	✗	✗
	Reported Issue 9	46min17s	✗	✗
	# Path	2,484	491	389
TP-Link WDR7660	CVE-2020-28877	26min45s	✗	190min57s
	CNVD-2021-39128	26min18s	✗	217min20s
	# Path	235	40	83
RICOH SP 221	CNVD-2021-42364	22min28s	✗	✗
	CNVD-2021-42365	10min12s	178min12s	97min04s
	# Path	122	95	38
FAST FAC1200R_Q	CVE-2021-33374	43min45s	✗	142min20s
	CNVD-2021-39243	4min36s	212min53s	16min05s
	CNVD-2021-39287	31min13s	✗	✗
	# Path	815	143	411
MERCURY M6G	CNVD-2021-39284	33min17s	✗	✗
	CNVD-2021-41097	16min11s	226min24s	46min54s
	# Path	53	107	72

5.3 Comparison with Symbolic Execution

We conduct comprehensive experiments to compare SFuzz with traditional symbolic execution (SE) techniques. Since no existing SE tools can directly test RTOSes, we implement a prototype by ourselves based on Angr [36], a popular and well-maintained SE tool. We list the results in Table 4. The result shows that the slicing method significantly boosts the efficiency of bug discovery. We also add the Control Flow Nodes Handler of SFuzz to the SE tool. The result shows that the handler can boost the performance of SE on bug discovery and path exploration.

Bug number. SFuzz can find 34 bugs in five vendors' samples. Symbolic execution (SE) only reveals eight bugs. With the help of the Control Flow Nodes Handler, SE+Handler can discover 19 bugs.

Time-to-Expose. SFuzz reveals bugs in a shorter time than the other methods. For Tenda samples, SFuzz reveals one bug in 33 mins, SE+Handler takes 52 mins, and SE needs 72 mins; For TP-Link, SFuzz discovers one bug in 27 mins, SE+Handler takes 204 mins, and SE cannot find any bug; For RICOH samples, SFuzz reveals one bug in 16 mins, SE+Handler takes 97 mins, and SE needs 178 mins; For FAST, SFuzz finds one bug in 27 mins, SE+Handler takes 79 mins, and SE needs 213 mins; For MERCURY, SFuzz finds one bug in 38 mins, SE+Handler takes 47 mins, and SE needs 226 mins.

Path exploration. Our slicing method helps fetch more paths than others and is more suitable in the fuzzing scenario than in symbolic execution. For Tenda samples, SFuzz can explore 2,484, SE+Handler executes 389, and SE runs 491 paths; For TP-Link, SFuzz can explore 235, SE+Handler performs 83, and SE only runs 40 paths; For RICOH, SFuzz can explore 122, SE+Handler runs 38, and SE runs 95; For Fast, SFuzz can explore 815, SE+Handler runs 411, and SE only executes 143; For MERCURY, SFuzz can explore 53, SE+Handler runs 72, and SE runs 107. In total, SFuzz explores 3,709, SE+Handler executes 993, and SE only runs 876 paths in these samples. It should be noted that SFuzz runs fewer paths in MERCURY because it gets bugs in quite a shorter time than others.

5.4 Accuracy and Efficiency

In this section, we evaluate the accuracy and efficiency of each part of SFuzz, including Forward Slicer (i.e., Semantic Reconstruction and Forward Slicing), Micro Fuzzing, and Concolic Analyzer.

Semantic Reconstruction. Among our dataset, 31 samples can be analyzed by SFuzz. The base address recognition model can correctly recognize the base addresses of 25 firmware samples. And six models (FAST FW325R, FAST FW313R, MERCURY MW325R, TP-Link WDR5660, Tenda AC5, and Tenda AC6V2) cannot be recognized automatically, and their base addresses are determined by manual analysis. Through verification using symbol tables and manual effort, we found that most of the semantics automatically recovered by SFuzz are accurate, and the cross-validation accuracy rate is more than 90%. In detail, the semantics of seven models were recovered via the symbol file recovery method. The web service semantic recovery method recognized the semantics of web input functions of seven Tenda devices. The virtual execution method can be used to restore the semantics of 24 samples. Eight models use the log function patterns to restore their function semantics. Especially in RICOH-SP330 (a printer), SFuzz finds only one user-input data reading function (i.e., `os_file_get`). Hence it only extracts one corresponding sensitive call graph. Additionally, we present a list of all revealed *Input Sources* and *Sink Functions* in our dataset[11].

Forward Slicer. To understand the accuracy of the slicing method, we need to check whether this method leads to missing sensitive data-flow influence based on global variables, which are referenced in the scope of patched functions handled by the control flow code handler. Thus, we measure the possible impact on the data flow and control flow, and specifically, we count how much the patched function can affect the input data and branch conditions through global variables. Finally, we count each proportion to the patched functions. As shown in Table 6, the two ratios in these models all maintain a low range (4.67% and 4.33%), meaning the possibility of sensitive data-flow leakage through global variables is sustainable.

To review the efficiency of the slicing method, we need to compare the size of our slices with the entire binary and check how many function call instructions and condition branches are handled in our slices, and in these handled positions, how many sites will be

[11] https://github.com/NSSL-SJTU/SFuzz/blob/main/source_sink.md

Table 5: Performance of the static analysis part. `#CG` represents the number of input-related call graphs. `Rate.Func` represents the ratio of the number of functions in the call graphs to the total functions. `Rate.Call` represents the proportion of the function call instructions handled by SFuzz in call graphs. `Rate.CJmp` represents the proportion of the condition branches handled by SFuzz in call graphs. `TRate.Call` represents the proportion of the function call instructions (handled by SFuzz) triggered in fuzzing. `TRate.CJmp` represents the proportion of the condition branches (handled by SFuzz) triggered in fuzzing.

Vendor	Model	#CG	Rate.Func	Rate.Call	Rate.CJmp	TRate.Call	TRate.CJmp	Time
D-Link	DIR100	8	1%(34/2507)	84%(158/188)	20%(39/197)	29%(46/158)	14%(1/7)	22.4
D-Link	DIR613	62	6%(227/4059)	95%(7853/8269)	6%(299/5302)	12%(943/7853)	13%(6/45)	3304.7
FAST	FAC1200R_Q	71	3%(307/9719)	74%(2559/3449)	50%(1054/2116)	30%(766/2559)	13%(58/437)	945.9
MERCURY	M6G	7	1%(138/11588)	76%(613/806)	24%(173/720)	9%(58/613)	7%(8/120)	309.4
RICOH	SP 221	6	2%(421/19134)	65%(1497/2313)	27%(732/2718)	2%(30/1497)	0%(0/59)	910.3
RICOH	SP 330	1	0%(27/36112)	87%(65/75)	12%(6/49)	22%(14/65)	0%(0/2)	31
TP-Link	WDR7660	27	2%(151/9425)	90%(1078/1196)	44%(304/684)	10%(112/1078)	15%(15/100)	2459.8
TP-Link	WDR7661	26	2%(144/9152)	90%(1048/1159)	44%(291/662)	11%(120/1048)	16%(15/94)	2411.5
Tenda	AC6V2	44	2%(156/7164)	94%(4416/4685)	9%(216/2411)	33%(1446/4416)	29%(16/56)	789.8
Tenda	AC8	42	2%(150/8531)	95%(4304/4550)	9%(202/2301)	41%(1750/4304)	22%(12/55)	641
Tenda	AC11	42	2%(156/8554)	95%(4443/4697)	9%(210/2402)	37%(1642/4443)	22%(13/58)	976.5
Average	-	**31**	**1.52%**	**89.32%**	**18.02%**	**24.71%**	**13.94%**	**1163.8**

Table 6: Data-flow influence of global variables. `#Input` represents the num of functions by `FuncCall` patched that can impact input data through global variables. `#Branch` represents the num of functions by `FuncCall` patched that can impact condition branches through global variables. `#Patched` represents the total number of functions by `FuncCall` patched. `Rate.Input` represents the proportion of patched functions that impact input data to all patched functions. `Rate.Branch` represents the proportion of patched functions that impact branch conditions to all patched functions.

Vendor	Model	#			Rate.	
		Input	Branch	Patched	Input	Branch
D-Link	DIR100	0	0	158	0%	0%
	DIR613	14	9	7,853	0%	0%
FAST	FAC1200R	216	200	2,559	8%	8%
MERCURY	M6G	115	115	613	19%	19%
RICOH	SP 221	52	47	1,497	3%	3%
	SP 3500SF	14	14	301	5%	5%
TP-Link	WDR7660	64	52	1,078	6%	5%
	WDR7661	63	51	1,048	6%	5%
Tenda	AC6V2	59	51	4,416	1%	1%
	AC8	54	47	4,304	1%	1%
	AC11	61	53	4,443	1%	1%
Average	-	-	-	-	**4.67%**	**4.33%**

triggered in the following fuzzing process. As shown in Table 5, the ratio of the number of functions in the sliced call graphs to the total functions is 1.52% on average. Thus, it shows that our slices are small enough to save analysis effort. In these call graphs, the proportion of the function call instructions and condition branches handled is 89.32% and 18.02% on average. Moreover, 24.71% and 13.9% of handled call instructions and condition branches are triggered in the subsequent fuzzing process. Thus, it proves these pruned sites are necessary and indeed make efforts in the following process.

Micro Fuzzing. In Table 7, the Forward Slicer of our tool can find 340 unique execution trees that could introduce bugs among 11 different models. The Micro Fuzzing engine identifies 245 vulnerable sink functions and constructs crash inputs corresponding to these potential bugs. The average analysis time on one execution tree varies from less than 7 minutes to over half an hour. And the total analysis time for one model ranges from 20 minutes to 21 hours, depending on the complexity of the execution trees to explore.

Table 7: The result of Micro Fuzzing. `#Tree` represents the number of execution trees which are found by the Forward Slicer. `#VSink` represents the number of vulnerable sink function call sites identified by Micro Fuzzing. `Avg.Time` represents the average time Micro fuzzing spends on one execution tree. `Total Time` represents the total time Micro fuzzing spends on one model. `Total Paths` represents the number of all explored execution paths.

Vendor	Model	#Tree	#VSink	Avg. Time	Total Time	Total Paths
D-Link	DIR100	8	7	658.75	5,270.00	108
	DIR613	62	43	668.48	41,446.00	721
FAST	FAC1200R	71	46	1,069.97	75,968.00	756
MERCURY	M6G	7	5	494.86	3,464.00	34
RICOH	SP221	6	1	454.17	2,725.00	98
	SP330	1	1	1,244.00	1,244.00	20
TP-Link	WDR7660	29	19	2,075.34	60,185.00	2,640
	WDR7661	28	19	2,175.23	56,556.00	2192
Tenda	AC6V2	44	34	674.86	24,970.00	1,062
	AC8	42	34	1,034.39	39,307.00	1,398
	AC11	42	36	1,150.93	48,339.00	1,841
Total	-	**340**	**245**	-	**359,474**	**10,870**

Concolic Analyzer. Micro Fuzzing module ignores other input data that may influence the execution path of bugs, and the patched control flow nodes may affect inputs we mutate. Thus, the number of unique crashes acquired from fuzzing in sink function call sites is often larger than real bugs. As shown in Table 8, SFuzz can find 115 alerts in 302 unique crashes and capture 67 other inputs in PoC results. By manual effort, we locate 16 false-positive cases among these alerts and eight false-negative cases that SFuzz cannot reveal. Due to page restrictions, we present the reason and how to determine these cases on Github[12]. Finally, SFuzz can discover 99 real bugs (One bug may exist in multiple devices of one vendor. Thus, the sum of unique bugs is 69—precisely, three duplicates in D-Link, two duplicates in RICOH, and 25 duplicates in Tenda) among 107 bugs of these devices (the unique bugs count is 75).

[12] https://github.com/NSSL-SJTU/SFuzz/blob/main/discussion.md

Table 8: The result of Concolic Analyzer. `#CI` represents the unique crash inputs found by Micro Fuzzing. `#OI` represents the number of other inputs in PoC results. `#Alert` represents the bug number verified by Concolic Analyzer. `#FP` represents the number of false-positive cases. `#FN` represents the number of false-negative cases. `#Bugs` represents the number of real bugs. `Avg.Time` represents the average time spent on concolic testing.

Vendor	Model	#						Avg.
		CI	Alert	OI	FP	FN	Bugs	Time(s)
D-Link	DIR100	7	6	0	2	1	5	36.86
	DIR613	43	22	6	9	0	13	294.02
FAST	FAC1200R	57	3	11	0	1	4	776.66
MERCURY	M6G	6	3	0	1	1	3	73.25
RICOH	SP221	6	2	0	0	0	2	181.00
	SP330	18	2	0	0	0	2	635.00
TP-Link	WDR7660	19	2	0	0	0	2	2,226.84
	WDR7661	20	1	0	0	0	1	2,240.00
Tenda	AC6V2	39	22	14	2	1	21	264.82
	AC8	41	26	20	1	2	27	712.31
	AC11	46	26	16	1	2	27	661.91
Total	-	**302**	**115**	**67**	**16**	**8**	**107**	-

6 RELATED WORK

RTOS Security. Armis Labs [2] reveals critical zero-days that can remotely compromise the most popular real-time OS, Vxworks [33], and demonstrates how to take over an entire factory by leveraging these discovered vulnerabilities [18]. Zhu et al. [45] introduce how to find vulnerabilities with fuzzing and debugging VxWorks devices. However, current methods are not generic and rely on equipment for debugging [45] or need heavy labor for manual analysis [33].

Symbolic Execution. Under-constrained symbolic execution [28] and compositional symbolic execution [1, 27] can analyze programs in the UNIX operating system, such as UC-KLEE [28], RWSet [3], and their improved methods [4, 22, 40]. They identify critical data (e.g., Read&Write Sets [3], Relevant Location Set [4]) that affect reachability to new code, and detect and eliminate redundant states and paths when exploring code space. Our slicing method prunes paths based on whether the current function is related to handling the external input or not. Exploring these irrelevant functions does not help bug discovery, but may make the instruction emulation fail or make the testing stuck. Therefore, eliminating paths in SFuzz is designed to boot fuzzing in RTOS and independent of these works in principle and target. UC-KLEE [28] performs a function-scope analysis and suffers from missing inter-procedural data flow. Moreover, these advanced approaches [1, 3, 4, 27, 28] are all designed for checking source code or IR, which are popular at full-fledged OS but scarce at RTOS. Thus, these methods need massive effort for scaling on the binary of RTOS and have high computation complexity when running on low-level instructions. Note that SFuzz performs the symbolic execution based on symbolizing concrete inputs no matter in the Micro Fuzzing engine and Concolic Analyzer module. Thus, they iterate branches in a path triggered by this concrete value (e.g., crash input) and mitigate the path explosion problem.

Greybox Fuzzing & Dynamic testing. AFLGo [5] proposes directed greybox fuzzing, which makes a fuzzer generate inputs to efficiently reach a given set of target program locations (i.e., vulnerable functions). Hawkeye [8] evaluates exercised seeds based on static information and the execution traces to generate the dynamic metrics, which help Hawkeye achieve better performance to touch the target sites. However, directed fuzzing aims to reach sensitive locations, regardless of roadblocks in execution paths that hinder efficient fuzzing and steady emulation in RTOS. Similarly, IntelliDroid [39] can directly generate inputs that trigger targeted Android APIs as an over-approximation for malicious behaviors and allow the dynamic analysis to decide whether they are malicious. However, it must work with full-system dynamic analysis tools (e.g., TaintDroid), which is hard to be satisfied in RTOS. HARVESTER [29] integrates program slicing with dynamic execution to automatically extract runtime values from highly obfuscated Android malware. These advanced testing methodologies work well on full-fledged OS. However, due to the lack of a stable system-wide emulation solution for RTOS, they can not succeed without a greybox environment for inspecting the context of the target program on the fly. Note that our slicing determines a coarse scope of the tainted data and tailors roadblocks that hinder efficient fuzzing and steady emulation in the RTOS binary. They make the subsequent fuzzing process work fluently and effectively on code snippets in terms of instruction flow without a system-wide emulation. Applying the directed fuzzing strategies may improve efficiency, and we will integrate them in future work.

Code Fragment Execution. Several methods have been proposed to directly test vulnerable functions hidden in the "deep" code. Ispoglou et al. [19] present a tool FuzzGen that can automatically synthesize fuzzers for triggering deep code in libraries within a given environment. However, FuzzGen needs to compile the source code of the target library and its consumers to infer the library's interfaces. Voss [25] designs UnicornAFL that adds the Unicorn-based test harness to normal AFL. Thus, it can fuzz binary codes with many CPU architectures, including ARM, X86, etc. However, UnicornAFL only emulates instructions, cannot emulate peripheral interaction and inter-procedural scheduling, and usually fails in executing related instructions (e.g., interrupt) either.

7 CONCLUSION

We propose SFuzz, a novel slice-based fuzzing method, to detect security vulnerabilities in RTOS. Based on the insight that an RTOS monolithic system can be split into meaningful code slices, SFuzz leverages forward slicing to construct a tailored execution tree that is small enough to drive greybox fuzzing on the emulator and utilizes forward and backward slicing to perform concolic testing to verify unique crashes from fuzzing. SFuzz has successfully discovered 77 zero-day software vulnerabilities in 20 RTOS devices, and 67 have been assigned CVE or CNVD IDs. Our evaluation result shows that each part of SFuzz helps it outperform the state-of-the-art tools (e.g., UnicornAFL) in discovering bugs in RTOS.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of this work for their helpful feedback. This research is supported, in part, by Shandong Provincial Natural Science Foundation under Grant No. ZR2020MF055, ZR2021LZH007, ZR2020LZH002 and ZR2020QF045, and Science and Technology Commission of Shanghai Municipality Research Program under Grant 20511102002.

REFERENCES

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *2008 14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. ETAPS, 367–381.

[2] Armis. 2021. Home - Armis. https://www.armis.com/. (2021).

[3] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *2008 14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. ETAPS, 351–366.

[4] Suhabe Bugrara and Dawson Engler. 2013. Redundant State Detection for Dynamic Symbolic Execution. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference (USENIX ATC '13)*. 199–212.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Vol. 8. 209–224.

[7] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*, Vol. 1. 1–1.

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, and Xiuheng Wu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[9] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*.

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.

[11] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–303.

[12] Abraham A Clements, Logan Carpenter, William A Moeglein, and Christopher Wright. 2021. Is Your Firmware Real or Re-Hosted?. In *Workshop on Binary Analysis Research (BAR)*, Vol. 2021. 21.

[13] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. 1201–1218.

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*.

[15] eCos. 2021. eCos Home Page. https://ecos.sourceware.org/. (2021).

[16] FreeRTOS™. 2021. Real-time operating system for microcontrollers. https://www.freertos.org/. (2021).

[17] Ghidra. 2021. Ghidra. https://ghidra-sre.org/. (2021).

[18] Barak Hadad and Dor Zusman. 2020. From an URGENT/11 Vulnerability to a Full Take-Down of a Factory, Using a Single Packet. In *Black Hat Asia*.

[19] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.

[20] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *NDSS*.

[21] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Annual Computer Security Applications Conference*. 733–745.

[22] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the 2017 USENIX conference on Annual Technical Conference (USENIX ATC 17)*. 689–701.

[23] Microsoft. 2006. Data Execution Prevention (DEP). (2006). http://support.microsoft.com/kb/875352/EN-US/.

[24] NASA. 2021. Command & Data-handling Systems. https://mars.nasa.gov/mro/mission/spacecraft/parts/command/. (2021).

[25] Nathan Voss. 2017. afl-unicorn: Fuzzing Arbitrary Binary Code. https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf. (2017).

[26] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[27] Dawei Qi, HOANG D. T. NGUYEN, and Abhik Roychoudhury. 2013. Path exploration based on symbolic output. *ACM Transactions on Software Engineering and Methodology 22*, 32 (2013), 1–41.

[28] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium (USENIX Security 15)*. 49–64.

[29] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *NDSS*, Vol. 16. 21–24.

[30] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 24th Network and Distributed System Security Symposium*. The Internet Society.

[31] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1544–1561.

[32] Majid Salehi, Danny Hughes, and Bruno Crispo. 2020. 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)*. 381–395.

[33] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. 2019. Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS. *White paper, ARMIS, URGENT/11* (2019).

[34] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[35] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, Vol. 16. 1–16.

[36] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.

[37] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 167–180.

[38] WindRiver. 2021. VxWorks: The Leading RTOS for the Intelligent Edge. https://www.windriver.com/products/vxworks. (2021).

[39] Michelle Y Wong and David Lie. 2016. Intellidroid: a targeted input generator for the dynamic analysis of android malware. In *NDSS*, Vol. 16. 21–24.

[40] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2018. Eliminating Path Redundancy via Postconditioned Symbolic Execution. *IEEE Transactions on Software Engineering* 44, 1 (2018), 25–43.

[41] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1145–1152.

[42] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. 2019. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1038–1049.

[43] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. 2019. SrFuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 544–556.

[44] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1099–1114.

[45] Wenzhe Zhu, Zhou Yu, Jiashui Wang, and Ruikai Liu. 2019. Dive into VxWorks Based IoT Device: Debug the Undebugable Device. In *Black Hat Asia*.