Enhancing Real-Time Operating System Security Analysis via Slice-based Fuzzing

Jialu Li, Haoyu Li, Yuchong Xie, Yanhao Wang, Qinsheng Hou, Libo Chen, Bo Zhang, Shenghong Li, and Zhi Xue

***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 designed to detect security vulnerabilities in RTOS. Leveraging the insight that RTOS tasks are typically independent, single-purpose, and deterministic, SFUZZ++ extracts task-specific code slices for targeted testing. Specifically, SFUZZ++ first identifies external input points that manage user input, with assistance from LLMs, and constructs call graphs from these points. Then, it leverages forward slicing to build the sensitive call graph and prune the paths independent of sink points (e.g., memcpy). Further, it detects and handles roadblocks within the coarse-grain scope that hinder effective fuzzing, such as call sites unrelated to the user input or conditional branches unrelated to sink points. At the same time, it attempts to restore the context of the slices to recreate the actual runtime state. 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 82 zero-day bugs on 35 RTOS samples, and 78 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.

***Index Terms*—RTOS, Slice-based fuzzing, Taint analysis, Concolic execution.**

I. INTRODUCTION

Real-Time Operating System (RTOS) is designed for real-time applications and is widely used in embedded microcontrollers and CPUs, with billions of installations globally. For example, VxWorks, a leading RTOS [6], powers over two billion devices [14]. Critical applications, such as NASA's InSight Spacecraft [15], demand RTOS for its deterministic real-time performance and stringent safety and security certifications. However, integrating traditional security mechanisms into RTOS is challenging due to its design constraints. To ensure immediate task responses, RTOS sacrifices kernel-user space isolation [10] and operates in a flat memory model, minimizing context-switching overhead but allowing unrestricted memory access [8]. This monolithic design was once acceptable in isolated local networks with minimal external threats.

The rise of the Internet of Things (IoT) has changed this landscape, directly connecting RTOS devices to the Internet and exposing them to external attackers. This shift makes proactive vulnerability detection critical. Unfortunately, most existing bug detection methods for embedded systems [12], [20]–[23], [50], [51] are ill-suited for RTOS. Typically distributed as monolithic firmware blobs, RTOS operates as a single execution environment encompassing the kernel, scheduler, and task modules. This structure complicates traditional bug detection approaches, which rely on modular or multi-layered architectures.

Traditional static analysis methods [12], [20], [52], [53] face significant challenges when applied to large, monolithic RTOS binaries. Classic approaches like symbolic execution [1], [2], [5], [54] often encounter path explosion issues, while the absence of explicit function symbols and the complexity of RTOS architectures make it difficult to interpret function semantics at the binary level. This limits the ability to identify sensitive data modules or perform detailed analysis. Dynamic analysis techniques [18], [21]–[23], [56]–[59], such as fuzzing, require either physical devices or accurate emulation to test firmware and services. However, the diversity of peripherals and interfaces across RTOS vendors [6], [7], [38] makes comprehensive emulation impractical.

Existing methods offer limited solutions [55]. For instance, Zhu et al. [17] use debugging to detect vulnerabilities in VxWorks-based IoT devices; Liu et al. [56] leverage concolic co-execution on real MCU hardware to enhance vulnerability detection; and Wen et al. [18] focus on identifying configuration errors in bare-metal BLE firmware. Scharnowski et al. [59] further investigate fuzzing peripheral (MMIO) inputs through full-system emulation. However, these approaches tend to be device-specific, require physical hardware or system emulation [17], [56], [58], [59], and are often restricted to particular inputs and bug types [18]. Similarly, Salehi et al. [19] and Clements et al. [11] propose methods for

- *Jialu Li, Haoyu Li, Yanhao Wang, Qinsheng Hou, Libo Chen, Shenghong Li and Zhi Xue are with School of Cyber Science and Engineering, Shanghai Jiao Tong University, Shanghai, China.*
  E-mail: lijialu123@sjtu.edu.cn, learjet@sjtu.edu.cn, wangyanhao136@gmail.com, houqinsheng@sjtu.edu.cn, bob777@sjtu.edu.cn, shli@sjtu.edu.cn, zxue@sjtu.edu.cn
- *Yuchong Xie is with the Hong Kong University of Science and Technology, HongKong, China.*
  E-mail: yxiece@cse.ust.hk
- *Bo Zhang is with China Electric Power Research Institute, Beijing, China.*
  E-mail: zhangbo6@epri.sgcc.com.cn

(Corresponding author: Libo Chen and Bo Zhang)

observing memory corruption and extending analysis tools like HALucinator [24]. Yet, these methods often depend on RTOS source code, manual effort, or extensive development for scalability [19]. Hence, current approaches lack flexibility, scalability, and generality, leaving a significant gap in effectively discovering vulnerabilities in RTOS systems.

Despite the challenges, specific RTOS features offer unique opportunities to overcome testing barriers, particularly through its multi-tasking mechanism. RTOS applications are typically divided into numerous single-purpose tasks, each handling specific events in a deterministic and isolated manner. These tasks exhibit straightforward and independent control flows. Notably, tasks within the same category often share similar data flow patterns. Leveraging this characteristic, our basic idea is focusing on analyzing the data flow from various external data entry points to potential sink functions (e.g., `memcpy`). By slicing code snippets related to these flows across tasks, we isolate compact, critical segments for analysis. These slices are significantly smaller and more focused than the full RTOS binary, making them well-suited for existing testing techniques, such as greybox fuzzing and symbolic execution.

Unlike function-level testing, which often lacks the broader context of data processing, our slicing approach captures 'complete' data flow logic. This reduces the risk of false positives caused by incomplete testing contexts, providing a more accurate and holistic view of potential vulnerabilities. Meanwhile, this approach addresses the scalability and complexity issues of other traditional methods. By reducing the control-flow scope and simplifying emulation requirements, we enable more efficient and effective testing, bypassing the limitations of comprehensive emulation or manual analysis. This targeted slicing approach aligns with RTOS's inherent modularity, turning its structure into an advantage for vulnerability discovery. Nonetheless, achieving effective slice-based fuzzing still requires addressing the following several key challenges posed by the unique characteristics of RTOS.

External Input Points Identification. RTOS firmware, typically distributed as binary code without symbol information, presents challenges in identifying external input entry points. To address this, we use heuristic methods to recover function symbols for explicit inputs, such as network data, and accurately locate these entry points. For implicit inputs, like external data processed by callback functions or interrupt handlers and stored in global memory buffers, the lack of explicit control flow often causes them to be overlooked. To mitigate this, we leverage LLMs to infer function semantics, categorizing relevant functions to identify potential implicit input entry points.

Code Snippet Scope and Context Retrieval. To analyze RTOS firmware effectively, an automated method is needed to identify functionally independent code snippets while capturing their runtime dependencies for accurate testing. To achieve this, SFUZZ++ constructs a call graph rooted at external input points and applies forward slicing to isolate relevant paths. It eliminates paths unrelated to inputs, adjusts conditional branches to guide control flow toward potential sink functions, and dynamically links execution paths across tasks, including those involving direct physical memory access. Additionally, SFUZZ++ traces the memory constraints required during snippet execution to define precise context information for subsequent testing, supported by large language models (LLMs).

Path Exploration and Issue Detection. The constraints within RTOS code snippets can hinder efficient path exploration during testing, while the lack of effective mechanisms makes detecting errors challenging. For instance, conditional branches based on variables unrelated to external inputs cannot be influenced by input mutation, limiting fuzzing effectiveness. To address these challenges, SFUZZ++ initializes the runtime environment based on its analysis and uses a Control Flow Node Handler to guide the fuzzer, bypassing function calls and branches unrelated to external inputs. By integrating fuzzing with symbolic execution, it generates valid seeds capable of navigating complex conditional guards. Additionally, SFUZZ++ enforces memory safety checks for stack frames and global regions, ensuring violations are detected within the reconstructed environment. This approach enhances path reachability and improves error detection during testing.

Potential Vulnerability Validation. During fuzzing, we use several techniques to mask the influence of other variables or path conditions to enhance fuzzing effectiveness. However, this approach can make it challenging to assess whether the triggered issues would occur in a real-world scenario. To address this, when a bug-triggering input is identified, SFUZZ++ restores the modified conditional branches and symbolizes the context of any ignored functions. It then uses symbolic execution to recover the omitted context from the pruned call graph, evaluating the relevance of the corresponding conditions. This enables SFUZZ++ to generate a complete and accurate path condition, allowing for effective vulnerability validation.

Building on this foundation, we introduce SFUZZ++, a novel slice-based fuzzing framework tailored for RTOS. SFUZZ++ processes monolithic firmware binaries by leveraging large language models (LLMs) to identify various types of external data entry points. It then combines data flow analysis with forward slicing to extract relevant code segments along with their broader execution context. Using this tailored code space, SFUZZ++ initializes the runtime environment with context information and applies greybox fuzzing to uncover potential vulnerabilities. To ensure thorough validation, SFUZZ++ employs backward slicing to conduct concolic execution, verifying crash-inducing inputs identified during fuzz testing.

We implement our prototype of SFUZZ++ based on Ghidra [39], angr [5] and UnicornAFL [26] with around 10,800 lines of Python code, 4,600 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 82 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.

Summary of Changes. This paper is an extension of our conference version appearing in the Proceedings of the ACM CCS 2022. In this extension, we added lots of new content according to the feedback from the reviewers and readers after publication, including a refined and more comprehensive definition of external input points in the RTOS, improvements to the Code Snippet Scope Extractor for a more extensive scope and context, and optimized initialization and strategies for the fuzzer to achieve more accurate and efficient fuzz testing. Furthermore, we reimplemented the prototype of the improved analysis tool and updated the experimental results using the latest versions of relevant tools (e.g., unicornafl and angr). We also conducted new experiments, presented fresh analyses, and provided insights derived from the latest discoveries. Additionally, many sections of the paper were rewritten and refined to improve readability.

In summary, we make the following contributions:

- We propose a slice-based fuzzing method for testing real-time operating systems (RTOS). This method leverages LLM to identify external input points, employs forward slicing to prune the control flow and restore the context environment for efficient fuzz testing, and incorporates backward slicing to validate alerts generated during 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 82 unknown bugs. 78 bugs have been assigned CVE/CNVD IDs.

II. PROBLEM AND SOLUTION

In this section, we first provide the background of vulnerabilities in RTOS. Then, we discuss the key challenges and propose corresponding solutions.

A. Security Risk Detection in RTOS

Real-Time Operating Systems (RTOS) are widely used in embedded devices such as printers, switches, and routers due to their ability to ensure deterministic task execution and meet real-time requirements. To optimize performance under constraints like limited memory and rapid task scheduling [10], [12], many vendors compile RTOS into a monolithic binary encompassing all functionalities. To further reduce the binary size, system symbols are often stripped during the compilation process. While these practices enhance efficiency, they pose significant challenges for researchers attempting to emulate entire RTOS-based systems or analyze their security.

While the monolithic and stripped nature of RTOS binaries poses significant challenges for security analysis, their frequent interaction with external data sources such as networks or Bluetooth further amplifies security risks. These interactions create potential attack vectors, enabling malicious actors to exploit vulnerabilities in data processing. Furthermore, many of these devices serve as critical nodes in home or Local Area Networks (LANs), yet they often lack advanced defense mechanisms like Executable Space Protection [48] or stack canaries [47] that are standard in modern operating systems. This combination of exposure to external data and insufficient defenses makes RTOS-based devices high-value targets for attackers, underscoring the importance of developing effective vulnerability detection methods tailored to these systems.

```
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,
8          (struct sockaddr *)&src_addr, (socklen_t *)&addrlen);
9    if ( len != ERROR )
10     ret = protocol_handler((packet *)(Global_addr+0x1c));
11     if (ret == ERROR)
12       logOutput("devDiscoverHandle Error!");
13 }
14 int protocol_handler(packet *data) {
15   bytes[4] = {0xe1, 0x2b, 0x83, 0xc7};
16   if (header_check(data))
17     if (magic_check(data->magic_bytes, bytes, 4))
18       if (checksum(data))
19         return msg_handler(data);
20   return ERROR;
21 }
22 int msg_handler(packet *data) {
23   int ret = ERROR;
24   if (data->version == 0x01)
25     ret=parse_advertisement(data->payload, data->payloadLen);
26   return ret;
27 }
28 int parse_advertisement(uint8 *payload, int payloadLen) {
29   char* dst;
30   char* var_addr;
31   char buffer[64];
32   int index;
33   var_addr = DAT_404d33a8;
34   msg_element *element;
35   msg_element_header *element_header;
36   element = parse_msg_element(payload, payloadLen);
37   element_header = element->header;
38   if (element_header) {
39     index = (int)*(var_addr+4));
40     dst = buffer+index;
41     if (copy_msg_element((char *)element->data, dst,
42         element_header->len) == 0) //Stack Overflow
43       return SUCCESS;
44   }
45   return ERROR;
46 }
```

Listing 1: Pseudocode of the simplified motivation example.

Motivation Example. To illustrate the security risks in RTOS-based devices, consider a buffer overflow vulnerability we identified in the TP-Link WDR7660's RTOS using our tool, SFUZZ++. This vulnerability, which has been fixed and assigned CVE-2020-28877, highlights the risks associated with processing external data. The issue arises in a function responsible for handling incoming packets, as shown in the code snippet in Listing 1. The function first validates the packet's format (protocol_handler in Line 10) via verifying header size (Line 16), magic bytes (Line 17), and checking integrity using a checksum (Line 18). If these checks are passed, the execution continues to a handler function (msg_handler in Line 19), which processes the packet based on its version (Line 24). Finally, the parse_advertisement function extracts the header of the element structure from the packet payload and copies it to a memory buffer (Line 36-42). The vulnerability occurs at Line 42, where the length of the element's data section, specified in its len field, can exceed the buffer size, allowing an attacker to trigger a stack overflow by crafting a malicious packet.

Limitations. Current bug-finding techniques for embedded systems struggle to detect such vulnerabilities effectively. Dynamic approaches like fuzzing or emulation face significant barriers when dealing with RTOS binaries due to their mono-

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

lithic structure, lack of hardware abstraction, and proprietary data formats. For instance, applying a tool like SRFuzzer [40] to locate this vulnerability would require extensive manual reverse engineering to identify all supported data formats, craft appropriate inputs, and trigger the specific handling code—a process that demands significant expertise and effort. Static approaches such as KARONTE [12] or SaTC [20] also encounter limitations. KARONTE primarily focuses on interprocess communication, while SaTC relies on keyword-based heuristics to identify user inputs. Neither approach is well-suited to analyzing monolithic RTOS binaries, especially when complex control flow paths, such as the one in this example, obscure the link between data entry points (e.g., `recvfrom`) and sink functions (e.g., `memcpy`).

B. Our Method and Challenges

Detecting vulnerabilities in RTOS-based embedded devices is inherently challenging. While dynamic methods are generally more effective than static analysis due to their ability to simulate real execution paths, they often demand extensive manual intervention and full-system emulation—requirements that are impractical for diverse RTOS implementations. This raises a critical question: *How can we dynamically detect vulnerabilities in RTOSes without relying on labor-intensive manual analysis or complete system emulation?*

To address this, we propose slice-based fuzzing, a method that focuses on isolating and fuzzing functionally independent code snippets to identify bugs in RTOSes. The core intuition is that program slices, comprising all functions responsible for receiving and processing external data, can be analyzed independently. This approach leverages fuzzing and instruction-level emulation, effectively bypassing the need for emulating entire hardware systems and embedded services. The validity of this intuition has been substantiated in our prior work [49], where experimental results demonstrated that function sets from RTOS-based systems exhibit functional independence in terms of control flow and data flow. To apply this method to RTOS of various embedded devices, we need to address following four challenges.

C1. How to identify the external data entry points? To locate external data input, we must identify both explicit and implicit data entry points in the RTOS codebase. Explicit input points, such as data read functions, can often be recognized even in the absence of symbol files. However, implicit input points, which directly reference buffers, require semantic analysis. For instance, in the code snippet shown in Listing 2, the function `wifi_config_set` accesses a global buffer at Line 18. Analyzing the semantics of such reference-point functions is necessary to determine whether they are used for external data parsing. While traditional static and dynamic analysis methods are labor-intensive, advanced code understanding capabilities of large language models can significantly automate this process.

C2. How to determine the scope of the snippets and their context? Code snippets reachable from external inputs often include irrelevant paths, unrelated functions, or paths that do not lead to critical sink functions, all of which reduce fuzzing efficiency. Moreover, these snippets often access global variables or local variables from their direct or indirect caller functions, which complicates the restoration of the execution context. For example, in Listing 2, `wifi_config_set` uses the global variable `Global_func_table` (Line 20) initialized in `Global_func_table_init` and treats parameters passed from the upper layer (Line 15) as a buffer for `strncpy` (Line 22). Without accurately extracting the memory constraints on these variables within the RTOS system logic and reconstructing their values, fuzzing may produce unintended or erroneous results.

```
1  void Global_func_table_init() { // Global environment init
2    registerFunc(0, Global_func_table, global_func_addr_0);
3    registerFunc(1, Global_func_table, global_func_addr_1);
4  }
5  int wifiRecvData(int sockfd, int offset) {
6    struct sockaddr_in addr;
7    int ret, len = sizeof(struct sockaddr_in);
8    memset((uint8 *)&addr, 0, 0x10);
9    ret = recvfrom(sockfd, 0x80004000, 0x5DC, 0, &addr, &len);
10   return ret;
11 }
12 void wifi_config_handler() {
13   char buf[4];
14   memset(buf, 0, sizeof(buf));
15   wifi_config_set(buf);
16 }
17 void wifi_config_set(char* buf) {
18   char* input = (char*)0x80004000;         // Implicit input
19   char* other_var = (char*)0x80008000;
20   if (global_match(input, Global_func_table))
21     if (*other_var)
22       strncpy(buf, input, len(input) + 1); // Buffer Overflow
23   else
24     logOutput ("wifi_config_set Error!");
25 }
```

Listing 2: Code Samples for Implicit Input Identification and Context Information Restoration.

C3. How to effectively conduct slice-based fuzzing? As discussed in C2, the constraints within RTOS code snippets directly influences the subsequent behavior during fuzzing, making it necessary to restore the contextual environment to ensure the fuzzing process accurately reflects the real execution state. Additionally, some function calls and conditional branches can impact the reachability of the execution path[2] and the overall efficiency of fuzzing. For example, some functions may be beyond the emulator's capability to emulate, thereby limiting path reachability. Moreover, certain conditional branches rely on variables that are not influenced by the input (e.g., Line 21 in Listing 2). Since these variables cannot be modified through seed mutation, it is not possible to control the direction of subsequent branches that depend on these comparisons. Moreover, given the absence of defense mechanisms in the RTOS, we need to establish a security check strategy to identify potential vulnerabilities.

C4. How to verify the vulnerabilities detected in the code snippets? Because fuzzing is performed on isolated code snippets, we must ensure that detected crashes represent real vulnerabilities in the original RTOS. This requires designing methods to evaluate whether a proof of concept (PoC) resulting in a crash corresponds to an actual security flaw.

Our Solutions. Consider the code snippet in Listing 1. By identifying the data-receiving function (Line 8) and all

[2] Execution Path refers to the sequence of instructions executed by the target firmware starting from a data-receiving point with a specific test case.

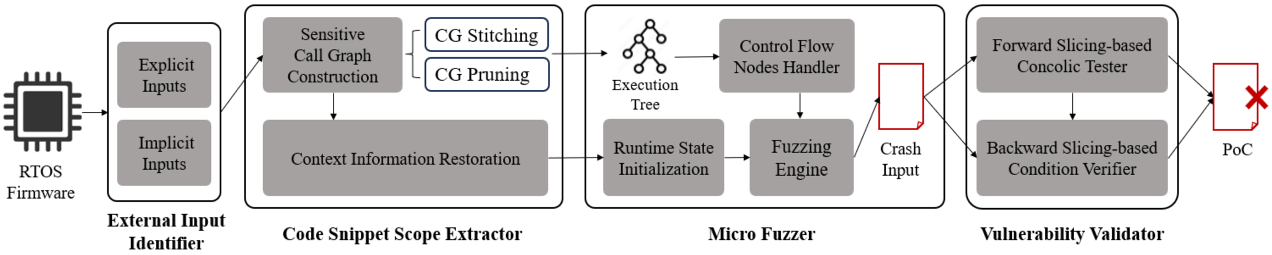

RTOS Firmware
Explicit Inputs
Implicit Inputs
External Input Identifier
Sensitive Call Graph Construction
CG Stitching
CG Pruning
Context Information Restoration
Code Snippet Scope Extractor
Execution Tree
Control Flow Nodes Handler
Runtime State Initialization
Fuzzing Engine
Crash Input
Micro Fuzzer
Forward Slicing-based Concolic Tester
Backward Slicing-based Condition Verifier
Vulnerability Validator
PoC

Fig. 1: **Overview of SFUZZ++.** It takes the firmware of the real-time embedded devices as input and outputs their bug reports.

functions involved in processing the data package (e.g., `copy` in Line 42), we construct a program slice tailored for analysis. Coverage-guided hybrid fuzzing is then used to generate inputs that can trigger vulnerabilities, such as the stack overflow shown in the example. To ensure efficient and stable fuzzing, we extract and solve the memory constraints related to the code snippets to restore the contextual environment with the support of large language models, and refine the control flow by handling nodes unrelated to external data processing, focusing solely on the snippet's relevant logic. Once a vulnerability is identified, we perform concolic analysis on the input that triggered it. This step extracts complete constraints for the relevant control flow nodes, enabling us to determine whether the detected issue is a true vulnerability or a false positive. By limiting the exploration scope to risky code snippets identified through static analysis, our method mitigates the scalability challenges of traditional static and symbolic execution approaches. Symbolic execution is employed selectively—either to generate new test cases when fuzzing is stuck or to verify crash results, ensuring efficient bug detection.

III. Design

Approach Overview. In this paper, we present SFUZZ++, a new tool for detecting vulnerabilities in RTOS-based embedded device firmware using a slice-based fuzzing approach. As illustrated in Figure 1, SFUZZ++ takes firmware as input and generates detailed bug reports as output. The process is divided into four key stages, detailed as follows:

(1) **External Input Identifier**: The workflow begins by identifying external input points. For explicit inputs, such as data-receiving call sites, we reuse the preprocessing module from our previous work, SFUZZ. In addition, SFUZZ++ introduces a new module to detect implicit inputs, including those arising from global memory access.

(2) **Code Snippet Scope Extractor**: Next, the *Code Snippet Scope Extractor* determines the relevant code snippets associated with these inputs. While this component adopts call graph analysis from SFUZZ, we further enhance its capabilities by adding a memory constraint retrieval mechanism. This module supplements the original Forward Slicer, enabling the restoration of a more complete context for subsequent analysis.

(3) **Micro Fuzzer**: The slice-based fuzzing engine, termed the *Micro Fuzzer*, builds upon its predecessor in SFUZZ. The primary improvement here is the integration of runtime state initialization, which allows the fuzzer to emulate a realistic execution environment. This enhancement leads to more accurate exploration of execution trees[3] and more effective handling of complex control flow.

(4) **Vulnerability Validator**: Finally, the *Vulnerability Validator*, directly inherited from SFUZZ, filters out false positives that may arise from exploration pruning or incomplete context. Our overall methodology remains consistent with SFUZZ, selectively integrating symbolic execution for test generation and crash validation. This approach ensures comprehensive bug detection while maintaining scalability.

Table I summarizes the inheritance and enhancement relationships for each component.

TABLE I: **New Features Added to Each Component of SFUZZ++.** `#Component` represents the key functional modules in SFUZZ++. `#Enhancement` indicates the new features added to each component in SFUZZ++.

Component	Enhancement
External Input Identifier	Implicit Input Handling
Code Snippet Scope Extractor	Memory Constraint Retrieval
Micro Fuzzer	Runtime State Initialization

A. External Input Identifier

To perform slice-based fuzzing on functional snippets within the target RTOS, the first step is to identify the various data input points within these snippets. We categorize these points into two types: explicit input points and implicit input points. Explicit input points refer to call sites of functions that clearly handle data reception or transmission, with well-defined roles in interacting with external data sources. Examples include functions like `recv` (for data reception, as shown Line 8 in Listing 1) or `nvram_get` (for global data sharing), which directly manage external inputs. In contrast, implicit input points involve data read-in operations based on hard-coded global addresses (e.g., Line 18-19 in Listing 2). These points are typically more difficult to detect, as they are not explicitly designated as functions for data reception. Nevertheless, they play a critical role in handling external data during execution.

1) Explicit Inputs Identification: Locating explicit input points typically involves identifying the call sites of standard data reception and transmission functions. However, this task becomes challenging when program symbols are stripped, and all functionality is compiled into a monolithic binary. To

[3] The execution tree represents all execution paths from a designated data-receiving point, where each node corresponds to instructions that fork new paths, such as branches and function calls.

overcome this, we recover the semantics of these functions by leveraging all available information and analyzing the code's behavior. This process primarily relies on the following four heuristic methods.

- Symbol File & Log Function. Some vendors, such as TP-Link, release symbol files that label function names within the firmware. Additionally, log functions used to output runtime errors can help recover function names. For example, in the log statement `logOutput`(ostream, "devDiscover: error, ret = %d", retcode), the function `devDiscover` can be identified.
- Virtual Execution. It identifies standard library functions through a three-step process. First, it compares the number of arguments and the return value of the target function with those of known standard library functions to identify potential matches. Next, it allocates memory, initializes register states, and sets initial argument values. Finally, it simulates the function's execution, analyzing output values and affected memory regions to confirm the match.
- Web Service Semantic. We use shared strings marking user input in front-end files (e.g., HTML, PHP, JavaScript) and back-end code to recover the semantics of functions involved in web services. This method is based on techniques proposed in SaTC [20].
- Open Source Firmware. For RTOS systems based on open-source projects (e.g., eCos, FreeRTOS), we can match functions in the firmware to their counterparts in the open-source code. After identifying the firmware version, we use binary-to-source matching tools like B2SFinder [44] to match functions based on strings, immediate values, and other embedded features.

2) Implicit Inputs Identification: Unlike explicit data entry points, implicit inputs lack clear patterns, such as recognizable function symbols (e.g., `recv`), making it challenging to differentiate functions that parse external data from those that merely access global memory for other purposes. Our analysis categorizes implicit input points into two types:

- Explicit Input Data Buffering. External inputs are first acquired through explicit input functions and stored in a global buffer. Data parsers subsequently retrieve these inputs by indexing the buffer's address. For example, in Listing 2, external values are stored in the global memory region `0x80004000` (Line 9) and later accessed by data parsers (Line 18).
- Peripheral Input Handling. For commonly used peripherals in the RTOS, input data is typically stored at specific memory-mapped addresses, with storage operations managed by the peripherals themselves. To facilitate access to these external inputs, the RTOS may map peripheral memory to a global buffer in RAM. It can then read data by directly accessing either the addresses linked to the peripherals or the mapped global addresses.

To identify implicit input points, we analyze the two address types associated with these inputs and leverage large language models to determine whether functions referencing these addresses exhibit data-parsing behavior. The process involves three systematic steps.

1. The initial step of our analysis is to identify global memory regions that serve as potential containers for external data. To this end, we analyze two primary sources of input: explicit input function calls (e.g., `recv`) and peripheral memory-mapped I/O (MMIO) regions. This process yields a `GlobalCandidateSet`, which comprises global memory regions directly reachable from these input primitives. The rationale for this approach is that by anchoring our analysis to well-defined input sources, the buffers identified within the `GlobalCandidateSet` are strong candidates for being directly involved in handling external data, thus ensuring their high relevance.
2. For each function referencing the global addresses in the `GlobalCandidateSet`, we extract its decompiled code along with the corresponding global data reference points. This information is organized into a structured prompt template (illustrated in Figure 2), which directs the large language model (LLM) to analyze the target function's code. Based on this analysis, the LLM delivers a binary judgment on whether the function demonstrates data parsing semantics. Functions identified as parsing functions ("Yes") are subsequently added to the `ParsingFuncSet`, along with their associated global data entries.
3. To further refine the identification of implicit input points, we conduct a lightweight taint analysis of the functions within the `ParsingFuncSet`. This analysis assesses whether data read from the identified global addresses can propagate to sensitive parameters in sink function calls. If a feasible taint path is detected based on over-approximated results, the corresponding global data reference points are treated as the final implicit input points. This approach reduces the risk of false negatives and enhances the comprehensiveness of the subsequent analysis.

B. Code Snippet Scope Extractor

Using the identified external data entry points as root nodes, we employ forward slicing techniques to construct an execution tree, representing a relatively independent testing unit. As illustrated in Figure 1, the scope of the code snippet and its context information are reconstructed through the following key steps.

1) Sensitive Call Graph Construction: The call graph constructor identifies all direct callers of functions containing external input points and uses these callers as root nodes to construct call graphs[4]. To improve the efficiency of fuzzing tests, the process retains only the branches that lead to potential sink functions (e.g., `memcpy`, `strcpy`, `sprintf`, etc.), while irrelevant branches are pruned. By focusing on these sensitive paths, the testing process more effectively targets areas with a higher likelihood of vulnerabilities, enhancing both precision and efficiency.

Call Graph Pruning. To determine whether external inputs or global data influence the parameters of sink functions and to

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

Q: "code snippet of function wifi_config_set"
Which global variables does the function reference?
A: The function accesses global data located at address 0x80004000 and 0x80008000 at Line 21 and 22.

Q: Does the function implementation follow one of the following patterns?

- The function validates the global data using one or more conditions.
- The function performs sub-pattern checking using strstr-like or strncmp-like APIs.
- The function reads global data and writes it to the local stack using memcpy-like APIs.

A: Let's analyze step by step:

- **null_check**: Yes. The function verifies the validity of the global data at 0x80004000 through the `global_match(input, Global_func_table)` condition and checks if the global data at 0x80008000 is null.
- **pattern_check**: No. The function does not explicitly check for specific patterns within the data.
- **substring_extraction**: Yes. The function uses `strncpy` to copy the content of the input variable into the stack buffer.

Q: Based on your analysis, does the function likely parse the global data?
A: Yes.
The function is indeed used to parse the global data at ......

Fig. 2: **Simplified LLM-based Data Parsing Function Identification.** The case-specific prompts and answers are highlighted in green, corresponding to the example in Listing 2.

refine the call graph, SFuzz++ employs lightweight, coarse-grained taint analysis. This analysis tracks each path in the call graph, from root to leaf, identifies the potential impact of external inputs and global data, and filters out paths that are independent of these influences.

The taint engine operates on each call path by analyzing the function body of every node along the path. Parameters or return values of explicit input functions, as well as address reference points of implicit inputs, are marked as taint sources based on their semantics. For instance, the parameter `Global_addr+0x1c` of `recvfrom` in Listing 1 is treated as a taint source, as the memory space it references stores input data. Our lightweight taint engine operates directly on Ghidra's P-code, which serves as a uniform intermediate representation (IR) to ensure our analysis is compatible across disparate architectures. The analysis process unfolds in two main stages: (1) IR Lifting: First, native instructions are lifted into their corresponding P-code representations. This essential step abstracts away architecture-specific details, providing a standardized semantic foundation for the subsequent taint propagation analysis. (2) Taint Propagation: The engine then processes each P-code instruction. If any input operand is tainted (i.e., influenced by external inputs), the taint is propagated to its output operands. This propagation model intrinsically handles inter-procedural analysis: for function calls, if any parameter is tainted, the taint is propagated to the function's return value. For function calls to callees outside the current call path, the taint engine propagates taint attributes from the tainted parameters to the return value.

```
/*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 3: Code Samples for Call Graph Stitching (dynamic method).

Finally, if any risky parameters of sink functions (e.g., the `count` parameter in `memcpy` (*dest, *src, count)) are influenced by external inputs, SFuzz++ retains the corresponding call path. This approach ensures that only relevant paths, where external inputs could propagate to sink functions, are included for further analysis.

Call Graph Stitching. To achieve a more comprehensive tracking of data flow, SFuzz++ restores missing edges that arise from the lack of direct correlations and connects nodes across different call graphs. Similar to KARONTE [12] and SaTC [20], we address interruptions in external input data flows caused by data-sharing paradigms (e.g., `set_env` and `get_env`). This challenge is also prevalent in RTOS environments.

Unlike prior approaches, which rely solely on static analysis to splice deterministically associated nodes, SFuzz++ incorporates dynamic techniques to capture non-deterministic correlations between data set and use points. (1) For data-sharing paradigms labeled with constant strings, we identify and match these labels based on their constant values. We then connect the respective call paths and introduce a virtual node, represented as a two-tuple (e.g., <`nvram_set`, `nvram_get`>), to symbolize the paradigm in the merged call graph. (2) For paradigms involving dynamically created variables, such as "wan%d_pppoe_username" in Listing 3, we use approximate string matching to detect such variables. A virtual condition node is created to represent the potential data-sharing paradigm. During emulation, the actual value of the variable is used to determine whether to jump to the corresponding global data read point. (3) When a set point corresponds to numerous get points, SFuzz++ constructs a virtual condition node to manage these relationships. The jump direction is determined probabilistically, ensuring that all potential data-sharing paths are considered.

2) Context Information Restoration: Effective emulation and fuzzing of code snippets rely on accurately retrieving stack layout and initialized global data. To achieve this, SFuzz++ extracts two key types of context (i.e., local and global context) information from the entire firmware binary. The local context encompasses register values, stack frames, and variable constraints that accumulate through preceding multi-level function calls, while the global context consists of the initialized contents of the global memory region set up during the RTOS bootstrap process.

To extract the local context, SFuzz++ performs lightweight symbolic execution starting from the topmost predecessor of a code snippet's root node in the call graph and continuing to the root node itself, which marks the slice-based fuzzing start point. During this process, it records stack frame information whenever a function is invoked along the path, facilitating the application of memory safety policies. At the same time, SFuzz++ gathers register values and constraints from upstream paths, which are essential for configuring the fuzzing environment and verifying vulnerability conditions. Without the restored local context, the *Micro Fuzzer* may miss crashes due to insufficient boundary information, resulting in false negatives. For instance, in Listing 2, the size of the variable `buf`, defined in the caller function `wifi_config_handler` (Line 13), plays a crucial role in assessing whether the `strncpy` operation leads to a buffer overflow vulnerability (Line 22).

For global context extraction, SFuzz++ focuses on identifying and processing global variable initializers. A representative example is the function `Global_func_table_init` in Listing 2, which initializes the global variable `Global_func_table`. This variable is later used in `wifi_config_set` for condition validation (Line 20). Following a similar approach to implicit input identification, SFuzz++ leverages LLMs to detect global variable initializers. Specifically, it establishes a mapping between each function and the global addresses it modifies, prioritizing write operations. Candidate functions are then analyzed by LLMs to determine whether they perform data initialization within the global space (as illustrated in Figure 3). If a function is confirmed to initialize global data, SFuzz++ leverages symbolic execution to extract constraints related to the global variables.

Q: "code snippet of function [Global_func_table_init]" The function references the global variable [Global_func_table]. Does the function implementation follow one of the following patterns?

- The function initializes global variables.
- The function initializes global function tables by passing function pointers to a registration function.

A: Let's analyze step by step:

- **global variable initialization**: No. The code does not explicitly initialize any global variables.
- **global function table initialization**: Yes. The function registers two functions via `registerFunc`, using function pointers `global_func_addr_0` and `global_func_addr_1`.

Q: Based on your analysis, does the function likely parse the global data?

A: Yes.

The function is indeed used to initialize this global variable at ......

Fig. 3: **LLM-based Global Variable Initializer Identification.** The case-specific prompt and answer are highlighted in green, corresponding to the example in Listing 2.

C. *Micro Fuzzer*

To effectively perform fuzzing on the extracted code snippets, SFuzz++ first processes various types of instructions (i.e., (i) function call, (ii) conditional branch) that influence path reachability and testing efficiency. This step ensures smooth fuzzing operations and minimizes unnecessary path exploration. Next, SFuzz++ uses the stored context information to restore the execution state required for fuzzing. Finally, it tests the code snippets against predefined memory safety policies to detect potential issues.

1) Control Flow Nodes Handler: Because of lacking full context 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. We process function calls in two distinct ways. (1) For functions whose parameters are unaffected by external input, we add the address of their call instructions to the `PatchedFunc` set and guide the fuzzer to skip these calls. Since their arguments and return values are unrelated to input mutations, skipping these functions reduces complexity and enhances fuzzing efficiency without affecting test accuracy. (2) For function calls within sensitive call graphs or with input-dependent arguments—such as `protocol_handler` and `header_check` in Listing 1—we retain and fully explore them. This ensures the fuzzer can reach sensitive execution paths, critical for effective vulnerability detection.

Conditional Branch. Conditional branches pose challenges during fuzzing, particularly when their constraints are independent of input data. In such cases, input mutations cannot influence the direction of the branch, making it necessary to apply tailored strategies to maintain testing effectiveness. Conventional fuzzers often waste cycles by indiscriminately exploring all execution paths. To address this inefficiency, our approach introduces a selective branch-handling mechanism informed by static slicing. Specifically, we instrument and control only the conditional branches that reside within the program slice leading to the target sink. This strategy effectively prunes the fuzzer's exploration space by steering it away from code segments irrelevant to the vulnerability being targeted, thereby significantly enhancing both efficiency and effectiveness.

- Single reachable branch. If only one branch leads to the sink function and its condition is input-dependent, we mark the unreachable branch's target address in the `PatchedJMP` set, instructing the fuzzer to avoid exploring that branch. If the condition is unrelated to input, we add the instruction's address to the `PatchedJMP` set and guide the fuzzer to replace the conditional jump with a fixed jump to the reachable branch.
- Both branches reachable. If both branches lead to sink functions and the condition is unrelated to input, we add the instruction's address to the `PatchedJMP` set, allowing the fuzzer to replace the conditional jump with a random jump. If the condition is input-dependent, no changes are made, ensuring exploration of all relevant paths.
- No reachable branches. If neither branch leads to the sink function, the target addresses of both branches are added to the `PatchedJMP` set, instructing the fuzzer to terminate path exploration upon encountering these addresses.

2) Runtime State Initialization: SFuzz++ utilizes the context information extracted in §III-B2 to accurately initialize the runtime environment for the target code snippet. Specifically, it assigns concrete values to registers and stack variables based on the local context, using stack frame offsets from upper-layer call sites to establish overflow checkpoints. Additionally, global data constraints are applied according to the previously gathered global context. For variables without predefined context constraints, SFuzz++ initializes them with random values to ensure comprehensive testing coverage.

3) Fuzzing Engine: The core component of our fuzzing framework is the slice-based fuzzing technique, referred to as micro fuzzing. This method systematically explores execution paths within extracted code snippets while monitoring the context of sink function calls and identifying crash inputs when memory access violates predefined safety policies. The micro fuzzing approach enhances efficiency by pruning unnecessary paths and stabilizing the emulation process. It skips input-irrelevant function calls, avoids executing masked conditional branches and emulation-hard[5] instructions, and allocates concrete values to uninitialized memory references. These optimizations enable the engine to concentrate on code segments critical to handling target input data, thereby improving the accuracy and stability of the fuzzing process.

Upon loading the RTOS firmware, the built-in image loader preprocesses the tailored code snippets. Call instructions marked in `PatchedFuncset` are replaced with NOP-like instructions to bypass unnecessary computations, while branches flagged in `PatchedJMPset` are augmented with `AvoidExplore` statements at their target addresses, ensuring termination of the current path exploration. Branches requiring fixed or random jumps are handled with the corresponding modifications.

During execution, the core fuzzing engine initializes the RTOS environment and iteratively executes target code snippets starting from the root node of the execution tree. Random data is generated for input entry points, and leveraging UnicornAFL [26], the engine performs coverage-guided fuzzing and emulates instruction execution on the tailored execution tree. To overcome situations where the engine becomes stuck, the hybrid fuzzer invokes its concolic execution component. This component selects a seed input, symbolizes its bytes, and traces the corresponding execution path. It then employs a constraint-solving engine to generate inputs that direct execution toward unexplored paths. The fuzzing process terminates if no new path is identified within a predefined time threshold.

4) Enhanced Memory Safety Policies: Given that bare-metal [19] and RTOS devices typically lack memory sanitization due to resource constraints, SFuzz++ incorporates a lightweight safety-checking mechanism tailored for its fuzzing engine. This mechanism targets memory-related sink functions by applying custom safety policies directly at their call sites. By monitoring these critical points for violations (e.g., buffer overflows), SFuzz++ significantly enhances its ability to detect memory corruption vulnerabilities without imposing the high overhead typical of full-scale sanitizers.

To detect memory bugs effectively, we classify memory buffers into two categories: those with statically determinable sizes and those without. Buffers with identifiable sizes include stack buffers, dynamically allocated buffers (e.g., via `malloc`-like functions), and global variables with sizes inferred from adjacent variables. For such buffers, we check for overflows by determining whether the buffer boundary data is modified after executing the sink function.

```
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 4: Code Samples for Vulnerability Validator

For buffers with indeterminate sizes, we first check whether the variable resides on the stack and appears in the recovered local context. If so, the restored stack frame information is used to validate the current sink point. Otherwise, an alarm is triggered, and the issue is further analyzed for satisfiability in the subsequent *Vulnerability Validator* module.

D. Vulnerability Validator

When the *Micro Fuzzer* detects potential overflows or other alarms, the *Vulnerability Validator* analyzes the crash-triggering inputs to confirm if they represent genuine vulnerabilities. It uses these inputs as concrete values to guide concolic execution along the corresponding paths for constraint solving.

As the pruned function calls (in `PatchedFuncset`) and conditional branches (in `PatchedJMPset`) have already changed context in the original execution tree, we must check whether a crash input triggers a real vulnerability in the original firmware. To conduct this check, the validator first restores the patched code sections. It then symbolizes all parameters and return values at the patched function call sites and conducts concolic testing by combining forward and backward slicing.

As illustrated in Algorithm 1, the workflow consists of two stages: the *Forward Slicing-Based Concolic Tester* analyzes the execution path from input to sink (Line 6, 20–25, and 27), while the *Backward Slicing-Based Condition Verifier* refines constraints and validates object sizes (Line 9–18). We demonstrate this process using CVE-2021-32186, a stack buffer overflow caused by improper handling of NVRAM data. As shown in Listing 4, when the web variable `ledStatus` is equal to 2 (Line 13), the `ledClsTime` variable is written to NVRAM using `nvram_set` at Line 17. Subsequently, in the function `vulnGet`, this value is retrieved via `nvram_get` and copied to the stack without any size checks, triggering the buffer overflow (Line 25).

1) Forward Slicing-Based Concolic Tester: The forward slicing phase begins with a crash input and uses concolic

[5] Emulation-hard instructions, typically related to hardware interactions or HAL modules (e.g., CPU scheduler signals), are excluded as they do not affect the data flow from input sources.

Algorithm 1 The workflow of Vulnerability Validator.

```
function VulnerabilityValidator(CrashInput, RTOS)
    Trace $\leftarrow$ Tracer(CrashInput, RTOS)
    TargetSink $\leftarrow$ GetSinkPoint(Trace)
    CompletePoC $\leftarrow \emptyset$
    State $\leftarrow$ SimulationStart(RTOS)
    State.AddConcreteConstraints(CrashInput)
    while State.active do                                   ▷ State still satisfies all constraints
        if isTargetSinkFunc(State, TargetSink) then
            StateConstraints $\leftarrow$ BackwardSlicing(State).Constraints()
            for constraint $\in$ StateConstraints do
                if ReSymExec(RTOS, constraint.invert(), CrashInput, TargetSink) then
                    StateConstraints.remove(constraint)
                end if
            end for
            if SinkBufferDeterminable(StateConstraints) then
                OutputCompletePoC(StateConstraints, CrashInput, RTOS)
                return
            end if
        else
            if State $\in$ PatchedFuncset then
                SymValues $\leftarrow$ SymRetValue(State) $\bigcup$ SymArgsValue(State)
                State.AddNewSymbols(SymValues)
            else if State $\in$ PatchedJMPset then
                State.SetJumpDirection(Trace)
            end if
        end if
        State.Step()                                        ▷ Step to next concolic state
    end while
    OutputFailedInfo(CrashInput, RTOS)                      ▷ State cannot reach sink
end function
```

execution to trace the path from the input entry point to the sink function. It collects path constraints, including symbolic expressions for function parameters and return values from other input-reading functions[6]. When encountering patched functions (i.e., those in PatchedFuncset), their arguments and return values are symbolized to maintain analytical precision.

In the case of CVE-2021-32186, the entry point is at Line 8, and the sink function is the call to nvram_set at Line 17. Forward slicing traces the path from Line 8, collecting path constraints related to other inputs at Line 9, 10, and 14. These constraints are then incorporated into branch conditions along the path, such as those at Line 11, 13, and 16.

2) Backward Slicing-Based Condition Verifier: Backward slicing initiates from the sink function and traces the execution path in reverse to achieve two goals: optimizing the path predicate of other inputs and validating the size constraints of objects written by the sink. By inverting each predicate of constraints one by one, (i.e., the web variable ledStatus is equal to 2 or not at Line 13), and rerunning symbolic execution, it assesses whether these constraints related to other inputs are essential for reaching the sink, discarding those that are not. Meanwhile, memory constraints for objects at the sink are verified to mitigate false positives.

For the details in CVE-2021-32186, backward slicing starts at the sink (nvram_set at Line 17) and traces back to identify constraints associated with other inputs (Line 9, 10, and 14). It then individually inverts these constraints and reruns symbolic execution to check satisfiability. If the sink remains reachable, the corresponding constraint is deemed unnecessary and discarded. Ultimately, this process discards the constraint at Line 11 while retaining those at Line 13 and 16.

The backward slicing phase also validates buffer size constraints at the final sink point. In vulnGet, we start from the identified memory operation (Line 25) and trace back to the relevant memory allocation site (e.g., the stack frame setup for vulnGet). Using symbolic execution, the analyzer calculates the allocated size for the destination buffer and compares it with the symbolic size of the incoming data to determine if an overflow can occur.

IV. Implementation

We implement the prototype system of SFuzz++ with around 10,800 lines of Python code, 4,600 lines of C code, and 5,100 lines of Java code. The external input identifier and code snippet scope extractor are implemented using Ghidra [39] and LLM. Our fuzzing engine is built on UnicornAFL and Driller [30]. For context information restoration and vulnerability validation, we utilize Angr [41]. We extended Driller to support RTOS images by re-implementing its trace logger with a custom RTOS loader, enabling effective tracking of execution traces for 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 [17] and used in some related works, such as FirmXRay [18]. 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 utilize both string pointers and function addresses to help recognize the base address, which yields relatively better results (as shown in §V-C). (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.

Global References Analysis. To obtain the comprehensive reference information needed for implicit inputs identification and global context restoration, we implement global references analysis. Since RTOS firmware is often stripped binaries and does not contain areas like RAM, relevant analysis tools such as Ghidra are unable to automatically analyze the reference relationships in these sections. Before indexing global addresses, assembly code typically uses computational instructions (e.g., addiu, ldr) to update the actual address values in the registers. Based on this observation, we perform lightweight state simulation at the instruction level using Ghidra, updating the register states for specific computational instructions and checking whether the updated values correspond to potential RAM segment addresses. Through this process, we establish

[6] Here, "other input" refers to external data unrelated to the crash input.

TABLE II: **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**

a mapping between functions and global data, enabling us to identify implicit inputs and restore the global context.

V. Evaluation

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

- **RQ1.** How effective is SFuzz++ in discovering real-world vulnerabilities across diverse RTOS-based embedded devices? (§V-A)
- **RQ2.** How does SFuzz++ perform compared to state-of-the-art fuzzing approaches in RTOS vulnerability detection? (§V-B)
- **RQ3.** What is the contribution of each SFuzz++ component to vulnerability discovery in terms of accuracy and efficiency? (§V-C)

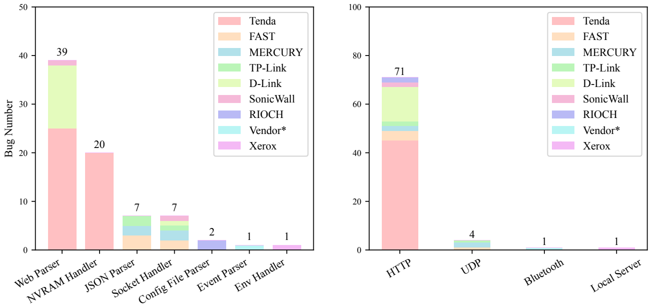
Dataset. As shown in Table II, we collected 35 firmware samples from 17 series in 11 vendors. These devices cover three RTOS types and supply various services, including 23 routers, seven printers, two firewalls, two switches, and one BCI (Brain-Computer Interface). Among these samples, seven devices 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 22.04 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.

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.

A. Real-world Vulnerabilities

SFuzz++ found 82 new bugs[7] in 20 firmware samples of different devices, including router, printer, firewall, and BCI. By the time of submission, all of them have been confirmed by the vendors, and 78 have been assigned CVE or CNVD IDs (54 CVE and 24 CNVD, 64 are high severity); 3 bugs are still under review. Figure 4 shows the types of data receiving points and tasks of the snippets that correspond 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].



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

Fig. 4: **Statistics of Input Receiving Points types and Service Protocols types corresponding to the real-world vulnerabilities**.

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

Experiment	Symbolic Execution	Handler for FuncCall	Handler for CBranch	Environment Recovery
UnicornAFL	✗	✗	✗	✔
SFuzz++-H	✔	✗	✗	✔
SFuzz++-FH	✔	✗	✔	✔
SFuzz++-CH	✔	✔	✗	✔
SFuzz++-DH	✗	✔	✔	✔
SFuzz	✔	✔	✔	✗
SFuzz++	✔	✔	✔	✔

B. Comparison with Existing Methods

To answer `RQ2`, we compare SFuzz++ with existing vulnerability detection approaches. We first implement `UnicornAFL` [26] as our baseline, enhancing its program loader to support RTOS firmware analysis. Due to direct compatibility issues with RTOS environments, we adapt two state-of-the-art methods: T-Fuzz [33] and Driller [30]. Specifically, we implement SFuzz++-FH to represent T-Fuzz's conditional branch handling strategies, and SFuzz++-H as an adaptation of Driller's approach for code fragment execution. To complete our comparative analysis, we also evaluate SFuzz++-CH, which exclusively handles function call instructions. To systematically evaluate our contributions, we assess three configurations. We use our prior work, SFuzz, as the baseline, which represents the system without our new environment recovery components. To measure the impact of symbolic execution-guided fuzzing, we also evaluate SFuzz++-DH, a version of SFuzz++ with its concolic execution capabilities disabled.

[7] All bugs are listed at https://github.com/NSSL-SJTU/SFuzz_Pro/blob/main/rw_vuls.md.

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

TABLE IV: **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	284	3	3	2
	TP-Link WDR7660	1,599	3	192	2
	RICOH SP221	36	0	0	0
	FAST_FAC1200R_Q	906	2	17	2
	MERCURY_M6G	56	0	0	0
	Total	**2,881**	**8**	**212**	**6**
SFuzz++-H	Tenda AC11	288	3	3	2
	TP-Link WDR7660	3,158	3	764	2
	RICOH SP221	37	0	0	0
	FAST_FAC1200R_Q	1,230	2	22	2
	MERCURY_M6G	59	0	0	0
	Total	**4,772**	**8**	**789**	**6**
SFuzz++-FH	Tenda AC11	273	4	6	2
	TP-Link WDR7660	1,675	4	533	3
	RICOH SP221	533	0	0	0
	FAST_FAC1200R_Q	1,543	5	35	3
	MERCURY_M6G	148	0	0	0
	Total	**4,172**	**13**	**574**	**8**
SFuzz++-CH	Tenda AC11	1,024	17	62	14
	TP-Link WDR7660	5,050	5	1,054	4
	RICOH SP221	33	0	0	0
	FAST_FAC1200R_Q	1,553	3	60	3
	MERCURY_M6G	51	0	0	0
	Total	**7,711**	**25**	**1,176**	**21**
SFuzz++-DH	Tenda AC11	1,476	23	316	14
	TP-Link WDR7660	635	4	13	3
	RICOH SP221	239	1	29	1
	FAST_FAC1200R_Q	1,413	5	36	3
	MERCURY_M6G	117	0	0	0
	Total	**3,880**	**33**	**394**	**21**
SFuzz	Tenda AC11	1,793	25	337	25
	TP-Link WDR7660	4,506	19	1,354	2
	RICOH SP221	449	2	390	2
	FAST_FAC1200R_Q	2,081	45	339	3
	MERCURY_M6G	123	4	33	2
	Total	**8,952**	**95**	**2,453**	**34**
SFuzz++	Tenda AC11	1,305	31	339	27
	TP-Link WDR7660	737	19	147	6
	RICOH SP221	200	2	38	2
	FAST_FAC1200R_Q	1,672	45	304	7
	MERCURY_M6G	133	4	23	2
	Total	**4,047**	**101**	**851**	**44**

Finally, SFuzz++ represents our full system, integrating the enhanced environment recovery, selective branch handling, and symbolic execution into a comprehensive fuzzing framework. The comparison of different approaches is shown in Table III.

All approaches operate on identical original execution trees[9] from the initial state, with each implementing its specific strategies. For our evaluation, we selected five representative firmware images, each from a different RTOS vendor, to ensure comprehensive testing. We assess SFuzz++ against six other baseline and variant fuzzers: SFuzz, SFuzz++-DH, SFuzz++-H, SFuzz++-FH, SFuzz++-CH, and UnicornAFL. Our analysis focuses on three key dimensions: effectiveness, stability, and efficiency.

Effectiveness. As shown in Table IV, all 7 tools can find vulnerabilities in real devices to a certain extent, ranging from 6 to 44. Both UnicornAFL and SFuzz++-H can only find bugs from 8 execution trees, and UnicornAFL only explored 2,881 execution paths. Although SFuzz++-H explored 4,772 paths with the assistance of Driller, it still failed to effectively identify new vulnerability paths. When we applied `Cbranch` patch on the basis of SFuzz++-H (i.e., SFuzz++-FH or T-Fuzz), the path exploration remained largely consistent, and we identified only two additional bugs from 13 execution trees. When we applied the `FuncCall` patch to SFuzz++-H (i.e., SFuzz++-CH), we can see that the number of explored paths increased by 62% to 7,711, and the actual number of bugs that increased by 250% to 21. Our complete method, combining the above modes, discovers 44 bugs across 5 models and outperforms all compared tools. In SFuzz++-DH, where the concolic execution component was disabled, crashes were found in only 33 execution trees, along with a significantly lower number of explored execution paths. This result underscores the critical role of concolic execution in enhancing path exploration and, consequently, vulnerability detection. Conversely, when the environment recovery component was disabled, SFuzz explored a slightly higher number of paths than SFuzz++ and found a similar number of crashes. However, manual verification revealed this higher path count to be deceptive. We found that the lack of proper environment initialization led to execution errors that incorrectly diverted the control flow into code segments that are unreachable in a valid run. As a result, while SFuzz's path count was inflated, its ability to find unique, real-world vulnerabilities was not improved, demonstrating the importance of accurate environment modeling.

Stability. To compare the stability among these tools, we examined the successful simulation ratios across all function call trees on five devices. The results indicate that the stability varies significantly among different devices and tools[10]. Both UnicornAFL and SFuzz++-H exhibited relatively low stability across all firmware. In comparison, CHandler (which handles conditional branches) improved stability to some extent for all firmware. Additionally, FHandler (which manages function call instructions) produced more variable results. In detail, the stability of the Tenda AC11 increased from 17.02% (using SFuzz++-H) to 59.57% (using SFuzz++-CH), while TP-Link WDR7660 experienced a slight decline. Therefore, SFuzz++ combines both FHandler and CHandler, allowing it to maintain satisfactory stability. Meanwhile, SFuzz and SFuzz++-DH also combine FHandler and CHandler, achieving results similar to SFuzz++.

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[11] demonstrates the average fuzzing time different tools spend on each device. In terms of processing time per execution tree, our tools show a clear trend related to analytical depth. On average, the times were: UnicornAFL (598s), SFuzz++-DH (781s), SFuzz++-H (830s), SFuzz++-CH (869s), SFuzz

[9] An original execution tree comprises all paths originating from an external data entry point (instruction) in the target RTOS.

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

[11] https://github.com/NSSL-SJTU/SFuzz_Pro/blob/main/avg_time.md.

TABLE V: **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	10	2%(49/2507)	83%(213/257)	21%(48/225)	18%(38/213)	11%(1/9)	38.9
D-Link	DIR613	62	6%(228/4059)	95%(7859/8311)	6%(298/5341)	6%(442/7859)	7%(5/72)	3230.5
FAST	FAC1200R_Q	77	3%(312/9719)	74%(2653/3575)	50%(1088/2206)	4%(94/2653)	7%(32/454)	811.3
MERCURY	M6G	7	1%(138/11588)	76%(613/806)	24%(173/720)	7%(42/613)	14%(17/120)	331.6
RICOH	SP 221	6	2%(421/19134)	65%(1497/2313)	27%(732/2718)	2%(24/1497)	0%(0/596)	846.39
RICOH	SP 330	1	0%(27/36112)	87%(65/75)	12%(6/49)	9%(6/65)	50%(1/2)	36.02
TP-Link	WDR7660	34	0%(158/9425)	89%(1208/1363)	43%(338/788)	10%(117/1208)	14%(17/123)	2380.84
TP-Link	WDR7661	33	2%(151/9152)	89%(1178/1326)	42%(325/766)	10%(123/1178)	16%(19/117)	2314.34
Tenda	AC6V2	49	3%(212/7164)	94%(6204/6566)	9%(329/3484)	35%(2149/6204)	38%(30/78)	806.83
Tenda	AC8	47	3%(224/8531)	95%(5885/6226)	10%(327/3276)	6%(327/5885)	11%(9/81)	604.16
Tenda	AC11	47	3%(219/8554)	95%(5867/6206)	10%(327/3259)	9%(502/5867)	14%(11/81)	956
Average	-	**34**	**2.34%**	**85.37%**	**23.11%**	**10.32%**	**16.59%**	**1123.4**

(1,551s), and finally, the full SFUZZ++ system (1,809s). This data shows that while SFUZZ++ requires more time, its overhead is a direct consequence of its deeper analysis capabilities. Considering its superior vulnerability detection rate, this additional time constitutes a reasonable trade-off for significantly improved effectiveness.

C. Accuracy and Efficiency

In this section, we evaluate the accuracy and efficiency of each part of SFUZZ++, including External Input Identifier(i.e.,Semantic Reconstruction and Implicit Input Identification), Code Snippet Scope Extractor, Micro Fuzzer, and Vulnerability Validator.

Semantic Reconstruction. Among our dataset, 31 samples can be analyzed by SFUZZ++. The base address recognition model successfully identifies all the base addresses of the 31 firmware samples, while SFuzz can recognize only 25. 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[12].

External Input Identifier. We selected four firmware from the experimental set that can pinpoint implicit input points as the dataset for this part, while other firmware were difficult to locate due to insufficient function symbolic information. As shown in Table VI, we identified a total of 20 implicit input points across four firmware samples, with a minimum of 5 and a maximum of 7 implicit input points per sample. With the assistance of the LLM, we identified 32 potential parse functions related to these input points out of a total of 108 functions that reference them, representing a ratio of 29.63%. On average, the cost of using the LLM (calculated using the official pricing for GPT-4o) per firmware is $0.1809. Our manual review of the LLM's judgments revealed 2 false positives and 7 missed detections, primarily due to the challenges in firmware analysis without detailed symbol information. The accuracy of LLM analysis is heavily influenced by the quality of Ghidra's decompilation results, where variable and function names often lack meaningful characteristics. This leads to two main types of errors: false positives occur when the model misclassifies functions as parsing functions based solely on their global variable access patterns, while false negatives arise from the model's difficulty in accurately tracing data flow within larger, complex functions. Our experiment with function name recovery using external symbol tables for wdr7660 and wdr7661 supports this observation: these two samples accounted for only two errors, whereas the remaining samples, which contained relatively few function symbols, had seven errors. This demonstrates the impact of symbol information quality on LLM performance.

TABLE VI: **Performance of Implicit Inputs Identification.** `#Implicit` represents the number of implicit input points found in binary. `Parse` represents the ratio of functions used for parsing implicit input to functions that reference implicit points. `#Parse.FP` represents the number of false-positive cases. `#Parse.FN` represents the number of false-negative cases. `#Tree` represents the number of execution trees that originate from implicit input in parsing functions. `Bugs` represents the ratio of the number of bugs from implicit inputs to the number of bugs from explicit input points.

Vendor	**Model**	**#Implicit**	**Parse**	**#Parse.FP**	**#Parse.FN**	**#Tree**	**Bugs**
D-Link	DIR100	5	23%(6/26)	1	3	2	0/1
FAST	FAC1200R_Q	7	30%(9/30)	1	2	6	4/3
TP-Link	WDR7660	4	35%(9/26)	0	1	5	4/2
TP-Link	WDR7661	4	31%(8/26)	0	1	5	4/2
Total	-	**20**	**29.63%**	**2**	**7**	**18**	**12/8**

We extracted a total of 18 new execution trees and discovered 12 actual bugs caused by the implicit input points, compared to 8 bugs found through explicit input points in these

[12] https://github.com/NSSL-SJTU/SFuzz_Pro/blob/main/source_sink.md

TABLE VII: **The result of Global Context Extraction.** `#GVar` represents the number of global variables related to execution trees found by the Code Snippet Scope Extractor. `Init` represents the ratio of the number of initialization functions confirmed by the LLM to the number of functions that reference related global variables. `#Func` represents the number of initialization functions that successfully extracted global variables. `#CVar` represents the number of global variables successfully restored.

Vendor	Model	#GVar	Init	#Func	#CVar
D-Link	DIR100	28	35%(19/55)	0	0
D-Link	DIR613	19	48%(10/21)	8	9
FAST	FAC1200R	16	53%(85/160)	51	51
MERCURY	M6G	20	48%(88/183)	53	56
RICOH	SP 221	13	28%(19/68)	15	32
RICOH	SP 330	0	0%(0/0)	0	0
TP-Link	WDR7660	93	38%(41/109)	23	22
TP-Link	WDR7661	72	36%(39/108)	19	20
Tenda	AC6V2	52	68%(54/79)	26	19
Tenda	AC8	54	63%(48/76)	22	16
Tenda	AC11	54	63%(47/75)	21	18
Total	-	**421**	**48.18%**	**238**	**243**

firmware samples. The numerical comparison between implicit and explicit input-related vulnerabilities indicates the necessity of including implicit input points in firmware vulnerability analysis.

Code Snippet Scope Extractor. 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 triggered in the following fuzzing process. As shown in Table V, the ratio of the number of functions in the sliced call graphs to the total functions is 2.34% 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 85.37% and 23.11% on average. Moreover, 10.32% and 16.59% 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.

Context Information Restoration. As shown in Table VII, the Code Snippet Scope Extractor successfully identified 421 global variables related to the call graphs obtained through slicing methods. Additionally, the component extracted 450 potential variable initialization functions from 934 functions referencing these variables using LLM, achieving a 48.18% proportion, with an average cost of $0.1953 per firmware. Due to the lack of symbols in most firmware, manual analysis results may be inaccurate. Therefore, we conducted an analysis of false negatives and false positives only on the WDR7660 and WDR7661 firmware, where we successfully recovered function name information using external symbol tables. A total of 217 functions were assessed in these two firmware, resulting in 6 false negatives (2.76%) and 17 false positives (7.83%). The reasons for these errors are similar to those in the implicit input identification section: false positives occur when the model incorrectly interprets variable access as assignment, while false negatives arise from the model's challenges in accurately tracing data flow, particularly in cases of variable aliasing. Ultimately, we successfully recovered 243 global variables from 238 initialization functions, showcasing the effectiveness of this component.

TABLE VIII: **The result of Micro Fuzzing.** `#Tree` represents the number of execution trees which are found by the Code Snippet Scope Extractor. `#VTree` represents the number of the execution trees that contain crashes. `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	#VTree	Avg. Time	Total Time	Total Paths
D-Link	DIR100	10	9	1,149.23	10,343.03	135
D-Link	DIR613	62	45	837.12	37670.50	1,243
FAST	FAC1200R	77	45	2,648.25	119,171.40	1,672
MERCURY	M6G	7	4	5684.83	22,739.31	133
RICOH	SP221	6	2	1941.77	3883.53	200
RICOH	SP330	1	1	640.41	640.41	38
TP-Link	WDR7660	34	19	4314.49	81,975.27	737
TP-Link	WDR7661	33	18	4,228.66	76,115.95	5,252
Tenda	AC6V2	49	40	1195.66	47,826.31	3,533
Tenda	AC8	47	35	1,490.50	52,167.59	1,757
Tenda	AC11	47	31	1,808.90	56,075.94	1,305
Total	**-**	**373**	**249**	**-**	**508,609**	**16,005**

TABLE IX: **Compared with SFuzz++-Context.** `#Tree` represents the number of input-related call graphs. `#VTree++` represents the number of the execution trees that contain crashes for SFuzz++. `#VTree` represents the number of the execution trees that contain crashes for SFuzz++-Context. `#FN` represents the number of false-negative cases of SFuzz++-Context. `#FP` represents the number of false-positive cases of SFuzz++-Context.

Vendor	Model	#Tree	#VTree++	#VTree	#FN	#FP
D-Link	DIR100	10	9	9	0	0
D-Link	DIR613	62	45	44	2	1
FAST	FAC1200R	77	45	45	1	1
MERCURY	M6G	7	4	4	0	0
RICOH	SP 221	6	2	2	0	0
RICOH	SP 330	1	1	1	0	0
TP-Link	WDR7660	34	19	19	2	2
TP-Link	WDR7661	33	18	18	2	2
Tenda	AC6V2	49	40	31	9	0
Tenda	AC8	47	35	27	8	0
Tenda	AC11	47	31	25	8	2
Total	-	**373**	**249**	**225**	**32**	**8**

Micro Fuzzer. In Table VIII, the Code Snippet Scope Extractor of our tool can find 373 unique execution trees that could introduce bugs among 11 different models. The Micro Fuzzing engine identifies 249 execution trees with vulnerabilities and constructs crash inputs corresponding to these potential bugs. The average analysis time on one execution tree varies from less than 10 minutes to over one hour. And the total analysis time for one model ranges from 10 minutes to 33 hours, depending on the complexity of the execution trees to explore.

Runtime State Initialization. To evaluate the impact of runtime state initialization, we compared the testing results of SFuzz++ and SFuzz++-Context. As shown in Table IX, SFuzz++, with runtime state initialization, discovered 249 execution trees containing crashes in the dataset, while SFuzz++-Context, using basic state setup, identified 225

```
1  struct ElementFun { // Global Function Table Member Definition
2      uint32_t type;
3      uint32_t index;
4      uint32_t priority;
5      uint32_t callback;
6  };
7  int parse_discovery(uint8 *payload, int payloadLen) {
8    char buffer [64];
9    char* dst;
10   int start_idx;
11   bool validcheck;
12   struct ElementFun * elementfunc_table;
13   struct ElementFun * current_elementfunc;
14   msg_element * element ;
15   msg_element_header * element_header ;
16   start_idx = (payload == 0) || (payloadLen == 0);
17   elementfunc_table = (struct ElementFun *)0x40612C58;
18   for (int i = start_idx; i < 9; i++) {
19     current_elementfunc = &elementfunc_table[i];
20     validcheck = (current_elementfunc->type == 0 ||
21                   current_elementfunc->priority == 0);
22     if (!validcheck) {                    // Global Env Check
23       element = parse_msg_element(payload, payloadLen);
24       element_header = element->header;
25       if (element_header) {
26         dst = buffer + current_elementfunc->index;
27         if (copy_msg_element((char *)element->data, dst,
               element_header->len)) == 0) // Stack Overflow
28           return SUCCESS;
29       }
30     }
31   }
32   return ERROR;
33 }
```

Listing 5: Code Samples for Global Environment Recovery

execution trees. The difference resulted in 32 missed detections and 8 false positives in SFUZZ++-CONTEXT. Manual examination of the divergent execution trees revealed that these errors were primarily due to missing environmental information. The following two case studies demonstrate how global context gap and local context gap led to these errors.

Case Study. False Negative from Global Context Gaps Listing 5 shows a simplified snippet that contains a buffer overflow error in Line 27. This function is located in one of the traces related to the vulnerability CVE-2020-28877 in TP-Link WDR7660. As shown in Listing 5, the function `parse_discovery` receives external input obtained from the upper-level function through the parameter `payload`. Upon entering the function body, the first step is to check the validity of the parameters, which determines the starting index for subsequent loops (Line 16). The function then accesses the global function table, func_table (Line 17), initialized during the system startup phase, and sequentially verifies the validity of its members (Line 22). If valid members are identified, the function uses them to parse external input, ultimately leading to a stack overflow. (Line 27). In the two traces related to vulnerability CVE-2020-28877, only the trace containing this function checks the global environment. However, due to SFUZZ++-CONTEXT's lack of relevant environment settings, it cannot effectively bypass the checks within the function. Although SFUZZ++-CONTEXT attempts to use control flow nodes to navigate around conditional jumps, both subsequent branches of the validation condition (Line 22) present potential sink points, where even if validation fails, it is still possible to reach a sink point by continuing the loop. Additionally, the global function table members that this condition relies on are indexed based on external input, which determines the starting index (Line 16). As noted in III-C1, SFUZZ++-CONTEXT does not handle such branching statements, resulting in false negatives. In contrast, SFUZZ++ leverages global context recovery to satisfy the checks, thus identifying the vulnerability.

```
1  int tftpcUploadFile() {
2    char pbuffer[512];
3    int fd = socket(AF_INET, SOCK_DGRAM, 0);
4    struct sockaddr_in src_addr;
5    if (bind(fd, &src_addr, sizeof(src_addr)) == -1)
6      return -1;
7    tftpSendFile(src_addr, pbuffer, fd);
8    return 1;
9  }
10 int tftpSendFile(int src_addr, char *pbuffer, int fd) {
11   char cbuffer[512];
12   int addr_len = sizeof (struct sockaddr_in);
13   if (recvfrom(fd, cbuffer, 512, 0, &src_addr, &addr_len) < 4)
14     return -1;
15   memcpy(pbuffer, cbuffer, 512);
16   return 1;
17 }
```

Listing 6: Code Samples for Local Environment Recovery

Case Study. False Positive from Local Context Gaps Listing 6 shows a false positive from SFUZZ++-CONTEXT due to local context gaps. In the tftpcUploadFile function, a socket connection is first established, and then socket-related parameters along with pbuffer are passed to the child function tftpSendFile (Line 7). The tftpSendFile function begins by using the recvfrom function to receive external data (Line 13), which is stored in cbuffer. It then parses this data and copies the contents of cbuffer into pbuffer (Line 15). Because this operation is subject to length constraints, there is no risk of stack overflow. However, SFUZZ++-CONTEXT lacks contextual information about the parent function tftpcUploadFile, leading it to misinterpret the operation at the sink point as a write to an unknown address, resulting in a false positive. In contrast, SFUZZ++ utilizes local context recovery to restore function context, thereby eliminating this false positive.

TABLE X: **The result of Vulnerability Validator.** `#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.
		Alert	OI	FP	FN	Bugs	Time(s)
D-Link	DIR100	7	0	3	1	5	119.56
D-Link	DIR613	22	6	9	0	13	94.58
FAST	FAC1200R	4	11	1	4	7	653.16
MERCURY	M6G	3	0	1	1	3	3,616.25
RICOH	SP221	2	0	0	0	2	19.50
RICOH	SP330	2	0	0	0	2	12.0
TP-Link	WDR7660	3	0	0	2	5	2,351.74
TP-Link	WDR7661	2	0	0	2	4	2,370.44
Tenda	AC6V2	24	14	4	1	21	226.88
Tenda	AC8	28	20	3	2	27	444.57
Tenda	AC11	28	16	3	2	27	605.81
Total	**-**	**125**	**67**	**24**	**15**	**116**	**-**

Vulnerability Validator. 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 sink function call sites that can crash during fuzzing is often greater than the actual number of real bugs. As shown in Table X, SFUZZ++ can find 125 alerts and capture 67 other inputs in PoC results. By manual effort, we locate 24 false-positive cases among these alerts and 15 false-negative cases that SFUZZ++ cannot reveal. Due to page restrictions, we present the reason and how to determine these cases on Github[13]. Finally, SFUZZ++ can discover 108 real bugs (One bug may exist in multiple devices of one vendor. Thus, the sum of unique bugs is 75—precisely, three duplicates in D-Link, two duplicates in RICOH, three duplicates in TP-Link, and 25 duplicates in Tenda) among 116 bugs of these devices (the unique bugs count is 81).

VI. DISCUSSION

In this section, we discuss the limitation of SFUZZ++ and explore the directions of improvement in the future.

Limitations. (1) **Heuristics for Binary Analysis**: Analyzing stripped binaries is a fundamental challenge. While SFUZZ++ employs heuristics to automate load address and function identification, their effectiveness can be target-dependent. Consequently, fully automated analysis is not always guaranteed, and occasional manual intervention may be necessary for complex firmware. (2) **Precision-Focused Input Identification**: Our strategy for identifying inputs is intentionally designed to be precise, focusing on high-confidence sources like explicit input functions to minimize false positives. The inherent trade-off is that some inputs in heavily stripped binaries might be missed, potentially leading to false negatives. (3) **Scope Limited to Memory Vulnerabilities**: SFUZZ++ is currently focused on detecting memory corruption vulnerabilities and does not target logic bugs. This is a direct consequence of our slice-guided methodology, which requires clear data flow between a source and a sink. Logic vulnerabilities often lack such distinct characteristics and robustly detecting them poses a substantial challenge in resource-constrained RTOS environments.

Future Work. In our future work, we plan to address the current limitations. Firstly, to improve implicit input point recognition, we will explore leveraging Large Language Models (LLMs) for semantic analysis. By training models to understand the function of buffer variables from their context, we aim to overcome the constraints of current heuristic methods and improve the accuracy and completeness of input identification. Secondly, to extend our analysis to logic vulnerabilities, we intend to generalize the concepts of "source" and "sink". This will involve creating abstract models for slicing that can represent violations of program logic, thereby expanding the detection scope of SFUZZ++ beyond memory errors.

VII. RELATED WORK

RTOS Security. Armis Labs [13] reveals critical zero-days that can remotely compromise the most popular real-time OS, Vxworks [14], and demonstrates how to take over an entire factory by leveraging these discovered vulnerabilities [16]. Zhu et al. [17] introduce how to find vulnerabilities with fuzzing and debugging VxWorks devices. However, current methods are not generic and rely on equipment for debugging [17] or need heavy labor for manual analysis [14].

Symbolic Execution. Under-constrained symbolic execution [27] and compositional symbolic execution [3], [45] can analyze programs in the UNIX operating system, such as UC-KLEE [27], RWSet [46], and their improved methods [4], [28], [29]. They identify critical data (e.g., Read&Write Sets [46], Relevant Location Set [28]) 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 [27] performs a function-scope analysis and suffers from missing inter-procedural data flow. Moreover, these advanced approaches [3], [27], [28], [45], [46] 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. AFL [37] is a widely-adopted coverage-guided fuzzing framework that has influenced subsequent research, including T-Fuzz [33], Tortoise-Fuzz [36], ZTaint-Havoc [60], and FOX [9]. AFLGo [42] proposes directed greybox fuzzing, which makes a fuzzer generate inputs to efficiently reach a given set of target program locations (i.e., vulnerable functions). This approach has been applied in real-world software testing implementations, including VD-Guard [35] and DDRace [34]. Hawkeye [43] 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 [31] 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 [32] 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. Similarly, Fuzzware [58] and Hoedur [59] also face challenges as they rely on full-system emulation to perform fuzz testing of peripheral (MMIO) inputs. These tools encounter additional difficulties when applied to RTOS firmware, as the input sources for RTOS are mostly network inputs rather than peripheral in-

[13] https://github.com/NSSL-SJTU/SFuzz_Pro/blob/main/discussion.md

terfaces. 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. [25] 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 [26] 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.

VIII. 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 LLM to identify external input points, employs forward slicing to construct a tailored execution tree and restore the context environment for efficient fuzz testing, and utilizes forward and backward slicing to perform concolic testing to verify unique crashes from fuzzing. SFuzz++ has successfully discovered 82 zero-day software vulnerabilities in 20 RTOS devices, and 78 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

This research was supported by National Natural Science Foundation of China under Grant No. 62372297 and 62272306.

References

[1] C. Cadar, D. Dunbar, D. R. Engler, et al., "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[2] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[3] D. Qi, H. D. T. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 32, pp. 1–41, 2013.

[4] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Eliminating path redundancy via postconditioned symbolic execution," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 25–43, 2018.

[5] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al., "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symp. Security Privacy (SP)*, 2016, pp. 138–157.

[6] WindRiver, "VxWorks: The Leading RTOS for the Intelligent Edge," [Online]. Available: https://www.windriver.com/products/vxworks, 2021.

[7] FreeRTOS™, "Real-time operating system for microcontrollers," [Online]. Available: https://www.freertos.org/, 2021.

[8] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *2017 IEEE Symp. Security Privacy (SP)*, 2017, pp. 289–303.

[9] D. She, A. Storek, Y. Xie, S. Kweon, P. Srivastava, and S. Jana, "FOX: Coverage-guided Fuzzing as Online Stochastic Control," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 765–779, 2024.

[10] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, D. Xu, "Securing Real-Time Microcontroller Systems through Customized Memory View Switching," in *NDSS*, 2018.

[11] A. A. Clements, L. Carpenter, W. A. Moeglein, C. Wright, "Is Your Firmware Real or Re-Hosted?," in *Workshop on Binary Analysis Research (BAR)*, vol. 2021, pp. 21, 2021.

[12] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symp. Security Privacy (SP)*, 2020, pp. 1544–1561.

[13] Armis, "Home - Armis," [Online]. Available: https://www.armis.com/, 2021.

[14] B. Seri, G. Vishnepolsky, D. Zusman, "Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS," *White paper, ARMIS, URGENT/11*, 2019.

[15] NASA, "Command & Data-handling Systems," [Online]. Available: https://mars.nasa.gov/mro/mission/spacecraft/parts/command/, 2021.

[16] B. Hadad, D. Zusman, "From an URGENT/11 Vulnerability to a Full Take-Down of a Factory, Using a Single Packet," in *Black Hat Asia*, 2020.

[17] W. Zhu, Z. Yu, J. Wang, R. Liu, "Dive into VxWorks Based IoT Device: Debug the Undebugable Device," in *Black Hat Asia*, 2019.

[18] H. Wen, Z. Lin, Y. Zhang, "FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware," in *Proceedings of the 2020 ACM SIGSAC Conf. Computer Communications Security*, 2020, pp. 167–180.

[19] M. Salehi, D. Hughes, B. Crispo, "μSBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability," in *23rd Int. Symp. Research in Attacks, Intrusions, and Defenses (RAID 2020)*, 2020, pp. 381–395.

[20] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, Z. Xue, "Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems," in *30th USENIX Security Symp. (USENIX Security 21)*, 2021.

[21] D. D. Chen, M. Woo, D. Brumley, M. Egele, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *NDSS*, vol. 1, pp. 1–1, 2016.

[22] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, Y. Kim, "FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis," in *Annual Computer Security Applications Conf.*, 2020, pp. 733–745.

[23] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, L. Sun, "FIRM-AFL: High-throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *28th USENIX Security Symp. (USENIX Security 19)*, 2019, pp. 1099–1114.

[24] A. A. Clements, E. Gustafson, T. Scharnowski, J. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, M. Payer, "HALucinator: Firmware Re-hosting through Abstraction Layer Emulation," in *29th USENIX Security Symp. (USENIX Security 20)*, 2020, pp. 1201–1218.

[25] K. Ispoglou, D. Austin, V. Mohan, M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th USENIX Security Symp. (USENIX Security 20)*, 2020, pp. 2271–2287.

[26] N. Voss, "afl-unicorn: Fuzzing Arbitrary Binary Code," 2017, https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf.

[27] D. A. Ramos, D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *24th USENIX Security Symp. (USENIX Security 15)*, 2015, pp. 49–64.

[28] S. Bugrara, D. Engler, "Redundant State Detection for Dynamic Symbolic Execution," in *Proceedings of the 2013 USENIX Conf. on Annual Technical Conference (USENIX ATC '13)*, 2013, pp. 199–212.

[29] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, T. Kim, "CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems," in *Proceedings of the 2017 USENIX Conf. on Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 689–701.

[30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *NDSS*, vol. 16, no. 2016, pp. 1–16, 2016.

[31] M. Y. Wong, D. Lie, "Intellidroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware," in *NDSS*, vol. 16, no. 2016, pp. 21–24, 2016.

[32] S. Rasthofer, S. Arzt, M. Miltenberger, E. Bodden, "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques," in *NDSS*, vol. 16, no. 2016, pp. 21–24, 2016.

[33] H. Peng, Y. Shoshitaishvili, M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710.

[34] M. Yuan, B. Zhao, P. Li, J. Liang, X. Han, X. Luo, and C. Zhang, "DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing," in *32nd USENIX Security Symposium*, pp. 2849–2866, 2023.

[35] Y. Liu, S. Chen, Y. Xie, Y. Wang, L. Chen, B. Wang, Y. Zeng, Z. Xue, and P. Su, "VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1676–1687, 2023.

[36] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[37] Google, "AFL," 2021, https://github.com/google/AFL.

[38] eCos, "eCos Home Page," 2021. Available online: https://ecos.sourceware.org/.

[39] Ghidra, "Ghidra," 2021. Available online: https://ghidra-sre.org/.

[40] Y. Zhang, W. Huo, K. Jian, J. Shi, H. Lu, L. Liu, C. Wang, D. Sun, C. Zhang, B. Liu, "SrFuzzer: An automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities," in *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 544–556, 2019.

[41] F. Wang, Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 8–9, 2017, IEEE.

[42] M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, 2017.

[43] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2095–2108, 2018.

[44] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu, "B2SFinder: Detecting Open-Source Software Reuse in COTS Software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1038–1049, 2019, IEEE.

[45] S. Anand, P. Godefroid, N. Tillmann, "Demand-Driven Compositional Symbolic Execution," in *2008 14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 367–381, 2008, ETAPS.

[46] P. Boonstoppel, C. Cadar, D. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," in *2008 14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 351–366, 2008, ETAPS.

[47] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security Symposium*, 1998.

[48] Microsoft, "Data Execution Prevention (DEP)," 2006. Available online: http://support.microsoft.com/kb/875352/EN-US/.

[49] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang, Z. Xue, "SFuzz: Slice-based fuzzing for real-time operating systems," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pp. 485–498, 2022.

[50] A. Murali, N. Mathews, M. Alfadel, M. Nagappan, M. Xu, "FuzzSlice: Pruning false positives in static analysis warnings through function-level fuzzing," in *Proceedings of the 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[51] K. Chen, Y. Zhao, J. Guo, Z. Gu, L. Han, "Accelerating firmware vulnerability detection through directed reaching definition analysis," *ICT Express*, 2025.

[52] D. de Ruck, J. Jacobs, J. Lapon, V. Naessens, "dAngr: Lifting software debugging to a symbolic level," in *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 2025.

[53] J. Zhao, Y. Li, Y. Zou, Z. Liang, Y. Xiao, Y. Li, W. Huo, "Leveraging Semantic Relations in Code and Data to Enhance Taint Analysis of Embedded Systems," in *33rd USENIX Security Symposium*, pp. 7067–7084, 2024.

[54] W. Gibbs, A. S. Raj, J. M. Vadayath, H. J. Tay, J. Miller, A. Ajayan, R. Wang, "Operation Mango: Scalable Discovery of Taint-Style Vulnerabilities in Binary Firmware Services," in *33rd USENIX Security Symposium*, pp. 7123–7139, 2024.

[55] W. Zhou, S. Shen, P. Liu, "IoT firmware emulation and its security application in fuzzing: A critical revisit," *Future Internet*, vol. 17, no. 1, p. 19, 2025.

[56] C. Liu, A. Mera, E. Kirda, M. Xu, L. Lu, "CO3: Concolic co-execution for firmware," in *33rd USENIX Security Symposium*, pp. 5591–5608, 2024.

[57] Y. Wang, Y. Li, "DCGFuzz: An embedded firmware security analysis method with dynamically co-directional guidance fuzzing," *Electronics*, vol. 13, no. 8, p. 1433, 2024.

[58] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, A. Abbasi, "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing," in *31st USENIX Security Symposium*, pp. 1239–1256, 2022.

[59] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, T. Holz, "Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[60] Y. Xie, W. Zhang, D. She, "ZTaint-Havoc: From Havoc Mode to Zero-Execution Fuzzing-Driven Taint Inference," in *Proceedings of the ACM on Software Engineering*, pp. 917–939, 2025.