applied sciences

MDPI

Article

A Vulnerability Scanning Method for Web Services in Embedded Firmware

Xiaocheng Ma, Chenyv Yan, Yunchao Wang *, Qiang Wei and Yunfeng Wang

School of Cyberspace Security, Information Engineering University, Zhengzhou 450007, China; wuliangttkx@foxmail.com (X.M.); yan_cyu@163.com (C.Y.); funnywei@163.com (Q.W.); wangyunfeng_2402@163.com (Y.W.)
* Correspondence: w_yunchao@sina.com

Abstract: As the Internet of Things (IoT) era arrives, the proliferation of IoT devices exposed to the Internet presents a significant challenge to device security. Firmware is software that operates within Internet of Things (IoT) devices, directly governing their behaviors and functionalities. Consequently, the security of firmware is critical to shielding IoT devices from potential threats. In order to enable users to operate a device intuitively, firmware commonly provides a web interface. Consequently, this interface frequently serves as the primary attack goal in Internet of Things (IoT) devices, rendering them susceptible to numerous cyber-attacks. Unfortunately, web services have complex data interactions and implicit dependencies, and it is not easy to balance efficiency and accuracy during the analysis process, leading to heavy overhead. This paper proposes a lightweight vulnerability scanning approach, WFinder, designed explicitly for embedded firmware web services to perform vulnerability checks on backend binary files in firmware. WFinder uses static analysis to focus on identifying vulnerabilities in boundary binary files related to web services in firmware. Initially, the approach identifies boundary binary files and external data entry points based on front-end and back-end associativity features. Subsequently, rules are formulated to filter hazardous functions to narrow the analysis targets. Finally, the method generates sensitive call paths from the external data input points to the hazardous functions and conducts a lightweight taint analysis along these paths to uncover potential vulnerabilities. We implemented a prototype of WFinder and evaluated it on the firmware of ten devices from five well-known manufacturers. We discovered thirteen potential vulnerabilities, eight of which were confirmed by the CNVD, and assigned them CNVD identification numbers. Compared with the most advanced tool, SATC, WFinder was more efficient at discovering more bugs on the test set. These results indicate that WFinder is effective at detecting bugs in embedded web services.

Keywords: firmware security; vulnerability discovery; web services; static analysis

Citation: Ma, X.; Yan, C.; Wang, Y.; Wei, Q.; Wang, Y. A Vulnerability Scanning Method for Web Services in Embedded Firmware. *Appl. Sci.* **2024**, *14*, 2373. https://doi.org/10.3390/app14062373

Academic Editor: Gianluigi Ferrari

Received: 2 February 2024
Revised: 26 February 2024
Accepted: 27 February 2024
Published: 12 March 2024

1. Introduction

With the advent of the era of the Internet of Things (IoT), countless new devices are being connected to the network to meet different application requirements. IoT devices are now omnipresent in our lives. According to the "Mobile Economy 2020" report released by the GSMA [1], it is predicted that by around 2025, the total number of global IoT device connections will reach 24.6 billion. Though the scale of the Internet of Things (IoT) industry is continuously expanding, the development of related security standards has yet to keep pace. Furthermore, developers have yet to be completely aware of device security maintenance. As a result, current IoT devices carry many severe security risks.

The functionality of Internet of Things (IoT) devices is predominantly governed by firmware, which is the underlying software responsible for controlling hardware operations to ensure that the devices perform as intended. As the number of devices connected to the Internet of Things (IoT) increases, firmware-targeted attacks have become more

prevalent. If attackers gain access, they may remotely control devices, exfiltrate sensitive data, and use them as springboards for further network intrusions. Firmware typically restricts access to unauthorized external users, granting only authorized users or system administrators limited permissions for specific access and modification. To ensure the user-friendly operation of devices, it is a common practice to provide specific terminal applications or web interfaces. Therefore, web interfaces usually suffer a large number of attacks as exposed points of IoT devices in the network. This paper focuses on discovering vulnerabilities in firmware web servers.

In recent years, considerable work has been carried out on discovering firmware vulnerabilities, including dynamic [2–8] and static [9–14] analyses. Within dynamic analyses, employing emulation techniques for grey-box fuzz testing or conducting black-box fuzz testing on actual physical devices represent mainstream approaches to discovering vulnerability. Dynamic analysis techniques provide contextual information during program execution; however, due to the complexity and variability of the Internet of Things (IoT) environment, as well as the inaccessibility of source code and documentation, both black-box and grey-box testing methods often struggle to achieve a comprehensive exploration of code execution paths, thereby potentially resulting in overlooked vulnerabilities. Compared to dynamic analyses, static analysis techniques can achieve higher code coverage rates given their ability to avoid executing code in an actual runtime environment. Consequently, this makes them more practical and cost-effective methods for detecting vulnerabilities in embedded firmware security. Based on the findings of [11–13] and practical analyses of existing vulnerabilities we conducted, it has been indicated that there is a specific correlation between the input parameters of the web service front end of firmware and the data-receiving functions in the boundary binary programs in the firmware. Vulnerabilities in firmware are often the result of backend binary files calling dangerous functions when processing front-end data, leading to so-called taint-style vulnerabilities. Consequently, this study adopted static analysis as its principal methodological approach, aiming to investigate taint-style vulnerabilities in embedded firmware.

The efficacy of detecting taint-style vulnerabilities is largely contingent upon a robust data-dependency analysis tool. To uncover flaws, such a tool must establish a path that enables contamination to proliferate from an attacker-controlled source to a security-sensitive sink. Regrettably, the efficient analysis of web services in embedded systems for vulnerability detection encounters two primary hindrances: (1) the intricate interactions and implicit dependencies between the front end and back end preclude the precise determination of contamination sources [12]; (2) during data flow tracing, achieving an optimal balance between efficiency and accuracy proves difficult.

Our approach. For web services within embedded systems, we track data flows between the front end and back end and develop rules to filter out boundary binary functions that are deemed risky yet prove to be harmless. Additionally, we designed a lightweight taint analysis methodology to facilitate vulnerability discovery. Specifically, our approach uses the relationship between the front end and back end to identify boundary binary files. We then discern functions that introduce external data as source points based on summarized characteristics. Furthermore, we formulate rules to detect and reduce hazardous functions viewed as sink points. Finally, we apply a coarse-grained taint analysis to trace data flows, facilitating vulnerability queries from source to sink points. In summary, we have developed a vulnerability scanning method that enables data flow tracing between the front end and back end. The contributions of this paper are as follows:

- We offer data flow tracing based on front end–back end associations, introducing a novel method of pinpointing boundary binaries and external data entry points.
- We provide rule parsing based on abstract syntax tree (AST) nodes derived from decompiled code, utilizing these AST nodes to filter hazardous functions.
- We designed coarse-grained taint propagation rules to facilitate data flow tracking from external data entry points to hazardous functions.

- We designed and implemented a prototype system, WFinder, and evaluated it using 10 real-world firmware samples. This evaluation revealed 13 unknown bugs, 8 of which were assigned CNVD numbers.

2. Related Work

2.1. Dynamic Analysis-Based Approaches

Mainstream dynamic analysis methods can be categorized into fuzz testing based on real devices [3–5] and fuzz testing in simulated environments [6–8]. Fuzz testing on real devices usually involves a black-box fuzzy test conducted using the communication protocols supported by the device. Existing black-box fuzz testing works either depend on reverse engineering [3] and modifying associated applications or on APIs [4,15] and firmware usage documents [5] disclosed by manufacturers. This approach results in substantial blind spots within the testing process, and the functional coverage of the devices under test is notably limited. IOTFUZZER [3] analyzes the supporting application of the firmware to derive rich agreement information for communication and mutates the test case by identifying and reusing program-specific logic to discover memory damage vulnerabilities. Snipuzz [4] runs as a client communicating with the device and infers mutated message fragments based on the response. WMIFuzzer [15] captures GUI messages as an initial seed and constructs an abstract syntax tree for mutation. It can test running IoT firmware without a predetermined data model. HUBFUZZER [5] is designed for scenarios in which IoT devices and cloud back-end communication are coordinated through a hub. It utilizes messages exchanged between the hub and IoT devices to discover all functions automatically and then initiates a feature-oriented message-semantics-guided fuzz test. In the latest WiFi-security-related works [16–19], wireless access points (APs) are taken as a research entry point to review the security of actual device WiFi networks, especially the security of web interfaces related to access points (APs).

Fuzz testing based on emulation primarily relies on an emulation environment. Firmadyne [6] is a well-established emulation tool that supports ARM and MIPS architectures; however, its success rate in emulation is relatively low. Although FIRMAE [7] has greatly improved the simulation success rate compared to Firmadyne, this improvement comes at the cost of modifying firmware configuration files and detaching from the actual device operating environment, resulting in misreports. The grey-box fuzz testing tool for Internet of Things (IoT) devices, FirmAFL [8], demonstrates notable improvements in the efficiency of vulnerability mining. However, it is limited to firmware successfully emulated by Firmadyne, narrowing its applicability. Although the FirmFuzz [20] project primarily focuses on fuzzing, it employs QEMU [21] for comprehensive system emulations, resulting in significant system overhead. The implementation process of fuzzing also demands extensive expertise in emulation, diverting focus away from firmware code analysis. Chen et al. [22] proposed SFuzz, which leverages coarse-grained taint propagation to eliminate paths unrelated to external inputs. It conducts forward slicing on call graphs, followed by fuzz testing on these slices. However, due to limitations inherent in the emulation platform, this methodology is solely applicable to RTOS. IoTHunter [23] employs multi-stage message generation techniques for coverage-guided grey-box fuzz testing aimed at protocols, yet it still requires feedback from emulation tools for execution. Consequently, the current state of firmware emulation technology is a significant limiting factor in advancing dynamic analysis techniques.

2.2. Static Analysis-Based Approaches

Current efforts addressing taint-style vulnerabilities in firmware primarily focus on taint analyses. Cheng et al. [9] developed a taint analysis tool, Dtaint, built on angr [24], which employs static taint analysis for vulnerability mining. Dtaint conducts data flow analysis based on a control flow graph, forms a data flow graph, and performs a backward depth-first search to identify execution paths from source to sink points. It then evaluates path constraints to detect potential taint-style vulnerabilities. However, it adheres to the

traditional static analysis rules of personal computer programs for identifying source and sink points, leading to a higher rate of false positives in firmware vulnerability mining. FIoT [10], developed based on angr [24], facilitates the detection of memory corruption vulnerabilities using fuzzy testing methods. It employs backward code-slicing techniques to traverse a binary program's control flow graph (CFG) within firmware. This process constructs hazardous code segments extending from input sources to calls of sensitive functions. Subsequently, it analyzes these segments through symbolic execution and dynamic fuzz testing. Nonetheless, FIoT's localization of source points remains dependent on user-defined criteria. Inaccurate judgments of dangerous code segments could lead to false positives and false negatives in subsequent analyses. REDINI et al. [11] have addressed vulnerabilities in Web services of Internet of Things (IoT) devices by introducing the concept of "boundary binary programs". They designed a method, KARONTE, to identify boundary binary programs. KARONTE analyzes numerous indicators, such as the count of basic blocks, the number of branches, and the quantity of network feature strings within a program. It then employs the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) clustering algorithm to selectively identify binary programs within firmware that process web input data, which are subsequently designated as targets for taint analysis. However, their method does not consider the impact of web front-end factors, resulting in reduced identification accuracy. Based on KARONTE, Chen et al. [12] factored in front-end considerations and developed a vulnerability mining approach centered on keywords shared between the front end and back end. This method identifies binary programs with the highest numbers of matching strings between the front end and back end, designating these as boundary binary programs. It uses the occurrence points of these key terms within the program as origin points for a taint analysis. By examining the function call graph, the method pinpoints potential vulnerability trigger paths and conducts a data flow analysis on these paths. Nevertheless, this approach overlooks the fact that user input data are ingested via data import functions, leading to a substantial number of false positives in the analysis. Liu et al. [13] and Cheng et al. [14] have researched the identification of these external data import functions and have proposed relevant identification techniques.

In summary, current static analysis methods for firmware vulnerabilities primarily rely on manually defined criteria to locate the origin points of vulnerabilities, including boundary binary programs and data import functions. Moreover, most studies arbitrarily designate the call locations of hazardous functions as sink points, lacking specificity. Such imprecise identifications of origin and termination points frequently lead to false positives and false negatives.

3. Background and Motivation

In this section, we first provide a background for and examples of embedded system vulnerabilities, followed by a discussion of the challenges faced in vulnerability analysis and our proposed methodology.

3.1. Motivating Example

Firmware in embedded devices are user-oriented, and to facilitate access to operating system features and hardware functionality, most embedded devices offer a web interface that allows users to configure and manage the device. These web interfaces are often the primary vectors for attacks on embedded devices. Typically, a web interface consists of front-end pages, back-end files, and middleware components. The front-end files construct a complete web page, are displayed to the user via a browser, and offer an interactive interface for device operation. Back-end files handle web requests originating from the front end, communicating with the front-end pages to deliver the full functionality of the web application. Middleware refers to a layer of software that resides between the front-end pages and back-end files; it processes web requests and responses, enabling the functionalities of the web interface.

The web services of IoT devices comprise two main components: the front end and the back end. The front end presents the device's configuration options and functionalities to the end user, and the back end parses requests from the front end and carries out corresponding services. Figure 1 illustrates an example wherein a user employs an interactive web interface to rapidly configure the Wi-Fi settings of a router. The user interacts with the front-end interface, automatically generating a request containing the "*timeZone*" field and sending it to the back end. Upon receiving this request, the back-end web server parses it and invokes the function "*form_fast_setting_wifi_set*" to process the request. The function "*form_fast_setting_wifi_set*" retrieves the value corresponding to the "*timeZone*" field on line 4, and on line 11, it formats this value as an argument for the "*_isoc99_sscanf*" function, which is then utilized in subsequent parts of the program.

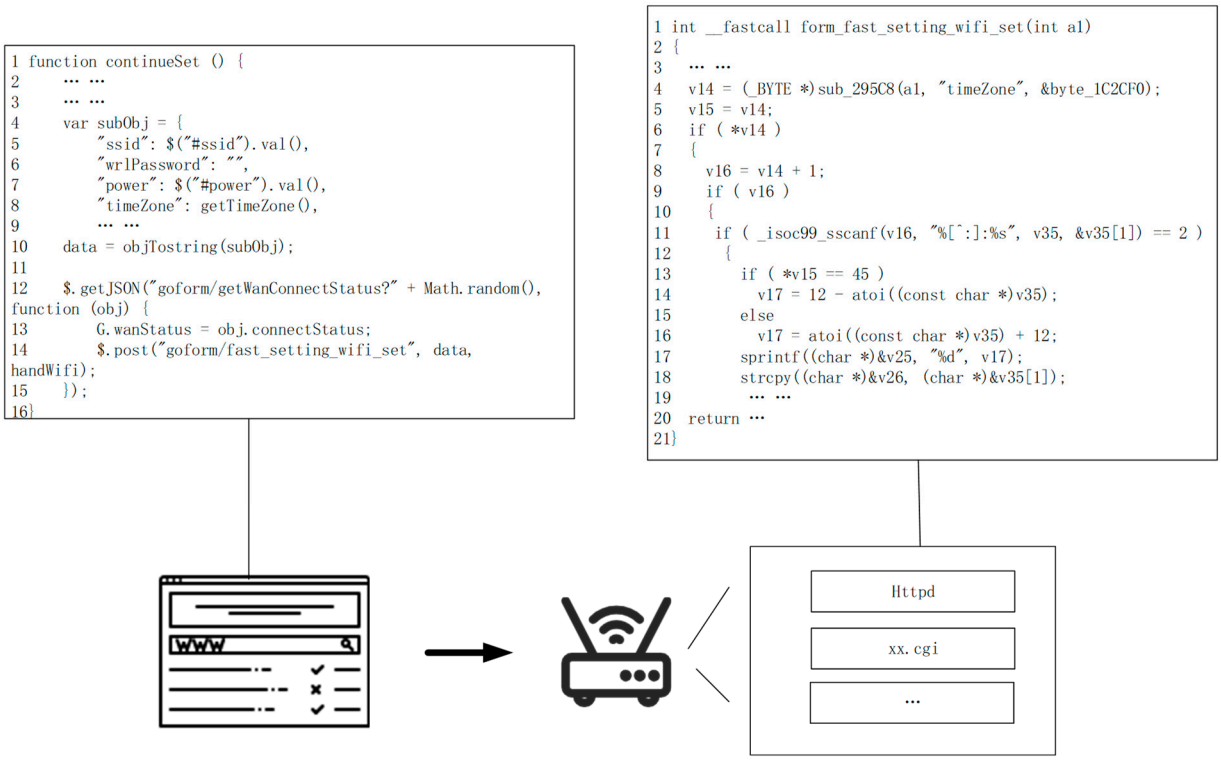

Figure 1. Motivating example. On the left is the web source code related to the WiFi configuration for a certain router's front end, and on the right is the back-end code for message processing. The timeZone string is used by both front-end and back-end code, and attackers can send carefully crafted timeZone values to trigger a buffer overflow.

Regrettably, the web service contains a classic stack overflow vulnerability. The initial stack overflow is on line 11, where the function "*form_fast_setting_wifi_set*" fails to check the length of the "*timeZone*" field before utilizing it as an argument in the "*_isoc99_sscanf*" for formatted input. This oversight results in stack overflow during the formatting of input into the variable list (the third and fourth arguments). Consequently, an attacker could induce a system crash or exploit the vulnerability further by sending a specially crafted data packet to trigger this stack overflow.

The relationship between the front end and the back end. Back-end binaries processing user input exhibit evident characteristics associated with HTTP services manifested by using keywords related to front-end files. As illustrated in Figure 1 with the example "*timeZone*", user inputs are labeled as keywords on the front end and encoded within the data packet. The back end then uses the same or similar keywords to extract user input from the data packet. Data are commonly transmitted in a key–value format. In addition to retrieving keywords for incoming front-end data—whether as request body parameters or

HTTP header parameters—the back-end binary files also output page information, which can be leveraged to extract characteristic words.

Boundary binary. Firmware encompasses numerous binary programs; however, the subset specifically responsible for processing web service data, referred to as "boundary binary programs", could be more sparse. These boundary binary programs are primarily tasked with processing web service data, which necessitates logic for handling requests from external sources. Consequently, their level of relevance with front-end script files is generally more significant than that of other program types. Such files are predominantly found within web server environments (e.g., GoAhead, httpd) or CGI scripts.

External data introduction functions. Boundary binary files ingest user input data through functions designed explicitly for input retrieval. These functions are typically manifested by retrieving the value associated with a keyword index, as illustrated by the function "*sub_295C8*", a custom data input function implemented by the developers. A statistical analysis of data input functions associated with known vulnerabilities reveals a variety of functions used for this purpose. These include generic C standard library functions such as "*websGetVar*", "*getenv*", and "*nvram_get*", as well as proprietary functions from different manufacturers, including "*get_cgi*" from Cisco, "*httpGetEnv*" from TP-Link, and "*find_var*" from NETGEAR [13].

Taint-style vulnerabilities. Vulnerabilities categorized as "taint-style" arise from the inadequate filtering of user input which consequently reaches sensitive functions, leading to issues such as command injection and buffer overflow vulnerabilities. The taint propagation process involves both source points and sink points. Source points refer to locations in the program where external input is received, and these may be manipulated by malicious users, turning them into taints, as exemplified by the function "*sub_295C8*". Sink points are defined as locations in the program where tainted data may be operated upon in a manner that poses risks, such as the potentially dangerous function "*_isoc99_sscanf*" depicted in the diagram.

3.2. Challenges and Methods

Current research on embedded web services primarily faces two challenges. We manually analyzed firmware from six manufacturers, comprising 25 distinct versions, and integrated these findings with the relevant literature to synthesize and propose our solutions.

Challenge 1: Given the complexities of interaction and implicit dependencies between the front end and back end, how can we accurately trace the entry points of front-end data?

Within unpacked firmware, front-end code is relatively distinguishable; however, a significant challenge arises from the plethora of binary files. Identifying code that processes front-end data amid such a high volume of binaries—precisely tracing the data entry points amid intricate front-end and back-end interactions—remains a problem that requires urgent resolution. Existing research has made strides toward this aim. For example, in DTaint [9], the authors manually specify vendor-customized functions (e.g., "*find_var*" and "*websGetVar*") as taint sources. In KARONTE [11], a pre-defined list of network encoding strings (like "*soap*" or "*HTTP*") are utilized as keywords to infer taint sources. SaTC [12] proposes utilizing shared keyword-aware taint checking to track user input data flow between the front end and back end. Researchers like Liu Lingxiang [13] and Cheng Kai [14] have suggested identifying external data input functions to increase analytical precision. Through our analysis of the associations between the front end and back end, we deduced that boundary binary files, which facilitate data interactions with the front end, exhibit salient HTTP service-related characteristics. Invariably, front end-transmitted data manifest as key–value pairs in which boundary binary files typically read values by indexing keys through external data input functions associated with key–value pairs. Thus, by extracting and assessing features, we can identify boundary binary files and external data input functions, effectively pinpointing external data entry points.

Challenge 2: During the process of data flow tracking, how can we achieve a better balance between efficiency and accuracy?

A program's complexity and size limit the precision and speed of taint analysis. Intricate rules for taint propagation have been established in existing work which, when applied to substantial and complex programs, can result in a time-consuming analysis with the potential for reduced accuracy. Previous methodologies often designated dangerous functions in the program as sinks. However, within the voluminous code of real-world firmware, such functions may be called extensively, making the indiscriminate designation of these functions as sinks analytically expensive. We have identified that for firmware with higher complexity, taint analysis outcomes necessitate further manual or automated verification due to the inherent limitations of static analysis. The unwarranted focus on accuracy is not advisable; thus, we propose developing lightweight taint propagation rules. This approach aims to balance accuracy and efficiency, allowing for swift tracking and querying of potential vulnerabilities. To increase the precision of queries, we not only focus on accurately locating external data entry points but also implement the filtering of dangerous functions. Our observations indicate that analysts prefer perusing decompiled code as it presents information more amenable to analysis, such as data types and higher-level control flows. Therefore, deploying rules on decompiled code's abstract syntax tree (AST) nodes for dangerous function filtration could be valuable.

4. Design

4.1. Overview

In this section, we will elaborate on the design of our system. Figure 2 illustrates the detailed design, which inputs firmware images using mainstream architectures (such as ARM and MIPS) in embedded systems and outputs reports on various potential threats. The system initially utilizes Binwalk [25] to decompress firmware images, subsequently separating and extracting frontend and backend files from the decompressed firmware. An analysis correlating the extracted files between the front end and back end is conducted to identify boundary binary files. Subsequently, the system identifies external data entry points in boundary binary files through the rule-based matching of external data import functions. It also interprets rules tailored for dangerous functions and filters these functions within the boundary binary files. Finally, the system conducts a coarse-grained taint analysis, identifying external data entry points as source points and filtered dangerous functions as sink points. In summary, WFinder comprises four main steps: boundary binary identification, external data entry point detection, dangerous function filtering, and taint analysis.

4.2. Boundary Binary Recognition

We identify boundary binary files by analyzing association relationships between the front end and the back end. After firmware unpacking, front-end code can generally be recognized by its unique file extensions. However, not all executable binary files within a firmware's file system are capable of processing front-end data except boundary binaries. Therefore, it is necessary to discern these boundary binary files through carefully designed methods.

In the firmware back end, boundary binary files export device functionalities to the front end and handle user inputs received from the front end. Upon receiving data packets from the front end, the boundary binary files are required to parse and respond to the packets accordingly. Therefore, we can distill signature characteristics to recognize boundary binary files. We use the examples in Figures 3 and 4 to illustrate possible features. Figure 3 shows a packet submitted by a router configuration page which is used to set the wanMTU, wanSpeed, cloneType, and MAC parameters of the device. When the back-end boundary binary receives the data packet, it will process the data in it, and the processed code may contain features related to the front end, as shown in Figure 4. In Figure 4a, if an incorrect URL is requested in the packet, an HTML error page will be outputted because

the URL cannot be found. In fact, boundary binary files can generate various HTML pages based on different response situations, and some of the code of these HTML pages exists in the form of strings in the boundary binary. In Figure 4b, the boundary binary checks whether some header fields in the packet exist. In fact, a boundary binary usually checks various headers of HTTP requests to complete packet verification, so the boundary binary file contains strings related to the HTTP header. In Figure 4c, the boundary binary obtains the parameters passed in the request body, where the parameters in the request body usually exist in the form of "key1 = value 1 & key2 = value 2 & . . .". The boundary binary obtains the corresponding value through the key, which is used for subsequent processing in the program. The three scenarios in Figure 4 correspond to three features associated with boundary binaries and the front end, and we can use these three points to identify boundary binary files.

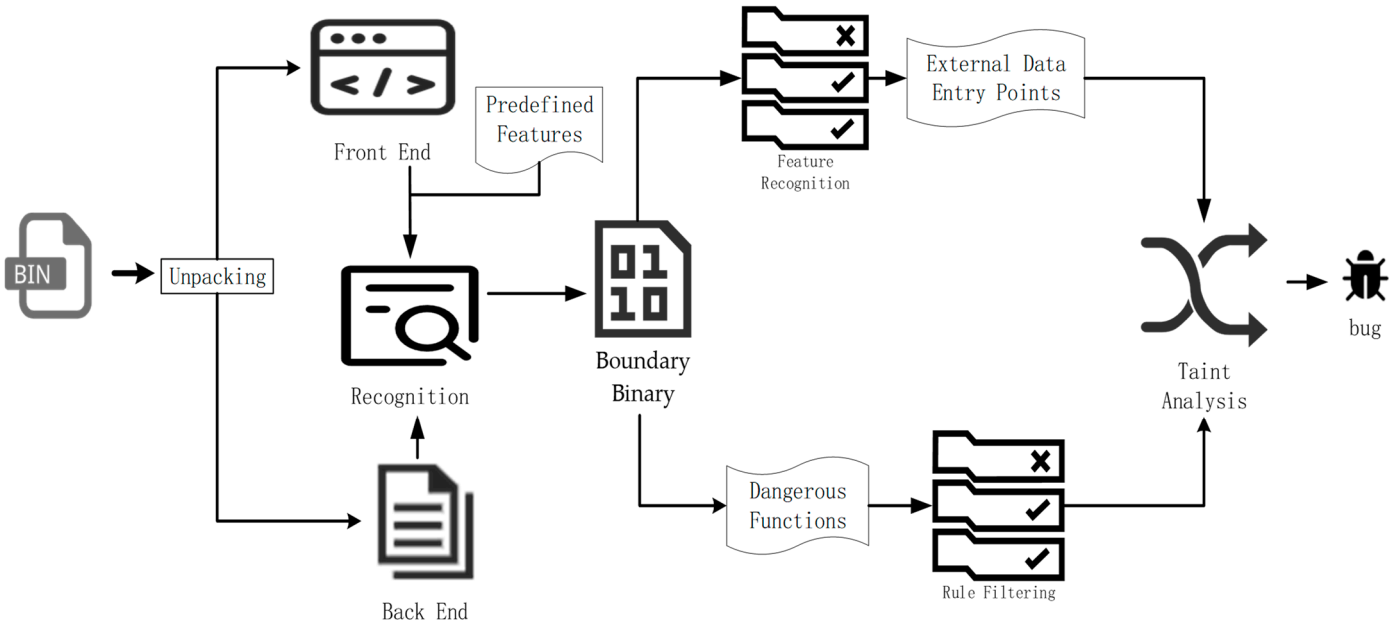

Figure 2. Structure of WFinder. WFinder inputs firmware from actual devices and outputs potential bugs. WFinder identifies back-end boundary binary files by analyzing the correlation between the front end and back end and accurately identifies contaminated source and sink points through specific features and rules for an input-sensitive taint analysis.

```
POST /goform/AdvSetMacMtuWan HTTP/1.1
Host: 192.168.0.140
Content-Length: 58
Accept: */*
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159
Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Origin: http://192.168.0.140
Referer: http://192.168.0.140/mac_clone.html?random=0.8051087400423418&
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9
Connection: close

wanMTU=1280&wanSpeed=0&cloneType=2&mac=00:0C:29:42:61:32
```

Figure 3. Packets from the front end. The data packet is sent by the front-end configuration page of a certain router. The data package includes parameter configuration information, which is used to set the device's wanMTU, wanSpeed, cloneType, and mac parameters.

```
1 … …
2 if ( sub_20DF8(a1, v8, a6, 0, 438) < 0 )
3 {
4     sub_2A878(a1, 404, "Cannot open URL");
5     return 1;
6 }
7 … …
8 int sub_2A878(int a1, int a2, const char *a3)
9 {
10    … …
11    sub_20C10(&v14, 20480,
12        "<html><head><title>Document Error: %s</title></head>\r\ n"
13        "        <body><h2>Access Error: %s</h2>\r\n"
14        "        <p>%s</p></body></html>\r\n", v9);
15    … …
16}
```

(a). Feature 1 of boundary binary

```
1 … …
2 for ( i = (char *)a1[1]; i && *i; i = v15 )
3 {
4     v16 = strtok(i, ": \t\n");
5     if ( v16 )
6     {
7       v17 = strtok(0, "\n");
8       … …
9       v19 = v17;
10      v22 = v19++;
11    if ( !strcmp(v16, "user-agent") )
12    {
13        a1[52] = sub_1C174(v22);
14        continue;
15     }
16    if ( sub_2BAB8(v16, "authorization") )
17    {
18        if ( !strcmp(v16, "content-length") )
19        {
20          v39 = atoi(v22);
21          … …
22     … …
23}
24… …
```

(b). Feature 2 of boundary binary

```
1 … …
2 int __fastcall sub_658D8(int a1, int a2)
3 {
4   … …
5   const char *v5;
6   int v9[8];
7   … …
8   v5 = (const char *)sub_295C8(a1, "wanMTU", &byte_1C2CF0);
9   strcpy((char *)v9, v5);
10  … …
11}
12… …
```

(c). Feature 3 of boundary binary

Figure 4. Possible responses from the back end. The processing logic that may exist in the back end binary code for data packets, including generating page-corresponding information (**a**), HTTP header validation (**b**), and obtaining the value of parameters passed in the request body (**c**).

Observations indicate that each characteristic of a boundary binary includes specific feature keywords present within the binary file, thus enabling the identification of binaries through keyword matching. For Characteristics 1 and 2, we can collect HTTP header fields and HTML tags to serve as matching keywords. Characteristic 3 requires the extraction of parameter keywords used for data transfer in the front-end files. We then extract strings from each back-end binary and attempt to match them with our established feature keywords. We prioritize back-end files that possess three distinct characteristics, and then binaries with the highest numbers of matching feature keywords are considered boundary binaries, as expressed in Equation (1), where K1, K2, and K3 represent the sets of keywords derived from the three characteristics, and Si represents the set of keywords extracted from each binary file in the firmware.

$$\text{Border} = \text{MAX}\ (K1 \cup K2 \cup K3) \cap Si \tag{1}$$

4.2.1. Keywords of Feature 1 and Feature 2

We collected a list of common HTML tags and HTTP/1.1 general header fields as keywords for Characteristics 1 and 2, respectively, as shown in Table 1. For the keywords of Characteristic 1, we only focus on structural tags. This decision is based on the fact that of the many types of HTML tags, including structural, text, and list tags, structural tags are the most commonly used and an indispensable part of any HTML document. As for the keywords related to Characteristic 2, we comprehensively collected the general header fields of HTTP/1.1 to ensure broad coverage of potential keywords.

Table 1. Keywords of Feature 1 and Feature 2. The structural tags of HTML tags are collected as the keyword for Feature 1, and the HTTP header fields are comprehensively collected as the keyword for Feature 2.

HTML tags	<html>, <head>, <title>, <meta>, <link>, <style>, <script>, <body>
HTTP Header	Accept, Accept-Charset, Accept-Encoding, Accept-Language, Accept-Ranges, Age, Authorization, Cache-Control, Connection, Keep-Alive, Content-Encoding, Content-Language, Content-Length, Content-Range, Content-Type, Etag, Expired, Host, If-Match, If-None-Match, If-Modified-Since, If-Unmodified-Since, If-Range, Last-Modified, Location, Pramga, Proxy-Authenticate, Proxy-Authorization, Range, Referer, Server, User-Agent, Transfer-Encoding, Vary, Via

4.2.2. Keywords of Feature 3

Regarding the keywords associated with Characteristic 3, we analyze front-end scripts to extract potential keywords derived from user input. Currently, front-end script files for Internet of Things (IoT) devices predominantly consist of HTML, JavaScript, and XML; therefore, this study focuses primarily on these types of files for analysis. Owing to the standardized format of HTML files, we employ regular expressions to extract keywords, precisely values of attributes such as their id, name, and action. Services based on XML typically employ a fixed format within their XML files to delineate input data. Therefore, we employ regular expressions to extract keywords based on their format. Given the highly variable nature of JavaScript formatting, regular expressions cannot reliably identify keywords. Therefore, for JavaScript files, we parse the files into an Abstract Syntax Tree (AST) and scrutinize each literal node to extract values from the value attributes.

Strings collected from HTML, XML, and AST present a significant challenge due to a plethora of spurious keywords. These not only impose a substantial burden on subsequent string-matching processes but also introduce false positives in error detection. For instance, commonly used front-end strings may lack corresponding objects in the backend. To filter out these irrelevant keywords, we have devised several rules based on empirical insights. Firstly, we exclude strings containing special characters, such as "!" and "@", which typically escape during the generation of HTTP requests on the front end. Secondly, for strings ending with "=", we retain the left part and discard the right part. Take, for example, "timeZone=" from Figure 1, where only the parameter name is reused in the backend.

Even after applying filters, the candidate list may still contain many distractors that are not intended to be used as input keywords. To reduce the complexity of subsequent modules, we employ two heuristic methods to identify and exclude these from the keyword set. When a JavaScript file is referenced by numerous HTML documents, we consider it a generic shared library, similar to chart libraries. As such files typically do not include input keywords, we disregard all candidates within them. Furthermore, if a keyword is referenced by multiple front-end files, such as "Button" and "Cancel", it is likely to be a common string rather than an input keyword. These keywords are also removed from the candidate list.

4.3. Identification of External Data Entry Points

Upon locating the boundary binary program, the subsequent step is to identify the data entry points within the program. Typically, external data are introduced through parameters found in HTTP packet headers and bodies, frequently formatted as key–value pairs. Functions that import external data typically refer to those that retrieve the value associated with a user-requested keyword, enabling access to values corresponding to the key: value-type data transmitted from the front end, including the values of HTTP headers and parameters in the request body that align with Features 2 and 3 of the boundary binary file. We categorize functions introducing external data into three groups: those

handling request body parameters, those for HTTP header values, and functions setting configuration information. Further, we document their respective characteristics.

The first category encompasses functions for introducing request body parameters. Commonly, these are fixed functions characterized by the following attributes:

- High-frequency characteristics: Functions that introduce external data are invoked repeatedly throughout various sections of the program. These functions typically reference keywords shared between the front end and back end, matching corresponding data from external input sources. Due to their reference to a range of keywords, these functions experience extensive utilization within the program. Therefore, we consider functions with a higher frequency of referenced parameter keywords candidate functions.
- Functions introducing request body parameters may utilize C standard library functions such as *websGetVar*, *getenv*, and *nvram_get*. Initially, we examine whether the candidate functions utilize C standard library functions; if so, we classify them as functions that import external data. If standard library functions are not used, we review the names of candidate functions, prioritizing those named with keywords related to WEB input for subsequent examination, such as Web, http, get, var, and similar terms.
- Functions for importing external data typically resemble the strcmp function or obtain the address index of values corresponding to keywords from external inputs by invoking functions similar to strcmp (we collectively refer to these as strcmp-like functions). Therefore, we assess whether the candidate functions are strcmp-like, prioritizing those marked for examination. Simultaneously, we evaluate whether these functions' parameters or return values contain pointers that store values. We identify functions characterized by strcmp-like features and with parameters or return values that retain values as functions for importing external data.

HTTP header fields are utilized less frequently in boundary binaries compared to request body parameters, potentially lacking dedicated import functions. Initially, we identify the locations of functions referencing HTTP header fields and subsequently examine whether these functions are similar to strcmp. If they resemble strcmp functions, we classify the functions referencing request body parameters at these locations as external data import functions.

Configuration information setting functions are typically library functions and usually manifest in pairs, serving the purpose of either assigning a value to a particular parameter (referred to as "set" functions) or retrieving a configuration parameter (referred to as "get" functions). Sometimes, the "set" functions retrieve values from external data inputs to configure device parameters, while "get" functions may process configuration information read from the device. Consequently, we can ensure the comprehensiveness of vulnerability exploration by integrating data flows of "set" and "get" functions that reference the same external keyword. In naming conventions, the "set" and "get" functions are identical except for their "set" or "get" prefixes. This characteristic enables us to identify such functions within imported library functions.

4.4. Hazardous Function Filtering

In vulnerability analysis, functions that serve as execution endpoints, termed "sink" functions, often use hazardous functions. However, not all instances of hazardous function calls are necessarily harmful. Our analysis focuses on risky functions associated with three types of taint-based vulnerabilities: buffer overflows, command injections, and format string vulnerabilities. We find that taint-based vulnerabilities arise when specific argument positions within hazardous functions can be controlled by the user. Therefore, we posit that a vulnerability only occurs when these dangerous parameters of hazardous functions are user-controllable. In many practical scenarios, the dangerous parameters of such functions are often constants or parameter types that do not lead to overflows. For example, in the case of the "strcpy" function associated with buffer overflows, the critical parameter

(second argument) could be a number or a constant string. Even if not constant, it might be a string whose length has been evaluated by the "strlen" function.

To diminish the number of sink points in the vulnerability analysis, we can devise rules that filter based on the hazardous parameter positions of dangerous functions. In composing these rules, we craft them with coarse granularity to avoid generating false negatives. For instance, with "strcpy" and similar functions, a rule could be "the second parameter is neither a constant nor derived from the 'strlen' function." For "strncpy" and analogous functions, the rule could be "the second parameter is neither a constant nor sourced from 'strlen', and the third parameter is either not a constant or is a constant of significant size." Regarding "sprintf" and similar format input functions, we examine whether the first parameter is a constant string. If it is not, there is potential for format string vulnerabilities; if it is a constant string, we search for "%s" within the string and assess whether its corresponding parameters meet the criteria of being non-constant and not derived from "strlen", concurrently tracking the parameters that could potentially induce vulnerabilities. We have categorized common dangerous functions and developed filtering rules tailored to their parameter characteristics. Table 2 showcases a sample of the rules we have developed.

Table 2. Example of dangerous function filtering rule. Examples of filtering rules for dangerous functions corresponding to vulnerabilities, such as buffer overflow, command injection, and formatted strings, are shown.

Vulnerability Types	Function Types	Rules
Buffer Overflow	Strncpy-like Functions	not param[1].is_constant() and not param[1].used_in_call_after(['strlen']) and (not param[2].is_constant() or param[2].is_morethan(256))
	Sprintf-like Functions	any([(not param[i+1].used_in_call_before(['strlen']) and not param[i+1].is_constant()) for i in range(len(param[1].string_value().split('%'))) if param[1].string_value().split('%')[i].startswith('s')]
Command Injection	System-like Functions	not param[0].is_constant()
Formatted String	Print-like Functions	not param[0].is_constant()

4.5. Vulnerability Path Exploration

After identifying external data introduction functions and filtering hazardous functions, we treat the external data introduction functions that reference the keywords of Feature 2 and Feature 3 as source points and hazardous functions as sink points for exploring potential vulnerability paths. Initially, we generate function call paths for both source and sink points. Subsequently, we rely on these function call paths and control flow graphs within functions to conduct a taint analysis, aiming to uncover potential vulnerabilities.

Although the preceding modules have narrowed down the target of the contamination analysis, a substantial number of pollution sources still require a trace analysis. To enhance the efficiency of this analysis, we first search for function call traces from source to sink points—that is, the sequence of function calls from external data introduction points to potential hazardous functions—before exploring paths. We designate the external data introduction function associated with an introduction point as the root node and employ depth-first traversal to search for calls to hazardous functions within the function's call tree, thus generating a sequence of calls. Should an external data introduction point lack any function call sequences, it is considered to have no reachable paths to potential hazardous functions and, consequently, can be removed from the collection of pollution sources.

Our taint analysis relies on the generated function call paths and the control flow graphs within functions. The main factors influencing the efficiency and accuracy of taint analysis are the taint propagation specifications for function calls. As shown in Algorithm

1, we categorize functions into three types: downward-propagating functions, inward-propagating functions, and terminator functions. Functions involved in the function call paths are designated as inward-propagating functions, within which we track the propagation of tainted parameters. Hazardous functions are considered terminator functions; for these, we verify whether tainted parameters correspond to dangerous ones, and if so, we issue an alert. Other functions are classified as downward-propagating functions. For these, we do not trace into the function; instead, we contaminate their parameters and return values and continue the downward propagation of taint in the originating function. The external data introduction functions within the pollution sources serve as the initial downward-propagating functions. As illustrated in Figure 5, the data reception function is in "Func_A", and from "Func_A" to the hazardous call point "Func_sink", there is a call path "Func_A -> Func_B -> Func_sink1". We regard "Func_B" on the call chain as a downward-propagating function. "Func_B" calls "Func_D" before calling "Func_sink1", but since "Func_D" is not on the call chain, it is considered a downward-propagating function. In the diagram, Func_C and Func_E are paired in calling the set and get functions of the configuration information setting function. Therefore, we continue to explore the data flow from Func_A through Func_C and Func_E to Func_sink2.

Algorithm 1: Taint Specifications.

```
1. Function Taint(Ins,Taint_Map)
2.   if IS_FUNCALL then
3.       func ← GETFUNADDR(Ins)
4.       (retv,params) ← GERPARAMS(Ins)
5.       taint_set ← HAS_TAINT(params)
6.       if taint_set = NULL then return
7.       end if
8.       if IS_CALLNOTEFUNC(func) then
9.          STEPINTO(func,Taint_Map,taint_set)
10.      else if IS_SINKFUNC(func) then
11.         alert()
12.      else then
13.         if IS_POINTER(retv) && IS_USED(retv) then
14.            T(retv)
15.      else
16.         T(params,retv)
17.      end if
18.   end if
19.   else
20.      TAINT_RULE(Ins,Taint_Map)
21.   end if
22. end function
```

Considering factors of efficiency and accuracy, we address the issue of cycles within the control flow graph caused by loops and similar constructs, which can lead to infinite loops during path exploration. To avoid this occurrence, we record the visitation status of each basic block in the control flow graph and limit the number of visits to each basic block to once.

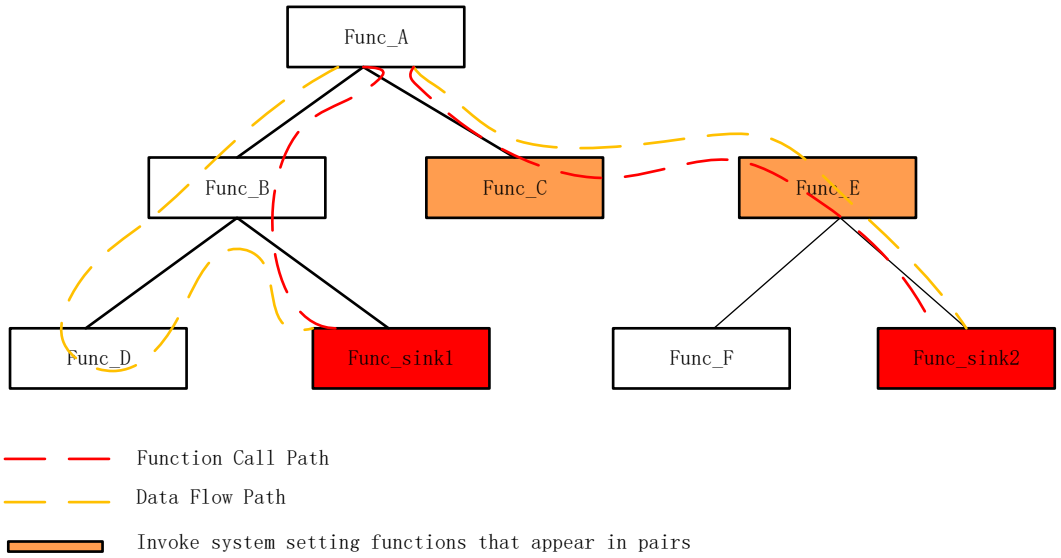

Figure 5. Function call graph. The figure shows the possible function call paths and data flow paths from Func-A, which calls the external data import function, to the potentially dangerous function. Finc_B called Func_D before calling Finc_sink1. Func_C and Func_E contain a pair of function calls for system configuration functions.

5. Evaluation

5.1. Experiment Setup

We have developed a prototype system utilizing Python on the foundation provided by the reverse-engineering tool IDA [26]. The front-end keyword extraction module is implemented based on SaTC [12], while the hazardous-function-filtering module utilizes the IDA plugin VulFi [27]. The taint analysis engine is constructed atop angr [24]. We evaluate WFinder in real-world embedded systems and answer the following questions:

Q1: Can SaTC find real-world vulnerabilities? (Section 5.2.) How effective is it compared to a state-of-the-art tool? (Section 5.2.)

Q2: Can WFinder accurately detect boundary binary functions and external data introduction functions? (Section 5.3.)

Q3: Do the detection of external data input points and the filtering of hazard functions provide positive significance? (Section 5.4.)

Q4: How efficient and accurate is our taint analysis? (Section 5.5.)

Dataset. As shown in Table 3, the dataset for this study is sourced from IoT device firmware provided online by five vendors: Tenda [28], D-Link [29], Netgear [30], ToToLink [31], and Vivotek [32]. These firmware architectures include ARM and MIPS, the mainstream architectures used in embedded devices. As shown in Table 3, we selected ten firmware samples, six of which use the ARM architecture and four of which use the MIPS architecture.

Existing Tool. We compared our tool with SaTC, the state-of-the-art static bug hunter for embedded systems. It searches for input entries in back-end binary files by analyzing keywords shared between the front end and back end and utilizes a taint analysis to track data flow to detect vulnerabilities.

Bug Confirmation. Each alert generated by WFinder contains a call trace from the starting point to the receiving function, as well as corresponding input keywords. We distinguish between true and false positives based on whether the path is reachable. If we can manually generate a proof of crash (PoC) based on alerts and validate it on physical devices, we consider it a real bug.

Table 3. Dataset of device samples. We selected 10 device samples from 5 vendors: 9 routers and 1 camera. SizeP and SizeUP represent the averages size before and after unpacking, respectively.

Vendor	Device Series	Architecture	SizeP	SizeUP
Tenda	AX1803	ARM	41.7 MB	102.7 MB
	AX1806	ARM	46.8 MB	114.5 MB
	G1/G3	ARM	10.4 MB	62.5 MB
D-Link	DIR823G	MIPS	6.2 MB	31.8 MB
	DIR816	MIPS	3.9 MB	20.5 MB
ToToLink	X2000R	MIPS	12.5 MB	72.9 MB
	A3002R	MIPS	6.9 MB	37.5 MB
Netgear	R7000	ARM	32.3 MB	192.9 MB
	R7000P	ARM	41.7 MB	227.2 MB
vivotek	CC8160	ARM	36.4 MB	51.7 MB

5.2. Real-World Vulnerabilities

As shown in Table 4, WFinder detected 13 previously unknown bugs. These 13 bugs were first discovered by WFinder during the experimental process. We report newly discovered vulnerabilities to the CNVD in a non-public manner, and the CNVD will verify and confirm the vulnerabilities and be responsible for contacting the manufacturer for handling. At the time of writing this paper, 8 of these have been reported to the CNVD and assigned identifier numbers; all 13 are buffer overflow vulnerabilities.

Table 4. Zero-day discoveries using WFinder. We identified 13 new buffer overflow vulnerabilities; 8 of them were reported to the CNVD and assigned identifier numbers.

Device Series	Total	Bug IDs
AX1803	8	CNVD-2022-89238, CNVD-2022-89237, CNVD-2022-89236, CNVD-2023-03805, CNVD-2023-03806, CNVD-2023-03807, CNVD-2023-03809, CNVD-2023-00833
X2000R	2	unassigned
AX1806	3	unassigned

The 13 bugs we discovered are all buffer overflow vulnerabilities, including heap overflow and stack overflow. They are all web service programs on the firmware back end which, when processing data packets, obtain the parameter values and flow into dangerous functions without limiting their length, resulting in buffer overflows. We use real devices to verify vulnerabilities, and 13 vulnerabilities can cause DOS or even more harm to the device. In fact, the harm that buffer overflow vulnerabilities can cause is related to the protection mechanisms enabled by binary programs. Due to the limited resources provided by embedded devices, most defense mechanisms will not be enabled. In the firmware we analyzed, most of the firmware programs only enabled NX, while ASLR and canary were hardly enabled. This paper only describes the relevant work of vulnerability discovery, and the discovered vulnerabilities can be triggered through the constructed PoC. Further discussion on vulnerability exploitability analysis will not be conducted.

Comparison with SaTC. We compared WFinder with the state-of-the-art static analysis tool SaTC with respect to discovering vulnerabilities. In order to make the comparison fair, we added the danger function defined in SaTC to align it with WFinder. Table 5 shows our evaluation results. WFinder raised a total of 644 alerts, of which 346 were true positives and 225 were verified to be real vulnerabilities. SaTC raised 660 alerts, of which 422 were true positives, but only 183 were verified to be real vulnerabilities. The results indicate that WFFinder can identify more vulnerabilities. At the design level, WFinder adopts a similar approach to SaTC, identifying the correlation between the front end and back end to locate

the starting point of the taint analysis. However, there are significant differences in the final results. This is because SaTC did not consider that the user input data were introduced by an external input function. Therefore, although the result contains many true positives that can be reached by the path, the starting point may not be controlled by the user and cannot lead to vulnerabilities. In addition, SaTC's recognition of boundary binary files is also not entirely accurate, resulting in CC8160 being unable to detect vulnerabilities. Furthermore, we will discuss the impact of the taint analysis on the experimental results in Section 5.5.

Table 5. Comparison with SaTC. We list the number of alerts (Alerts), true positives (TPs), and vulnerabilities.

Device Series	Boundary Binary	SaTC			WFinder		
		Alert	TP	Vulnerability	Alert	TP	Vulnerability
AX1803	tdhttpd	64	42	16	45	31	29
AX1806	tdhttpd	61	45	19	51	29	25
G1/G3	httpd	53	31	14	59	29	17
DIR823G	goahead	64	43	21	70	39	23
DIR816	goahead	78	39	25	64	37	27
X2000R	boa	87	66	20	112	48	21
A3002R	boa	33	15	0	20	8	1
R7000	httpd	96	78	32	121	64	41
R7000P	httpd	124	63	36	98	58	40
CC8160	httpd	0	0	0	4	3	1
Total		660	422	183	644	346	225

5.3. Inferring External Data Entry Points

This paper analyzes the front-end and back-end correlation of the file systems of 10 firmware types and obtains each firmware's boundary binary programs and data import functions. According to the actual situation of each firmware, this paper compares the analysis results of boundary binary files with SATC, as shown in Table 6. Both Wfinder and SATC can recognize most boundary binaries. However, for vivotek_cc8160, SATC failed to accurately identify it because the boundary binary file httpd in cc8160 contains fewer parameter keywords. However, WFinder can successfully identify it by introducing HTTP headers and HTML tags as feature keywords. For the boundary binary file boa identified by dir823, we manually analyzed it and found that it was not the boundary binary file used by the device. After the file was excluded, we identified it again and successfully identified the boundary binary file Goahead. In addition, this paper lists the identified external data import functions. The table only lists the functions specifically used to extract the parameters of the HTTP request body and the identified configuration information setting functions. Since most HTTP header import functions are not fixed, they are not listed in this paper.

5.4. Sensitive Path Reduction Analysis

Because this paper filters the source point and sink point, the sensitive path is greatly reduced, as shown in Table 7. We count the number of danger functions filtered by WFFinder, the number of accurately identified external data input points, and the number of sensitive paths. We compare the results with SaTC. Using unfiltered hazard functions and external data input points containing many false positives as starting and ending points in SaTC generates a large number of sensitive paths, increasing the burden of stain analysis. On the contrary, due to the reduction in starting and ending points, WFinder obtains significantly fewer sensitive paths, which can greatly improve the efficiency of the taint analysis.

Table 6. External data entry point recognition. BB is the boundary binary; EDIF is the external data introduction function; CSF is the settings configuration function.

Device Series	Device Series	SaTC	Wfinder		
		BB	BB	EDIF	CSF
Tenda	AX1803	tdhttpd	tdhttpd	sub_4F4F4	Getvalue Setvalue
	AX1806	tdhttpd	tdhttpd	sub_295C8	Getvalue Setvalue
	G1/G3	httpd	httpd	websGetVar	Getvalue Setvalue
D-Link	DIR823G	boa	boa goahead	sub_40DF84 sub_41EA84	apmib_set apmib_get
	DIR816	goahead	goahead	websGetVar	nvram_get nvram_set nvram_bufset nvram_bufget
ToToLink	X2000R	boa	boa	sub_40F1F0	
	A3002R	boa	boa	sub_410510	
Netgear	R7000	httpd	httpd	sub_19644	acosNvramConfig_set acosNvramConfig_get
	R7000P	httpd	httpd	sub_1A760	acosNvramConfig_set acosNvramConfig_get
vivotek	CC8160	onvifd	httpd	sub_1A760	

Table 7. Sensitive path reduction. DF is the dangerous function, IE is the input entry, SP is the sensitive path, and FDF is the filtered dangerous function.

Series	Binary	SaTC			WFinder		
		DF	IE	SP	FDF	IE	SP
AX1803	tdhttpd	649	2847	1324	457	345	257
AX1806	tdhttpd	612	2721	977	429	367	276
G1/G3	httpd	1268	4304	63,523	462	598	1236
DIR823G	goahead	254	670	672	208	48	143
DIR816	goahead	316	427	1782	217	395	1322
X2000R	boa	909	1067	4,760,783	552	822	253,677
A3002R	boa	913	1171	3,524,763	505	785	245,343
R7000	httpd	3813	5509	170,774	2907	1774	3566
R7000P	httpd	3615	5621	175,822	2758	1764	3687

5.5. Efficacy of Taint Analysis

Based on reducing sensitive paths, we adopt lightweight function pollution specification which reduces the exploration path of the taint analysis and dramatically improves the efficiency of finding embedded web server vulnerabilities in firmware. As shown in Figure 6, compared with the well-known work SATC, it is found that the efficiency is improved by 40% on average; x2000r and a3002r were not completed within the specified time (12 h) due to too many sensitive call paths.

As shown in Table 8, we calculated the proportion of true positives in alerts (TP/Alert) and the proportion of vulnerabilities in true positives in the analysis results of each sample. We believe that the value of TP/Alert only reflects the accuracy of whether the taint analysis exploration path is reachable. In fact, whether the path is reachable does not necessarily mean that a vulnerability exists, so we also calculated Vulnerability/TP. We compared the statistical results of WFinder and SaTC. It was found that WFinder outperforms SaTC on

Vulnerability/TP. WFinder reduces false positives caused by factors other than the taint analysis by accurately locating the starting and ending points of the taint analysis. Therefore, true positives in WFinder results are more likely to be verified as vulnerabilities. In fact, the false positives of the taint analysis are difficult to avoid, and the cost of improving its accuracy is not worth it. We need to balance the efficiency and accuracy of the taint analysis. This is why WFinder is generally not as good as SaTC on TP/Alert. WFinder simplifies the function call specification for the taint analysis in exchange for higher analysis efficiency. But we also use accurate positioning of the starting and ending points to compensate for their false positives, reducing them to an acceptable range for manual verification. In fact, according to Table 5, the number of alerts we issued is not significantly different from SaTC, but our analysis efficiency has dramatically improved. In addition, due to the simplified function call specification, which can avoid path explosions encountered during the analysis process, we explored more vulnerabilities than SaTC.

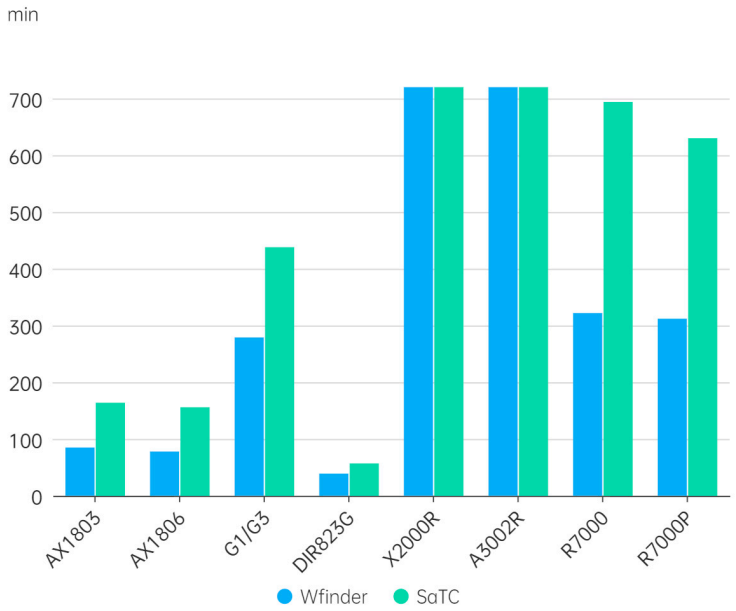

Figure 6. The efficiency of the taint analysis. We have listed the analysis time of each sample for WFFinder and SaTC.

Table 8. The accuracy of the taint analysis. We have listed the proportion of true positives in alarms and the proportion of verified vulnerabilities in true positives for each sample.

	Vendor	**AX1803**	**AX1806**	**G1/G3**	**DIR823G**	**DIR816**	**X2000R**	**A3002R**	**R7000**	**R7000P**
WFinder	TP/Alert	0.68	0.56	0.49	0.55	0.57	0.43	0.40	0.53	0.59
	Vulnerability/TP	0.93	0.86	0.58	0.58	0.72	0.43	0.13	0.64	0.69
SaTC	TP/Alert	0.66	0.73	0.58	0.67	0.5	0.76	0.45	0.81	0.50
	Vulnerability/TP	0.38	0.42	0.45	0.48	0.64	0.30	-	0.41	0.57

6. Discussion

We devised the vulnerability scanning tool WFinder specifically for embedded web servers. We analyze the correlation between the front end and back end, identify boundary binary files and external data inflow points based on the features of the front end and back end correlation, scan and filter the dangerous functions in the boundary binary files, and finally perform a lightweight taint analysis to identify potential vulnerabilities. WFinder successfully uncovers thirteen zero-day vulnerabilities across ten firmware samples, with eight of them already assigned CNVD identifiers. Our evaluation results indicate that WFinder surpasses the capabilities of cutting-edge tools in identifying errors within the firmware samples.

Future Work: In this study, we implement a preliminary vulnerability-scanning approach crossing the front end and back end. We analyze and observe real firmware programs, summarize the feature recognition of external data entry points in the back end, and design methods to balance efficiency and accuracy in the process of data flow analysis. Due to the balance between accuracy and efficiency, our method can efficiently explore more vulnerabilities, making it highly practical for firmware vulnerability analyses. However, to enhance efficiency during the taint analysis phase, we accepted a certain level of false positives, which can be manually reviewed. Moreover, while WFinder assesses the reachability of paths during the scanning process, it does not account for potential constraints on user inputs along these paths, which is also why some true positives cannot become vulnerabilities. Therefore, future efforts will focus on developing an appropriate automatic verification method to address these issues. We consider extracting relevant information from our static analysis process and combining it with dynamic fuzz testing to achieve the automated verification of potential vulnerabilities and PoC generation.

Author Contributions: Conceptualization: X.M. and Y.W. (Yunchao Wang); methodology: X.M.; formal analysis: X.M. and C.Y.; resources: Y.W. (Yunfeng Wang); writing—original draft preparation: X.M.; writing—review and editing: C.Y. and Y.W. (Yunchao Wang); supervision: Q.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research study was funded by National Key Research and Development Program of China, grant number 2019QY0500.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available upon request from the corresponding author. The data are not publicly available due to privacy.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. GSMA. June 2020. Available online: https://www.gsma.com/ (accessed on 1 June 2020).
2. Boofuzz. 2018. Available online: https://github.com/jtpereyda/boofuzz (accessed on 16 April 2018).
3. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2018.
4. Feng, X.; Sun, R.; Zhu, X.; Xue, M.; Wen, S.; Liu, D.; Nepal, S.; Xiang, Y. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, 15–19 November 2021; pp. 337–350.
5. Ma, X.; Zeng, Q.; Chi, H.; Luo, L. No More Companion Apps Hacking but One Dongle: Hub-Based Blackbox Fuzzing of IoT Firmware. In Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Service, Helsinki, Finland, 18–22 June 2023; pp. 205–218.
6. Chen, D.D.; Woo, M.; Brumley, D.; Brumley, D. Towards automated dynamic analysis for linux-based embedded firmware. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 21–24 February 2016; Volume 1, pp. 1.1–8.1.
7. Kim, M.; Kim, D.; Kim, E.; Kim, S.; Jang, Y.; Kim, Y. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In Proceedings of the Annual Computer Security Applications Conference, Austin, TX, USA, 7–11 December 2020; pp. 733–745.
8. Zheng, Y.; Davanian, A.; Yin, H.; Song, C.; Zhu, H.; Sun, L. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In Proceedings of the USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019; pp. 1099–1114.
9. Cheng, K.; Li, Q.; Wang, L.; Chen, Q.; Zheng, Y.; Sun, L.; Liang, Z. DTaint: Detecting the taint-style vulnerability in embedded device firmware. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 430–441.
10. Zhu, L.; Fu, X.; Yao, Y.; Zhang, Y.; Wang, H. FIoT: Detecting the memory corruption in lightweight IoT device firmware. In Proceedings of the 2019 18th IEEE International Conference On Trust, Security and Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), Rotorua, New Zealand, 5–8 August 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 248–255.

11. Redini, N.; Machiry, A.; Wang, R.; Spensky, C.; Continella, A.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1544–1561.
12. Chen, L.; Wang, Y.; Cai, Q.; Zhan, Y.; Hu, H.; Linghu, J.; Hou, Q.; Zhang, C.; Duan, H.; Xue, Z. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In Proceedings of the 30th USENIX Security Symposium, Virtual Event, 11–13 August 2021; pp. 303–319.
13. Liu, L.; Pan, Z.; Li, Y.; Li, Z. A static localization method for firmware vulnerabilities based on front-end and back-end correlation analysis. *Inf. Netw. Secur.* **2022**, *22*, 44–54.
14. Cheng, K.; Fang, D.; Qin, C.; Wang, H.; Zheng, Y.; Yu, N. Automatic inference of taint sources to discover vulnerabilities in soho router firmware. In Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection, Oslo, Norway, 22–24 June 2021; Springer International Publishing: Cham, Switzerland, 2021; pp. 83–99.
15. Wang, D.; Zhang, X.; Chen, T.; Li, J. Discovering vulnerabilities in COTS IoT devices through blackbox fuzzing web management interface. *Secur. Commun. Netw.* **2019**, *2019*, 1–19. [CrossRef]
16. Kampourakis, V.; Chatzoglou, E.; Kambourakis, G.; Dolmes, A.; Zaroliagis, C. Wpaxfuzz: Sniffing out vulnerabilities in wi-fi implementations. *Cryptography* **2022**, *6*, 53. [CrossRef]
17. Chatzoglou, E.; Kambourakis, G.; Kolias, C. Your wap is at risk: A vulnerability analysis on wireless access point web-based management interfaces. *Secur. Commun. Netw.* **2022**, *2022*, 1833062. [CrossRef]
18. Chatzoglou, E.; Kampourakis, V.; Kambourakis, G. Bl0ck: Paralyzing 802.11 connections through Block Ack frames. *arXiv* **2023**, arXiv:2302.05899.
19. Chatzoglou, E.; Kambourakis, G.; Kolias, C. How is your Wi-Fi connection today? DoS attacks on WPA3-SAE. *J. Inf. Secur. Appl.* **2022**, *64*, 103058. [CrossRef]
20. Srivastava, P.; Peng, H.; Li, J.; Okhravi, H.; Shrobe, H.; Payer, M. Firmfuzz: Automated iot firmware introspection and analysis. In Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, New York, NY, USA, 15 November 2019; pp. 15–21.
21. Bellard, F. QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference, FREENIX Track, Anaheim, CA, USA, 10–15 April 2005; Volume 41, p. 46.
22. Chen, L.; Cai, Q.; Ma, Z.; Wang, Y.; Hu, H.; Shen, M.; Liu, Y.; Guo, S.; Duan, H.; Jiang, K.; et al. SFuzz: Slice-based Fuzzing for Real-Time Operating Systems. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 7–11 November 2022; pp. 485–498.
23. Yu, B.; Wang, P.; Yue, T.; Tang, Y. Poster: Fuzzing iot firmware via multi-stage message generation. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 2525–2527.
24. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 138–157.
25. Binwalk. 2015. Available online: https://github.com/ReFirmLabs/binwalk (accessed on 1 February 2024).
26. Interactive Disassembler Professional. Available online: https://hex-rays.com/ida-pro/ (accessed on 1 February 2024).
27. VulFi. 2021. Available online: https://github.com/Accenture/VulFi (accessed on 1 February 2024).
28. Tenda. Available online: https://www.tenda.com.cn/ (accessed on 14 July 2022).
29. D-Link. Available online: http://www.dlink.com.cn/ (accessed on 6 August 2015).
30. Netgear. Available online: https://www.netgear.com/ (accessed on 27 May 2016).
31. ToToLink. Available online: https://www.totolink.cn/ (accessed on 28 February 2023).
32. Vivotek. Available online: https://www.vivotek.com/ (accessed on 24 July 2019).