


## Article

# Finding Taint-Style Vulnerabilities in Lua Application of IoT Firmware with Progressive Static Analysis

Xixing Li <sup>1,\*</sup> , Qiang Wei <sup>1,\*</sup>, Zehui Wu <sup>1,2</sup> and Wei Guo <sup>1,2</sup><sup>1</sup> School of Cyber Science and Engineering, Information Engineering University, Zhengzhou 450000, China<sup>2</sup> State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China

\* Correspondence: lxx.scholar@gmail.com (X.L.); prof\_weiqiang@163.com (Q.W.)

**Abstract:** With the rapid growth of IoT devices, ensuring the security of embedded firmware has become a critical concern. Despite advances in existing vulnerability discovery methods, previous research has been limited to vulnerabilities occurring in binary programs. Although an increasing number of vendors are utilizing Lua scripting language in firmware development, no automated method is currently available to discover vulnerabilities in Lua-based programs. To fill this gap, in this paper, we propose FLuaScan, a novel progressive static analysis approach specifically designed to detect taint-style vulnerabilities in Lua applications in IoT firmware. FLuaScan first heuristically locates the code that handles user input, then divides the code into different segments to conduct a progressive taint analysis. Finally, a graph-based search method is applied to identify vulnerable code that satisfies the conditions of taint propagation. To comprehensively compare FLuaScan with state-of-the-art tool TscanCode, we conducted various experiments on a dataset consisting of 13 real-world firmware samples from different vendors. The results demonstrate the superior performance of FLuaScan in terms of accuracy (increased TP rate from 0% to 42.50%), effectiveness (discovered 21 vulnerabilities, of which 7 are unknown), and practicality (acceptable time overhead and visual output to assist in manual analysis).

**Keywords:** vulnerability detection; IoT security; firmware; taint analysis; embedded devices



**Citation:** Li, X.; Wei, Q.; Wu, Z.; Guo, W. Finding Taint-Style Vulnerabilities in Lua Application of IoT Firmware with Progressive Static Analysis. *Appl. Sci.* **2023**, *13*, 9710. <https://doi.org/10.3390/app13179710>

Academic Editor: Agostino Forestiero

Received: 2 August 2023

Revised: 19 August 2023

Accepted: 21 August 2023

Published: 28 August 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

The rapid development of Internet of Things (IoT) technology has resulted in the proliferation of embedded devices in everyday life. Reports project that the number of IoT terminals worldwide will increase to 1.67 billion by 2023 and to 4.16 billion by 2025 [1]. Unfortunately, IoT manufacturers exhibit a lack of software security awareness, resulting in many IoT devices deployed on the Internet with either known or unknown vulnerabilities. Recent network attacks [2] have exposed the dire state of IoT security vulnerability, which significantly endangers the normal operation of both the network and the physical world. In 2016, Mirai compromised hundreds of thousands of DVRs and home routers to launch a DDoS attack against Internet mainstays like Netflix, Dyn, and GitHub [3]. Although numerous new technologies (e.g., edge computing [4–6] and artificial intelligence [7,8]) have been proposed and used to enhance network security, the current security situation of IoT devices remains weak [9]. It is crucial for the community to develop techniques that facilitate automatic vulnerability analysis of firmware.

Two fundamental methods for identifying firmware vulnerabilities are static analysis [10,11] and dynamic analysis [12–14]. Despite the progress made in dynamic analysis—for example, Firmadyne has successfully emulated the execution of over 1900 firmware images—this technique has several limitations, including low success rates and distorted simulation environments, leading to abnormal program behavior. Consequently, static analysis, which can ignore the complex environmental dependencies of firmware while detecting multiple vulnerabilities, is gaining interest in the

research community as a complementary method to dynamic analysis. For example, KARONTE [11] is a static analysis framework for embedded firmware that can discover vulnerabilities due to multibinary interactions. SaTC [10] also performs taint analysis to discover bugs in binary programs. It utilizes shared keywords related to user input in the front end and back end to infer the taint source.

Currently, the analysis target of most research on static analysis is limited to binary programs. This limitation is troublesome because IoT manufacturers are increasingly using scripting languages (especially Lua) in their development. Meanwhile, vulnerabilities in script programs are more harmful and easier to exploit, since they are often unrestricted by traditional security defense mechanisms such as ASLR, DEP, and Stack Canary. Despite the possibility of numerous high-risk vulnerabilities that could compromise device security, the academic community lacks methods to identify such bugs automatically.

To fill this gap, we propose *FLuaScan*, the first scalable software system to automatically expose vulnerabilities in the Lua applications of IoT firmware. *FLuaScan* first locates input processing code from complex multifile code based on the characteristics of dispatch. Then, we devised a progressive taint analysis method to address the limitations of current static taint analysis engines that hinder cross-functional and cross-file analysis. This method facilitates the examination of taint propagation from processing handlers to sink points and effectively performs static vulnerability analysis.

We conducted a comparative analysis of *FLuaScan* and existing tools on a total of 13 devices. The experiments demonstrate that *FLuaScan* exhibits higher accuracy, efficacy, and practicality than state-of-the-art tool *TscanCode*. *FLuaScan* achieves an average accuracy of 42% in detecting vulnerabilities within a duration of 30 s for real-world firmware images. Finally, we employed *FLuaScan* to identify a total of 21 vulnerabilities, 7 of which were previously undocumented.

The contributions of this paper are summarized as follows:

- We observed and analyzed the distribution patterns employed by vendors for different functional code, subsequently proposing a novel technique for precise localization of service handlers within vendor-made service programs;
- We designed and implemented *FLuaScan*, which utilizes progressive taint analysis to effectively discover taint-style vulnerabilities in the Lua applications of IoT firmware;
- We evaluated our tool against 13 firmware programs in popular, real-world embedded devices. Experiments demonstrate that *FLuaScan* outperforms the state-of-the-art tool in terms of correctness, effectiveness, and practicality. Specifically, we found 21 vulnerabilities, 7 of which were previously unknown.

## 2. Background

### 2.1. WEB Application in Firmware

IoT devices usually provide a web-based interface for end users to configure the system. In the early stage, the web server is usually written in C language. However, now, more and more vendors tend to use powerful scripting languages in development, especially Lua. Although there are no standards for how to implement this interface, many vendors prefer to utilize web technology because of its flexibility and simplicity. However, it has been proven that developing secure web applications is a non-trivial task [15]. Web servers have become the primary targets for hackers due to their vulnerabilities that can be remotely exploited, allowing for the acquisition of elevated privileges.

### 2.2. Taint-Style Vulnerabilities

Web applications often provide many features for end users and act correspondingly on user inputs and interactions, e.g., form submissions or clicks. A security vulnerability occurs when user-supplied data (i.e., taint) are not sufficiently sanitized and used in critical operations (i.e., sinks) of the application. Such vulnerabilities are known as taint-style vulnerabilities [16,17]. Taint-style vulnerabilities are a persistent security threat to web applications. Common types of vulnerabilities, such as command injection (CI),

SQL injection (SQLi), etc., are instances of taint-style vulnerabilities. An attacker might exploit such a flaw by providing malicious inputs to change the expected behavior of the application, e.g., injecting malicious code.

### 2.3. Vulnerability Detection Methods in Embedded Devices

In recent years, numerous efforts have been made to identify vulnerabilities in embedded devices. This section provides a detailed analysis of these methods, highlighting their strengths and weaknesses. Furthermore, we point out the shortcomings in the existing research.

We use Table 1 to organize existing vulnerability detection literature. The “Literature” column records the abbreviation of the literature. The “Dependence” column indicates the conditions on which the work relies. The “Technique” column represents the core technology used in the work. The “Purpose” column indicates the research objectives of the work. The “Target” column records the objects analyzed in the work.

**Table 1.** Summary of existing vulnerability detection methods in embedded device.

Category	Strategy	Literature	Dependence			Target	Technology	Purpose	
			Device	Fw	App				
Static analysis	Taint analysis-based method	KARONTE [11]		✓		Binary program in firmware	Multibinary interaction and taint analysis	taint style vulnerabilities (BOF/ CI)	
		SainT [18]		✓	✓	IoT app	Taint analysis	Sensitive information leak	
		DTaint [19]		✓		Binary program in firmware	Taint analysis	Taint-style vulnerabilities (BOF/ CI)	
		SATC [10]		✓		Binary program in firmware	Taint analysis	Taint-style vulnerabilities (BOF/CI)	
		CryptoREX [20]		✓		Binary program in firmware	IR-based analysis	Crypto misuse	
	pattern-matching-based method	Costin [21]		✓		Firmware	Correlation analysis	Backdoor	
		Stringer [22]		✓			Static data comparison	Backdoor	
		Firmalice [23]		✓			Symbolic execution and program slicing	Backdoor	
	dynamic Analysis	analysis basis	Firmadyne [24]		✓		Firmware	Full system emulation	Emulation purpose
			FirmAE [25]		✓			Arbitration emulation	Emulation purpose
Avatar [26]			✓	✓		Forward I/O and full-separation mode emulation		Dynamic analysis framework	
Fuzzing		FIRMAFL [12]		✓		Firmware	Greybox fuzzing	BOF/NPD	
		IoTFuzzer [27]	✓		✓		Blackbox fuzzing	BOF/NPD	
		RPFuzzer [28]		✓			Greybox fuzzing	DoS	
		SRFuzzer [29]	✓				Blackbox fuzzing	BOF/NPD/XSS	
		Snipuzz [14]	✓				Blackbox fuzzing and message snippet inference	NPD/DoS	
		Diane [30]	✓		✓		Blackbox fuzzing and app static analysis	BOF/DoS	
		FirmFuzz [31]		✓			taint analysis and structured symbolic expression	BOF/NPD CI/XSS	

BOF: buffer overflow; CI: command injection; NPD: null pointer dereference; DoS: denial of service; XSS: cross-site scripting.

According to the technical approach, there are two main categories: dynamic analysis and static analysis.

Dynamic analysis involves running the embedded devices in a certain way and conducting analysis during the runtime [32]. Such methods can be divided into two categories based on research strategy: emulation and fuzzing. Emulation focuses on running firmware to mimic the behavior of a real device as closely as possible. FIRMADYNE [24] is a firmware emulation framework designed for automated dynamic analysis in large-scale embedded firmware. Although it is promising, its emulation rate of network reachability and web service availability is considerably low. FIRMAE [25] uses several heuristics to address the problems encountered in FIRMADYNE and increases the emulation success rate. However, it can only handle observed cases and may not handle problems in new devices. AVATAR [26] enables complex dynamic analysis of embedded devices by orchestrating the execution of an emulator, together with the real hardware. FirmFuzz [31] employs a greybox-based generational fuzzing approach coupled with static analysis and system introspection to provide targeted and deterministic bug discovery within a firmware image. The above examples of dynamic analysis rely on emulation, but accurate emulation is a non-trivial task in practice due to the diversity of embedded devices [33]. Many works [34–39] have made advances in providing large-scale and reliable emulation environments for dynamic analysis, although many problems remain unsolved [13]. Emulation does not directly discover vulnerabilities, whereas fuzzing can [40]. Table 1 introduces some typical fuzzing works. RPFuzzer [28] is a fuzzing framework specifically designed to find protocol vulnerabilities in router devices. IoTfuzzer [27] attempts to find memory corruption vulnerabilities in IoT devices via their official apps; therefore, it is firmware-free. Snipuzz [14] represents a novel automated blackbox fuzz testing approach to overcome the challenge of applying syntax-based fuzz testing to various non-standard communication protocol formats widely used in IoT devices. Diane [30] views the functions that are executed before data-transforming functions but after input validation, as fuzzing triggers and uses them to fuzz IOT devices automatically.

Static analysis refers to the process of discovering vulnerabilities in a program without executing it. They can be divided into two categories based on research strategy: pattern-matching-based methods and taint-analysis-based methods. Pattern-matching-based methods identify inherent patterns in vulnerabilities and convert them into rules for automated vulnerability mining. Such methods are usually designed to find a backdoor in firmware. Costin [21] uses fuzzy hashing to match weak keys in the firmware and finds the similarity between different firmware images across four different dimensions through association analysis. Stringer [22] proposed a method based on static data comparison analysis to detect undisclosed features and hard-coded authentication backdoors in COTS device firmware. Taint analysis [17,41] is an effective approach to finding taint-style vulnerabilities. It tracks the propagation of taints introduced from external sources (e.g., untrusted user-supplied data) during program execution, checks if the tainted data could flow to critical program locations (sinks), and reports the sinks that could potentially be manipulated by attackers [42]. Several works have been proposed utilizing taint analysis methods for vulnerability detection in IoT devices. Saint [18] detected privacy-related vulnerabilities in IoT devices by tracing the information flow between sources and external sinks. To find cryptographic misuse vulnerabilities, CRYPTOREX [20] lifts binary code to a unified IR and performs static taint analysis across multiple executables. KARONTE [11] leverages static analysis techniques to perform multibinary taint analysis. SaTC [10] determines whether variable names are commonly shared between front-end files and back-end functions and leverages this information to enhance taint analysis.

In general, dynamic analysis has the advantage of low false-positive rates when discovering vulnerabilities, but its disadvantage is the need to address numerous dependency conditions and the difficulty of conducting large-scale analysis. Static analysis, on the other hand, has the advantage of requiring fewer dependency conditions and being suitable for large-scale analysis, but it has a higher false-positive rate. In this study, we chose to use

the static analysis approach to explore vulnerabilities in embedded devices, focusing on high-impact taint-style vulnerabilities.

However, in the academic community, the current research focus in vulnerability discovery is primarily limited to binary programs, leaving vulnerabilities in Lua applications reliant on manual identification.

#### 2.4. Tscancode and Semgrep

Although no tools have been designed to detect vulnerabilities in Lua applications of IoT firmware, several works support the analysis of Lua applications. To the best of our knowledge, TscanCode [43], which was released on GitHub with 1.7 k stars, can perform static analysis of Lua applications. It was developed by Tencent to help programmers identify code defects at the beginning of the design process. However, it was not designed for IoT scenarios, leading to a high-false negative rate when detecting firmware programs. Semgrep [44] is a fast, open-source static analysis engine for finding bugs, detecting dependency vulnerabilities, and enforcing code standards. Analysts can extract specific code snippets or perform taint analysis on programs using the interface provided by Semgrep. Additionally, Semgrep supports various programming languages, including Go, Java, C, and C++, among others. Owing to its robust capabilities and open nature, Semgrep has garnered widespread adoption and currently boasts 8.4k stars on GitHub. However, for the Lua language, Semgrep only supports intraprocedural analysis. Semgrep [44] plays a crucial role as an import component of the prototype system presented in this paper. Moreover, semgrep [44] offers an online playground (<https://semgrep.dev/playground/> accessed on 20 August 2023) that facilitates a more profound comprehension of the code snippets provided in the subsequent sections.

### 3. Empirical Analysis and Problem Statement

In this section, we first present an empirical analysis of our research object. Then, based on the results, we make a detailed statement of our research problem.

#### 3.1. Empirical Analysis

We conducted an empirical analysis to gain a precise understanding of the usage details of the Lua language in firmware. First, 552 firmware programs from 7 vendors were selected as the initial dataset. We wrote a program to automatically extract the system(rootfs) file from these firmware programs. Then, we traversed their rootfs to identify and record the language type of each script program (Lua, asp, php, etc.). The results regarding the Lua language are recorded in Table 2. The first row represents the brand of the images, and the second row represents the total number of images involved in the empirical analysis. If the number of Lua programs in the file system exceeds 50, we classify an image as special, while the third row records the number of special images. The fourth row (%) records the percentage of special images among all surveyed images, while the last row records the average number of Lua programs in the special images. The results presented in Table 1 reveal three facts.

**Table 2.** Information about representative experimental platforms in empirical analysis.

Brand	Xiaomi	Tplink	Trendent	Linksys	Netgear	Totolink	Zyxel
Images	192	133	81	53	39	23	31
Lua > 50	184	69	25	16	5	5	9
%	95.83%	51.88%	30.86%	30.19%	12.82%	21.74%	29.03%
Avg.	209.39	168.39	173.68	149.63	125	117.34	95

Fact-1: IoT vendors widely use Lua programs in their product development, accounting for up to 56.70% of all surveyed images.

Fact-2: When vendors use Lua for development, it is usually in the form of multiple files.

Fact-3: The use of Lua programs is not limited to a single vendor but multiple vendors.



### 3.2. Research Scope and Research Goals

Fact-1 mentioned above indicates the significant prevalence of Lua program usage in firmware. However, the detection of vulnerabilities in these programs lacks an effective approach. Vulnerabilities in web servers are commonly high-risk, and Lua is widely used in IoT development. Consequently, in this work, we aim to strengthen static analysis methods for the identification of taint-style vulnerabilities within the Lua-based web server of firmware. We focus on data-flow analysis, despite the absence of a dedicated solution for Lua programs. Nevertheless, its theoretical foundation has found extensive utilization in both academic [10,11,45,46] and industrial [47] domains.

### 3.3. Research Challenge

Based on the facts presented in Section 3.1, designing and implementing a static taint analysis tool for Lua applications in firmware is challenging.

First, the development environment for IoT devices is heavily fragmented, with manufacturers exhibiting preferences for diverse frameworks. Most existing static analysis methods are designed for desktop systems, while their web applications often utilize popular frameworks. These frameworks have comprehensive documentation, which considerably aids in conducting static analysis. However, these methods cannot be directly ported to firmware objects. Meanwhile, as Fact-3 shows, multiple vendors use Lua to develop IoT devices. Furthermore, usually, they have their respective framework but without documentation. Consequently, data-flow analysis encounters challenges in pinpointing the initial input source from users.

Second, we need to propose a new interprocedural and cross-file taint-tracking algorithm to compensate for failures of existing methods. As Fact-2 shows, Lua applications in firmware are complicated and usually consist of various files. Therefore, a taint analysis with interprocedural and cross-file support is a prerequisite for accurate vulnerability detection. However, existing tools can only support intraprocedural analysis.

Thirdly, it is necessary to establish vulnerability models in the IoT field to rectify the shortcomings of current approaches. However, this is a formidable undertaking due to the intricate interdependence between software and hardware within IoT devices. Moreover, the web servers of these devices necessitate intricate interactions with back-end programs and underlying system modules.

## 4. Motivation

In this section, our core insight for solving this problem is demonstrated by utilizing motivating examples. After analyzing the web server based on Lua of multiple vendors, we discovered a common characteristic. In Lua applications, a dispatch module is commonly utilized to parse requested packets from users and direct them to corresponding handlers for real processing. Although different vendors may implement similar functionalities differently in web server development, a general convergence can be observed in their dispatch modules. Two typical approaches for implementation of the dispatch module are found, as illustrated in Listings 1 and 2.

As lines 2–4 show in Listing 1, the web server in the Xiaomi R3600 router dispatches different service handlers by registering different entries. Service handler refers to the code snippet implementing the desired work of a single interaction, such as the function *setConfigIotDev* in Listing 1, which is responsible for configuring the router's wireless network. The term "registering entries" is coined to describe this type of dispatch method. As for the TP-Link Archer AXE75 router, its web server dispatches different service handlers through the *dispatch\_tbl* function in the Listing 2. In the same way, we named this kind of dispatch a "handler table".

There is a command injection vulnerability at line 21 of Listing 1. However, existing tool TscanCode [43] fails to find this vulnerability. The root cause is that TscanCode fails to locate the service handler and therefore does not apply precise data-flow analysis. Therefore, our insight is leveraging the dispatch module to enhance taint analysis. Specifically, we

first locate the dispatch module by analyzing its appeal characteristics, then further extract the service handler. Afterward, taint analysis is launched starting from the service handler.

**Listing 1.** Code snippet from Xiaomi R3600.

```

1 function index()
2   entry({"api", "misystem", "set_band"}, call("setBand"), (""), 123)
3   entry({"api", "misystem", "set_config_iotdev"}, call("setConfigIotDev"), (""), 221)
4   entry({"api", "misystem", "get_unconfig_iotdev"}, call("getunConfigIotDev"), (""), 222)
5 end
6 function setConfigIotDev()
7   local XQFunction = require("xiaoqiang.common.XQFunction")
8   local LuciUtil = require("luci.util")
9   local result = { ["code"] = 0 }
10  local ssid = LuciHttp.formvalue("ssid")
11  local bssid = LuciHttp.formvalue("bssid")
12  local uid = LuciHttp.formvalue("user_id")
13  if XQFunction.isStrNil(ssid)
14    or XQFunction.isStrNil(bssid)
15    or XQFunction.isStrNil(uid) then
16    result.code = 1523
17  end
18  if result.code ~= 0 then
19    result["msg"] = XQErrorUtil.getErrorMessage(result.code)
20  else
21    XQFunction.forkExec("connect -s " .. ssid .. " -b " .. bssid .. " -u " .. uid)
22  end
23  LuciHttp.write_json(result)
24 end

```

**Listing 2.** Code snippet from TP-Link Archer AXE75.

```

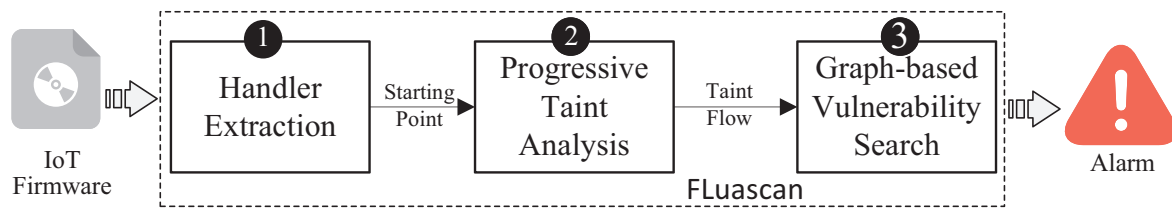
1 local dispatch_tbl = {
2   lang = {
3     ["read"] = {cb= get_sysinfo}, ["write"] = {cb = set_sysinfo}, ["mcu_read"] = {cb = get_mcu_sysinfo}},
4   index_lang = {
5     ["read"] = {cb= get_sysinfo}, ["write"] = {cb = set_sysinfo}, ["mcu_read"] = {cb = get_mcu_sysinfo}},
6   mobile_lan = {
7     ["read"] = {cb = get_sysinfo_mobile}, ["write"] = {cb = set_sysinfo_mobile} },
8   country = {
9     ["read"] = {cb = get_country}, ["write"] = {cb = set_country} },
10  country_list = {
11    [".super"] = {cb = get_country_list}, ["read"] = {cb = get_country_list} }
12 }
13 function dispatch(http_form)
14   return ctl.dispatch(dispatch_tbl, http_form)
15 end
16 function set_country(http_form)
17   local uci_r = uci.cursor()
18   local country = http_form.country
19   local old_country = uci_r:get(LOCALECFG, "sysinfo", "country") or "no_country"
20   uci_r:set(LOCALECFG, "sysinfo", "country", country)
21   uci_r:commit(LOCALECFG)
22   local config = require "luci.sys.config"
23   config.merge_config_by_country(old_country)
24   return true
25 end

```

## 5. Design and Implementation

### 5.1. Overall Design

Based on the above insights, we designed a prototype system, FLuaScan, which takes IoT firmware as initial input and ultimately outputs suspicious vulnerability points, along with their data flow details. As shown in Figure 1, FLuaScan consists of three tightly coupled modules.



**Figure 1.** Overview of FLuaScan.

As indicated by ❶ in Figure 1, the handler location module automatically extracts Lua applications from the firmware. Then, FLuaScan utilizes heuristic methods based on the characteristics described in Section 4 to identify the dispatch mode and locate service handlers.

Then, FLuaScan carries out a progressive taint analysis (❷ in Figure 1) starting from these service handlers. During the analysis process, our system divides multiple Lua files into segments and performs taint analysis in a segmented manner.

The final module is the graph-based vulnerability search module (❸ in Figure 1). It records the taint flows from module ❷ and organizes them as a graph based on their relationships. Then, FLuaScan utilizes graph search algorithms to identify the execution paths in dangerous functions that accept user input without sanitization and reports them as potential vulnerability points.

### 5.2. Handler Extraction

Given an IoT firmware, FLuaScan first extracts rootfs from the image file. FLuaScan subsequently traverses the rootfs, extracting all Lua files and recording their respective paths.

Thirdly, FLuaScan conducts a scan of each source file using semgrep. By applying the rules presented in Listing 3, two dispatch methods can be identified. In these cases, \$F represents the function name of the handler, and the locations of these functions are recorded. Then, FLuaScan conducts an initial analysis of each handler to identify user inputs and pinpoint the initial point for data flow analysis. The user input of the web server sources from the HTTP protocol, so FLuaScan identifies the user-controllable portions in the HTTP request packets. The web server receives user input through the HTTP protocol. FLuaScan extracts the user-controllable portions from the HTTP request packets to find the taint source. As shown in lines 6 to 8 shown in Listing 4, three parts are user-controllable: the form data, environmental variables, and cookies. To record the user-controllable variables and support the following graph-based search, a novel concept called the analysis node is introduced.

**Listing 3.** Code snippet to extract service handler utilizing semgrep.

```

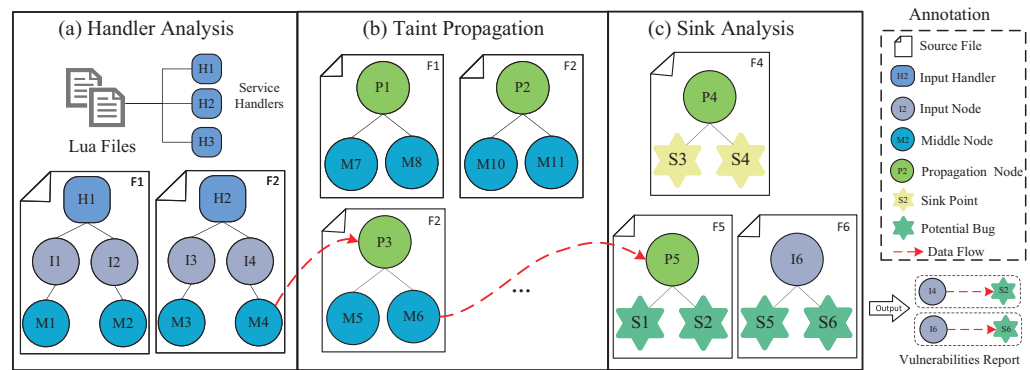
1 rules:
2   - id: FLuaScan-handler-extract
3     languages: [lua]
4     pattern-either:
5       - pattern: entry($ENTRY, call($F),...)
6       - pattern: $ENTRY = {cb = $F}
7     message: $ENTRY -> $F

```

$$N_p :< attr, pos, type > \quad (1)$$

As shown in Equation (1), the analysis node is defined as a 3-tuple, including *attr*, *pos*, and *type*. The *type* field is used to distinguish between different analysis nodes. For user-controlled variables, the type is denoted as *I*, and the node name is represented as *I<sub>x</sub>* as an abbreviation (like I1 in Figure 2). The *attr* field is used to record additional information. Taking user-controllable variables as an example, the *attr* field records the function name of the handler. The *pos* field is responsible for recording the specific position of the node, including the file path and the starting and ending offset of the variable in the file.





**Figure 2.** The overview of progressive taint analysis engine.

**Listing 4.** Code snippet to execute handler analysis utilizing semgrep.

```

1 rules:
2   - id: fluascan-handler-analysis
3     mode: taint
4     pattern-sources:
5       - pattern-either:
6         - pattern: $FV = $HTTP.formvalue(...)
7         - pattern: $FE = $HTTP.getenv(...)
8         - pattern: $FV = $HTTP.getcookie(...)
9     pattern-sinks:
10      - patterns:
11        - pattern: $MOD.$FUNC(...)
12        - pattern-not: $HTTP.formvalue(...)
13        - pattern-not: $HTTP.getenv(...)
14        - pattern-not: $HTTP.getcookie(...)
15        - pattern-not: LuciHttp.write_jsonp(...)
16        - pattern-not: lutil.split(...)
17        - pattern-not: $M.isStrNil(...)
18        - pattern: uci:set(..)
19    pattern-sanitizers:
20      - pattern-either:
21        - pattern: XQFunction._cmdformat(...)
22        - pattern: tonumber(...)
23    languages: [lua]

```

### 5.3. Progressive Taint Analysis

Once user-controllable variables have been identified, FLuaScan employs taint analysis to expose vulnerabilities. However, the current limitation of semgrep [44] is that it exclusively supports intraprocedure analysis, potentially leading to false positives in vulnerability detection. To overcome this issue, we designed a progressive taint analysis engine. This engine divides multiple Lua files into segments and conducts taint analysis on each segment. It also records taint propagation across various functions and files to establish their relationships. FLuaScan initially employs the ctags [48] syntax analysis tool to parse each Lua file and assigns a distinct identifier to each object. Subsequently, this identifier is utilized for correlation analysis. Next, as shown in Figure 2, FLuaScan performs three rounds of taint analysis on all files: (a) handler analysis, (b) taint propagation, and (c) sink analysis.

During the (a) handler analysis process, FLuaScan aims to identify the data flow from user input to middle nodes. Middle nodes are defined as analysis nodes that have the potential to facilitate taint propagation between different functions. These nodes can be categorized into two types: (1) Invocation of subfunctions; during this process, taint can

potentially propagate with the transfer of parameters. (2) Database write operations; during this process, taint can potentially propagate after being written to a database, then retrieved again. The *type* field of the middle mode is set as M (like M2 in Figure 2). The semgrep configuration used to accomplish this task is illustrated as Listing 4. We employed the taint mode of semgrep and designated the three user-controllable parts in the HTTP protocol as the sources of tainted data. As for the sink points, in Listing 4, lines 11–17 aim to find the invocation of subfunctions, while line 18 aims to find the database write operations. In Listing 4, lines 21–22 show the designs of sanitizers, including two categories: (1) vendor-customized security filtering functions, which perform security checks and filtering on variable content, and (2) string conversion functions, such as *tonumber*, making the user input unable to trigger vulnerabilities after the conversion.

During the (b) taint propagation process, *FLuaScan* aims to identify data flows within the invocation of subfunctions. The semgrep configuration utilized to achieve this task is depicted in Listing 5. The taint mode of semgrep is consistently used, while the configuration of sinks and sanitizers remains unchanged. However, the taint sources are been changed, consisting of two types: (1) parameters passed when a subfunction is called (lines 5–10 in Listing 5) and (2) data retrieved from the database (line 12 in Listing 5). The taint sources from subfunctions are defined as propagation nodes (like P1 in Figure 2), with their *type* field set to P.

**Listing 5.** Code snippet to execute handler analysis utilizing semgrep.

```

1 rules:
2   - id: fluascan-taint-propagation
3     mode: taint
4     pattern-sources:
5       - patterns:
6         - pattern-inside: function $FNAME($VAR,...).end
7         - focus-metavariable: $VAR
8       - patterns:
9         - pattern-inside: function $FNAME($VAR).end
10        - focus-metavariable: $VAR
11      - patterns:
12        - uci:get(...)

```

There is ample evidence to suggest that the incorrect utilization of vulnerable functions serves as the root cause of numerous vulnerabilities. Hence, precise modeling of vulnerable functions is imperative to detect vulnerabilities effectively. Current methods disregard the specificities of IoT scenarios, only perceiving a limited number of sensitive system APIs as vulnerable. During the development process, it has been observed that numerous manufacturers tend to encapsulate low-level system APIs into higher-level custom APIs. Overlooking these interfaces can lead to erroneous identification of vulnerabilities. Hence, we decided to identify these custom APIs and continue to employ taint analysis with semgrep to accomplish this goal. In the specific configuration shown in Listing 6, the tainted sources are defined as the function's parameters, while the underlying foundational command execution functions (like *io.popen* and *os.execute*) are designated as the sink points. Finally, *FLuaScan* discovered many custom APIs that are specific to various vendors, such as *XQFunction.forkExec*, *LuciUtil.exec*, and *subprocess.call*. Furthermore, these sensitive APIs are defined as sink nodes (like S1 in Figure 2).

**Listing 6.** Code snippet to locate custom APIs utilizing semgrep.

```

1 rules:
2   - id: fluascan-sensitive-api-find
3     mode: taint
4     pattern-sources: <same as listing 5 >
5     pattern-sinks:
6       - pattern-either:
7         - pattern: io.popen(...)
8         - pattern: os.execute(...)
9         - pattern: Nixio.exec(...)

```

Once these custom APIs are identified, the (c) sink analysis process aims to identify the incorrect usage of vulnerable functions. Unsafe usage can be classified into two types: direct propagation, which involves the direct propagation of user inputs to vulnerable APIs within a function, and indirect propagation, which refers to the propagation of user inputs across functions via middle nodes to access vulnerable APIs. *FLuaScan* utilizes semgrep to achieve this objective with the configuration presented in Listing 7.

**Listing 7.** Code snippet to locate custom APIs utilizing semgrep.

```

1 rules:
2   - id: fluascan-sink-analysis
3     mode: taint
4     pattern-sinks:
5       - pattern-either:
6         - pattern: XQFunction.forkExec(...)
7         - pattern: LuciUtil.exec(...)
8         - pattern: io.popen(...)
9         - pattern: os.execute(...)
10        - pattern: sys.fork_exec(...)
11        - pattern: subprocess.call(...)
12        - pattern: sys.call(...)
13        - pattern: luci.sys.call(...)
14        - pattern: sys.fork_call(...)
15        - pattern: db.exec(...)
16     pattern-sources:
17       - patterns:
18         - pattern-inside: function $FNAME($VAR,...)...end
19         - focus-metavariable: $VAR
20       - patterns:
21         - pattern-inside: function $FNAME($VAR)...end
22         - focus-metavariable: $VAR
23       - pattern-either:
24         - pattern: $FV = $HTTP.formvalue(...)
25         - pattern: $FE = $HTTP.getenv(...)
26         - pattern: $FV = $HTTP.getcookie(...)
27         - pattern: db.exec(...)

```

Therefore, in the taint source setup, lines 24–27 in Listing 7 focus on the direct propagation, while lines 17–22 focus on indirect propagation. In the sink point setup, both previously discovered custom vendor APIs and common sensitive APIs are taken into account.

#### 5.4. Graph-Based Vulnerability Search

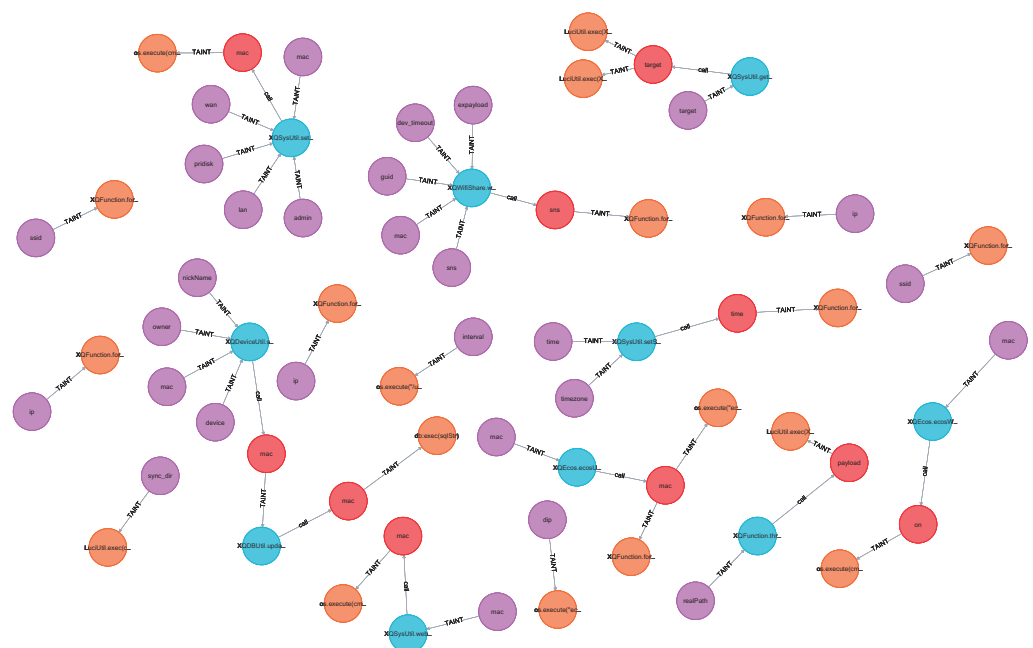
Upon completion of progressive taint analysis, the details of taint propagation between distinct snippets were elucidated. However, vulnerability mining requires finding a com-

plete path from the input to the sink point. Therefore, FLuaScan proposes a graph-based vulnerability search method. Drawing on the previously mentioned concept of analysis nodes, vulnerability detection is redefined as a graph search problem. Specifically, our initial step involves utilizing the Neo4j database to store all nodes that have been analyzed in prior stages. Then, *FLuaScan* creates three labels (INode, MNode, PNode, and SNode) in the Neo4j database corresponding to three different types of nodes. *FLuaScan* inserts INode and MNode into the Neo4j database, creating the internal taint relationships, based on the result of the handler analysis process. Subsequently, based on the results shown in (b) and (c), the relationships between PNode and MNode, as well as between PNode and SNode, are processed in a similar manner. The necessary condition for the existence of a vulnerability is to find a path from INode to SNode. Currently, finding such a path is not possible due to our limitation in obtaining only the relationships of direct contamination within different code snippets. The analysis of subfunctions is essential to connect these distinct code snippets into a whole path. During the progressive taint analysis, in the (a) process, the invocation of subfunctions acts as the sink point and is represented as MNodes in the Neo4j database. In contrast, in the (b) and (c) processes, the parameters of subfunctions function as taint sources and are represented as PNodes in the Neo4j database. *FLuaScan* examines the *pos* attributes of all MNodes and PNodes. A relationship between the MNode and the PNode is established if they pertain to the same subfunction. Finally, *FLuaScan* employs a Cypher query to search for paths that start from the INode, traverse multiple MNodes, and end at the SNode. Detail of the Cypher query are demonstrated in Listing 8.

**Listing 8.** Cypher query to detect vulnerability.

```
1 MATCH p=(a:INode)-[*]->(b:SNode) return p
```

To assist users in manual analysis, FLuaScan provides an interactive web page displaying all identified suspicious vulnerability points. Figure 3 (raw figure at <https://sm.ms/image/L8VRorXCgnUSefk> accessed on 20 August 2023) shows a screenshot of the web page, which represents the paths that meet the vulnerability requirements.



**Figure 3.** The visual result of a graph-based vulnerability search in FLuaScan.

The purple, red, blue, and yellow nodes, respectively, represent INodes, MNodes, PNodes, and SNodes, respectively. In addition to this web page, FLuaScan also includes a

VSCode plugin, assisting in manual analysis. With the assistance of this plugin, comments are automatically added at the positions of analysis nodes and displayed with different highlight colors based on their type.

## 6. Experiment

To ensure thorough testing of FLuaScan, three evaluation questions are proposed.

Q1: **Correctness:** What is the number of true positives generated by FLuaScan? (Section 6.1);

Q2: **Effectiveness:** Can FLuaScan detect vulnerabilities in real-world devices? (Section 6.2);

Q3: **Practicability:** How are the time overhead and actual user experience of FLuaScan? (Section 6.3).

To bolster the credibility of the experiment, we painstakingly curated a diverse test set consisting of multiple devices sourced from various vendors. As shown in Table 3, the test set comprises firmware from 13 devices obtained from 7 distinct vendors. The final row documents the count of Lua files present within the firmware. The test set has an average of 147 Lua files per image, satisfying the requirement of the research objective for multiframe detection. In order to evaluate the advancement of FLuaScan, we also compared it with current state-of-the-art tool TscanCode in the experiment.

**Table 3.** Information about the test set.

#	Brand	Device	Version	Lua Files
1	xiaomi	r3a	V2.12.8	202
2	xiaomi	r3600	V1.0.17	209
3	xiaomi	r4c	V2.28.48	207
4	tplink	TL-WR1043N	V5_161118	140
5	tplink	Archer-C1900	V2.0_220524	156
6	tplink	Archer-AXE75	V1_220405	215
7	zyxel	NBG6716	V1.00	69
8	zyxel	NBG6604	V1.00	117
9	linksys	EA6350V2	V2.1.2.184309	137
10	totolink	A6000R	V1.0.1	140
11	totolink	X5000R	V9.1.0cu.2089	114
12	trendent	TEW-821DAP	V1.08B04	115
13	netgear	EX8000	V1.0.1.164	91

### 6.1. Correctness

We conducted vulnerability detection on each firmware in the test set using FLuaScan and TscanCode. These two tools output some alerts; then, we will hire three expert vulnerability hunters to conduct a second confirmation of these suspicious vulnerability points. If experts determine that the danger can be excluded, it is classified as a false positive (FP); otherwise, it is classified as a true positive (TP). The detailed experimental results are recorded in Table 4, where the “alerts”, “TP”, and “FP” columns represent the total number of potential vulnerability points identified by the tool, the number of confirmed vulnerabilities after secondary verification, and the number of false positives, respectively. In addition, the “Handlers” column records the number of service handlers identified by FLuaScan from Lua applications.

As shown in Table 3, TscanCode exhibits a significantly higher number of alerts compared to FLuaScan. However, TscanCode fails to discover actual vulnerabilities. FLuaScan

achieved an average accuracy rate of 42.50% on this test set. The success of FLuaScan in achieving exceptional results can be attributed to its precise extraction of handlers. In contrast, TscanCode not only fails to achieve this but also struggles to identify vulnerabilities accurately. Meanwhile, for the ninth firmware, as FLuaScan was unable to locate the handler that processes user input, no vulnerabilities were discovered.

**Table 4.** Detailed result of the test set.

#	Brand	Device	FLuaScan				TscanCode		
			Alerts	TP	FP	Handlers	Alerts	TP	FP
1	xiaomi	r3a	16	5	11	448	149	0	149
2	xiaomi	r3600	17	6	11	488	179	0	179
3	xiaomi	r4c	10	5	5	399	166	0	166
4	tplink	TL-WR1043N	3	1	1	347	141	0	141
5	tplink	Archer-C1900	2	1	1	321	138	0	138
6	tplink	Archer-AXE75	2	1	1	820	297	0	297
7	zyxel	NBG6716	3	2	1	92	90	0	90
8	zyxel	NBG6604	3	2	1	65	118	0	118
9	linksys	EA6350V2	0	0	0	0	134	0	134
10	totolink	A6000R	7	3	4	113	152	0	152
11	totolink	X5000R	7	3	4	91	163	0	163
12	trendent	TEW-821DAP	2	1	1	37	122	0	122
13	netgear	EX8000	8	4	4	43	96	0	96

### 6.2. Effectiveness

After organizing all the suspicious vulnerabilities identified by FLuaScan, we ultimately discovered 21 vulnerabilities, of which 7 are unknown. We meticulously compiled a comprehensive summary of the information about these vulnerabilities and organized it in Table 5. The third, fourth, and fifth columns represent the affected devices, the function names where the vulnerabilities are located, and the types of vulnerabilities, respectively. The vulnerabilities FLuaScan found can be categorized into two types: command injection (CI) and SQL injection (SQLi). The sixth column (MNode) in the table reveals the count of middle nodes through which the vulnerability traverses from user input to the sensitive function. Each middle node signifies a cross-file invocation. The results indicate that the majority of vulnerabilities involve cross-file invocations, highlighting the necessity of cross-file static analysis. However, current state-of-the-art tool TscanCode is unable to accomplish this. The seventh column represents whether the vulnerability is known, while the last column records the severity level of the vulnerability. The results highlight two distinct characteristics. First, a vulnerability can potentially affect multiple products within the same vendor. Second, the vulnerabilities we uncovered exhibit a significant level of risk, all falling into moderate to high-severity categories. These two characteristics are similar to those summarized in other literature reports, indicating the universality of our findings.

### 6.3. Practicality

To evaluate the practicality of FLuaScan, we adopted a combined approach of quantitative and qualitative evaluation. In quantitative terms, we separately recorded the runtime

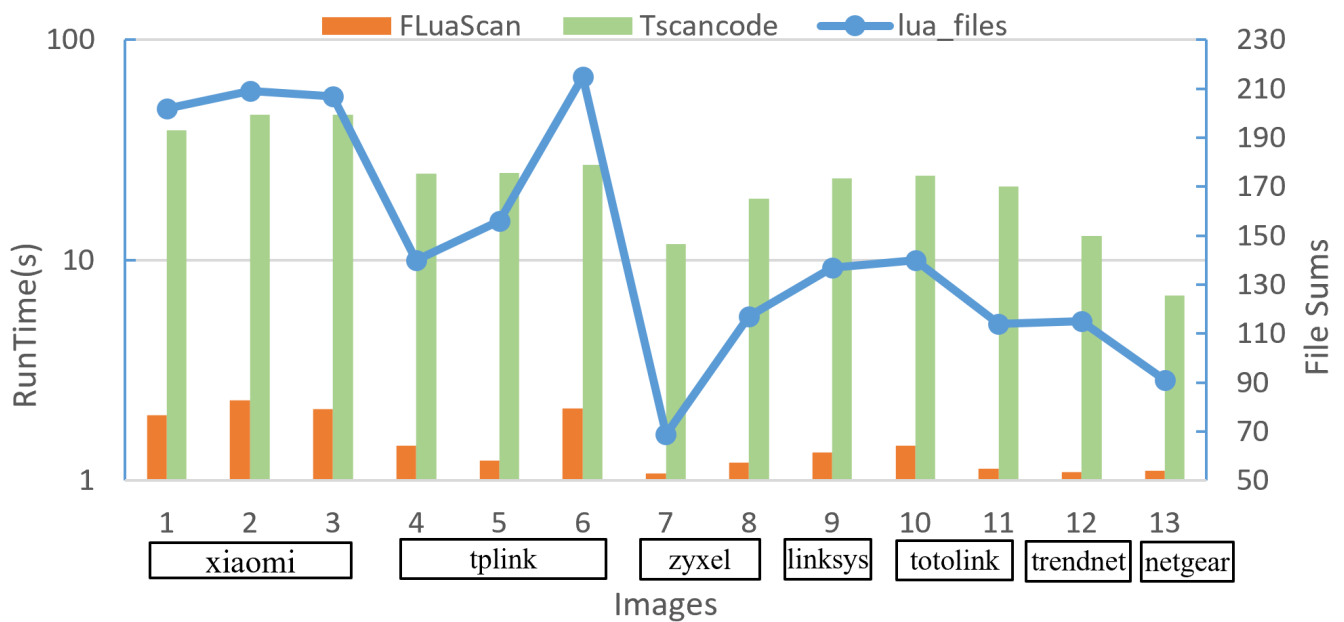


of FLuaScan and Tscancode to detect the same image. In Figure 2, we present these results in the form of bar and line charts. The brown and green bars represent the runtime of FLuaScan and Tscancode, respectively, while the blue line represents the number of Lua files.

As shown in Figure 4, within the same brand, as the number of Lua files increases, the runtime of both Tscancode and FLuaScan also increases. However, regarding the same image, FLuaScan always takes more time than Tscancode. FLuaScan has an average runtime of 25.16 s, while Tscancode has an average runtime of 1.51 s. The primary factor contributing to the higher time overhead of FLuaScan is its greater analytical depth in comparison to Tscancode, in addition to the extra operations involving database writing and analysis. Meanwhile, on average, FLuaScan can complete vulnerability detection for a real-world device image in less than half a minute. Considering the higher accuracy of FLuaScan, we believe that this time overhead level is acceptable. Since security analysts typically need to manually verify the results provided by automated tools, it is essential for the tool to be able to collaborate with manual code auditing.

**Table 5.** The vulnerabilities found by FLuaScan.

#	Vendor	Device	Vuln Function	Type	MNode	0 Day	Severity
1	xiaomi	r3a,r4c	setconfigIoTDev	CI	0	Known	High
2	xiaomi	r4c,r3a,r3600	wifi_access	CI	2	Known	High
3	xiaomi	r3a,r4c	getThumb	CI	1	Known	High
4	xiaomi	r3a,r4c	setSyncRouterFile	CI	2	Unknown	High
5	xiaomi	r3a,r4c	setDeviceNickName	SQLi	3	Unknown	Middle
6	xiaomi	r3a,r4c	updateDeviceNickname	SQLi	3	Unknown	Middle
7	xiaomi	r3a,r4c	backupData	CI	3	Known	High
8	tplink	TL-WR1043N,Archer-AXE75, Archer-C1900	set_country	CI	3	Known	High
9	tplink	Archer-AXE75	vpn_user_update	CI	0	Known	High
10	zyxel	NBG6604,NBG6716	multiple_ssid	CI	1	Known	High
11	zyxel	NBG6604,NBG6716	multiple_ssid5G	CI	1	Known	High
12	zyxel	NBG6604,NBG6716	wlanwpsstation	CI	1	Known	High
13	zyxel	NBG6604,NBG6716	action_wireless	CI	1	Known	High
14	totolink	A6000R	enable_wsh	CI	0	Known	High
15	totolink	A6000R,X5000R	action_reboot	CI	0	Unknown	High
16	totolink	A6000R,X5000R	reset_wifi	CI	0	Known	High
17	netgear	EX8000	iface_shutdown	CI	1	Unknown	High
18	netgear	EX8000	iface_reconnect	CI	1	Unknown	High
19	netgear	EX8000	iface_delete	CI	1	Unknown	High
20	netgear	EX8000	action_bandwidth	CI	1	Known	High
21	trendnet	TEW-821DAP	action_bandwidth	CI	0	Known	High



**Figure 4.** The runtime of FLuaScan and Tscancode to detect vulnerabilities in the test set.

We also compared the usage mode of TscanCode and FLuaScan in qualitative terms. The output of TscanCode is relatively simple, as it only includes the locations of the issues and is separated from the original code, making it difficult to assist in manual analysis directly. FLuaScan offers two visual output options: (1) using interactive graphs to display the specific propagation of vulnerabilities and tainted data flow, as shown in Figure 3, and (2) As shown in Figure 5, highlighting sensitive snippets of vulnerable code based on their attributes with the assistance of the VSCode plugin. In conclusion, FLuaScan's visual output options and acceptable time cost make it more practical than TscanCode.

```
function setConfigIotDev()
    local XQFunction = require("xiaoqiang.common.XQFunction")
    local LuciUtil = require("luci.util")
    local result = {
        ["code"] = 0
    }

    local ssid = LuciHttp.formvalue("ssid")--[TAINT-SRC]
    local bssid = LuciHttp.formvalue("bssid")--[TAINT-SRC]
    local uid = LuciHttp.formvalue("user_id")--[TAINT-SRC]

    if XQFunction.isStrNil(ssid)..
        or XQFunction.isStrNil(bssid)..
        or XQFunction.isStrNil(uid) then
        result.code = 1523
    end
    if result.code ~= 0 then
        result["msg"] = XQErrorUtil.getErrorMessage(result.code)
    else
        XQFunction.forkExec("connect -s "..ssid.." -b "..bssid.." -u "..uid)--[TAINT-SINK]
    end
    LuciHttp.write_json(result)
end
```

**Figure 5.** The vulnerability source code highlighted with different colors supported by FLuaScan's VSCode plugin.

Based on the above results, we have presented the differences between FLuaScan and TscanCode regarding correctness, effectiveness, and practicability in a tabular form. As shown in Table 6, compared with existing tool TscanCode, the results demonstrate the superior performance of FLuaScan in terms of accuracy, effectiveness, and practicality.

**Table 6.** Comparison of FLuaScan and TscanCode in terms of correctness, effectiveness, and practicability.

	Correctness	Effectiveness	Practicability
TscanCode	0% TP Rate	0 actual vulnerabilities	time cost: average 1.51 s output: simple text
FLuaScan	42.50% TP Rate	21 vulnerabilities(9 unknown) moderate to high severity risk	time cost: average 25.16 s (acceptable) output: two visual output assist manual analysis

## 7. Discussion

This section aims to examine the constraints of our methodology and provide insights into potential directions for future development. During the operation of FLuaScan, it leverages existing software tools like Semgrep and Neo4j. However, the methodology employed in FLuaScan is entirely innovative, and no method described in existing literature can do the same. At present, FLuaScan exclusively focuses on supporting the Lua programming language. However, the fundamental theories discussed in this paper regarding vulnerability mining are applicable to various programming languages. Furthermore, the semgrep we employ demonstrates the capability to support multiple programming languages, facilitating seamless portability to other script languages. After conducting surveys, we discovered that many IoT manufacturers also employ languages such as ASP and PHP in their product development. Therefore, enhancing FLuaScan to support multiple scripting languages is our next research direction. Meanwhile, some manufacturers are developing products by packaging compiled Lua bytecode as firmware rather than directly using Lua source files. The absence of symbol information in this scenario poses numerous challenges for static analysis. The next step of our research is to find ways to improve FLuaScan to adapt to Lua programs without sources.

## 8. Conclusions

In conclusion, this paper provides a thorough study of the security concerns associated with Lua applications in IoT firmware. This paper introduces an innovative progressive taint analysis method that precisely identifies service handlers, performs segmented taint analysis, and utilizes graph search to identify vulnerability paths. Furthermore, we developed a prototype system, FLuaScan, and conducted experiments to demonstrate its superiority over current state-of-the-art tool TscanCode in terms of correctness, effectiveness, and practicality. FLuaScan discovered 21 vulnerabilities during the experiment, of which 7 are unknown. The work presented in this paper provides a solid foundation for future research on vulnerability detection in other language scripts programs of IoT firmware.

**Author Contributions:** Conceptualization, X.L. and Z.W.; Methodology, X.L., W.G. and Q.W.; Software, X.L. and Q.W.; Validation, W.G.; Writing—original draft, X.L.; Writing—review & editing, X.L. and W.G. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable

**Informed Consent Statement:** Not applicable

**Data Availability Statement:** Readers could send emails to us to access the related data.

**Conflicts of Interest:** The authors declare that they have no conflict of interest and no competing financial interest or personal relationships that could have appeared to influence the work reported in this paper.

## References

1. IoT Analytics. State of IoT 2023: Number of Connected IoT Devices Growing 16% to 16.7 Billion Globally, 2023. Available online: <https://iot-analytics.com/number-connected-iot-devices/> (accessed on 20 August 2023).
2. Antonakakis, M.; April, T.; Bailey, M.; Bernhard, M.; Bursztein, E.; Cochran, J.; Durumeric, Z.; Halderman, J.A.; Invernizzi, L.; Kallitsis, M.; et al. Understanding the mirai botnet. In Proceedings of the 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, 16–18 August 2017; Kirda, E., Ristenpart, T., Eds.; USENIX Association: Boston, MA, USA, 2017; pp. 1093–1110.
3. Microsoft. Microsoft Digital Defense Report 2022. In *Technical Report*; 2023. Available online: <https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE5bUvv> (accessed on 20 August 2023).
4. Deng, S.; Zhao, H.; Fang, W.; Yin, J.; Dustdar, S.; Zomaya, A.Y. Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence. *IEEE Internet Things J.* **2020**, *7*, 7457–7469. [\[CrossRef\]](#)
5. Khan, M.N.; Rahman, H.U.; Khan, M.Z.; Mehmood, G.; Sulaiman, A.; Shaikh, A.; Alqhatani, A. Energy-Efficient Dynamic and Adaptive State-Based Scheduling (EDASS) Scheme for Wireless Sensor Networks. *IEEE Sens. J.* **2022**, *22*, 12386–12403. [\[CrossRef\]](#)
6. Hameed, M.; Yang, F.; Ghafoor, M.I.; Jaskani, F.H.; Islam, U.; Fayaz, M.; Mehmood, G. IOTA-Based Mobile Crowd Sensing: Detection of Fake Sensing Using Logit-Boosted Machine Learning Algorithms. *Wirel. Commun. Mob. Comput.* **2022**, *2022*, e6274114. [\[CrossRef\]](#)
7. Mian, T.; Choudhary, A.; Fatima, S.; Panigrahi, B.K. Artificial intelligence of things based approach for anomaly detection in rotating machines. *Comput. Electr. Eng.* **2023**, *109*, 108760. [\[CrossRef\]](#)
8. Chang, Z.; Liu, S.; Xiong, X.; Cai, Z.; Tu, G. A Survey of Recent Advances in Edge-Computing-Powered Artificial Intelligence of Things. *IEEE Internet Things J.* **2021**, *8*, 13849–13875. [\[CrossRef\]](#)
9. Stergiou, C.; Psannis, K.E.; Gupta, B.B.; Ishibashi, Y. Security, privacy & efficiency of sustainable Cloud Computing for Big Data & IoT. *Sustain. Comput. Inform. Syst.* **2018**, *19*, 174–184. [\[CrossRef\]](#)
10. 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 (USENIX Security 21), Virtual Event, 11–13 August 2021; pp. 303–319.
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 IEEE Symposium on Security & Privacy (S&P), San Francisco, CA, USA, 18–21 May 2020; pp. 1544–1561. [\[CrossRef\]](#)
12. 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 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019; pp. 1099–1114.
13. Zhang, C.; Wang, Y.; Wang, L. Firmware Fuzzing: The State of the Art. In Proceedings of the 12th Asia-Pacific Symposium on Internetwork, Singapore, 1–3 November 2020; Internetwork’20; Association for Computing Machinery: New York, NY, USA, 2020; pp. 110–115. [\[CrossRef\]](#)
14. 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 CCS’21: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, 15–19 November 2021; CCS’21; Association for Computing Machinery: New York, NY, USA, 2021; Volume 1. [\[CrossRef\]](#)
15. Xie, W.; Chen, J.; Wang, Z.; Feng, C.; Wang, E.; Gao, Y.; Wang, B.; Lu, K. Game of Hide-and-Seek: Exposing Hidden Interfaces in Embedded Web Applications of IoT Devices. In Proceedings of the ACM Web Conference, New York, NY, USA, 25 April 2022; WWW ’22; Association for Computing Machinery: New York, NY, USA, 2022; pp. 524–532. [\[CrossRef\]](#)
16. Schwartz, E.J.; Avgerinos, T.; Brumley, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In Proceedings of the 2010 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 16–19 May 2010; SP ’10; pp. 317–331. [\[CrossRef\]](#)
17. Cheng, K.; Liu, T.; Guan, L.; Liu, P.; Li, H.; Zhu, H.; Sun, L. Finding Taint-Style Vulnerabilities in Linux-based Embedded Firmware with SSE-based Alias Analysis. *arXiv* **2022**, arXiv:2109.12209.
18. Celik, Z.B.; Babun, L.; Sikder, A.K.; Aksu, H.; Tan, G.; McDaniel, P.D.; Uluagac, A.S. Sensitive information tracking in commodity IoT. In Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018; Enck, W., Felt, A.P., Eds.; USENIX’18; USENIX Association: Berkeley, CA, USA, 2018; pp. 1687–1704.
19. 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 Proceedings-48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018; DSN’18; pp. 430–441. ISSN 2158-3927. [\[CrossRef\]](#)
20. Zhang, L.; Chen, J.; Diao, W.; Guo, S.; Weng, J.; Zhang, K. *CryptoREX: Large-Scale Analysis of Cryptographic Misuse in IoT Devices*; USENIX Association: Berkeley, CA, USA, 2019; pp. 151–164.
21. Costin, A.; Zarras, A.; Francillon, A. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, Xi’an China 30 May–3 June 2016; ASIA CCS ’16; Association for Computing Machinery: New York, NY, USA, 2016; pp. 437–448. [\[CrossRef\]](#)

22. Thomas, S.L.; Chothia, T.; Garcia, F.D. Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality. In Proceedings of the Computer Security-ESORICS 2017-22nd European Symposium on Research in Computer Security, Oslo, Norway, 11–15 September 2017; Proceedings, Part II; Foley, S.N., Gollmann, D., Sneekenes, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10493, pp. 513–531. [\[CrossRef\]](#)
23. Shoshitaishvili, Y.; Wang, R.; Hauser, C.; Kruegel, C.; Vigna, G. Fimalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, CA, USA, 8–11 February 2015. [\[CrossRef\]](#)
24. Chen, D.D.; Woo, M.; Brumley, D.; Egele, M. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In Proceedings of the 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, CA, USA, 21–24 February 2016. [\[CrossRef\]](#)
25. 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; ACSAC '20; Association for Computing Machinery: New York, NY, USA, 2020; pp. 733–745. [\[CrossRef\]](#)
26. Zaddach, J.; Bruno, L.; Francillon, A.; Balzarotti, D. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, CA, USA, 23–26 February 2014. [\[CrossRef\]](#)
27. 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 2018 Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2018. [\[CrossRef\]](#)
28. Wang, Z.; Zhang, Y.; Liu, Q. RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing. *KSII Trans. Internet Inf. Syst.* **2013**, *7*, 1989–2009. [\[CrossRef\]](#)
29. Zhang, Y.; Huo, W.; Jian, K.; Shi, J.; Lu, H.; Liu, L.; Wang, C.; Sun, D.; Zhang, C.; Liu, B. 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, {ACSAC} 2019, San Juan, PR, USA, 9–13 December 2019; ACSAC'19; Balenson, D., Ed.; ACM: New York, NY, USA, 2019; pp. 544–556. [\[CrossRef\]](#)
30. Redini, N.; Continella, A.; Das, D.; Pasquale, G.D.; Spahn, N.; Machiry, A.; Bianchi, A.; Kruegel, C.; Vigna, G.; De Pasquale, G.; et al. Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices. In Proceedings of the 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021; Volume 2021, pp. 484–500. ISSN 10816011. [\[CrossRef\]](#)
31. 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, London, UK, 15 November 2019; pp. 15–21.
32. Li, X.; Wei, Q.; Wu, Z.; Guo, W. A comprehensive survey of vulnerability detection method towards Linux-based IoT devices. In Proceedings of the 2023 2nd International Conference on Networks, Communications and Information Technology, Xi'an, China, 7–9 June 2023; CNCIT '23; Association for Computing Machinery: New York, NY, USA, 2023; pp. 35–41. [\[CrossRef\]](#)
33. Muench, M.; Stijohann, J.; Kargl, F.; Francillon, A.; Balzarotti, D. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In Proceedings of the Proceedings 2018 Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2018; NDSS'18. [\[CrossRef\]](#)
34. Fasano, A.; Ballo, T.; Muench, M.; Leek, T.; Bulekov, A.; Dolan-Gavitt, B.; Egele, M.; Francillon, A.; Lu, L.; Gregory, N. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, Hong Kong, China, 7–11 June 2021; pp. 687–701. [\[CrossRef\]](#)
35. Clements, A.A.; Gustafson, E.; Scharnowski, T.; Grosen, P.; Fritz, D.; Kruegel, C.; Vigna, G.; Bagchi, S.; Payer, M. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020, Boston, MA, USA, 12–14 August 2020; USENIX'20; Capkun, S., Roesner, F., Eds.; USENIX Association: Boston, MA, USA, 2020; pp. 1201–1218.
36. Scharnowski, T.; Bars, N.; Schloegel, M.; Gustafson, E.; Muench, M.; Vigna, G.; Kruegel, C.; Holz, T.; Abbasi, A. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; Security'22; USENIX Association: Boston, MA, USA, 2022.
37. Zhou, W.; Guan, L.; Liu, P.; Zhang, Y. Automatic firmware emulation through invalidity-guided knowledge inference. In Proceedings of the 30th USENIX Security Symposium, USENIX Security 2021, Vancouver, BC, Canada, 11–13 August 2021; USENIX'21; Bailey, M., Greenstadt, R., Eds.; USENIX Association: Boston, MA, USA, 2021; pp. 2007–2024.
38. Feng, B.; Mera, A.; Lu, L. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In Proceedings of the 29th USENIX Security Symposium, USENIX Security 2020, Boston, MA, USA, 12–14 August 2020; Capkun, S.; Roesner, F., Eds.; USENIX Association: Boston, MA, USA, 2020; pp. 1237–1254.
39. Gustafson, E.; Muench, M.; Spensky, C.; Redini, N.; Machiry, A.; Fratantonio, Y.; Balzarotti, D.; Francillon, A.; Choe, Y.R.; Kruegel, C.; et al. Toward the Analysis of Embedded Firmware through Automated Re-Hosting. In Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Beijing, China, 23–25 September 2019; pp. 135–150.



40. Qin, C.; Peng, J.; Liu, P.; Zheng, Y.; Cheng, K.; Zhang, W.; Sun, L. UCRF: Static analyzing firmware to generate under-constrained seed for fuzzing SOHO router. *Comput. Secur.* **2023**, *128*, 103157. [[CrossRef](#)]
41. Chen, S.; Lin, Z.; Zhang, Y. SelectiveTaint: Efficient data flow tracking with static binary rewriting. In Proceedings of the 30th USENIX Security Symposium, USENIX Security 2021, Vancouver, BC, Canada, 11–13 August 2021; Bailey, M., Greenstadt, R., Eds.; USENIX Association: Boston, MA, USA, 2021; pp. 1665–1682.
42. Luo, C.; Li, P.; Meng, W. TChecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in PHP applications. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, 7–11 November 2022; Yin, H.; Stavrou, A., Cremers, C., Shi, E., Eds.; ACM: New York, NY, USA, 2022; pp. 2175–2188.
43. Tencent; TscanCode: A Static Code Analyzer for C++, C#, Lua. Available online: <https://github.com/Tencent/TscanCode> (accessed on 20 August 2023).
44. Semgrep—Find Bugs and Enforce Code Standards. Available online: <https://semgrep.dev/> (accessed on 20 August 2023).
45. Medeiros, I.; Neves, N.F.; Correia, M. DEKANT: A static analysis tool that learns to detect web application vulnerabilities. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, 18–20 July 2016; Zeller, A., Roychoudhury, A., Eds.; ACM: New York, NY, USA, 2016; pp. 1–11.
46. Yao, M.; Cui, B.; Chen, C. Research on IoT Device Vulnerability Mining Technology Based on Static Preprocessing and Coloring Analysis. In Proceedings of the Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2020), Lodz, Poland, 1–3 July, 2020; Barolli, L., Poniszewska-Maranda, A., Park, H., Eds.; Advances in Intelligent Systems and Computing; Springer International Publishing: Cham, Switzerland, 2020; pp. 254–263. [[CrossRef](#)]
47. Bandara, V.; Rathnayake, T.; Weerasekera, N.; Elvitigala, C.; Thilakarathna, K.; Wijesekera, P.; De Zoysa, K.; Keppitiyagama, C. Large scale analysis on vulnerability remediation in open-source JavaScript projects. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, 15–19 November 2021; pp. 2447–2449. [[CrossRef](#)]
48. Universal Ctags. 2023. Available online: <https://github.com/universal-ctags/ctags> (accessed on 20 August 2023).

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.