FirmCross: Detecting Taint-Style Vulnerabilities in Modern C-Lua Hybrid Web Services of Linux-based Firmware

Runhao Liu*, Jiarun Dai†, Haoyu Xiao†, Yuan Zhang†, Yeqi Mou*, Lukai Xu*, Bo Yu*, Baosheng Wang*, Min Yang†
*National University of Defense Technology, †Fudan University
runhaoliu@nudt.edu.cn, {jrdai, hyxiao20, yuanxzhang}@fudan.edu.cn
{myq_2000, xulukai17, yubo0615, bswang}@nudt.edu.cn, m_yang@fudan.edu.cn

***Abstract*—Static taint analysis has become a fundamental technique to detect vulnerabilities implied in web services of Linux-based firmware. However, existing works commonly oversimplify the composition of firmware web services. Specifically, only C binaries (i.e., those extracted from the target firmware) are considered within the scope of vulnerability detection. In this work, we observe that modern firmware extensively combines Lua scripts/bytecode and C binaries to implement hybrid web services, and obviously, those C-binary-oriented vulnerability detection techniques can hardly achieve satisfactory performance. In light of this, we propose FirmCross, an automated taint-style vulnerability detector dedicated for C-Lua hybrid web services. Compared to existing detectors, FirmCross can automatically de-obfuscate the Lua bytecode in target firmware, additionally identify distinctive taint sources in Lua codespace, and systematically capture the C-Lua cross-language taint flow. In the evaluation, FirmCross detects 6.82X ~ 14.5X more vulnerabilities than SoTA approaches (i.e., MangoDFA and LuaTaint) in a dataset containing 73 firmware images from 11 vendors. Notably, FirmCross helps identify 610 0-day vulnerabilities among target firmware images. After reporting these vulnerabilities to vendors, till now, 31 vulnerability IDs have been assigned.**

I. Introduction

Internet of Things (IoT) devices have become deeply integrated into daily life, with projections indicating that the number of connected IoT devices will reach 40 billion by 2030 [1], and Linux-based devices are the mainstream within the IoT ecosystem [2], [3]. However, their inherent vulnerabilities (e.g., remote code execution[4], [5], denial-of-service[6], [7]) pose significant risks to production and living infrastructures (e.g., routers, IP cameras, VPN devices, etc.).

Given that these firmware vulnerabilities usually arise from web services [8], [9], [10], [11], [12], [13] (i.e., attackers can compromise the victim device by sending crafted network requests to the vulnerable web services), security researchers are actively developing static and dynamic techniques to detect vulnerabilities in IoT web services. Technically, dynamic approaches [14], [15], [16], [17] work by literally executing the target firmware services either through emulated environments or directly on physical devices. However, these methods face inherent challenges including labor-extensive firmware rehosting [3], [18] and limited code coverage [14], [19]. In comparison, static approaches [10], [11], [13], [2], [20], which do not require the establishment of a dynamic execution environment, have been embraced as a complementary solution to detect vulnerabilities of IoT web services. Specifically, these works mostly leverage the taint analysis technique to accomplish this task, which involves two key phases: ❶ **Source and Sink Identification** pinpoints user-controllable inputs (sources) and security-sensitive operations (sinks), and ❷ **Taint Propagation** verifies whether attacker-controlled data can flow from sources to sinks through execution paths, causing potentially exploitable vulnerabilities.

Although these static taint analysis techniques [8], [9], [2], [13], [12], [20] have helped identify various vulnerabilities in firmware web services, they usually over-simplify the composition of firmware web services, inevitably hurting the completeness of vulnerability detection. To be more specific, existing works merely consider C-binaries (i.e., those extracted from the target firmware) as the scope of vulnerability detection. However, as highlighted in recent studies [21], [22], [23], Lua has emerged as one of the most popular languages for implementing web services [24], [25], [26], [27], [28], due to its high performance and flexibility. As demonstrated in our large-scale empirical study (see §II) on 4012 commercial device firmware, 38% firmware samples typically adopt a hybrid C-Lua framework to implement modern web services. Among these firmware, Lua scripts/bytecode are extensively leveraged to implement a wide array of functions (e.g., URI dispatching and handling). In fact, these long-neglected Lua-involved attack surfaces [29], [30] have become one of the bottlenecks in firmware security.

Due to the fundamental differences between C binaries and Lua scripts/bytecode, as well as the complex cross-language interactions in C-Lua hybrid web services, existing static taint analysis approaches [8], [9], [2], [13], [12], [20] are difficult to directly apply to this scenario. Hence, in this work, we are highly motivated to re-design the taint-style

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231251
www.ndss-symposium.org

vulnerability detection, making it comply with C-Lua hybrid web services. However, it is technically a non-trivial task, due to the following challenges (detailed in §III-C):

- **C1: Prevalent and Diverse Lua Bytecode Obfuscation.** Based on our empirical study (see §II-C) on commercial firmware, 34% of Lua code exists in bytecode form, and 98% of Lua bytecode has been guarded with vendor-customized obfuscation, largely hindering the static taint analysis. Bytecode obfuscations include structure obfuscations (i.e., shuffled bytecode structure orders) and data obfuscations (i.e., transformed data field values). Here, the most recent work, LuaHunt [31], sheds light on Lua bytecode deobfuscation. However, LuaHunt necessitates heavy expert efforts, as it introduces reverse engineering to understand how the bytecode structure is shuffled. In addition, LuaHunt can only deal with one type of data obfuscation (opcode obfuscation) and can hardly extend to multiple coexisting obfuscations.
- **C2: Lua-specific Source Identification.** The methods for obtaining the taint sources (i.e., variables that represent specific input fields) are significantly different between C and Lua. Existing works [9], [12], [11] suggest that C-binaries of firmware web services commonly implement input access functions (e.g., `websGetVar(wp, "time")`) to parse the specific input field in an on-demand manner, which provides clear guidance for source identification (e.g., the return variable of input access functions). Generally, these on-demand input access functions are easily recognizable with well-summarized features (e.g., shared string constants [9], [12], or internal code patterns [11]). However, Lua scripts/bytecode usually parse all input fields in a uniform way before network request dispatching, and URI handlers (i.e., handler function for each specific request path) would fetch these parsed fields as input parameters through diverse callback mechanisms. In this instance, Lua code fetches inputs from variable fields instead of functions, requiring a new source identification approach. The most recent work LuaTaint [22] aggressively views all input parameters of URI handlers as taint sources. Obviously, this heuristic would induce numerous false positives. Even worse, LuaTaint's framework-specific design (i.e., can only handle services implemented upon LuCI [28] framework) would cause significant false negatives.
- **C3: C-Lua Data Flow Modeling.** Furthermore, the C-binary and the Lua script/bytecode usually exhibit frequent and complex communication, including the data flow through the application programming interfaces (API) and the inter-process communications (IPC). Although there have been studies [32], [33], [34], [35] on cross-language communication modeling (e.g., C-Python, C-Java), none of them has constructed the data flow model between C and Lua. Without a precise and complete communication model, we can hardly identify those potential C-Lua vulnerabilities.

Our Work. To address these challenges, we present FirmCross, an automatic static vulnerability detection approach for modern C-Lua hybrid web services of Linux-based firmware. First, FirmCross addresses multiple coexisting obfuscation mechanisms in Lua bytecode via an invariant-based structure deobfuscation technique and a static bytecode diffing-based data deobfuscation technique. The rationale behind this is that obfuscated bytecode still contains invariant features (e.g., length-header and fixed-termination structure, site-preserving characteristics, prototype structure consistency, see §II-C) enabling field order recovery, while differences between obfuscated and normal bytecode reveal data transformation rules. Second, based on the key observation that Lua URI handler registration follows deterministic patterns and source parameters of URI handlers exhibit distinct usage characteristics, FirmCross identifies Lua-specific sources by analyzing the registration behaviors of URI handlers and data usage patterns of handler parameters. Third, FirmCross leverages the deterministic patterns of C-Lua communication to construct the communication model covering API and IPC. It enables cross-language data flow tracking, thereby supporting cross-language vulnerability detection.

Evaluation Results. We have implemented a prototype of FirmCross and evaluated it on datasets from existing works [9], [8], [12] and real-world firmware, including 73 firmware images from 11 vendors. The evaluation results indicate that FirmCross can deobfuscate all custom interpreters extracted from real-world firmware, while LuaHunt [31] failed to handle anyone due to the insufficient support for obfuscation types and the lack of automation capabilities. In addition, compared to LuaTaint [22], we achieved 12.96 times as many identified Lua sources with a 4.67X improvement in accuracy. Benefiting from cross-language taint tracking capabilities and enhanced Lua source identification, FirmCross achieves superior performance in vulnerability detection by detecting more vulnerabilities while maintaining comparable precision compared to existing works. Specifically, FirmCross detected a total of 696 vulnerabilities with a 33.27% precision rate. The number of vulnerabilities is 6.82X that of MangoDFA [20] and 14.5X that of LuaTaint [22], while maintaining comparable precision. Notably, FirmCross discovered 610 0-day vulnerabilities, with 31 vulnerability IDs assigned until now. We have responsibly reported these 0-day vulnerabilities to relevant vendors and received official acknowledgments from Xiaomi, TP-Link, D-Link, and Tenda.

Contributions. We make the following major contributions:

- We propose three key techniques to boost the effectiveness of detecting taint-style vulnerabilities in C-Lua hybrid web services of Linux-based firmware, including Lua bytecode deobfuscation based on structure invariant features and bytecode diffing, Lua table-based source identification, and C-Lua cross-language communication modeling.
- We implemented the three techniques into a prototype FirmCross, and extensively evaluated it on real-world Linux-based firmware. Results show that FirmCross significantly outperforms SoTA approaches in Lua bytecode deobfuscation, Lua source identification, and vulnerability detection.
- We detected 696 vulnerabilities during experiments, among which 610 are 0-day vulnerabilities. After responsibly re-

porting these 0-day vulnerabilities, till now, we have received official acknowledgments from Xiaomi, TP-Link, D-Link, and Tenda, with 31 vulnerability IDs assigned.

We have released the code and dataset [1] to facilitate future research.

II. Empirical Study and Background

As mentioned in §I, existing works commonly over-simplify the composition of modern hybrid firmware web services, particularly omitting Lua-based components in their analysis. Hence, to comprehensively understand the security risks of C-Lua hybrid firmware web services, we performed a large-scale empirical study within IoT ecosystems. Accordingly, we also provide necessary background information about these hybrid web services for ease of understanding our solutions.

A. Study Design

Research Questions. In this study, we seek to answer the following questions:

- RQ1: How prevalent are Lua-C hybrid web services in modern firmware, and what service-delivery mechanisms have been employed? (see §II-B)
- RQ2: How prevalent is the Lua bytecode obfuscation in modern firmware, and what obfuscation mechanisms have been employed? (see §II-C)

Methodology. First, we leveraged Firmadyne Scraper [3] to collect a large amount of firmware images from six major vendors, including TP-Link, D-Link, Ruijie, Xiaomi, Netgear and Tenda, which are the main testing targets of previous works [8], [9], [13], [20]. Following existing threshold criterion [23], a given firmware image is considered to contain Lua components only when it includes more than 50 Lua files, either in source or bytecode format. After that, we manually examined firmware web services and confirmed whether each service was implemented using both C binaries and Lua scripts/bytecode. Finally, we analyzed whether the Lua bytecode has been obfuscated. To faithfully confirm the existence of bytecode obfuscation, we used the official Lua interpreter to load each under-test Lua bytecode file. If the bytecode cannot be successfully loaded, it must have been guarded with obfuscation techniques.

B. C-Lua Service Prevalence and Mechanisms

Prevalence of C-Lua Hybrid Services. As shown in Table I, we collected 4,012 firmware images in total and successfully extracted the corresponding filesystem for 2,461 of them, while the others failed due to firmware encryption or the nonexistence of the filesystem. Among these 2,461 analyzable firmware images, we identified 38% firmware instances (i.e., 937/2,461) containing C-Lua hybrid web services. Considering the wide existence of such hybrid services, we are highly motivated to detect implied vulnerabilities among them.

C-Lua Service-delivery Mechanisms. As shown in Figure 1, C-Lua hybrid firmware web services, similar to C-only ones, commonly rely on a C-binary network-facing server(①) to receive and preprocess network requests. After that, the network-facing server(①) would dispatch requests to three distinct targets, including internal server handler functions(①), external Common Gateway Interface (CGI) C-binaries(②) or external CGI Lua scripts/bytecode(③). Finally, specific service functional components(④) would be executed to handle each dispatched request.

[1] https://github.com/prankster009/FirmCross

TABLE I: Statistic Results of Empirical Study.

Vendor	# FW Total	#FW with Extractable FS	# FW with Hybrid Web Service	# FW with Lua Bytecode	#FW with Lua Obfuscation
XIAOMI	42	42	42	40	40
TP-Link	1957	1256	469	273	267
Ruijie	623	342	323	0	0
Netgear	501	365	27	9	9
Tenda	715	349	66	0	0
D-Link	174	107	10	0	0
ALL	4012	2461	937	322	316

FW: Firmware Image; **FS**: Filesystem;

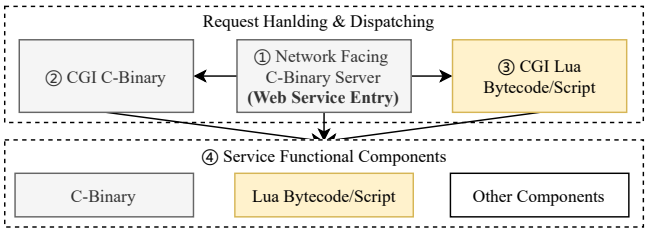


Fig. 1: General Architecture of C-Lua Firmware Web Service.

Long-neglected Lua-involved Attack Surfaces. Under the aforementioned C-Lua service architecture, obviously, there do exist vulnerability patterns that cannot be handled by existing firmware vulnerability detectors [8], [9], [2], [13], [12], [20]. To be more specific, existing works mainly focus on the attack vectors associated with C-binaries. According to Figure 1, the vulnerability triggering paths of these C-binary-only vulnerabilities generally include ①, ①→②, ①→④, and ①→②→④, while those involving Lua scripts/bytecode are totally omitted (e.g., ①→③ and ①→③→④). Hence, we aim to additionally detect these Lua-involved vulnerabilities to further enhance the completeness of firmware vulnerability detection.

C. Lua Obfuscation Prevalence and Mechanisms

Prevalence of Lua Bytecode and Lua Bytecode Obfuscation. The C-Lua hybrid service implementations exhibit two Lua code formats, including Lua source scripts and Lua bytecode files. Statistically, as listed in Table I, 66% (i.e., 615/937) firmware images with C-Lua web services only contain human-readable Lua source scripts. The remaining 34% (322/937) contain bytecode-form Lua files. Notably, following the methodology introduced in §II-A, we confirmed that 98% (316/322) firmware images implement vendor-specific bytecode obfuscation schemes. Hence, we should design reliable deobfuscation solutions to ensure the feasibility of static taint analysis for C-Lua hybrid firmware services.

Background Information about Lua Bytecode. As shown in Figure 2, a Lua bytecode file is generated by compiling the Lua source script through a Lua interpreter or compiler.

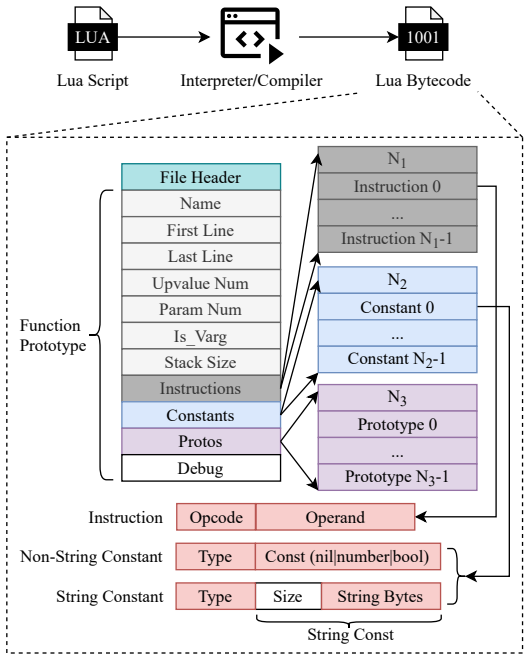


Fig. 2: Lua Bytecode Specification.

During runtime, the Lua interpreter also serves as the executor of Lua bytecode files. Hence, if the firmware image contains obfuscated Lua bytecode, it must also include a vendor-customized interpreter to execute such obfuscated bytecode. For ease of understanding the mechanisms of Lua bytecode obfuscation, we here also provide a brief introduction to the Lua bytecode specification. As illustrated in Figure 2, the Lua bytecode file consists of a 12-byte file header and multiple function prototypes which are organized in a prototype-chain form. To be more specific, the file header stores essential compiler metadata (e.g., endianness and version), and each function prototype contains metadata about a specific Lua function (e.g., number of parameters, constants, and instructions). Besides, Figure 2 also illustrates the structure of each single bytecode instruction and constant.

Understanding Real-world Lua Obfuscation Mechanisms. As mentioned above, once Lua bytecode is obfuscated, there always exists a corresponding vendor-customized interpreter packed in the firmware image, serving as the executor of the obfuscated bytecode. Besides, the Lua obfuscator commonly resides within the corresponding vendor-customized interpreter. Thus, we can easily obtain the obfuscated bytecode by using the vendor-customized interpreter to compile arbitrary source code inputs. These trials potentially provide valuable clues for investigating the landscape of real-world Lua obfuscation mechanisms. In total, we collected 316 interpreters from firmware images with Lua bytecode obfuscation. Besides, we leverage a representative Lua source script dataset to reproduce the vendor-customized obfuscation procedures. Specifically, we first manually wrote source scripts encompassing comprehensive usages of all instruction and constant types. Moreover, we also extracted Lua source scripts from all collected real-world firmware. In total, this dataset includes 204 Lua source code files with 164,305 instructions and 45,583 constants, which are sufficient to cover all usage scenarios. Here, we obtain the following key observations:

- *Observation-I: Coexisting structure obfuscation and data obfuscation.* (1) Structure obfuscation reorders the 10 fields within the function prototype while preserving each field's internal organization. For instance, the instructions array still maintains its original sequence order, and individual instruction instances retain their structures. (2) Data obfuscation alters specific values within fields, such as opcode, operand, constant type, and const (highlighted in red in Figure 2), containing three data transformation rules, including data shift that maps a number to another, XOR operation which is commonly used for strings, and storage format conversion which mainly refers to the conversion between integer format and floating-point number format.
- *Observation-II: Bytecode invariants under multiple obfuscations.* (1) Length-header and fixed-termination structure: The Lua specification mandates that all array structures within bytecode (e.g., instructions, constants, protos, string-bytes) must begin with a length header, and all instructions regions are terminated by a fixed `RETURN` instruction. (2) Size-preserving characteristics: Obfuscations do not modify length indicators for array structures (e.g., instructions, constants, protos, string-bytes). (3) Prototype Structure Consistency: All prototypes within the bytecode maintain identical structure field order.

III. Design Overview

In this section, we first introduce the target scope and threat model of our work. Next, we present a real-world vulnerability as a motivating example to illustrate how to detect cross-language vulnerabilities in modern web services and the associated technical problems. Finally, we explain in detail the challenges of detecting vulnerabilities in modern IoT web services crossing Lua and C, and then offer our key insights for addressing these challenges.

A. Target Scope and Threat Model

Similar to existing work [8], [9], [13], [20], [12], [2], we assume the adversary can send crafted network requests to a device's web service, aiming to compromise the victim device. By leveraging IPC and cross-language API, the adversary may exploit vulnerabilities in various web service components, including network-facing binary server ①, CGI binary ②, CGI Lua bytecode/script ③, and generic functional components ④, as shown in Figure 1. Therefore, this paper considers all these web service components as vulnerability detection targets. Since taint-style vulnerabilities are quite prevalent in web services [8], [9], [12], [10], [13], this paper focuses on detecting such vulnerabilities, including command injection (CWE-78) and buffer overflow vulnerabilities (CWE-119).

```
01. -- URL handler in Lua
02. function wol_wake(input_table, URI_path)
03.   log("request URI path" .. URI_path)
04.   local user_if = input_table.user_if -- get the user_if field
05.   local mac = input_table.mac -- get the mac field
06.   local cmd1 = string.format([[/usr/bin/ether-wake -b -i %s %s]],
07.                 user_if, mac)
08.   lib.forkExec(cmd1) -- command injection in Lua
09.   local ip = input_table.ip
10.   local cmd2 = "/usr/sbin/portscan "..lib._cmdformat(ip).." "
11.                 ..lib._cmdformat(mac).." portscan"
12.   lib.forkExec(cmd2) -- execute vulnerable binary portscan
13. end
14. -- vulnerable function in binary portscan
15. int sub_4027CC(char *mac)
16. {
17.   char file_name[512];
18.   memset(file_name, 0, sizeof(file_name));
19.   // buffer overflow
20.   sprintf(file_name, "/tmp/portscan_result/%s", mac);
21.   ...
22. }
```

Fig. 3: Motivating example including two 0-day vulnerabilities discovered by FirmCross. For readability, we restructure the decompiled code and rename the symbols.

B. Motivation Example

We showcase two 0-day vulnerabilities detected by FirmCross in Figure 3. Applying taint analysis to these vulnerabilities requires three key steps: Lua bytecode deobfuscation, Lua-specific source identification, and C-Lua cross-language communication modeling. **First**, the code of both vulnerabilities was compiled to bytecode and obfuscated, which could not be recognized by taint analysis tools. Therefore, the bytecode must be deobfuscated to recover the original bytecode before actual taint analysis. **Second**, taint analysis typically starts from an adversary-controllable variable, known as a taint source, which should be first identified. URI handlers (e.g., `wol_wake()`) in Lua scripts/bytecode commonly receive network input fields (e.g., `input_table` in Figure 3, parsed before request dispatching) through their parameters. Specifically, the input is parsed and stored in fields of `input_table`, whose type is a Lua table. Only by identifying these adversary-controllable variables/fields, the taint analysis can track their propagation for vulnerability detection. For example, the `user_if` and `mac` fields extracted from `input_table` at line 4-5 are propagated to `cmd1` string, then executed at line 8 without strict input sanitization; this allows attacker to send malicious inputs like `user_if=";cat /var/passwd;"` to remotely execute `"cat /var/passwd"` command. By tracking such a taint propagation flow, taint analysis detects this vulnerability as a command injection. **Third**, another vulnerability is triggered by the Lua taint source `input_table` as well, but the vulnerable code is located in the `portscan` binary. In this case, the taint analysis will fail to detect the vulnerability without modeling the cross-language communication of C-Lua. More specifically, the inputs stored to `ip` and `mac` are passed to `portscan` after input format checks introduced by `lib._cmdformat()` at line 10-11, so the taint analysis must further analyze the `portscan` binary to see whether these inputs will cause vulnerabilities. By modeling that the `ip` and `mac` are passed through arguments, the taint analysis can further find that the `mac` input is propagated to the `sprintf()` function call at line 20 without size checking, and detect the vulnerability as a buffer overflow.

C. Key Challenges and Insights

In light of the technical problems of the novel scenario mentioned above, we summarize the following three challenges and then offer our key insights to deal with them.

C1: How to deal with diverse obfuscations of the Lua bytecode automatically and efficiently. As detailed in §II-C, bytecode obfuscation consists of structure obfuscation (shuffling bytecode structure orders) and data obfuscation (transforming bytecode field values). For structure obfuscation, LuaHunt [31] first leverages binary reverse engineering to understand how the bytecode structure is shuffled (i.e., what is the shuffled structure field order). However, because it introduces reverse engineering, LuaHunt necessitates heavy expert efforts. This is because the obfuscated interpreter binary likely lacks symbols and involves syntactic differences introduced by customization, version divergence, and compiler optimizations, where existing automatic reverse engineering techniques (e.g., binary diffing) likely fall short [36], [37], [38].

Regarding data obfuscation, LuaHunt leverages a mutation-based semantic testing technique to identify data transformation rules for deobfuscation. Specifically, it selects opcode regions of bytecode compiled from pre-constructed Lua source gadgets as mutation targets; then it randomly mutates the opcode regions. After that, LuaHunt uses the obfuscated interpreter to execute the mutated bytecode to get outputs; if the output is equivalent to the expected source code output, then the mutation operation is used for deobfuscation. However, the mutation-based approach faces fundamental limitations when handling multiple co-existing obfuscations. Specifically, as the vendor-specific custom interpreter may simultaneously obfuscate multiple kinds of data elements (e.g., opcode and operand of instructions, type and const of constants), the mutated regions expand. The worse case occurs when the string constants are obfuscated, which further makes the exploration space for mutations grow exponentially. As a result, the mutation-based technique will fail to recover bytecode involving co-existing data obfuscation schemes.

Insight-I: Leverage invariant features and bytecode difference for Lua bytecode deobfuscation. The obfuscated interpreter, responsible for executing obfuscated bytecode, must reside in the firmware. By leveraging this interpreter to compile pre-constructed source code, we can generate controllable obfuscated bytecode.

To address the challenges of structure deobfuscation, we propose an invariant-based approach. As detailed in §II-C, bytecode exhibits invariant features even under multiple types of obfuscation. Based on these invariant features, we can identify the actual field orders in obfuscated bytecode structures.

Specifically, FirmCross extracts signatures of the field invariant features from pre-constructed Lua source gadgets, and then matches the signatures in obfuscated bytecode compiled from the same source gadgets to identify corresponding structure fields. By using this technique, FirmCross automatically infers structure field orders of obfuscated bytecode without reversing the obfuscated interpreter binary.

To address the challenges of data deobfuscation, we propose a static diffing-based approach. The other key observation is that the data transformation rules can be directly inferred by statically comparing obfuscated bytecode and normal bytecode, without relying on heavy mutation-based testing. Specifically, by comparing value differences in the same fields (e.g., opcode and operand of instructions, type and const of constants) between obfuscated bytecode and normal bytecode generated from the same source gadget, FirmCross can infer the data transformation rules for corresponding fields. This static bytecode diffing-based method can handle scenarios where multiple data obfuscations coexist, while avoiding the exponential exploration space problem inherent to mutation-based approaches.

C2: How to automatically identify Lua-specific intermediate sources. As described in Figure 3, the Lua-specific source like *inputs_table* is essential for source identification. Such a source, denoted as intermediate source (IS), is identified from the middle of network input handling instead of the start point of receiving network inputs, which greatly improves the effectiveness of source-to-sink taint propagation, and has been adopted by many previous work [9], [11], [12], [13], [20]. However, Lua scripts/bytecode and C binaries exhibit fundamentally distinct language-specific characteristics. C binaries predominantly utilize pointer-based memory access, necessitating comparatively lengthy code implementations and typically encapsulating frequently used operations into dedicated functions. In contrast, Lua scripts/bytecode leverage table structures to achieve complex data access with minimal code. Consequently, the IS access method is different between C binaries and Lua scripts/bytecode within hybrid firmware web services. The C binaries predominantly employ the dedicated function, namely the input access function (e.g., *websGetVar(wp, "time")*), to retrieve input fields from structured data (e.g., HTTP requests), which is easily recognizable (e.g., keywords and function signatures) and provides clear guidance (e.g., the return variable of input access functions). However, Lua scripts/bytecode usually parse all input fields in a uniform way before network request dispatching, and URI handlers would fetch these parsed fields as input parameters through diverse callback mechanisms. Consequently, existing source identification methods [9], [11], [12] based on the input access function cannot locate Lua-specific IS.

To address this issue, LuaTaint [22] leverages a framework-specific and file path-based approach to identify Lua-specific ISs. Specifically, it assumes that all function parameters under specific file paths within the LuCI [28] framework are sources. However, this approach is coarse-grained and not scalable. First, because not all parameters of URI handlers are sources and its file path-based heuristic solution is rough, it leads to many false positives. In addition, it can hardly locate sources within web services under other frameworks [24], [39], [27], which leads to significant false negatives.

Insight-II: Leverage registration mechanisms of URI handlers for Lua table-based source identification. As shown in Figure 3, Lua-specific sources can be identified at table-structured parameters of URI handling functions. Therefore, the core problem is to identify URI handlers. We observe that Lua URI handlers strictly follow a registration-callback mechanism, exhibiting deterministic registration patterns. Based on how callbacks (i.e., URI handlers) are registered, our key idea is to identify URI handlers based on two characteristic patterns of their registration procedures. The first pattern is that the registration function is called with characteristic arguments (e.g., a nested registry table or a function name string). The second pattern is that the registration function exhibits specific registering code logic (see §IV-B1). After identifying registration procedures, we identify the registered URI handlers and then distinguish attacker-controllable parameters as taint sources, which is based on the common field-based data accessing paradigms of table-based source variables.

C3: How to accurately model the communication between C-binaries and Lua scripts/bytecode? There are frequent (e.g., 167 times per firmware on average as shown in §V-E) and diverse communications (via API and IPC) between C-binaries and Lua scripts/bytecode, which inherently introduces cross-language vulnerabilities. Although many studies [32], [33], [34], [35] have focused on cross-language communication modeling (e.g., C-Java, C-Python), there is no work that constructs the communication modeling between C and Lua. Without a comprehensive and accurate C-Lua communication model, we cannot detect potential cross-language C-Lua vulnerabilities within the firmware hybrid web service.

Insight-III: Leverage deterministic patterns to construct C-Lua cross-language communication models. Even though C-Lua communications are diverse in the real world, we find that they all follow standardized paradigms defined by the operating system (e.g., system calls) or the C/Lua specifications. Therefore, we design deterministic patterns to identify and model such cross-language communications. Specifically, there are two kinds of communications, i.e., API-based and IPC-based communications. For API-based communications, the C specification indicates that C binaries invoke functions of Lua scripts/bytecode via the *lua_State* structure, and the Lua specification indicates that Lua scripts/bytecode invoke C binary functions through standardized loading mechanisms. For IPC-based communications, we mainly focus on the IPC triggered by command execution, tracing command-line parameters used by command execution functions according to the operating system specification (e.g., `execve()` in C, `os.execute()` in Lua). We present more details in §IV-C.

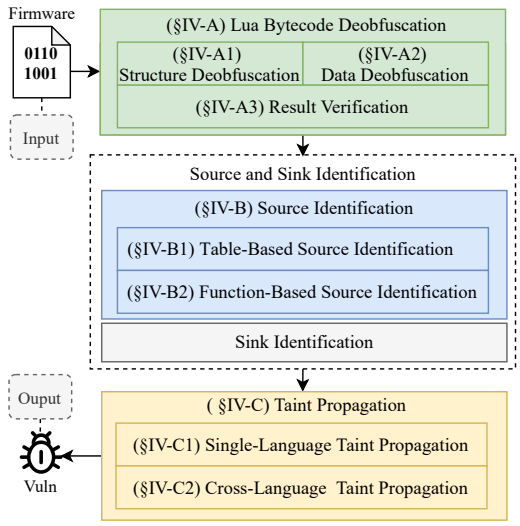


Fig. 4: FirmCross Workflow.

IV. Detailed Approach

Based on our key insights mentioned in §III-C, we design FirmCross, an automatic taint-style vulnerability detection approach for modern C-Lua hybrid web services of Linux-based firmware. As shown in Figure 4, FirmCross works in three steps: ❶ FirmCross takes firmware as input and checks whether Lua code within firmware is obfuscated. If obfuscation is detected, it conducts bytecode deobfuscation to convert obfuscated bytecode to normal bytecode (see §IV-A). ❷ After that, FirmCross identifies sources (see §IV-B) and sinks for C binaries and Lua scripts/bytecode within firmware. ❸ Subsequently, FirmCross constructs a C-Lua communication model and performs taint propagation within and across languages (see §IV-C). Finally, FirmCross generates a vulnerability report when it detects unsanitized tainted data flowing from a source to a sink.

A. Lua Bytecode Deobfuscation

Firmware containing obfuscated Lua bytecodes must have the vendor-specific custom interpreter to execute it. The custom interpreter allows us to generate controllable obfuscated bytecode by carefully constructing source code inputs. FirmCross first performs invariant-based structure deobfuscation (§IV-A1) and static diffing-based data deobfuscation (§IV-A2), then carefully verify whether the deobfuscation generates correct results. The detailed verification steps are presented in (§V-C).

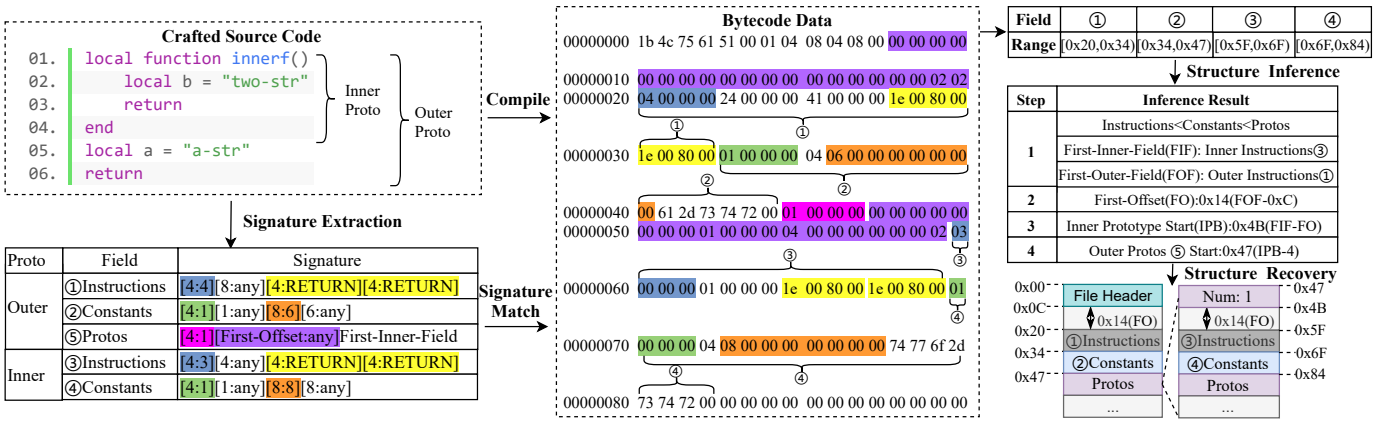
1) Invariant-Based Structure Deobfuscation: Structure deobfuscation needs to identify three structure fields, namely Instructions, Constants, and Protos fields. Instructions and Constants fields constitute the core data regions for bytecode execution logic; the Protos field contains instructions and constants of sub-functions, so is also essential for recovering the entire bytecode structure. Even though other structure fields can also be deobfuscated, they either serve debugging purposes (e.g., source line number), or can be recovered by analyzing the deobfuscated fields as described in §A.

We deobfuscate bytecode structures through 4 steps: First, we design signatures based on field invariants described in §II-C. Second, we construct a source gadget and extract its invariant-based signatures. Third, FirmCross matches the invariant-based signatures in the source gadget's obfuscated bytecode compiled with a given obfuscation interpreter. Finally, FirmCross extracts offsets between structure fields, which represent the bytecode structure obfuscation scheme, and then leverages this scheme to deobfuscate any bytecode compiled with the given obfuscation interpreter.

Invariant-based Field Signatures. Based on the bytecode invariants detailed by Observation-II in §II-C, we further design signatures to infer variant structure field characteristics introduced by structure obfuscation, i.e., the offsets between structure fields. For instructions and constants fields, the invariants suggest strong data characteristics, so we design signatures to directly determine their offsets in a structure. For the instructions field, the invariant suggests it should start with a 4-byte instruction number counter and terminate with 2 RETURN instructions when a return statement is used. For the constants field, the invariant suggests it should also start with a 4-byte constant number counter; besides, by definition, the constants must hold all constants from the specific source code scope. By using these data features, the signatures can determine the starting and ending offset of the instructions and constants field.

For the protos field, it represent functions defined in an inner scope of an outer code range (e.g., sub-functions defined within a function), and can be recursively nested. Therefore, inferring their offsets is more complex. Fortunately, there are multiple invariants suggesting that we can infer the offsets of protos fields based on the offsets of other fields. The most important invariant is that all protos fields are of the same structure type, which contain fields of sub-function bytecode, including instructions, constants, and inner protos. This invariant suggests we can determine the order of instructions, constants, and protos fields by comparing the offsets of inner and outer instructions/constants fields. Based on such order, we can further identify the first field of outer and inner protos, and thus calculate the offset between them. Since this offset is a sum calculated of the size of fields within outer protos fields and offsets between these fields, the offsets between protos and instructions/constants can be calculated using the known size (e.g., sizes of instructions and constants fields) and offset values (e.g., offsets between instructions and constants). Therefore, we leverage the offset between outer and inner protos as the signature to infer offsets between protos and other fields. We showcase the above signatures in Figure 5.

Field Signatures Extraction. We carefully constructed a source code gadget to extract distinct field signatures as shown in Figure 5. We use the outer protos field as an example. The instructions comprise four instructions ending with two identical `RETURN` instructions, hence its signature is [4:4][8:`any`][4:`RETURN`][4:`RETURN`] where `any` represents any value. Notably, the instructions signatures do



Proto	Field	Signature
Outer	①Instructions	[4:4][8:any][4:RETURN][4:RETURN]
	②Constants	[4:1][1:any][8:6][6:any]
	⑤Protos	[4:1][First-Offset:any]First-Inner-Field
Inner	③Instructions	[4:3][4:any][4:RETURN][4:RETURN]
	④Constants	[4:1][1:any][8:8][8:any]

[$\texttt{Size}_1$:$\texttt{Value}_1$][$\texttt{Size}_2$:$\texttt{Value}_2$] denotes that $\texttt{Value}_1$ occupies $\texttt{Size}_1$ bytes while $\texttt{Value}_2$ immediately follows, occupying $\texttt{Size}_2$ bytes.

Fig. 5: A Running Example of Structure Deobfuscation.

not leverage the specific value of the RETURN instruction but identify the two repeated 4-byte patterns, ensuring they work even when instructions are obfuscated. The constants field has only one string-type constant, and its signature is [4:1][1:any][8:6][6:any] (consistent with the string-type constant structure, see Figure 2). The constants signatures do not leverage the string constant content, thus remaining effective even under string content obfuscation.

Field Signature Matching. After obtaining the signatures of instructions and constants, we compile the source code gadget into bytecode and match signatures in the bytecode. As shown in Figure 5, we locate the outer prototype instructions region as [*0x20,0x34*) and its constants region as [*0x34,0x47*), while the inner prototype instructions region is [*0x5F,0x6F*) and its constants region [*0x6F,0x84*).

Structure Deobfuscation. We first infer the position of the outer protos field, and then recover the bytecode structure according to instructions, constants, and protos field of the outer prototype. We use the example in Figure 5 to explain how to infer the outer protos position following a 4-step procedure: ❶ Because field ① is prior to the field ②, the instructions field is located before constants field; because fields ①+② are prior to fields ③+④ while ③+④ are in the outer protos field, so the order of three fields is instructions, constants, protos. ❷ As the first outer field (FOF) is ① which starts from 0x20 and the outer prototype starts from 0xC, so the first field offset (FO) is 0x14 (0x20-0xC). ❸ As the first inner field (FIF) is ③ which starts from 0x5F, so the inner prototype starts from 0x4B (0x5F-FO). ❹ Since there is a number header with a length of 4 (*size_t*) between the outer protos field ⑤ start and the inner prototype start, so the outer protos field ⑤ starts from 0x47 (0x4B-4). Since the lengths of these three fields vary across different bytecode instances, we use relative offsets rather than absolute positions for localization. Thus, the final recovered bytecode structure is as follows: The first field is instructions, with an offset of 0x14, followed by constants field with an offset of 0 from the previous field, followed by protos, with an offset of 0 from the constants field.

2) Static Diffing-Based Data Deobfuscation:
Our static diffing-based obfuscation approach compiles source code with standard and obfuscated interpreters to generate bytecode for diffing-based analysis. So we first prepare a source code dataset to representatively cover all data usage scenarios. Specifically, we extensively collect 204 scripts following the method introduced in §II-C. After compiling the source code into standard and obfuscated bytecode, FirmCross extracts all constants and the instructions fields from the two bytecode files for diffing-based analysis. Subsequently, FirmCross iterates the constants and instructions fields and compares each constant and instruction element. Specifically, the compared regions are opcodes and operands of instruction elements, as well as the type and data regions of constant elements. If the comparison finds inconsistency, FirmCross identifies that the interpreter obfuscates the bytecode data.

When data obfuscation is identified, FirmCross further analyzes how the data is obfuscated. Based on our observation, the existing data transformations can be classified into three categories: (1) data shift that maps a number to another; (2) XOR operation which is commonly used for strings; and (3) storage format conversion, which mainly refers to the conversion between integer format and floating-point number format. If any two elements from two bytecode files with the same position are inconsistent, FirmCross automatically conducts inspections according to the above three categories of data transformations and records the specific transformation pattern that is applied.

```lua
-- register by nested table
dispatch_tbl.port_speed_supported = {
  [".args"] = "supported",
  [".super"] = {
    cb = port_speed_supported, -- the URI handler
  }
}

function dispatch(http_form)
  local ctl    = require "luci.model.controller"
  return ctl.dispatch(dispatch_tbl, http_form)
end

-- register by string, the name of URI handler is "wol_wake"
register_keyword_action("wol", "wake", "wol_wake")
```

Fig. 6: Two types of URI Handler Registration in Lua.

B. Taint Source Identification

1) Table-Based Source Identification:

FirmCross first leverages registration mechanisms of URI handlers to identify candidate Lua sources. To be specific, FirmCross utilizes the registration parameter patterns and registration behaviors to accurately locate the URI handlers and identifies all parameters of these handlers as candidate sources. Subsequently, FirmCross further uses data storage characteristics to distinguish attacker-controllable parameters as table-based ISs.

Handler Registration-Based Source Candidate Identification. FirmCross initially identifies candidate registration functions based on registration parameter patterns. The core idea here is that the registration function is invoked with parameters describing the URI handlers to be registered, and such parameters are easily identifiable. Specifically, we observe two types of these parameters. The first type directly conveys the function name (e.g., `wol_wake` at line 15 in Figure 6). The second type of parameter describes a URI handler using a registry table (e.g., `dispatch_tbl at line 2 in Figure 6)`, where the target URI handler function (e.g., `port_speed_suppoted` at line 5 in Figure 6) is stored in a table field. Therefore, both types of parameters refer to URI handlers. By analyzing the characteristics of function parameters, we first identify candidate registration functions and record the registered candidate URL handlers for further analysis (line 2 of Algorithm 1). Specifically, FirmCross analyzes the parameters of all functions: if a parameter is a string corresponding to a function name, or a table containing function objects internally, the function is considered a candidate registration function.

Simply leveraging the parameter patterns to identify registration functions easily leads to false positives as debug functions usually print function names. To filter out such false positives, FirmCross further analyzes the internal logic to confirm that the previously identified candidate registration functions exhibit registration behaviors. Specifically, FirmCross identifies two kinds of registration behaviors (line 3 in Algorithm 1). First, registry table-based registration functions (e.g., `ctl.dispatch` at line 11 in Figure 6) contain internal logic to execute the handler object stored within the table. FirmCross identifies such registration logic by detecting whether the stored handler is used by `CALL`-like instructions within the function. Second, name binding-based registration functions (e.g., `register_keyword_action` at line 15 in Figure 6) contain internal logic to store these function-name strings in a Lua table with a constant string/number, which will then be used to invoke corresponding URI handlers once the request path matches the registered path. FirmCross identifies such registration logic by detecting whether the function-name strings are used by `SETABLE`-like instructions with a constant key. If confirmed, FirmCross records the actual registration function. At this stage, FirmCross has located actual registration functions, *RealR*, and identifies all parameters of URI handlers registered by actual registration functions as candidate sources.

Algorithm 1 Lua Table-based Source Identification

Input: F - All Lua Functions
Output: $Source$ - The identified source list
1: $CandidateR, RealR, HandlerAll, S \leftarrow \emptyset$
2: $CandidateR, HandlerAll \leftarrow RegParamCheck(F)$
3: $RealR \leftarrow RegLogicCheck(CandidateR)$
4: **for** $RegF \in RealR$ **do**
5: $\quad HandlerList \leftarrow getHandler(HandlerAll, RegF)$
6: $\quad NewSource \leftarrow HandlerParamCheck(HandlerList)$
7: $\quad Source \cup IdentifiedSource$
8: **end for**
9: **return** $Source$

Parameter Usage-Based Source Filtering. After collecting candidate sources, FirmCross further distinguishes controllable sources from them. Based on our observation, source parameters commonly exist in the form of Lua table structure, due to the hierarchical nature of multi-field inputs (e.g., `input_table` at line 2 in Figure 3), while uncontrollable parameters are stored as primitive types, such as number and string (e.g., `URI_path` at line 2 in Figure 3). In addition, URI handlers registered by the same registration function at the same registration parameter indices exhibit consistent source parsing and propagation patterns, sharing identical parameter structures (i.e., which parameter serves as the source). Given that, if any parameter within a URI handler is found to exhibit nested access, all parameters at the corresponding indices of such URI handlers are marked as table-based sources in the form of Lua table structure. Specifically, FirmCross extracts all URI handlers, *HandlerList*, registered by the same registration function, *RegF* (line 5 in Algorithm 1). After that, FirmCross iterates through each parameter of every URI handler to detect nested access patterns (line 6 in Algorithm 1). If detected, FirmCross classifies all parameters at the same index position across such handlers as taint sources (line 6-7 in Algorithm 1).

2) Function-Based Taint Source Identification:

For C binaries, FirmCross strictly follows the mangoDFA [20] approach to identify sources, including both direct sources (e.g., `read()`) and intermediate sources (e.g., `getenv()`). For Lua scripts/bytecode, we have found that in the LuCI [28] framework, while sources are generally retrieved via table-based URL handler parameters, some sources are retrieved using dedicated functions like `luci.http.formvalue()`. To make source identification more comprehensive, we have also incorporated these LuCI-specific functions into our method, which is consistent with LuaTaint [22].

C. Taint Propagation

FirmCross incorporates both single-language and cross-language taint propagation. For single-language taint propagation, FirmCross leverages existing techniques [20] to analyze C code, and implements a bytecode-based taint propagation engine to analyze Lua code. During single-language taint propagation, FirmCross recognizes specific input sanitization functions by their names (e.g., `atoi()` in C code and `tonumber()` in Lua code), and untaints the flow that calls these functions to filter out false positives. This untainting approach is in line with prior works [8], [9], [20].

For cross-language taint propagation, FirmCross first identifies cross-language data propagation points (DPPs) and then constructs the communication model for each identified point.

Data Propagation Point Identification (DPPI). The data propagation across C binaries and Lua scripts/bytecode can be divided into two types, including API and IPC.

- *API-based DPPI.* FirmCross identifies Lua-to-C and C-to-Lua API-based DPPs accordingly. As Lua-to-C function calls can be dynamically registered, we must first identify C functions registered by Lua interpreter. When loading a C library for registration, the Lua interpreter will call the library's initialization function, which calls `luaL_register()` function to register C functions. Each `luaL_register()` function call registers a function table (an array containing function names and their addresses), which is passed through its arguments. Based on the calling convention, FirmCross can easily identify the registered functions and their names. Then, if any Lua function calls the registered function by its name, FirmCross identifies the call as a DPP. For C-to-Lua API-based DPPs, they propagate data by calling a set of specific APIs in order. Specifically, the C binary loads the Lua script/bytecode by calling `luaL_loadfile()`, then it obtains a reference to a Lua function by calling `lua_getglobal()`, and after that, it passes arguments with dedicated functions like `lua_pushnumber()` function and finally calls `lua_pcall()` to invoke the Lua function. Therefore, we directly identify such API call sequences as C-to-Lua DPPs.
- *IPC-based DPPI.* FirmCross focuses on command execution-triggered IPC. Specifically, FirmCross monitors whether the execution target in command strings (within command execution functions, e.g., `os.execute()` in Lua, `system()` in C) resolves to a binary executable and whether command-line parameters is not a constant. If both conditions are satisfied, FirmCross records the propagation point as *(target, cmd-param-index)*, where *target* is the target binary/script/bytecode and cmd-param-index represents the index of non-constant command-line parameter.

C-Lua Data Propagation Modeling. FirmCross constructs propagation models based on identified propagation points. For API-based propagation modeling, when FirmCross confirms that parameters passed to cross-language target functions are tainted, it marks these parameters as tainted and initiates taint propagation within the target environment (program/script/bytecode). For IPC-based propagation modeling, when FirmCross detects that command-line parameters passed to cross-language targets are tainted within command execution strings, it marks these command-line parameters as tainted and activates taint propagation in the target environment.

V. Evaluation

A. Experiment Setup

Implementation. We have implemented a prototype of FirmCross using more than 14,500 lines of Python code (i.e., about 4,500 LoC for Lua bytecode deobfuscation, and about 10,000 LoC for source/sink identification and taint tracking). Considering the co-existence of Lua source scripts and bytecode in given firmware images, we chose to first convert source scripts into bytecode using an official Lua interpreter, and then uniformly conduct taint analysis at the bytecode level. Notably, the prototype of FirmCross mainly put an emphasis on Lua code taint analysis and C-Lua cross-language taint analysis. As for the taint analysis among C binaries, which is not the key contribution of this work, we chose to directly integrate the SoTA tool mangoDFA [20] into FirmCross.

Evaluation Questions. Our evaluation is organized by answering the following research questions:

- **RQ1.** How effective is FirmCross in detecting taint-style vulnerabilities in real-world firmware? (see §V-B)
- **RQ2.** How effective is FirmCross in Lua bytecode deobfuscation? (see §V-C)
- **RQ3.** How effective is FirmCross in identifying taint sources among Lua code? (see §V-D)
- **RQ4.** How effective is FirmCross in C-Lua cross-language taint propagation? (see §V-E)

Dataset. To convincingly demonstrate the effectiveness of FirmCross, we tried our best to construct a representative dataset of firmware images as evaluation targets. Specifically, we considered those benchmark firmware images used by existing works (i.e., Karonte [8], SaTC [9], EmTaint [10], and LARA [12]), and finally collected 13 firmware images which contain C-Lua hybrid firmware web services. To further improve the diversity of target firmware images, we randomly selected 60 latest-version firmware images (i.e., 10 firmware images from each of the 6 vendors) from the dataset of our empirical study (see §II). As such, the final dataset contains 73 firmware images across 11 vendors.

Environment. All experiments were run on a host machine with a 56-core Intel(R) Xeon(R) CPU and 128GB of RAM running the Ubuntu 18.04 operating system.

B. Effectiveness of Vulnerability Detection (RQ1)

Experiment Overview. In this experiment, we considered MangoDFA [20] and LuaTaint [22] as baseline tools. MangoDFA [20] is the SoTA vulnerability detector for C-binary firmware, which has experimentally shown better performance than other C-binary-oriented detectors (e.g., Karonte [8] and SaTC [9]). LuaTaint [22] is the SoTA vulnerability detector for firmware web services implemented in Lua, which outperforms other Lua-oriented detectors (e.g., LuaCheck [40], TscanCode [41] and Semgrep [23], [42]). The comparative experiments were conducted on the 73 firmware images collected in our firmware dataset (see §V-A). We manually developed PoCs for all reported alerts to dynamically validate the existence of each identified vulnerability, either on physical devices or in emulated environments.

Overall Results. According to Table II, the evaluation results demonstrate that FirmCross detected 696 vulnerabilities in total with 96.67% vulnerability coverage (VC) ratio and 33.27%

TABLE II: Vulnerability Detection Results of FirmCross, MangoDFA and LuaTaint.

Vendor	#Firmware	Test Targets		FirmCross						MangoDFA						LuaTaint					
		#Bin	#Lua	#Alert	TP	FP	Prec.	VC	AVG Time	#Alert	TP	FP	Prec.	VC	AVG Time	#Alert	TP	FP	Prec.	VC	AVG Time
TPLink	11	12836	5382	229	56	173	24.45%	81.16%	184.58	23	8	15	34.78%	11.59%	163.86	39	13	26	33.33%	18.84%	101.84
XIAOMI	10	10076	6530	1016	145	871	14.27%	100.00%	15.13	0	0	0	-	0.00%	7.94	0	0	0	-	0.00%	0.00
NetGear	15	21156	2273	238	117	121	49.16%	92.86%	174.13	214	93	121	43.46%	73.81%	173.29	78	33	45	42.31%	26.19%	19.18
Tenda	10	6184	1102	366	250	116	68.31%	100.00%	55.90	12	0	12	0.00%	0.00%	55.10	0	0	0	-	0.00%	48.48
Dlink	10	6180	1025	156	86	70	55.13%	100.00%	50.33	2	0	2	0.00%	0.00%	49.93	0	0	0	-	0.00%	24.72
Ruijie	10	9940	3590	1	0	1	0.00%	-	84.98	0	0	0	-	-	81.34	18	0	18	0.00%	-	1.20
TOTOLink	2	2096	97	77	41	36	53.25%	100.00%	22.82	1	0	1	0.00%	0.00%	22.15	0	0	0	-	0.00%	120.00
Motorola	2	1260	112	8	0	8	0.00%	-	190.34	8	0	8	0.00%	-	190.12	0	0	0	-	-	0.45
H3C	1	496	30	0	0	0	-	-	44.60	0	0	0	-	-	44.37	0	0	0	-	-	0.56
Mercury	1	280	3	0	0	0	-	-	46.99	0	0	0	-	-	46.97	0	0	0	-	-	0.01
TRENDnet	1	936	124	1	1	0	100.00%	33.33%	97.28	1	1	0	100.00%	33.33%	96.73	2	2	0	100.00%	66.67%	0.38
SUM	73	53472	20268	2092	696	1396	33.27%	96.67%	100.29	261	102	159	39.08%	14.17%	95.31	137	48	89	35.04%	6.67%	32.73

#Alert: the number of reported vulnerabilities; **TP/FP**: the number of correctly/incorrectly identified vulnerabilities; **Prec.**: $\frac{\text{TP}}{\text{TP+FP}}$; **VC**: vulnerability coverage, calculated as $\frac{\text{TP}}{\text{TP of all tools}}$; **ACG Time**: average time cost of analyzing per firmware sample (minutes);

precision rate, achieving 6.82X and 14.5X improvements in VC ratio over MangoDFA [20] and LuaTaint [22] respectively, while maintaining comparable precision. To sum up, the vulnerability detection capability of FirmCross significantly outperforms existing approaches. In the following, we further carried out a detailed breakdown analysis of these results.

Breakdown Analysis (VC). Regarding vulnerabilities detected by FirmCross but overlooked by comparative methods, FirmCross uniquely identified C-Lua cross-language vulnerabilities, whereas MangoDFA [20] and LuaTaint [22] failed to detect vulnerabilities in languages outside their focus domains. Additionally, via automatic Lua bytecode deobfuscation (see §V-C) and advanced Lua source identification (see §V-D), FirmCross uncovered numerous vulnerabilities that LuaTaint could not detect. Furthermore, by modeling cross-language data flow between Lua scripts/bytecode and C binaries, FirmCross traces data propagation from Lua scripts/bytecode to C binaries, enabling detection of previously undetectable cross-language vulnerabilities triggered in C binaries that MangoDFA missed (see §V-E). Concerning vulnerabilities detected by comparative methods but missed by FirmCross, since FirmCross is built upon MangoDFA, FirmCross inherently detected all vulnerabilities discovered by MangoDFA. However, LuaTaint identified 24 vulnerabilities overlooked by FirmCross. Manual inspection revealed that this is attributed to incomplete modeling of the `MOD` instruction in our taint analysis framework.

Breakdown Analysis (Precision). We manually verified all the reported alerts as follows: (1) First, we checked whether the identified sources could truly be reached via network inputs. (2) We examined whether there was no sanitization along the taint propagation path. (3) If all the above checks passed, the vulnerability was considered an TP; otherwise, it was regarded as a FP. The FPs in FirmCross, LuaTaint, and MangoDFA can all be attributed to the following reasons: (1) There are misidentified sources that are actually not reached via network inputs. (2) During taint propagation, they assume that the return value of a function is tainted when its parameter is tainted, which may trigger false alarms. Existing works [12], [13], [20] also encounter this problem. (3) Developers conduct various checks during program/script/bytecode execution, such as format checks for IP and MAC inputs and content checks for malicious delimiters. Nevertheless, it is challenging to detect these checks through a unified method. In addition, apart from the above reasons, MangoDFA incorrectly flagged buffer overflow vulnerabilities due to unmodeled input-length-to-buffer-capacity relationships.

Identified 0-day Vulnerabilities. Among 696 TP vulnerabilities discovered by FirmCross, 610 of them were previously unknown 0-day vulnerabilities, including 557 command injection vulnerabilities and 53 buffer overflow vulnerabilities. We have responsibly reported all discovered vulnerabilities to relevant vendors. Until now, 31 vulnerability IDs have been assigned, which are detailed in §B. Besides, we also received official acknowledgments from Xiaomi and TP-Link, for uncovering the long-neglected Lua-involved attack surfaces in their products.

TABLE III: Lua Bytecode Deobfuscation Results for Real-World Vendor-Customized Interpreters.

Vendor	#Interp	FirmCross		LuaHunt	
		Pass Rate	AVG time	Pass Rate	AVG time
NetGear	9	100%	19.52	0	-
XIAOMI	40	100%	13.51	0	-
TP-Link	267	100%	21.07	0	-
SUM	316	100%	18.03	0	-

#Interp: the number of under-test interpreters; **Pass Rate**: what percentage of interpreters can pass the deobfuscation verification; **AVG Time**: the average time cost of deobfuscating per interpreter (minutes);

TABLE IV: Deobfuscation Capability Comparison between FirmCross and LuaHunt.

Tool	Ob-Structure	Ob-Data				
		❶	❷	❸	❹	❺
FirmCross	✓	✓	✓	✓	✓	✓
LuaHunt	✓	✗	✗	✗	✓	✗

Ob-Structure: structure obfuscation; **Ob-Data**: data obfuscation; ❶: signed int type addition; ❷: constant type modification; ❸: string xor; ❹: opcode obfuscation; ❺: operand obfuscation;

C. Effectiveness of Lua Bytecode Deobfuscation (RQ2)

Experiment Overview. To comprehensively evaluate the deobfuscation capability of FirmCross, we here did not use the firmware dataset introduced in §V-A, which contains only 16 vendor-customized interpreters. Alternatively, we extracted 316 distinct vendor-customized interpreters from firmware images collected in the empirical study §II, as the evaluation targets of this experiment. Here, we selected LuaHunt [31]

TABLE V: Obfuscation Mechanisms of Real-World Vendor-Customized Interpreters.

Vendor	#Interp	Ob-Structure	Ob-Data				
			❶	❷	❸	❹	❺
NetGear	9	0	9	0	0	0	0
XIAOMI	40	40	40	40	40	40	40
TP-Link	267	267	267	0	0	111	0
SUM	316	307	316	40	40	151	40

see table footnotes of Table IV;

(i.e., the most recent work in Lua bytecode deobfuscation) as the baseline. To automatically verify the correctness of bytecode deobfuscation, we followed a 3-step procedure: ❶ we first generate normal bytecode and obfuscated bytecode by compiling the same source script using the official interpreter and the vendor-customized interpreter respectively. ❷ Then, we deobfuscate the obfuscated bytecode to obtain deobfuscated bytecode. ❸ Finally, by directly comparing all field regions of the deobfuscated bytecode with those of the normal bytecode, we consider the deobfuscation result correct only when all regions are exactly the same. In addition, we select the Luahunt benchmark, which consists of 9 source scripts, as the basis for compiling bytecode. Besides, We provide the bytecode structure recovered by FirmCross to LuaHunt, eliminating the need for manual reverse-engineering in its testing process.

Overall Results. As shown in Table III, FirmCross passed the deobfuscation result verification on all 316 vendor-specific custom interpreters, achieving a 100% pass rate with an average time of 18.03s per interpreter. It means that the current version of FirmCross can correctly handle all deobfuscated Lua bytecode among these firmware images. In comparison, the SoTA approach LuaHunt [31], however, did not pass the deobfuscation verification on any custom interpreter, because the obfuscation types employed by vendor-specific interpreters exceed its supported scope. Besides, since LuaHunt requires manual efforts to reverse-engineer the interpreter to understand the bytecode structure, we did not report its time consumption here.

Table IV demonstrates the deobfuscation capability comparison between FirmCross and LuaHunt. Since FirmCross deobfuscates bytecode based on invariant field features and static bytecode diffing, it can handle all types of existing obfuscation techniques, including structure obfuscation and all types of data obfuscation techniques. However, LuaHunt [31] can only address structure obfuscation and a single data obfuscation technique (opcode manipulation) based on mutation-based testing.

In addition, we have statistically analyzed the obfuscation behaviors among all interpreters. As shown in Table V, among the 316 obfuscated interpreters, 307 implemented structural obfuscation, while all 316 exhibited data obfuscation: 316 added new signed integer constant types (notably with varying type identifiers across these interpreters), 40 modified constant type values, 40 xored constant string contents, 151 obfuscated opcodes, and 40 obfuscated operands. The Xiaomi obfuscated interpreters demonstrated the most comprehensive techniques, incorporating all identified obfuscation techniques.

Benefits of More Reliable Lua Bytecode Deobfuscation. Converting obfuscated bytecode into normal bytecode enables us to conduct vulnerability mining on it. In the dataset introduced in §V-A, Lua bytecode in 16 firmware is obfuscated. Benefiting from our deobfuscation technique, we discovered 154 vulnerabilities in them, including 148 Lua bytecode vulnerabilities and 6 C binary vulnerabilities where the data flow originated from the Lua bytecode.

Other Interesting Findings. We have also identified other interesting obfuscation phenomena. For the Xiaomi vendor, we identified obfuscation techniques diverging from publicly documented patterns [43], demonstrating that pattern-based deobfuscation approaches easily become outdated, which necessitates more generalized deobfuscation techniques. FirmCross models bytecode invariants and common data transformation schemes, thus can handle such cases. Additionally, in some interpreters, different compilation options resulted in varying obfuscation behaviors. Besides, we observed cases where a single instruction was obfuscated into two instructions, as well as cases where different instructions were obfuscated into the same instruction. Through manual inspection, we found that these involved the UNM and NOT instructions, which the vendor likely considered functionally equivalent.

D. Effectiveness of Lua Source Identification (RQ3)

Experiment Overview. In this experiment, we evaluated FirmCross against LuaTaint [22] on Lua taint source identification. We used the 73 firmware images described in §V-A as evaluation targets. Then we manually inspected the source identification results through dynamic execution to measure the precision. To reduce manual inspection results, for each vendor, we only randomly sampled 50 sources (or all sources if the overall number is less than 50) identified by a tool for analysis. For reliability, we followed a 2-step inspection procedure: ❶ recording the identified source locations (instruction numbers for bytecode and line numbers for source code); ❷ manually checking whether the identified sources are reachable via the network.

Overall Results. Table VI presents the Lua source identification results of FirmCross and LuaTaint [22]. Statistically, FirmCross successfully identified valid sources in the firmware of 10 vendors, with a total of 23,306 sources, while LuaTaint only identified 1,798 valid sources in the firmware of 6 vendors. The inspection revealed that FirmCross achieved a source accuracy rate of 0.84, compared to LuaTaint's 0.18. By analyzing the registration behavior and parameter usage of URI handlers, FirmCross can identify table-based sources in different implementations with high accuracy. In contrast, the source identification scheme of LuaTaint is designed for the specific LuCI framework, which makes it unable to identify sources in other frameworks and results in many false negatives. Additionally, its file-path-based heuristic method is coarse-grained and leads to a large number of false positives.

TABLE VI: Source Identification Comparison between FirmCross and LuaTaint.

Vendor	#Firmware	FirmCross			LuaTaint		
		#Vul	#Source	Prec.	#Vul	#Source	Prec.
TPLink	11	48	5572	1	13	401	0.58
XIAOMI	10	139	11470	0.84	0	0	-
NetGear	15	24	1885	1	33	707	0.4
Tenda	10	250	1466	0.84	0	0	-
Dlink	10	86	1098	0.92	0	248	0.2
Ruijie	10	0	1058	0.86	0	334	0.44
TOTOLink	2	41	532	0.92	0	0	-
Motorola	2	0	138	0.94	0	54	0.2
H3C	1	0	2	1	0	0	-
Mercury	1	0	0	-	0	0	-
TRENDnet	1	0	85	0.94	2	54	0.16
Total	73	588	23306	0.926	48	1798	0.33

#Vul: the number of identified Lua vulnerabilities; **#Source**: the number of identified Lua sources; **Prec.**: the precision of the Lua source identification;

TABLE VII: Cross-Language Date Propagation and Vulnerabilities.

#CL DP		#Vul		
#IPC-based DP	**#API-based DP**	**#CL Vul.**	**#SL C Vul.**	**#SL Lua Vul.**
1622	10584	6	102	588

CL: Cross-Language; **SL**: Singe-Language; **DP**: Data Propagation; **Vul.**: Vulnerability;

Benefits of More Reliable Source Identification. The evaluation results show that FirmCross discovered 12.25 times as many verified Lua vulnerabilities as LuaTaint. As shown in Table VI, we counted the total number of manually verified true vulnerabilities within Lua scripts/bytecode discovered by both methods across all firmware. Benefiting from more identified Lua sources, FirmCross identified 588 vulnerabilities, significantly outperforming LuaTaint, which only detected 48 vulnerabilities.

E. Effectiveness of Cross-Language Vulnerability (RQ4)

Experiment Overview. To systematically analyze cross-language data flows in hybrid web services, we performed a quantitative analysis of cross-language communication identified by FirmCross in 73 firmware images and documented new vulnerabilities discovered through cross-language communication modeling.

Overall Results. Table VII presents the evaluation results of cross-language communication, counting the number of cross-language data propagation via IPC and API. FirmCross has found 1,622 IPC-based communications and 10,584 API-based communications.

Benefits of C-Lua Cross Language Taint Tracking. Moreover, FirmCross discovered 6 cross-language vulnerabilities all triggered in C binaries. These vulnerabilities cannot be detected by simply analyzing C binaries because the data triggering the vulnerabilities originates from Lua bytecode.

VI. Discussion

Limitations of Lua Bytecode Deobfuscation. As a countermeasure against obfuscation, the deobfuscation technique of FirmCross is designed to combat specific obfuscation strategies and may prove ineffective when faced with novel obfuscation methods. The structural deobfuscation approach of FirmCross primarily relies on invariant features of Lua bytecode structures, such as the consistent counts of instructions/constants before and after obfuscation. Consequently, this approach becomes ineffective when obfuscation alters instruction/constant lengths. Additionally, the static bytecode diffing-based data deobfuscation approach employed by FirmCross focuses solely on byte mapping operations, such as XORing string bytes with a constant. This limits its capability to identify and recover from complex data obfuscation techniques like AES encryptions. Nonetheless, we have not encountered such obfuscation scenarios in real-world firmware images.

Common Taint Propagation Defects. During taint propagation, FirmCross primarily addresses the modeling of cross-language data flow between C and Lua. For single-language taint propagation, FirmCross can encounter common taint propagation defects similar to previous techniques, including over-tainting and under-tainting issues [20], [10], [22].

False Positive Mitigation. The FPs stem from two primary factors: the inherent over-tainting issue of taint analysis and the incorrect modeling of custom sanitizer logic. Although little light has been shed on these two obstacles in the firmware security community, existing vulnerability validation techniques [44], [45], [46] in the binary security community can potentially help improve the precision of FirmCross.

Scalability for Other Script Languages. Porting FirmCross to other script languages would require language-specific adaptations: in the source identification phase, we need to analyze source characteristics within the new language; in the cross-language taint propagation phase, we need to identify language-specific DPPI related to new language.

VII. Related Works

Static Vulnerability Detection of Linux-based Firmware. Static taint analysis has been extensively adopted for vulnerability detection in Linux-based firmware, primarily comprising two phases: source/sink identification and taint propagation tracking. Source identification techniques have been systematically studied through cross-domain optimizations: SaTC [9] identifies intermediate sources by correlating frontend keywords with backend input access functions, shortening the taint propagation path; LARA [12] further enhances SaTC with LLM-driven recognition of semantic relations in code and data, uncovering more previously undetectable sources; FITS [11] analyzes the function signatures of input access operations and then uses the signature to locate intermediate sources. For taint propagation tracking, significant advancements have been achieved: Emtaint [10] resolves indirect calls based on the structured symbolic expressions to reconstruct complete execution paths; HermeScan [2] enhances the detection efficiency via optimized reaching dataflow analysis; MangoDFA [20] introduces a sink-to-source strategy that prunes unreachable paths, enabling taint analysis across all firmware binaries with acceptable overhead.

Lua Bytecode Deofuscation. While code deobfuscation techniques are extensively studied across various programming languages [47], [48], [49], [50], the deobfuscation of Lua

bytecode remains largely unexplored. Based on our literature review, LuaHunt [31] represents the state-of-the-art in the field of Lua bytecode deobfuscation. LuaHunt introduces a Lua bytecode structure deobfuscation method that necessitates binary reverse engineering, demanding substantial expert efforts. Additionally, it offers a mutation-based semantic testing approach for data deobfuscation, which assumes that only instruction opcodes are subject to obfuscation. Consequently, this approach falls short in addressing other data obfuscation strategies, such as modifications to data types. In contrast, FirmCross tackles structure deobfuscation using invariant characteristics of Lua bytecode structure fields. Furthermore, FirmCross expands the scope of data deobfuscation to accommodate a wider array of obfuscation schemes by leveraging bytecode diffing-based techniques.

VIII. Conclusion

This paper introduces FirmCross, which incorporates three novel techniques to improve static taint-style vulnerability detection in C-Lua hybrid web services of Linux-based firmware: (1) Lua bytecode deobfuscation enables taint analysis on deobfuscated Lua bytecode; (2) Lua table-based source identification locates more taint sources to facilitate taint-style vulnerability detection; (3) C-Lua communication modeling supports cross-language vulnerability detection. Our evaluations demonstrate the three techniques help detect vulnerabilities in C-Lua hybrid firmware web services, and FirmCross significantly outperforms the SoTA approaches by detecting 6.82X ˜ 14.5X more vulnerabilities.

Acknowledgment

We appreciate the anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (61902416, U2436207, 62172105) and Natural Science Foundation of Hunan Province in China (2019JJ50729). Jiarun Dai, Baosheng Wang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Baosheng Wang is a professor with the School of College of Computer Science and Technology, National University of Defense Technology, Changsha, China. Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] "State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally," https://iot-analytics.com/number-connected-iot-devices, 2024.

[2] Z. Gao, C. Zhang, H. Liu, W. Sun, Z. Tang, L. Jiang, J. Chen, and Y. Xie, "Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis," in *Proceedings of the 2024 Network and Distributed System Security Symposium, San Diego, CA, USA*, vol. 26, 2024.

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

[4] "CVE-2024-20399," https://nvd.nist.gov/vuln/detail/CVE-2024-20399, 2024.

[5] "CVE-2025-20184," https://nvd.nist.gov/vuln/detail/CVE-2025-20184, 2025.

[6] "CVE-2025-30658," https://nvd.nist.gov/vuln/detail/CVE-2025-30658, 2025.

[7] "CVE-2025-30659," https://nvd.nist.gov/vuln/detail/CVE-2025-30659, 2025.

[8] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware," in *Oakland'20*, 2020.

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

[10] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, and L. Sun, "Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 360–372.

[11] P. Liu, Y. Zheng, C. Sun, C. Qin, D. Fang, M. Liu, and L. Sun, "Fits: Inferring intermediate taint sources for effective vulnerability analysis of iot device firmware," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023, pp. 138–152.

[12] J. Zhao, Y. Li, Y. Zou, Z. Liang, Y. Xiao, Y. Li, B. Peng, N. Zhong, X. Wang, W. Wang *et al.*, "Leveraging semantic relations in code and data to enhance taint analysis of embedded systems," in *USENIX Security'24*, 2024, pp. 7067–7084.

[13] A. Qasem, M. Debbabi, and A. Soeanu, "Octopustaint: Advanced data flow analysis for detecting taint-based vulnerabilities in iot/iiot firmware," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2355–2369.

[14] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith *et al.*, "Greenhouse:{Single-Service} rehosting of {Linux-Based} firmware binaries in {User-Space} emulation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5791–5808.

[15] W. Xie, J. Chen, Z. Wang, C. Feng, E. Wang, Y. Gao, B. Wang, and K. Lu, "Game of hide-and-seek: Exposing hidden interfaces in embedded web applications of iot devices," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 524–532.

[16] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 2525–2527.

[17] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 337–350.

[18] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 733–745.

[19] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 417–428.

[20] W. Gibbs, A. S. Raj, J. M. Vadayath, H. J. Tay, J. Miller, A. Ajayan, Z. L. Basque, A. Dutcher, F. Dong, X. Maso *et al.*, "Operation mango: Scalable discovery of {Taint-Style} vulnerabilities in binary firmware services," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7123–7139.

[21] "Lua as the Future of Web Development Insights and Trends — MoldStud," https://moldstud.com/articles/p-why-lua-is-the-future-of-web-development-insights-trends-and-\innovations, 2025.

[22] J. Xiang, L. Fu, T. Ye, P. Liu, H. Le, L. Zhu, and W. Wang, "Luataint: A static analysis system for web configuration interface vulnerability of internet of things device," *IEEE Internet of Things Journal*, 2024.

[23] X. Li, Q. Wei, Z. Wu, and W. Guo, "Finding taint-style vulnerabilities in lua application of iot firmware with progressive static analysis," *Applied Sciences*, vol. 13, no. 17, p. 9710, 2023.

[24] "mod_lua - Apache HTTP Server Support for Lua," https://httpd.apache.org/docs/trunk/mod/mod_lua.html, 2025.

[25] "OpenResty - an nginx distribution which includes the LuaJIT interpreter for Lua scripts," https://en.wikipedia.org/wiki/OpenRest), 2025.
[26] "civetweb - a powerful embeddable web server with Lua support," https://github.com/civetweb/civetweb, 2025.
[27] "lighttpd Support for Lua," https://doc.lighttpd.net/lighttpd2/core_lua.html, 2025.
[28] "LuCI - OpenWrt Configuration Interface," https://github.com/openwrt/luci, 2025.
[29] "CVE-2023-26317," https://nvd.nist.gov/vuln/detail/CVE-2023-26317, 2023.
[30] "CVE-2023-26319," https://nvd.nist.gov/vuln/detail/CVE-2023-26319, 2023.
[31] C. Luo, J. Ming, J. Fu, G. Peng, and Z. Li, "Reverse engineering of obfuscated lua bytecode via interpreter semantics testing," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 3891–3905, 2023.
[32] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, "{PolyFuzz}: Holistic greybox fuzzing of {Multi-Language} systems," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1379–1396.
[33] W. Li, J. Ming, X. Luo, and H. Cai, "{PolyCruise}: A {Cross-Language} dynamic information flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2513–2530.
[34] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1137–1150.
[35] S. Li and G. Tan, "Finding bugs in exceptional situations of jni programs," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 442–452.
[36] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," in *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*, 2024.
[37] "Diaphora, the most advanced Free and Open Source program diffing tool." https://github.com/joxeankoret/diaphora, 2025.
[38] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *NDSS*, 2023.
[39] "Nginx Support for Lua," https://docs.nginx.com/nginx/admin-guide/dynamic-modules/lua/, 2025.
[40] "luacheck: A tool for linting and static analysis of lua code." https://github.com/mpeterv/luacheck, 2025.
[41] "Tscancode: A static code analyzer for c++, c#, lua." https://github.com/Tencent/TscanCode, 2025.
[42] "Semgrep: Lightweight static analysis for many languages." https://github.com/semgrep/semgrep, 2025.
[43] "Exploit (Almost) All Xiaomi Routers Using Logical Bugs," https://hitcon.org/2020/agenda/638a09df-846b-4596-9600-e3727923974c/, 2020.
[44] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 999–1010.
[45] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for {use-after-free} vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 47–62.
[46] S. Yang, Y. He, K. Chen, Z. Ma, X. Luo, Y. Xie, J. Chen, and C. Zhang, "1dfuzz: Reproduce 1-day vulnerabilities with directed differential fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 867–879.
[47] R. Li, C. Zhang, H. Chai, L. Ying, H. Duan, and J. Tao, "Powerpeeler: A precise and general dynamic deobfuscation method for powershell scripts," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4539–4553.
[48] J. Lee and W. Lee, "Simplifying mixed boolean-arithmetic obfuscation by program synthesis and term rewriting," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2351–2365.
[49] L. Shijia, J. Chunfu, Q. Pengda, C. Qiyuan, M. Jiang, and G. Debin, "Chosen-instruction attack against commercial code virtualization obfuscators," in *Internet Society. https://www. ndss-symposium. org/ndss-paper/auto-draft-210*, 2022.
[50] G. Menguy, S. Bardin, R. Bonichon, and C. d. S. Lima, "Search-based local black-box deobfuscation: understand, improve and mitigate," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2513–2525.
[51] S. Alrabaee, M. Debbabi, P. Shirani, L. Wang, A. Youssef, A. Rahimian, L. Nouh, D. Mouheb, H. Huang, and A. Hanna, *Binary Analysis Overview*. Cham: Springer International Publishing, 2020, pp. 7–44. [Online]. Available: https://doi.org/10.1007/978-3-030-34238-8_2
[52] "Function prototype - Wikipedia," https://en.wikipedia.org/wiki/Function_prototype, 2025.
[53] "Calling convention - Wikipedia," https://en.wikipedia.org/wiki/Calling_convention, 2025.

APPENDIX A
META FIELD RECOVERY.

FirmCross primarily deobfuscates array-like fields, i.e., instructions, constants, and protos fields, during structure and data deobfuscation. Nevertheless, some other fields can also be obfuscated. We observe that all these fields, except fields that convey debugging information (e.g., file line numbers), represent meta characteristics of previously deobfuscated fields, and can be inferred from those array-like fields. Specifically, these meta fields contain *Upvalue Num*, *Param Num*, *Is_Varg*, and *Stack Size* fields. We infer the values of these fields by conducting further analysis of the recovered normal constants and instructions regions through mature call convention recovery techniques [51], [52], [53]. For example, the *Param Num* field represents the number of prototype parameters. Since Lua bytecode retrieves parameters through registers, we can analyze the number of registers that are directly accessed but not previously assigned values to determine the value of the *Param Num* field.

TABLE VIII: Assigned Vulnerability IDs

Vendor	Product	Assigned ID	Security Impact
xiaomi	Multiple Devices	CVE-2024-45350	CI
xiaomi	Multiple Devices	CNNVD-2025-01403377	CI
xiaomi	Multiple Devices	CNNVD-2025-19781912	CI
xiaomi	Multiple Devices	CNNVD-2025-63764701	CI
xiaomi	Multiple Devices	CNNVD-2025-88867500	CI
xiaomi	Multiple Devices	CNNVD-2025-65640701	CI
xiaomi	Multiple Devices	CNNVD-2025-09046796	CI
xiaomi	Multiple Devices	CNNVD-2025-81356786	CI
xiaomi	Multiple Devices	CNNVD-2025-95841416	CI
xiaomi	Home Mesh	CNVD-2025-14713	CI
xiaomi	Home Mesh	CNVD-2025-14162	CI
tplink	TL-R470GP-AC	CNNVD-2025-92053103	CI
tplink	TL-R498GPM-AC	CNNVD-2025-22752752	CI
tplink	Multiple Devices	CNNVD-2024-75859171	CI
tplink	Multiple Devices	CNNVD-2024-72222829	CI
tplink	Multiple Devices	CNNVD-2024-14275696	CI
netgear	Multiple Devices	CNNVD-2025-07587675	CI
netgear	Multiple Devices	CNNVD-2025-94673056	CI
netgear	Multiple Devices	CNNVD-2025-99184612	CI
netgear	Multiple Devices	CNNVD-2025-44664084	CI
tenda	5G03	CNNVD-2025-90302764	CI
tenda	5G03	CNNVD-2025-21518636	CI
tenda	5G03	CNNVD-2025-38206496	CI
tenda	5G03	CNNVD-2025-61543504	CI
tenda	5G03	CNNVD-2025-30568258	CI
tenda	5G03	CNNVD-2025-81178109	CI
tenda	5G03	CNNVD-2025-55608524	CI
tenda	5G03	CNNVD-2025-30123348	CI
tenda	5G03	CNNVD-2025-48547128	CI
tenda	5G03	CNNVD-2025-74078812	CI
dlink	823X	CNNVD-2025-08830318	CI

CI: OS Command Injection.

APPENDIX B
DISCOVERED 0-DAY VULNERABILITIES.

The 610 0-day vulnerabilities represent 610 unique (vulnerability, firmware version) pairs, which remain unreported in major vulnerability databases (e.g., CVE, CNVD, CNNVD). Based on our manual verification, 110/610 vulnerabilities share similar root causes (i.e., sink functions and vulnerability-triggering call chains) but affect different firmware versions of the same device. The assigned vulnerability IDs are presented in Table VIII.

APPENDIX C
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: FirmCross's source code is available on Github [2]. The artifact corresponding to the paper is available on the Zenodo [3].

2) Hardware dependencies: We conducted experiments on a host machine with a 56-core Intel(R) Xeon(R) CPU and 128GB of RAM.

3) Software dependencies: We implemented FirmCross using Python 3.11. The dependencies for FirmCross (and the following experiments) are detailed in `README.md`.

4) Benchmarks: None. The artifact is self-contained.

B. Artifact Installation & Configuration

Our repository includes README.md files with step-by-step installation instructions for FirmCross and its dependencies. We have also prepared a virtual machine with a pre-installed experimental environment.

C. Major Claims

We make the following claims in our paper:

- (C1): FirmCross can detect vulnerabilities within C-Lua hybrid web services of real-world firmware. This is proven by the experiment (E2) whose results are reported in Table II.
- (C2): FirmCross can identify Lua-specific sources within web services automatically. This is proven by the experiment (E3) whose results are reported in Table VI.
- (C3): FirmCross can detect the cross-language date propagation. This is proven by the experiment (E4) whose results are reported in Table VII.
- (C4): FirmCross can deobfuscate the obfuscated bytecode within real-world firmware. This claim has been verified and approved in the AE phase, with verification results consistent with Table III and V. The related code of this claim is not open-sourced, with reasons detailed in §C-E

D. Evaluation

You can find more detailed explanation for each experiment in the `README.md` in the artifact.

[2]https://github.com/prankster009/FirmCross

[3]https://doi.org/10.5281/zenodo.16950418

1) Experiment (E1): [Minimized Evaluation] [5 human-minutes + 2 compute-hours]: Since large-scale experiments cannot be completed within 24 hours, we first use a minimized test case to analyze a firmware instance, which allows us to get the minimized results related to C1, C2, and C3 in short time.

[Preparation] Go to the `firmcross_ae/minimize_testcase` folder.

[Execution] Run the command bellow.

```
$ sudo ./begin_vul_detection.sh
```

[Results] Run the command bellow.

```
$ python3
    do_statistic_for_minimize_testcase.py
```

A similar output as follows should be shown.

```
(C1)totoal_vul: 106
      lua_vul: 100
      c_vul: 0
      cross_vul: 6
(C2)identified source: 1148
(C3)total_times: 616
      IPC_times: 240
      API_times: 376
```

2) Experiment (E2): [Overall Vulnerability Detection] [5 human-minutes + 162 compute-hours]: The experiment corresponds to the evaluation of the overall vulnerability detection in Table II.

[Preparation] Go to the `firmcross_ae` folder.

[Execution] Run the command bellow.

```
$ sudo ./large_scope_c_vul_detection.sh
$ sudo python3
    large_scope_lua_vul_detection.py
$ sudo python3
    large_scope_cross_vul_detection.py
```

[Results] Run the command bellow, and the result is in the `vul_detection.xlsx`.

```
$ python3 large_scope_statistic.py -vul
```

3) Experiment (E3): [Source Identification] [5 human-minutes + 5 compute-minutes]: The experiment corresponds to the evaluation of the Lua-specific source identification in Table VI.

[Preparation] This can be done right after (E2).

[Execution] Run the command bellow.

```
$ python3 large_scope_statistic.py -si
```

[Results] The result is in the `source_identify.xlsx`.

4) Experiment (E4): [Cross-Language Data Propagation] [5 human-minutes + 5 compute-minutes]: The experiment corresponds to the evaluation of the cross-language data propagation in Table VII.

[Preparation] This can be done right after (E2).

[Execution] Run the command bellow.

```
$ python3 large_scope_statistic.py -clc
```

[Results] The result is in the `cross_communication.xlsx`.

E. Not Open-Sourced Code and Data

The following code and dataset are not open-sourced: (1) The deobfuscation code. Directly open-sourcing this code would expose manufacturers' protected code (which is guarded by obfuscation), harming their intellectual property rights. Additionally, manufacturers may struggle to design new obfuscation schemes to counter our deobfuscation technology in the short term. To ensure the smooth verification of subsequent experiments, we have provided already deobfuscated Lua bytecode in the dataset. (2) Ten firmware filesystem samples guarded by manufacturers' encryption protection. We have provided the file *dataset_60_fw.xlsx*, which records detailed information about all tested firmware that we newly collected to facilitate further research by other researchers.