Accurate and Efficient Recurring Vulnerability Detection for IoT Firmware

Haoyu Xiao*
Fudan University
China
hyxiao20@fudan.edu.cn

Yuan Zhang*
Fudan University
China
yuanxzhang@fudan.edu.cn

Minghang Shen
Fudan University
China
hadreys1007@gmail.com

Chaoyang Lin
Fudan University
China
cylin.cs@gmail.com

Can Zhang
State Key Laboratory of Mathematical Engineering and Advanced Computing
China
827249870@qq.com

Shengli Liu
State Key Laboratory of Mathematical Engineering and Advanced Computing
China
mr_shengliliu@163.com

Min Yang
Fudan University
China
m_yang@fudan.edu.cn

ABSTRACT

IoT firmware faces severe threats to security vulnerabilities. As an important method to detect vulnerabilities, recurring vulnerability detection has not been systematically studied in IoT firmware. In fact, existing methods would meet significant challenges from two aspects. First, firmware vulnerabilities are usually reported in texts without too much code-level information, e.g., security patches. Second, firmware images are released as binaries, making the analysis of known vulnerabilities and the detection of unknown vulnerabilities quite difficult.

This paper presents FirmRec, the first recurring vulnerability detection approach for IoT firmware. FirmRec features several new techniques to enable accurate and efficient vulnerability detection.

First, it proposes a new exploitation-based vulnerability signature representation for firmware, which does not use syntactic code features but the semantic features along the dynamic vulnerability exploitation procedure (thus is more resilient to binary code changes and fits the context of binary-only firmware). Second, given a vulnerability report, it designs concolic execution-based vulnerability signature extraction to understand the vulnerability exploitation procedure and generate an exploitation-based vulnerability signature. Third, based on known vulnerability signatures, it employs a two-stage pipeline to accurately and efficiently detect recurring vulnerabilities.

With a dataset of 320 firmware images, FirmRec efficiently detects 642 vulnerabilities. Till now, 53 CVEs have been assigned. Compared with SaTC, jTrans, and Greenhouse, FirmRec detects more vulnerabilities and is more accurate.

Our study shows that recurring vulnerabilities are quite prevalent in IoT firmware but require new techniques to detect.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

firmware security, firmware analysis, vulnerability discovery

ACM Reference Format:
Haoyu Xiao, Yuan Zhang, Minghang Shen, Chaoyang Lin, Can Zhang, Shengli Liu, and Min Yang. 2024. Accurate and Efficient Recurring Vulnerability Detection for IoT Firmware. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3658644.3670275

1 INTRODUCTION

Internet-of-things (IoT) has brought unprecedented convenience to human life through ubiquitous connected embedded devices, such as routers, web cameras, and smart printers. According to recent statistics [2], billions of IoT devices have been installed. However, vulnerabilities in the embedded software of IoT devices, i.e., firmware, pose severe threats to personal privacy, property, and public security. For instance, Mirai and its variants have successfully exploited firmware vulnerabilities of a large number of victim IoT devices for conducting large-scale DDoS attacks [10, 14, 33]. Therefore, it is essential to detect firmware vulnerabilities of IoT devices early to mitigate the devastating impact.

Third-party analysts play a crucial role in hunting vulnerabilities in IoT firmware. Actually, numerous IoT firmware vulnerabilities have been reported by third-party analysts. According to Claroty [8,

---

* co-first author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670275

9], at least 64% of IoT vulnerabilities are disclosed by the third party, while IoT device vendors only account for 28%. Typically, due to the closed-source nature of IoT firmware, third-party analysts have to reverse binaries for vulnerability discovery, which is time-consuming and requires extensive expertise.

To aid security analysts in detecting firmware vulnerabilities, existing works mainly rely on modeling vulnerability triggering conditions by static analysis [19, 20, 24, 51, 54] or dynamic testing [17, 22, 30, 37, 66, 68, 69], known as *model-based vulnerability detection*. Specifically, dynamic testing relies on sanitizers [56], while static analysis relies on rules and patterns. However, previous studies [56, 58] have shown that static and dynamic techniques are incomplete or unsound in detecting vulnerabilities due to their inherent limitations (e.g., limited coverage, path explosion).

Recurring vulnerability detection is a complementary technique to model-based vulnerability detection, which uses known vulnerabilities to detect similar new ones. The key insight is that vulnerabilities are usually caused by code reuse or similar buggy code logic. In the area of source code, both model-based vulnerability detection [15, 42, 70] and recurring vulnerability detection [39, 60] have been extensively studied. Their results show that recurring vulnerability detection can significantly extend the capability of model-based vulnerability detection. For instance, TRACER improves recall of CodeQL by 55.8% with a 28.4% precision increase [39]; MVP [60] detects 97 new vulnerabilities on frequently tested projects frequently tested by model-based approaches.

Similarly, IoT firmware is also severely threatened by recurring vulnerabilities. Reports [3, 11, 12] have revealed that IoT firmware from either the same or different vendors is threatened by similar severe vulnerabilities. Therefore, detecting recurring vulnerabilities in firmware is an important research problem. However, due to the closed-source nature of firmware and the lack of code-level vulnerability descriptions, detecting recurring vulnerabilities in firmware has not been studied yet. In general, recurring vulnerability detection works in two steps: extracting signatures of known vulnerabilities and matching vulnerability signatures to detect unknown but similar ones [39, 60]. Following this workflow, we summarize three challenges in firmware recurring vulnerability detection.

❶ The primary challenge is *how to represent the signatures of known firmware vulnerabilities for detecting recurring vulnerabilities across binary code*. Existing works mainly rely on source code tokens, e.g., normalized statements [60] and operators/expressions [39] to represent vulnerable code. However, most source code tokens are dropped during the compilation and thus cannot be used for firmware. As another line of related research, code clone detection is also used to detect recurring vulnerabilities and supports both source code and binary scenarios [23, 25, 27, 28, 32, 44, 59, 62, 67]. However, code clone detection usually leverages general code signatures (e.g., instruction/statement sequences), which fail to distinguish vulnerability code features from other code features (e.g., security patch features and functionality code features), leading to false positives and false negatives in detecting recurring vulnerabilities [60].

❷ The second challenge is *how to extract signatures for known firmware vulnerabilities*. Signature extraction usually requires fine-grained code-level vulnerability information. Most existing works [36, 40, 60, 61, 63] leverage security patches to extract such information. However, patches of firmware vulnerabilities are typically unavailable for third-party analysts. Alternatively, IoT firmware vulnerabilities often have public reports (e.g., NVD [6] and exploit-db [4]), which usually convey useful information to understand the vulnerability. For example, vulnerability reports have been used to synthesize attack signatures for intrusion detection [29]. However, since vulnerability reports only convey text-level vulnerability descriptions, it is still challenging to extract code-level vulnerability information from the reports. This necessitates a hybrid of text analysis with binary code analysis to uncover the code-level details embedded within the firmware, which has not been fully explored.

❸ The third challenge is *how to accurately and efficiently detect unknown recurring vulnerabilities with known vulnerability signatures*. In essence, recurring vulnerability detection is a search problem, i.e., using signatures to discover similar ones in a lot of targets, which suffers from accuracy and efficiency problems. To improve detection accuracy and efficiency, existing approaches mainly rely on semantic information of known vulnerabilities to reduce the search scope and guarantee accuracy. For example, MVP [60] relies on patches to filter target functions that potentially match the vulnerable functions for efficiency, and extract signatures from both vulnerable and patched functions to improve accuracy; Tracer [39] uses well-designed vulnerability-specific code tokens to improve both accuracy and efficiency. However, these approaches cannot be applied in the context of binary-only firmware vulnerabilities.

To address the above challenges, we present FirmRec, an automatic static recurring vulnerability detection approach for IoT firmware. First, FirmRec features an exploitation-based vulnerability signature representation for firmware vulnerabilities. The key observation is that recurring vulnerabilities not only share similar code features but also exhibit similar vulnerability-exploiting behaviors. Second, FirmRec uses vulnerability reports to extract the necessary information about vulnerability exploitation and leverages concolic execution to generate an exploitation-based vulnerability signature from a vulnerable binary. Third, FirmRec employs a two-staged design for vulnerability detection: a light-weight search stage and a heavy-weight validation stage. This design helps to accurately and efficiently detect recurring vulnerabilities.

We have implemented a prototype of FirmRec, and compared it with state-of-the-art vulnerability detection tools—SaTC [19] (a static firmware vulnerability detector), jTrans [59] (a binary code clone detector), and Greenhouse [57] (a firmware fuzzer). Our experiments show that FirmRec outperforms baselines by at least 28.8% in precision and 74.1% in recall, and is 4.2 times faster than SaTC. Besides, FirmRec has uniquely discovered 53 CVEs and 52 of them are regarded as high or critical severity. We have released the code and the dataset [1] to ease the follow-up research.

Contributions. We make the following major contributions:

- *New Approach.* We proposed the first recurring vulnerability detection approach for IoT firmware. Our approach employs several new techniques, including the exploitation-based vulnerability signature to represent known vulnerabilities, concolic execution-based signature extraction, and two-stage vulnerability detection.

[1] https://github.com/seclab-fudan/FirmRec

- *Comprehensive Evaluation.* We evaluated FirmRec with 40 known vulnerability reports and 320 real-world IoT firmware images. The results show that: 1) FirmRec is effective in discovering unknown recurring vulnerabilities; 2) FirmRec significantly outperforms two state-of-the-art baselines (SaTC and jTrans) in both accuracy and efficiency; and 3) FirmRec's internal designs significantly help improve detection accuracy and efficiency.
- *Zero-day Vulnerabilities.* We leveraged FirmRec to discover 642 real vulnerabilities from a wide range of IoT products. Until now, 53 CVEs have been assigned where 52 are labeled with high or critical severity.

2 BACKGROUND AND MOTIVATION

Before discussing the details of FirmRec, we first introduce firmware recurring vulnerabilities and a real-world example. Then, we analyze the limitations of existing works in detecting firmware recurring vulnerabilities. At last, we describe our key insights in firmware recurring vulnerability detection.

2.1 Scope: Firmware Recurring Vulnerabilities

Recurring vulnerabilities are vulnerabilities caused by similar buggy code logic. They have been extensively studied in the source code area [39, 60]. Similarly, IoT firmware is also threatened by recurring vulnerabilities. For example, a vulnerability in D-Link smart camera firmware recurs in TP-Link routers, making tens of thousands of devices exposed to similar remote attacks [11]. These vulnerabilities recur in IoT firmware due to two primary reasons. Firstly, IoT devices usually provide similar functionalities (e.g., configuration backup, file upload, and device reset), so their code logic would be quite similar. Secondly, developers commonly reuse and customize third-party components (e.g., BoA HTTP server [12]) in their own firmware. Considering recurring vulnerabilities can be severe and widespread, it is essential to detect them in an early stage. In this paper, we focus on the network-facing binaries of IoT firmware (e.g., HTTP and UPNP servers) because their vulnerabilities can be exploited remotely, rendering severe consequences.

2.2 Motivating Example

Figure 1 gives a real-world example of IoT recurring vulnerabilities. Specifically, Figure 1(a) details CVE-2019-20500, a command injection vulnerability discovered by third-party analysts in D-Link router firmware. Figure 1(b) presents a recurring vulnerability of CVE-2019-20500, which we have discovered in Netgear firmware. Both vulnerabilities were found in the HTTP server of the firmware.

In Figure 1(a), the vulnerable function downloads a configuration file from a user-specified server IP—*"downloadServerip"*. The *"downloadServerip"* field is retrieved from the HTTP request at Line 3 and is used to compose a tftp download command at Line 10, which is then executed by *system()*. Due to the lack of validation, an attacker can execute arbitrary commands. For example, an attacker can run *"cat /var/passwd"* command by crafting a malformed server IP *"downloadServerip=;cat /var/passwd;"*. Since HTTP download functionality is quite common in IoT firmware, such vulnerable code logic is prone to recur in other modules or firmware images.

As shown in Figure 1(b), we observe a recurring vulnerability of CVE-2019-20500 in the firmware update service of Netgear firmware. An attacker can inject the same malformed server IP to the *"firmwareServerip"* field and run arbitrary commands through this vulnerability, which is similar to exploit *"downloadServerip"* in (a). Besides, while both vulnerabilities share the same vulnerable logic, their vulnerable functions look significantly different because they implement different functionalities. We have renamed the pseudo-code variables and functions to ease the understanding.

Existing Works and Limitations. Due to the unique characteristics of recurring vulnerabilities (i.e., sharing similar vulnerable code logic), there has been a standalone research line for detecting recurring vulnerabilities. In this research area, known vulnerabilities are leveraged to detect similar unknown ones. For example, the vulnerability depicted in Figure 1(a) can be used to detect Figure 1(b). Compared with model-based vulnerability detection, recurring vulnerability detection automatically extracts code features from known vulnerabilities for vulnerability detection instead of solely relying on fixed and pre-defined vulnerability models. Thus, recurring vulnerability detection is an orthogonal and complementary research line to model-based vulnerability detection.

Recurring vulnerability detection generally works in two steps: (1) extracting signatures of known vulnerabilities and (2) matching these signatures to detect unknown vulnerabilities. According to vulnerability representations, existing approaches can be divided into two categories: token-based approach and code clone-based approach. The former one can only be applied to source code [39, 60] while the latter one works on both source code and binaries [25, 28, 35, 52, 59]. Unfortunately, these works would meet huge limitations in detecting recurring vulnerabilities of IoT firmware.

First, the token-based approach leverages source code tokens to represent known vulnerabilities. However, these tokens are usually dropped during compilation. In Figure 1(a), only a few names of dynamically linked external functions are reserved; other tokens are inferred by decompilers and may be incorrect. For example, decompilers may recognize the variable type *"char*"* at Line 3 as *"int"* or *"void*"* due to missing type information. Thus, it is unreliable to match source code tokens with vulnerable binary code.

Second, code clone detection, which aims to find recurrences of a code block (e.g., a function), is also used to detect recurring vulnerabilities. A common practice in this line of research is to use features of the whole code block to represent a known vulnerability. Taking Figure 1(a) as an example, existing code clone detectors usually extract the code features from the whole *config_download_handler()* function and then use these features to match other functions for detecting code clones. As discussed in MVP [60], such coarse-grained vulnerability representation has two shortcomings. Firstly, it cannot detect the recurring vulnerabilities that have similar vulnerable code logic but embed in functions with different functionality, e.g., Figure 1(b). Secondly, coarse-grained vulnerability representation cannot differentiate the patched version from the vulnerable version, introducing a lot of false alarms.

Third, both existing token-based approaches and code clone detection approaches rely on code-level information of known vulnerabilities, which at least encompasses two aspects: *vulnerability locations* and *vulnerability-relevant code. Vulnerability locations* are essential as they dictate the specific segments from which vulnerability signatures are extracted, serving as a cornerstone for the

```
01: int config_download_handler(void *ui) {
02:     ...
03:     char *serverIP = ui_get_input_value(ui, "downloadServerip");
04:     ret = check_suffix(var, "xml");
05:     if (!ret)
06:         return 0;
07:     ret = dump_config("/tmp/config.xml");
08:     if (ret)
09:         return 0;
10:     tftp("/tmp/config.xml", var, "-p", serverIP);
12:     ...
13: }
```

(a) CVE-2019-20500 in D-Link

```
bool tftp(char *local, char *remote, char *opt, char *ip) {
    char cmd[0x84];
    snprintf(cmd, 0x80, "/usr/bin/tftp -l %s -r %s %s %s"
             " >/dev/console 2>&1", local, remote, opt, ip);
    return !system(cmd);
}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
01: int upgrade_handler(void *ui) {
02:     ...
03:     char *serverIP = ui_get_input_value(ui,"firmwareServerip");
04:     tftp("/tmp/upgrade.tar", var, "-g", serverIP);
05:     ...
06: }
```

(b) A Recurring Vulnerability of CVE-2019-20500 in Netgear

Figure 1: Motivating Example. Both vulnerabilities belong to command injection and share similar vulnerable code logic. Note that the names of the variables and functions in the pseudo-code have been renamed to ease understanding.

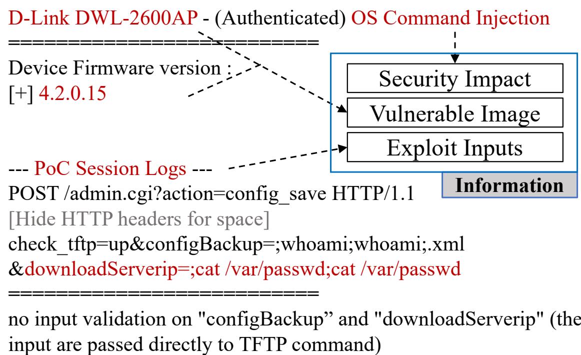

Figure 2: Vulnerability Report of CVE-2019-20500 [3]. We hide some redundant or descriptive texts for space limitation.

detection approach. *Vulnerability-relevant code* pertains to the segments of code that are directly associated with the vulnerability. This latter type of information is crucial for the refinement of vulnerability signatures, thereby enhancing the detection process. For example, MVP [60] leverages modified lines in security patches to derive enhanced vulnerability signatures, which significantly improves its performance over traditional code clone detection methods, achieving a remarkable increase of at least 75.6% in precision and 42.4% in recall. However, the dependency on rich vulnerable code information presents a notable limitation when dealing with closed-source IoT firmware. In such cases, known vulnerability information is mainly described by text-level vulnerability reports instead of code-level security patches. An illustrative example of this limitation is the vulnerability report for CVE-2019-20500 [3], as depicted in Figure 2. Lacking explicit references to vulnerable code slices, vulnerability reports complicate the task of pinpointing the precise location or the relevant code associated with a vulnerability.

2.3 Key Ideas

In light of the above limitations and the challenges mentioned in §1, we have the following key ideas to design an accurate and efficient recurring vulnerability detector for IoT firmware.

Idea-I: Leveraging Exploitation-based Vulnerability Signatures. As discussed in the previous section, the major limitation of existing works is that their vulnerability representations are not applicable in binary code or are too coarse-grained. To design an appropriate vulnerability signature representation for firmware, our observation is that recurring vulnerabilities not only share similar code features but also exhibit similar vulnerability-exploiting behaviors. Specifically, a vulnerability is typically exploited within three steps: reading malformed inputs from the "exploitation entry", driving the code execution along the "exploitation flow" and triggering the final "exploitation impact" (e.g., buffer overflow). Moreover, we observe that such a procedure is similar among recurring vulnerabilities because developers usually make similar mistakes when implementing semantically similar code.

Taking Figure 1 as an example, both vulnerabilities start by reading inputs from "exploitation entry", where both read a server IP; then, they can be exploited with the same malformed server IP values containing a ";" character, which drives code execution along similar "exploitation flows"; finally, they trigger the same "exploitation impact"—command injection, where the malicious inputs are used to invoke arbitrary commands. Thus, we propose using the exploitation procedure (including "exploitation entry", "exploitation flow" and "exploitation impact") to represent the signature of a known IoT vulnerability. By capturing the semantic features during the exploitation of a vulnerability, the new representation is more resilient to binary code changes than previous works. We present more details and examples in §3.1.

Idea-II: Extracting Vulnerability Signatures from Vulnerability Reports with Concolic Execution. After designing the exploitation-based signature representation for firmware vulnerabilities, the next challenge is to extract such signatures from vulnerability reports of IoT firmware, since there is no code-level information about the vulnerabilities, such as security patches. We observe that though the vulnerability reports do not mention any code-level information, they typically mention the exploiting method and the potential exploitation impact, which may help to locate the vulnerability in an affected firmware. For example, the vulnerability report shown in Figure 2 says that the vulnerability in Figure 1(a) could cause *"command injection"* and provides an exploit input to demonstrate the exploiting method of the vulnerability. Our key idea is to leverage concolic execution (a method that combines concrete and symbolic execution [50, 55]) to identify a feasible exploitation procedure in an affected firmware which triggers the described security impact with the described exploit input.

We use Figure 2 and Figure 1(a) as an example to illustrate our idea. Specifically, we first locate the code that may read the attack payloads. Since the constant *"downloadServerip"* occurs both in the exploit input (Figure 2) and the firmware code, we recognize Line 3 in Figure 1(a) as a potential vulnerability exploitation entry. Then, from this code point, we leverage concolic execution to verify there is an exploitable path. The concolic execution initializes the "serverIP" variable with the concrete value *";cat /var/passwd;"*, as the *"downloadServerip"* field in the exploit input is assigned to this variable. After that, the concolic execution explores the program paths from Line 3. During exploration, all uninitialized variables are marked as symbolic (representing arbitrary value) to improve code coverage. Finally, the execution could find a path that reaches Line 10 and triggers the *system()* syscall, which executes a command like *"...;cat /var/passwd;..."*. This shows an attacker-controlled command injection is triggered, just as described in the report (Figure 2). Thus, the discovered exploitation procedure can be used to extract the signature of this vulnerability. Note that the concolic execution part actually has more issues to handle (e.g., the discovery of undesired vulnerabilities and the slow path exploration) than the above description, which are given in §3.2.

Idea-III: Balancing Accuracy and Efficiency with Two-stage Vulnerability Detection. Accuracy and efficiency are two important properties of a vulnerability detector. However, since our vulnerability signatures are based on runtime exploitation behaviors, the detection phase also require to matching the runtime exploitation behaviors, which should leverage an accurate analysis and is time-consuming. To address this challenge, our idea is to employ a two-stage design: the first stage uses a lightweight analysis to find recurring vulnerability candidates and the second stage uses a heavyweight analysis to verify these candidates.

As shown in Figure 1(a), the attack input is injected into the *"downloadServerip"* field. Thus, we use a lightweight constant analysis to find similar input reading locations. In this way, we locate Line 3 of Figure 1(b) as a potential vulnerability candidate because it reads inputs from a similar field, i.e., *"firmwareServerip"*. From Line 3 of Figure 1(b), our heavyweight analysis uses concolic execution to validate whether there is an exploitation flow to trigger the same exploitation impact as the given vulnerability, i.e., command injection. To narrow the search scope of the concolic execution, the attack input value *";cat /var/passwd;"* is used to initiate the *serverIP* variable, and the exploitation flow of the given vulnerability is used as guidance (detailed in §3.3). Following the two-stage design, we could find a good balance between accuracy and efficiency.

3 DETAILED APPROACH

Based on our key ideas mentioned in §2.3, we design an automatic recurring vulnerability detection approach for IoT firmware, namely FirmRec. Figure 3 depicts its workflow. In general, FirmRec works in two steps. ❶ FirmRec automatically analyzes known vulnerability reports to retrieve some necessary information about the vulnerabilities and leverages concolic execution [50, 55] to extract vulnerability signatures from the vulnerable firmware images (see §3.2). To facilitate the detection of recurring vulnerabilities in firmware, FirmRec adopts a new exploitation-based vulnerability signature, which consists of three parts: exploitation entry,

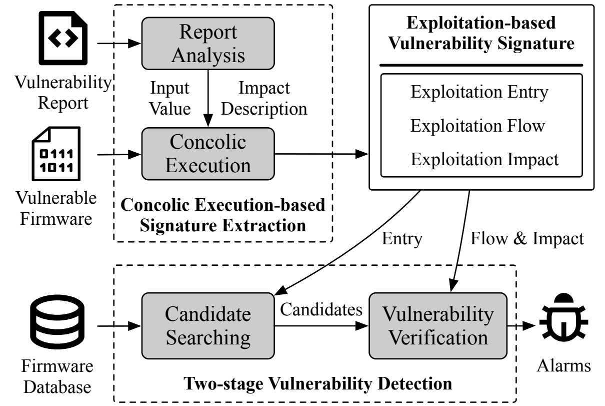

Figure 3: FirmRec Workflow.

exploitation flow, and exploitation impact (see §3.1). ❷ With the extracted signatures of known vulnerabilities, FirmRec automatically detects recurring vulnerabilities in other firmware images within two stages (i.e., a search stage and a verification stage), to guarantee both accuracy and efficiency (see §3.3).

3.1 Exploitation-based Vulnerability Signature

Our exploitation-based vulnerability signature consists of code features related to the dynamic exploitation procedure. Technically, we divide the exploitation procedure into three steps: reading malformed inputs from the exploitation entry, driving the code execution along the exploitation flow, and triggering the final exploitation impact. Accordingly, our exploitation-based vulnerability signature also consists of three parts: "exploitation entry", "exploitation flow", and "exploitation impact". Next, we will detail their designs. Figure 4 shows an example of such signature.

3.1.1 Exploitation Entry. A vulnerable program may have several entries (program locations) that read external inputs. The exploitation entry represents the program location that reads the exploit input that finally triggers the exploitation impact. For example, Line 3 of Figure 1(a) is the exploitation entry because it reads a malformed *serverIP* which triggers the attacker-controlled command injection at Line 10.

As explained later, the exploitation entry is the entry point to extract vulnerability signatures and detect recurring vulnerabilities, so it should be lightweight to locate in a firmware image. Meanwhile, it should be unique enough to stand out from benign input reading locations (denoted as input entries). We observe that firmware binaries usually use constants to read or validate inputs, and these constants usually describe the semantic meaning of the handled data. For example, Line 3 of Figure 1(a) and Line 3 of Figure 1(b) use the constant *"downloadServerip"* and *"firmwareServerip"* to read a server IP address, respectively. Therefore, we use such constants to represent the exploitation entry, such as *"downloadServerip"*.

3.1.2 Exploitation Flow. The exploitation flow describes the key execution trace during the exploitation. We only consider vulnerability-relevant code because vulnerability-irrelevant code behaviors (e.g., *"dump_config"* in Figure 1(a)) are useless to describe the vulnerable logic and even introduce noises in detecting vulnerabilities.

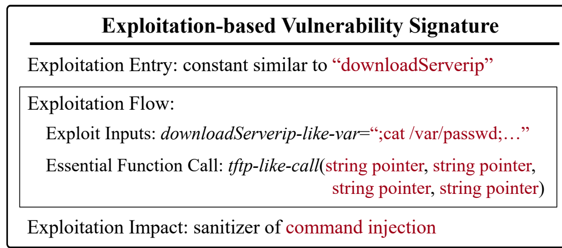

Figure 4: Exploitation-based Vulnerability Signature of CVE-2019-20500.

To this end, we consider two kinds of vulnerability exploitation behaviors to represent the exploitation flow. First, the input data read at the exploitation entry is included in the exploitation flow. Our consideration is that the exploit input data directly determines whether the vulnerability can be triggered or not, thus capturing the most important behaviors of the exploitation. Moreover, the exploit input is not specific to a concrete vulnerable code implementation (e.g., both command injection vulnerabilities in Figure 1 can be exploited by *serverIP=";cat /var/passwd;"*), and thus it is very helpful to detect more recurring vulnerabilities. Second, we include essential function calls that occurred during the exploitation in the exploitation flow. A function call is deemed as essential because it enables the exploitation. Specifically, FirmRec considers three kinds of essential function calls: i) it is the ancestor of the function that triggers the vulnerability (i.e., vulnerability-triggering function); or ii) it modifies the sensitive argument of the vulnerability triggering function; or iii) it modifies the data that affects the path conditions to the vulnerability-triggering function. FirmRec will automatically extract these essential function calls as part of the signature. Considering the function names may be stripped in the firmware binary, we match the prototypes of the essential function calls in the signature to address this issue.

3.1.3 Exploitation Impact. The exploitation impact is the security consequence caused by a vulnerability. It describes the purpose of the attacker and is an important feature of the vulnerability. Though the exploitation impact is described as text in a vulnerability report (e.g., NVD), we need to capture it at the code level to support vulnerability detection. Technically, FirmRec employs sanitizers to identify the triggering of a vulnerability, so we use the triggered sanitizer to represent the exploitation impact of a vulnerability. For example, both vulnerabilities in Figure 1(a) and Figure 1(b) can be captured by a command injection sanitizer. We present more sanitizer implementation details in §4.1.

3.2 Concolic Execution-based Signature Extraction

Before detecting recurring vulnerabilities, FirmRec first needs to extract the signatures of known vulnerabilities from vulnerability reports. As shown in Figure 3, FirmRec has two steps for signature extraction. First, it automatically analyzes a vulnerability report to extract the necessary information to understand the vulnerability. Second, it uses concolic execution to locate the vulnerability and the vulnerability-relevant code in an affected firmware image and then generates an exploitation-based vulnerability signature.

3.2.1 Vulnerability Report Analysis. To facilitate concolic execution-based vulnerability signature extraction, we extract three types of vulnerability information from a vulnerability report showcased by Figure 2: (1) affected products and versions of the vulnerability that are used to find a vulnerable image; (2) a description of the security impact when the vulnerability is triggered/exploited; (3) attack inputs that are used to trigger/exploit the vulnerability.

Inspired by IoTShield [29], we introduce a pattern-based approach to automatically extract such vulnerability information from reports. First, FirmRec extracts the information about affected products and versions. If FirmRec can find a link to a public vulnerability database (e.g., NVD) from the report, it directly extracts the product and version information (e.g., the Common Platform Enumeration) from the database. Otherwise, we use regular expressions to match the product/version string in the vulnerability report. The product/version strings are collected/constructed the same as IoTShield [29]. Second, FirmRec extracts the description of a vulnerability's security impact using two methods. For most cases, a public vulnerability database (e.g., NVD) will label the vulnerability category (e.g., CWE number), which describes its security impact. In these cases, FirmRec uses the vulnerability category's name to describe the security impact. In other cases, FirmRec uses common keywords that describe the security impact (e.g., buffer overflow) to match the vulnerability report. If such keywords are found, they are used to describe the security impact. Table 1 shows these keywords. Third, FirmRec extracts exploit inputs from vulnerability reports. As described in [29], the exploit inputs may be represented as network traffic logs or PoC/exploit scripts. We first define representative patterns of exploit inputs to locate them in the reports. In most cases, the traffic logs and scripts can be identified with special markers (e.g., "' in Markdown language). In other cases, they are usually introduced by special texts, such as *"Proof of Concept"*. Then, from the location of the exploit inputs, FirmRec extracts their concrete values, which will be used for concolic execution. The traffic log directly provides the concrete inputs we need, while for the scripts, FirmRec executes them and extracts the concrete values from the traffic by hooking network-facing APIs (e.g., *send()* in Python scripts).

3.2.2 Signature Generation. With the information extracted from the vulnerability report, signature generation is initialized by collecting a vulnerable firmware image. Specifically, FirmRec queries the extracted affected products and versions to find the depicted image in a firmware database, such as the one built by [17].

FirmRec then leverages concolic execution to discover the vulnerable code of the given vulnerability in the firmware. However, this step is non-trivial due to two reasons. First, the firmware image may contain multiple vulnerabilities, and it is difficult to confirm whether a discovered vulnerability is the given one. Second, concolic execution is usually quite slow in exploring many program states, leading to significant efficiency concerns.

To address these problems, we introduce three new designs to accurately and efficiently discover the given vulnerability with concolic execution. ❶ To eliminate the need to analyze every input entry, FirmRec only focuses on those mentioned in the vulnerability report. Besides, it starts concolic execution from these specific locations rather than the binary program's entry point to shorten

the execution path (inspired by the idea of under-constrained symbolic execution [50, 55]). ❷ During concolic execution, FirmRec adaptively prunes/explores function calls. It first employs an aggressive strategy that prunes all uninteresting function calls for efficiency. If no vulnerability is discovered, it restarts the execution and adopts a new strategy to gradually explore previously pruned function calls. This design prioritizes exploring shallow code logic while still guaranteeing that the deep code logic (especially that affects the triggering of the vulnerability) could be explored during the execution, albeit at a later stage. ❸ When a vulnerability is discovered, FirmRec features a verification step to confirm whether it matches the given vulnerability report.

Following the above designs, we depict the signature generation algorithm in Algorithm 1. Initially, Line 1 identifies the program locations that read exploit inputs in the vulnerable firmware. Then, for each identified location L, Line 4 adaptively explores the code from this point with concolic execution until a sanitizer triggers a vulnerability. Line 5 verifies whether the triggered vulnerability matches the vulnerability report. If the given vulnerability is discovered, Line 6 extracts its signature by inspecting the vulnerability-exploiting execution trace. Specifically, the vulnerability signature consists of the exploitation entries, the essential function calls that enable the triggering of the vulnerability, the input values fed into the exploitation entries, and the sanitizer that triggers the vulnerability. Next, we present more details about each major step.

Algorithm 1 Vulnerability Signature Generation

```
Procedure GenerateSignature(Image, Exp, Impact)
Input: Image - A vulnerable image,
    Exp - Exploit input,
    Impact - Exploitation impact
Output: Sig - Vulnerability signature
 1: LL ← IdentifyInputReadingLocations(Image, Exp)
 2: for L ∈ LL do
 3:     ExeInstance ← InitConcolicExecution(L, Exp)
 4:     Vul ← AdaptiveExploration(ExeInstance)
 5:     if IsReportedVulnerability(Vul) then
 6:         return ExtractSignature(Vul)
 7:     end if
 8: end for
```

Identify Input Entries and Initialize Concolic Execution (Line 1&3). FirmRec first identifies all input reading functions of a firmware binary, then it locates code that potentially reads the given exploit input (i.e., input entries) in the vulnerable firmware, and finally, it initializes the concolic execution with the input. The three sub-steps are described below.

(a) FirmRec first identifies two types of input reading functions that are often encountered in IoT firmware—raw byte reading functions and structure (data) reading functions. *Raw byte reading functions* directly retrieve raw bytes via well-defined OS APIs or system calls, such as *recvfrom()* (specified in the POSIX standard). Due to their employment of standardized specifications and well-established implementation practices, FirmRec swiftly identifies them through symbol or pattern matching.

Structure reading functions are also commonly used in IoT firmware as noted in [19]. These functions typically leverage constant names to extract structured data, such as *"downloadServerip"* in Figure 1. However, such functions lack standardization and vary widely due to different development practices, complicating the automated identification. Traditional methods either rely on pre-defined function symbols [51] or leverage likely external input names [19] to infer these functions. Nonetheless, these methods fall short of capturing all structure reading functions. FirmRec confronts this challenge by proposing a conservative behavior-based method to recognize as many structure reading functions as possible while eliminating false positives with follow-up analysis. Specifically, we observe that such functions usually take constants as input, and their outputs will be used in the binary. Thus, if the majority (more than 80%) of calls to a function conform to such behavior, it is recognized as a structure reading function.

(b) After the identification of input reading functions, FirmRec finds the locations that may read the given exploit input. The basic idea is to extract the constants representing individual data elements from the exploit input and use these constants to locate the reading locations in the firmware (as illustrated in §2.3). To comprehensively extract these constants, FirmRec first parses the exploit input using common parsers, e.g., HTTP, JSON, XML, URL, and YAML. Based on the parsed results, FirmRec collects constant names of the input data elements. When these parsing tools fall short, FirmRec treats the input as raw bytes and extracts all discovered constants. Specifically, FirmRec extracts uncommon 4-byte numbers (excluding 0, 2^n, and $2^n \pm 1$), visible name characters, and delimiters between invisible characters.

Then, FirmRec uses the extracted constants from the exploit input to match the constants discovered within the firmware binaries, pinpointing potential locations that may read the exploit input. Specifically, the matched constants can either be names of structure reading locations or constants used to process (e.g., validate) inputs at raw byte reading locations. These constants are extracted from the arguments of data reading functions or input comparison code with traditional binary data-flow analysis, without relying on any debug symbols. Given that an exploit input may be consumed with multiple readings at different locations, FirmRec groups reading locations on the same execution path and starts concolic execution from the first input entry. Besides, to improve the efficiency of discovering the target vulnerability, FirmRec prioritizes the input entry if its group has more relevant constants to the original vulnerability report.

(c) For each identified input entry, FirmRec feeds it with concrete values provided by the exploit input for concolic execution. According to the type of input reading function, FirmRec feeds the exploit input data in different ways. For functions that read raw bytes, FirmRec follows their API standards to load and store the input data. For functions that read structured data, FirmRec uses the constant name to get the data in the exploit input and feed the data into its output variables. By using the concrete values, FirmRec can use the concolic execution to only explore targeted program paths (that can trigger the given vulnerability).

Adaptively Prune/Explore Function Calls (Line 4). When the concolic execution is initialized, FirmRec starts to discover the given vulnerability by exploring the code. In theory, symbolic/concolic execution should step into every function call to explore all

the possible program paths. However, this is prone to cause the classical path explosion problem. In fact, not all the functions met are important to the vulnerability discovery; thus, such functions could be safely skipped during the exploration. In this way, since few useless code are explored, symbolic/concolic execution could explore deeper code space and probably detect more vulnerabilities. That is why existing symbolic/concolic execution-based firmware vulnerability detectors adopt function call pruning during the code exploration [19, 51]. However, the most challenging problem is determining whether a function call can be safely pruned.

By inspecting existing approaches, we find both SaTC [19] and Karonte [51] employ an *aggressive function call pruning strategy*. Specifically, they taint the variables that are read from the external and propagate the taint tags during the symbolic execution. When a function call is met, they will prune the function call if it does not have a tainted argument. The pruned call is replaced with stubs that only return symbolic values. Though this strategy is lightweight to implement, it would miss some function calls that are useful to detect vulnerabilities. For example, a function call may manipulate tainted variables through global variables (and thus affects the triggering of a vulnerability), even if its argument does not have a tainted argument. Therefore, such an aggressive pruning strategy would cause false positives/negatives to vulnerability detection.

To mitigate the above limitation, we propose an *adaptive function call pruning strategy*. The core idea is to first explore shallow function calls and then dive into deeper function calls gradually if no desired vulnerability is detected. Specifically, FirmRec first adopts an aggressive function call pruning strategy during concolic execution (i.e., same to SaTC [19] and Karonte [51]). Then, if no vulnerability is discovered, FirmRec updates the strategy by adding previously pruned function calls. This continues until a vulnerability is discovered or timeout.

Verify Known Vulnerabilities (Line 5). During the concolic execution, FirmRec may trigger some abnormal program behaviors that can be captured by sanitizers and reported as vulnerabilities. Actually, many of such abnormal behaviors might be false alarms due to several reasons, e.g., the concolic execution doesn't start from the entry point, or the input read location is wrong. Thus, we choose to only identify the vulnerability with aligned behaviors to the vulnerability report, which has been confirmed to be a true vulnerability. To this end, FirmRec requires two properties on the triggered vulnerability. First, FirmRec ensures that the vulnerability is controlled by the exploit input. For example, if a command injection vulnerability is triggered but the command cannot be controlled by the exploit input, it is not a real vulnerability. Technically, this requirement is achieved with a taint analysis to track the data flow from the exploit input. Second, FirmRec guarantees that the vulnerability is triggered by the sanitizers that match the security impact described in the given vulnerability report. For example, FirmRec shall ignore a buffer overflow vulnerability when the report describes a command injection vulnerability.

3.3 Two-stage Vulnerability Detection

Based on the extracted signatures, FirmRec detects unknown recurring vulnerabilities with a two-stage design to guarantee both the detection accuracy and efficiency. Technically, in the first stage, FirmRec adopts a lightweight analysis to efficiently search for vulnerability candidates as many as possible. Then, in the second stage, FirmRec adopts a heavyweight concolic execution to accurately verify whether a vulnerability candidate is a recurring vulnerability. Meanwhile, both stages are guided by the exploitation-based vulnerability signatures to further reduce the search scope.

3.3.1 Stage 1: Vulnerability Candidates Searching. FirmRec systematically searches for input entries semantically similar to known exploitation entries and flags them as potential recurring vulnerability candidates. First, FirmRec identifies the input entries in a target firmware binary and extracts the constants that are used to read or validate inputs for each input entry, using the same technique for signature extraction (see §3.2.2). We differentiate two kinds of constants that are extracted at each input entry: *named* and *unnamed* constants. The named constants are used as names to read input data. For example, *"downloadServerip"* is a named constant to read an IP address, and *"addr_list"* is a named constant used to read addresses in a list format. The unnamed constants are used to process input data. For instance, the punctuation symbol "dot" is an unnamed constant used to divide name spaces or file suffixes, and "comma" is an unnamed constant often used to separate list elements during parsing. These constants encode either meanings or formats of the input data, which are closely relevant to the semantics of the input entries. Thus, FirmRec then uses the semantic similarity of these constants to find input entries semantically similar to a given exploitation entry.

Specifically, for the *unnamed constants*, FirmRec only identifies input entries with exactly the same constants, because unnamed constants are highly sensitive to alterations (e.g., "dot" and "comma" differ one ASCII bit but have significantly different meanings). For the *named constants*, simply identifying similar names by their literal words may cause false negatives, because semantically similar names can vary significantly in their literal forms (e.g., both *"password"* and *"auth_secret"* represent passwords). To avoid such false negatives, our idea is to leverage the large language model (LLM) [1] to compare names by their underlying semantics. We design an LLM prompt that uses two key requirements for identifying similar names. First, the names' basic meaning, i.e., denotation, must be analogous to discover similar pairs like *"password"* and *"auth_secret"*. Second, the names should potentially convey similar data formats, suggesting their potential use in analogous data processing logic. The detailed prompt is presented bellow. If the LLM answers *Yes*, the compared names are deemed as similar and the corresponding input entry will be identified as a vulnerability candidate. Owing to LLMs' strong semantic understanding capability, this helps to discover vulnerability candidates as many as possible. For example, in our experiments, these name pairs are deemed as similar: *key* vs *passphrase*, *dev_name* vs *deviceId*, *ssid* vs *wifiFilterListRemark*.

Infer whether two variables from IoT software may satisfy both requirements:

(1) have similar denotation words;
(2) may convey similar data formats.

The answer is 'Yes' or 'No': $\{Constant1\}\{Constant2\}$

A practical issue here is efficiency. According to our experiment, it averagely costs 1 minute to compare about 30 name pairs with LLMs. To improve efficiency, FirmRec uses two steps for name similarity comparison. First, it uses semantic-aware sub-word matching to recognize name pairs that share no sub-word and deem them dissimilar. This greatly reduces the need to use LLM, as most name pairs are dissimilar in semantics. Then, only a small part of name pairs require the LLM-powered semantic comparison. To avoid overlooking semantically-similar names, the sub-word matching considers not only the words that form the named constant but also their abbreviation and synonym words. To obtain such sub-words, FirmRec employs LLMs to segment words and find abbreviations/synonyms. The segmented words and their abbreviations/synonyms words are kept in a dictionary, so the semantic-reserving sub-word matching only takes linear time to finish. In our experiments, this two-step approach reduced 94.4% of needs for using LLMs while still maintaining high accuracy.

3.3.2 Stage 2: Vulnerability Verification. Each vulnerability candidate identified in the first stage has a similar input entry to the exploitation entry of a known vulnerability. In this stage, FirmRec verifies whether the input entry of each vulnerability candidate can trigger a recurring vulnerability. To guarantee detection accuracy, FirmRec applies concolic execution for verification. Besides, to improve the efficiency of the concolic execution, we use the exploitation flow (i.e., exploit input data and essential function calls) of the given vulnerability signature as guidance.

Technically, FirmRec starts the concolic execution from the input entry of a vulnerability candidate and feeds the exploit input data (which is kept in the exploitation flow of the given vulnerability signature) into the entry. Then, the concolic execution explores the program states path-sensitively and precisely tracks data flows. During the exploration, FirmRec uses static sanitizers detailed in §4.1 to capture the triggering of a vulnerability. If a candidate shares the same exploitation impact as the given signature and is triggered by the exploit input, FirmRec reports it as a recurring vulnerability.

Guided Function Call Pruning Strategy: To mitigate path explosion during concolic execution and avoid pruning some essential function calls to trigger the vulnerability, FirmRec leverages the known exploitation flow to guide the pruning. Specifically, FirmRec retains essential function calls given by the vulnerability signature and prunes non-essential ones. However, a key technical challenge arises when attempting to match function calls from a vulnerability signature to those that occur in a different firmware image. Traditional binary code similarity techniques, while useful, often meet efficiency and accuracy issues. These techniques must analyze the whole function code for calculating code similarity, but they still miss essential function calls particularly when functions share similar purposes but differ in implementations.

Instead, we propose matching functions by their prototypes. Specifically, we infer the function prototype by recognizing the argument types based on their run-time values. For example, when an argument points to a constant string, FirmRec classifies it as a string pointer. Similarly, FirmRec also identifies various argument types encountered during execution, including concrete number/number pointer, symbolic value/value pointer, etc. This strategy, while may keep some inessential function calls (due to the false positives in function matching), will not hurt the vulnerability detection capability. Compared with the adaptive function call pruning strategy used in signature extraction (see §3.2.2), such a guided strategy is more efficient and appropriate for detecting recurring vulnerabilities.

4 EVALUATION

We conducted extensive experiments to evaluate FirmRec's performance in detecting real-world vulnerabilities (**RQ1**), compare it with existing vulnerability detection tools (**RQ2**), and measure the contributions of its internal designs (**RQ3**).

4.1 Experiment Setup

Prototype. We implemented a prototype of FirmRec with about 9,000 lines of Python code and 3,600 lines of Java code. For a vulnerability report, the prototype used Python scripts to extract necessary vulnerability information. For a firmware image, the prototype first used Binwalk [7] to extract binaries; then it used Ghidra [5] to identify and locate all input reading locations and used the concolic execution engine of Angr [55] to generate known vulnerability signatures and detect recurring vulnerabilities. Currently, the prototype focuses on Linux-based firmware. To extend the prototype to other types of firmware, e.g., RTOS firmware, Angr needs more firmware information such as load address, architecture, and memory layout. When generating vulnerability signatures and verifying a vulnerability candidate, the concolic execution engine was set with a timeout of 5 and 10 minutes correspondingly. When identifying vulnerability candidates, the prototype leverages ChatGPT LLM [1] with its temperature parameter set to zero. This makes the results more deterministic and reproducible.

Similar to existing firmware vulnerability detectors like SaTC [19], the prototype detects 4 kinds of vulnerabilities with sanitizers during the concolic execution: stack, heap, data segment buffer overflow, and command injection. The core idea is to detect the direct security impact of vulnerabilities during exploitation. Specifically, the stack overflow sanitizer detects whether the return pointer is overwritten by malicious inputs using a return stack; the heap overflow sanitizer checks whether a write operation (e.g., *memcpy()*) writes multiple heap chunks, thus corrupting the heap data, by maintaining heap boundary information; the data segment overflow sanitizer checks whether a write operation writes across data segments or variables, where we only consider data segments and variables whose boundaries can be recovered through binary analysis; and the command injection sanitizer checks whether attacker-controlled inputs are used to invoke command execution APIs (e.g., *system()*). All these checks are performed at specific program locations using a lightweight instruction-level hook during symbolic execution. For example, the stack buffer overflow sanitizer is only called at function returning instructions. Besides, we have summarized the security impact description for each sanitizer to facilitate the mapping between them (See Table 1).

Firmware Images. For the evaluation, we collected 9,993 firmware images from the official websites of 12 popular IoT vendors. Utilizing the binwalk tool [7], we extracted file systems from 4,708 images. To effectively evaluate FirmRec, we employed a two-step sampling process on the extracted binaries. First, to ensure a broad representation of the vendor's product range, we randomly selected

Table 1: The sanitizers and their corresponding security impact descriptions.

<table>
<tr><th>Sanitizer</th><th colspan="3">Security Impact Description</th></tr>
<tr><td>Stack Buffer Overflow</td><td>stack overflow, stack-based buffer overflow, CWE-121</td><td rowspan="3">buffer overflow, overflow a buffer, denial of service, DoS, out-of-bounds, OOB, improper boundary check, CWE-119, CWE-120, CWE-306, CWE-787, CWE-788, CWE-823</td><td rowspan="4">code execution, elevate privileges, bypass authentication, execute arbitrary code, execution of arbitrary code, improper input validation, improper validation, RCE, CWE-826</td></tr>
<tr><td>Heap Buffer Overflow</td><td>heap overflow, heap-based buffer overflow, heap buffer overflow, CWE-122</td></tr>
<tr><td>Data Segment Buffer Overflow</td><td>overwriting of sensitive memory</td></tr>
<tr><td>Command Injection</td><td colspan="2">command injection, command execution, execute arbitrary command, execute arbitrary OS command, run arbitrary commands, execute system command, CWE-20, CWE-77, CWE-78</td></tr>
</table>

approximately $\frac{1}{15}$ of the images from each vendor. This approach aligns with standard practices in data representativeness and helps mitigate selection bias. Next, we focused on extracting network-facing binaries from these images, given they are more susceptible to network-based attacks. This sampling approach is in line with methodologies adopted in prior research [19, 51]. Finally, we collected a dataset of 2,111 unique binaries from 320 firmware images. Our dataset covers a variety of IoT products, such as VPN, camera, AP, smart switch, WiFi extender, router devices.

Baselines. To answer RQ2, we need to select several baselines. To the best of our knowledge, FirmRec is the first recurring vulnerability detector for firmware. Thus, we have to choose baselines from related research areas. After a literature review, we found three most related research areas: binary code clone detection, static analysis-based firmware vulnerability detection, and grey-box fuzzing.

- *Binary code clone detection.* We chose jTrans [59], which adopts a Transformer architecture for binary code similarity detection and outperforms previous approaches in various settings. Specifically, we collected known vulnerable functions and directly used the model trained by jTrans to identify their vulnerable code clones.
- *Static firmware vulnerability detection.* We chose SaTC [19], which is the state-of-the-art firmware vulnerability detector and outperforms the previous tool—Karonte [51]. It takes a firmware image as input and identifies the shared variable names between the front-end files and back-end functions. Then, it locates the input reading locations of the variables with the identified names and leverages symbolic execution to find sensitive sinks (same as FirmRec) reachable from these locations. If a sensitive sink is found and its critical argument (e.g., the *command* argument to the *system()* syscall) can be controlled by the front end, SaTC reports it as a vulnerability. We ran SaTC with its default settings.
- *Grey-box firmware fuzzing.* We chose Greenhouse [57]—the leading firmware fuzzing tool. By addressing several key roadblocks in emulation, Greenhouse can effectively rehost firmware user-space services and fuzz them with AFL++ [13]. We ran Greenhouse for 24 hours for each binary under test.

Known Vulnerabilities. Both jTrans and FirmRec require known vulnerabilities as inputs. We collected public IoT vulnerabilities from NVD [6]. First, we searched all the vulnerability entries reported from 2019 to 2023 with common IoT vendor names (e.g., *dlink*) and IoT-related keywords (e.g., *router*). Second, we manually checked the search results against four conditions to build a known vulnerability dataset for comparing jTrans and FirmRec: (1) it is a true IoT firmware vulnerability; (2) there is at least one affected firmware image in our collected dataset; (3) we could locate the vulnerable functions in the affected firmware image; (4) we could find a vulnerability report that describes (partial) exploit inputs and the exploitation impact. Note that the first and second conditions are natural requirements for our evaluation, the third condition is only required by jTrans, and the fourth condition is automatically verified by FirmRec with given vulnerability reports. By checking 54 vulnerabilities, we finally collected 40 known vulnerabilities from 8 vendors that satisfied all four conditions. They are used as inputs for jTrans and FirmRec in the following experiments.

Environment. We ran FirmRec and SaTC on a Ubuntu 20.04 LTS machine equipped with a 2.30 GHz Intel Xeon(R) Gold 5218 CPU and 245 GB memory. Since jTrans relies on a neural network model, we ran jTrans on a Ubuntu 18.04.6 LTS machine equipped with a 3.20 GHz Intel(R) Xeon(R) Silver 4215R CPU, an NVIDIA GeForce RTX 3070 GPU, and 58 GB memory.

Table 2: FirmRec vulnerability detection results (RQ1).

Vendor	Stage I		Stage II		
	# All Entries	# Found Entries	# Alarms	# Sampled[1]	# Vulns[1]
ASUS	92,360	6,129	24	6	2
Buffalo	139,466	3,681	502	126	101
Cisco	49,038	4,252	131	33	9
Dahua	7,032	89	0	0	0
D-Link	132,214	7,081	331	83	58
Linksys	86,254	6,679	200	50	20
Mikrotik	494	5	0	0	0
Netgear	720,324	66,960	2,653	663	241
Tenda	190,944	25,708	733	183	155
Tomato	59,522	4,464	104	0	0
TOTOLink	25,784	3,074	78	20	17
TP-Link	115,514	17,479	214	54	39
Summary	1,618,946	145,601	4,970	1,244	642

[1] The results are reported by randomly sampling 25% of all alarms.

4.2 RQ1: Vulnerability Detection Experiments

In this experiment, we applied FirmRec to detect vulnerabilities in real-world firmware images. First, FirmRec generated vulnerability signatures for the 40 known vulnerabilities in 18,428 seconds. Then, using the extracted vulnerability signatures as input, FirmRec spent 11,259,535 CPU seconds to finish the analysis of 2,111 binaries.

Table 2 presents the vulnerability detection results. In stage I, FirmRec found 145,601 vulnerability candidates, representing a significant reduction to just 9% of all 1,618,946 input entries. In stage II, FirmRec verified all these candidates and generated 4,970 unique alarms.

To measure the precision of FirmRec, we manually investigated the reported alarms against three conditions. (1) The input entry of an alarm is reachable from the program entry point. (2) The exploit inputs of the alarm are read directly from the network or indirectly via particular program paths. This step ensures that the exploit

Table 3: Effectiveness comparison results of jTrans, SaTC, and FirmRec (RQ2).

Vendor	# Img	# Bin	Precision						Recall			
			# jTrans		# SaTC		FirmRec		# Union TPs	jTrans	SaTC[3]	FirmRec[3]
			K[1]=10	K[1]=20	Alarms	Precision[2]	Alarms	Precision[2]		K[1]=20		
ASUS	5	35	7.7%	3.4%	8	50.0%	24	33.3%	4	25.0%	25.0%	50.0%
Buffalo	11	66	-	-	0	-	502	80.2%	101	0.0%	0.0%	100.0%
Cisco	5	37	-	-	53	16.7%	131	27.3%	11	0.0%	18.2%	81.8%
Dahua	7	19	-	-	0	-	0	-	0	-	-	-
D-Link	29	220	2.9%	3.3%	200	64.0%	331	69.9%	87	2.3%	37.9%	66.7%
Linksys	39	216	-	1.2%	4	50.0%	200	40.0%	22	4.5%	4.5%	90.9%
Mikrotik	4	10	-	-	0	-	0	-	0	-	-	-
Netgear	108	914	14.28%	9.21%	641	15.0%	2,653	36.3%	291	8.9%	8.6%	83.2%
Tenda	47	213	32.61%	24.74%	258	12.5%	733	84.7%	189	24.9%	5.3%	82.5%
Tomato	21	97	-	-	0	-	104	0.0%	0	-	-	-
TOTOLink	8	60	-	-	0	-	78	85.0%	17	0.0%	0.0%	100.0%
TP-Link	36	224	5.13%	2.5%	68	11.8%	214	72.2%	42	4.8%	4.8%	95.2%
Summary	320	2,111	13.5%	9.88%	1,232	22.8%	4,970	51.6%	764	10.3%	9.7%	84.4%

[1] The top K items in the ranked list are deemed as vulnerabilities.
[2] The precision of SaTC and FirmRec are reported by sampling 25% of all alarms.
[3] The recall of SaTC and FirmRec are reported by evaluating all alarms on the ground truth.

inputs can be controlled externally. (3) All security checks can be satisfied in some specific program paths. The alarm is deemed a true positive if all three conditions can be satisfied.

Following the above steps, we manually checked 25% of all the alarms (i.e., 1,244 alarms). These alarms were randomly sampled for each vendor, and thus are representative. Our investigation confirmed 642 vulnerabilities, demonstrating a precision of 51.6% for FirmRec. We believe such precision is quite acceptable for a static analysis-based firmware vulnerability detector.

Responsible Disclosure. We have responsibly disclosed the confirmed vulnerabilities to the vendors and the CVE database. Among these vulnerabilities, 47 belong to command injection while the left ones are buffer overflows. These vulnerabilities could lead to severe security consequences, such as allowing an attacker to fully control the device. Till now, 53 CVEs have been assigned, including 35 with critical and 11 with high severity (CVSS) score. Besides, all these vulnerabilities are not reported by jTrans nor SaTC in §4.3.

Cross-architecture/-optimization Performance. We further analyzed the verified 1,244 alarms to measure FirmRec's performance under cross-architecture/-optimization settings. Specifically, by parsing the ELF headers, we found 883 alarms were reported in a cross-architecture setting (i.e., the alarm is reported in a binary with a different architecture to the signature binary) and the detection precision is 47.0%. Moreover, we used a deep learning model [49] to infer the optimization level of a binary. As a result, 11 alarms were found to be reported under a cross-optimization setting and the detection precision is 54.5%. We also found that most (about 93%) firmware binaries were compiled with *"-Os"*. This explains why few alarms were reported in a cross-optimization setting. In summary, FirmRec achieves comparable performance to the overall dataset even under cross-architecture/-optimization settings.

4.3 RQ2: Comparison Experiments

We first conducted the effectiveness comparison experiments between FirmRec and static analysis-based approaches (jTrans and SaTC) and then with fuzzing-based approaches (Greenhouse). Finally, we reported the efficiency comparison results.

4.3.1 Effectiveness Comparison with jTrans and SaTC. In this experiment, we used the same dataset used in RQ1. As a binary code similarity detection model, jTrans cannot directly detect vulnerabilities. Instead, it outputs a ranked list of binary functions according to their similarity with the given vulnerable binary functions. To identify vulnerabilities from the ranked list, we followed the same method used by jTrans in its original paper [59] to evaluate its effectiveness. Briefly, jTrans defines a threshold K and recognizes the top-K similar functions in the list as vulnerable.

Due to the large manual effort to analyze all the alarms, we randomly sampled 25% of alarms reported by SaTC and FirmRec for each vendor to check. This approach is in line with the practice of the previous work [60]. Besides, since jTrans identifies the top-K functions as vulnerable, we could tune the K parameter to determine the number of alarms that require manual checks. Specifically, we set K to 20. In all, we manually checked 800, 307, and 1,244 alarms reported by jTrans, SaTC, and FirmRec respectively.

To compare the effectiveness of the three tools, we measure both their precision and recall. To measure the precision, we followed the same manual investigation methodology in RQ1 to verify true and false alarms. For SaTC and FirmRec, since the alarms were randomly sampled for verification, the precision on these samples should be, probabilistically, close to their real effectiveness. Measuring the recall of a vulnerability detector is hard, which requires manually finding all vulnerabilities in the selected firmware images. Instead, we used the true alarms that have been checked as the ground truth. By combining all the true alarms reported by three tools, we collected 764 unique vulnerabilities and measured the recall of each tool on this ground truth. We calculated recall on all alarms instead of the sampled ones to eliminate potential biases caused by random sampling.

Table 3 presents the measured results of the three tools. We found that FirmRec achieved the best precision of 51.6%, while the precision of jTrans, SaTC is only 13.5% and 22.8%, respectively. The results demonstrate the advantages of using known vulnerabilities to detect unknown ones. FirmRec also achieved the highest recall of 84.4%, while SaTC and jTrans only discovered 9.7% and 10.3% vulnerabilities accordingly. We also found that the vulnerabilities discovered by different tools seldom overlap. For example, only 23 vulnerabilities have been detected by both FirmRec and jTrans, and similarly, only 10 vulnerabilities have been detected by both FirmRec and SaTC. This clearly shows that FirmRec is an important complementary tool to existing vulnerability detectors.

We further investigated the causes of false positives (FPs) and false negatives (FNs) for different tools. For jTrans, we analyzed the 40 FPs with the highest rank for each known vulnerability signature and randomly sampled 40 FNs. For SaTC and FirmRec,

we randomly sampled 40 FPs and 40 FNs. By analyzing these cases, we find several key factors of the inaccuracy for the three tools.

Shared Inaccuracy Factors. We have identified the following three factors shared by different tools:

- *Limitations of Disassembling Tools.* jTrans, SaTC, and FirmRec rely on binary code disassembling tools for function identification or control-flow graph (CFG) construction. Thus, imperfections in these tools have lead to 2, 15, and 8 FNs respectively. SaTC is more prone to such FNs due to its dependency on precise CFG construction, which is used to optimize the model-based exploration. Unlike SaTC, jTrans and FirmRec mainly leverage known vulnerability signatures for vulnerability detection, which has less dependency on CFG analysis.
- *Imprecise Input Entry Identification.* Both SaTC and FirmRec identify input reading locations as the entries for vulnerability detection. However, it is hard to define a precise model for input entries. As a result, both tools have incorrectly identified some entries that are actually not attacker-controllable, causing 33 and 32 FPs respectively. Although precisely identifying attacker-controllable input entries is an important and common requirement for firmware vulnerability detection, it is not the research problem of this paper.
- *Limitations of Symbolic Execution.* The limitations of symbolic execution affect the taint analysis of SaTC and concolic execution of FirmRec. First, the symbolic execution engine of both tools—Angr [55] typically replaces common library functions with simplified versions to avoid path explosion. However, this optimization often neglects security checks in the original functions, leading to 7 FPs for SaTC, and 8 for FirmRec. Second, the symbolic execution involves path-sensitive analysis, which may exceed the time budgets in some cases, leading to 4 and 1 FNs for SaTC and FirmRec respectively.

Unique Inaccuracy Factors of jTrans. To understand the FPs of jTrans, we carefully compared the identified vulnerable functions with the given vulnerable functions. Interestingly, we found that only 7 pairs have a similar CFG, while the remaining 33 pairs looked significantly different. After a deep code investigation of the 7 pairs, we found they all have different code logic. Thus, all the FPs in jTrans result from its inaccuracy in detecting code clones. For the 38 FNs unmentioned before, we found their vulnerable functions have different code logic from the given vulnerable functions (e.g., implementing different functionalities), so they have not been detected as clones by jTrans. This finding shows the necessity of semantic-based vulnerability detection.

Unique Inaccuracy Factors of SaTC. Except for FPs/FNs caused by the shared factors, SaTC has two unique inaccuracy factors, which cause 21 FNs. First, the methodology of SaTC failed to cover the input entries for 12 cases because it relies on variable names shared between front-end files and back-end binary code, which these cases do not mention the same/similar variable names in front-end files. Second, SaTC met 9 FNs due to the flaws of taint analysis. Specifically, SaTC failed to discover 7 vulnerabilities because its aggressive function call pruning strategy created infeasible paths/data-flow conditions, which finally affected the vulnerability identification. For the other 2 cases, SaTC failed to recognize essential loops (e.g., memcpy-like sinks), so the variables processed by these loops were not properly tainted (i.e., under-tainted).

In contrast, the following three designs featured by FirmRec help to discover all these FNs. First, FirmRec analyzed the semantics of exploitation entries to identify similar candidate input entries, which pinpointed the 12 recurring vulnerabilities entries overlooked by SaTC. Second, FirmRec leveraged known vulnerability signatures to keep vulnerability-relevant functions for the 7 vulnerabilities, which were aggressively pruned by SaTC. Third, with concrete inputs from vulnerability signatures, FirmRec efficiently explored all potential execution paths, which prevented the undertainting issue and discovered the 2 vulnerabilities missed by SaTC.

Unique Inaccuracy Factors of FirmRec. Except for the FPs/FNs mentioned in the shared factors, FirmRec failed to discover 31 FNs due to the lack of corresponding vulnerability signatures. Specifically, the exploitation entries of the missed vulnerabilities were dissimilar to any exploitation entry in our generated vulnerability signatures. Thus, FirmRec misses the detection of these vulnerabilities. By using more vulnerabilities for signature generation, FirmRec can be extended to discover these vulnerabilities.

4.3.2 *Effectiveness Comparison with Greenhouse.*

Since Greenhouse requires a long emulation period to repeatedly run and patch the binary and the fuzzing period is also time-consuming (i.e., 24 hours per binary), it is infeasible to run it on the whole firmware dataset (which has 2,111 unique binaries). Considering web servers are popular exploitation targets [29], we chose to evaluate Greenhouse on the web server binary for each firmware image. In all, only 33 web servers have been successfully tested by Greenhouse, while the left ones failed due to emulation/rehosting failures (242 cases) or fuzzing errors (45 cases).

During the testing of 33 web servers, Greenhouse reported 381 unique crashes. After manual analysis, we found 209 crashes represent true vulnerabilities (precision: 54.9%) while the other crashes are caused by incorrect/improper emulation. As a comparison, FirmRec reported 958 alarms on the same set of binaries. Among these alarms, 226 ones (23.6%) have been verified in RQ1 and 143 ones were true alarms (precision: 63.3%). Among the 209 crashes reported by Greenhouse, we found that they were caused by 24 unique input entries. Thus, Greenhouse only discovered 24 vulnerabilities. In contrast, although only 23.6% alarms reported by FirmRec have been checked, FirmRec has discovered 143 vulnerabilities. These results show that FirmRec is more effective than Greenhouse in firmware vulnerability detection.

4.3.3 *Efficiency Comparison.*

We only compared the efficiency of FirmRec with SaTC, because jTrans mainly ran on GPU cores, and Greenhouse's running time was set to a fixed fuzzing budget. Table 4 presents the efficiency comparison results. In the head-to-head assessment, SaTC took 21,743,233 CPU seconds to finish the detection while FirmRec spent 11,259,535 CPU seconds. This means FirmRec only required nearly half as much time as SaTC. We also compared the time cost of SaTC and FirmRec per input entry. In all, SaTC and FirmRec analyzed 53,936 and 145,601 input entries, respectively. On average, SaTC needed 403 CPU seconds to analyze an input entry, while FirmRec only cost 77 CPU seconds. It shows FirmRec was 4.2 times faster than SaTC. The better efficiency of FirmRec was mainly attributed to the guidance of known vulnerabilities.

Table 4: Efficiency comparison results between SaTC and FirmRec in seconds (RQ2).

Tool	Signature Extraction	Vulnerability Detection			
		Total	per Sig	per Bin	per Entry
SaTC	-	21,743,233	-	10,300	403
FirmRec	18,428	11,259,535	281,488	5,334	77

4.4 RQ3: Ablation Experiments

We have several important designs in FirmRec to guarantee good signature extraction and vulnerability detection accuracy and efficiency. In this experiment, we measured the contributions of these designs through ablation experiments. To this end, we implemented three variants of FirmRec.

- FirmRec *w/o Adaptive*: This variant disables the adaptive function call pruning strategy used in signature generation (§3.2.2). Instead, it uses the same strategy as SaTC [19] and Karonte [51], which aggressive prunes function calls during concolic execution.
- FirmRec *w/o SimSearch*: This variant disables similarity-based input entry searching when locating vulnerability candidates (§3.3.1). Instead, it uses an exact match to locate the input entries, similar to §3.2.2.
- FirmRec *w/o Guidance*: This variant does not use the guided function call pruning strategy when verifying vulnerabilities (§3.3.2). Instead, it employs the adaptive function call pruning strategy in signature generation.

Table 5: Ablation experiment on signature extraction.

Settings	# Vul. Reports	# Sigs.	Total Time(s)	Time/Sig(s)
FirmRec	40	40	50,920	73
FirmRec *w/o Adaptive*	40	36	62,920	66

Signature Extraction. We compared FirmRec with FirmRec *w/o Adaptive* to measure the contribution of our adaptive function call pruning strategy used in the signature generation. As shown in Table 5, FirmRec *w/o Adaptive* failed to generate signatures for 4 vulnerabilities. Besides, due to such failures, FirmRec *w/o Adaptive* had to analyze more input entries, costing more analysis time than FirmRec, even though FirmRec *w/o Adaptive* was more lightweight in analyzing a single input entry. The results show that the adaptive function call pruning strategy helps to generate more vulnerability signatures without introducing additional time costs.

Vulnerability Detection. We compared FirmRec with FirmRec *w/o SimSearch* and FirmRec *w/o Guidance* to understand the contributions of the similarity-based (LLM-based) input entry searching strategy when locating vulnerability candidates (Stage 1) and the guided function call pruning strategy when verifying vulnerabilities (Stage 2). We run the experiments on the 320 images used in RQ2. Table 6 presents the results. First, compared with FirmRec *w/o SimSearch*, FirmRec has found 18.6 times more vulnerability candidates in the first stage, so it took more analysis time. Meanwhile, FirmRec reported 4,519 more alarms than FirmRec *w/o SimSearch*. The result shows that similarity-based input entry searching significantly helps to detect more vulnerabilities. Second, FirmRec and FirmRec *w/o Guidance* report similar numbers of alarms while FirmRec *w/o Guidance* took 74.5% more analysis time. The results show that the guided function call pruning strategy significantly improves the efficiency of vulnerability detection.

Table 6: Ablation experiment on vulnerability detection.

Settings	# Candidates	# Alarms	Total Time(s)
FirmRec	145,601	4,970	10,356,022
FirmRec *w/o SimSearch*	7,430	451	529,491
FirmRec *w/o Guidance*	145,601	4,923	18,071,646

5 DISCUSSION

Requirements on Vulnerability Reports. For signature generation, FirmRec relies on extracting specific information from vulnerability reports, including the methods used to exploit vulnerabilities (e.g., exploit inputs) and the potential security impacts. Our study, detailed in Appendix A, reveals that the vast majority (over 94.9%) of firmware vulnerability reports provide this critical information. Note that, FirmRec can effectively generate vulnerability signatures even with partial exploit inputs, as long as they are sufficient to demonstrate the basic exploiting method.

Furthermore, our findings suggest that FirmRec does not depend on an extensive corpus of vulnerability reports to detect recurring vulnerabilities. Utilizing as few as 40 reports, FirmRec significantly complemented existing approaches, contributing to uncovering at least 566 more vulnerabilities and 53 new CVEs.

Other Types of Firmware Vulnerabilities. The current prototype of FirmRec supports four types of firmware vulnerabilities because these are severe and primary vulnerability types and are also supported by existing firmware detectors (e.g., SaTC). Actually, the approach is not specific to these vulnerability types. To support new types, we just need to design corresponding sanitizers for them, e.g., Cross-Site Scripting (XSS). Yet, it is hard to support some logic errors (e.g., cryptography misuse) due to the difficulty of checking corresponding security impacts during symbolic execution.

False Positives and Countermeasures. As described in §4.3, FirmRec's FPs were caused by two kinds of general firmware analysis limitations—imprecise input entry identification and imperfect static analysis tools. The former can be mitigated by leveraging additional firmware knowledge, e.g., binary dependencies [51]. The latter can be mitigated by improving the implementation of static analysis tools. Another countermeasure is to automatically reproduce the vulnerability alarms using directed grey-box fuzzing (DGF) [34, 64], e.g., by specifying the alarm sites as fuzzing targets. However, due to emulation roadblocks [17, 57], DGF is immature in the firmware area and thus has not been applied to FirmRec.

6 RELATED WORKS

Source Code-based Recurring Vulnerability Detection. Previous studies such as MVP [60] and TRACER [39] have proposed recurring vulnerability detection in the source code area. MVP relies on security patches to extract enhanced code signatures for vulnerability detection. TRACER leverages known vulnerable traces to verify whether an alarm discovered by traditional vulnerability detectors is indeed a recurrence of the known vulnerability.

Code Clone-based Vulnerability Detection. Code clone detection techniques are extensively studied in both the source code and binary area and can be adapted to discover vulnerability clones. Many code clone detection approaches [35, 38, 40, 41, 52, 53], by design, only detect code clones in source code. Binary code clone detection approaches can deal with binary code. Their main focus is

to design code representations robust to non-semantic changes introduced by compilations, architectures, etc. For this purpose, these approaches either leverage domain knowledge [16, 23, 26, 27, 31, 43, 45, 47, 48, 67] or learning-based approaches [25, 28, 32, 44, 46, 59, 62] to extract code representations. Instead of using such general code syntactic features, FirmRec captures dynamic semantic features that are more robust to syntactic code changes. Patch presence testing [36, 61, 63] is a technique that further identifies whether a code clone is a patched version, but such a technique can not work in close-sourced IoT firmware due to lack of security patches.

Dynamic Analysis for Firmware Vulnerability Detection. Dynamic analysis, by capturing actual program states and outputs, automates the identification of vulnerabilities. Black-box fuzzing [30], despite its direct application to embedded devices, is limited by the devices' constrained computing capabilities [68]. Grey-box fuzzing, which collects program execution feedback through instrumentation, encounters challenges with device instrumentation. To circumvent this, research often resorts to re-hosting program execution in simulated environments [22, 24, 37, 57, 68, 69], a solution hindered by the complexity of modeling device dependencies. White-box fuzzing, exemplified by SFuzz [18], employs symbolic execution to mitigate some emulation challenges but, like all dynamic analysis methods, struggles with achieving comprehensive code coverage.

Static Analysis for Firmware Vulnerability Detection. Static analysis examines code without executing it, playing a crucial role in identifying vulnerabilities within firmware [19–21, 51, 54, 65]. Firmalice [54] and CINDY [65] target specific vulnerability classes, e.g., authentication bypass and command injection. Karonte [51] and SaTC [19] rely on input entries to detect a broader range of taint-style vulnerabilities. They differ in input exploitation entry identification. For example, Karonte uses manually-curated API lists while SaTC uses shared keywords to identify input reading functions. FirmRec significantly differs from them by identifying semantically similar input entries to known exploitation entries, which helps identify more input entries and vulnerabilities.

7 CONCLUSION

In this paper, we introduced FirmRec, a novel approach for detecting recurring vulnerabilities in IoT firmware, an area previously unexplored. Leveraging exploitation-based vulnerability signatures and signature-driven vulnerability detection, FirmRec effectively addresses the challenges posed by the unique characteristics of firmware vulnerability reports and the closed-source nature of firmware, enhancing both accuracy and efficiency in vulnerability detection. Our evaluations demonstrate FirmRec's superior performance over existing methods, identifying 642 vulnerabilities and contributing to 53 CVEs. As a pioneering effort in this field, FirmRec sets the foundation for future research and will be made publicly available to support ongoing advancements.

ACKNOWLEDGMENTS

We appreciate the anonymous reviewers for their valuable comments. This work was supported in part by National Natural Science Foundation of China (62172105,62172104,62102091, 62102093) and the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Yuan Zhang was supported in part by the Shanghai Rising-Star Program 210A1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

REFERENCES

[1] 2023. https://chat.openai.com.
[2] 2023. Current IoT Forecast Highlights - Transforma Insights. https://transformainsights.com/research/forecast/highlights.
[3] 2023. CVE-2019-20500. https://www.exploit-db.com/exploits/46841.
[4] 2023. exploit-db. https://www.exploit-db.com/.
[5] 2023. Ghidra. http://ghidra-sre.org.
[6] 2023. NVD. https://nvd.nist.gov/vuln.
[7] 2023. ReFirmLabs – Binwalk. https://github.com/ReFirmLabs/binwalk.
[8] 2023. State of XIoT Security: 1H 2022. https://claroty.com/resources/reports/state-of-xiot-security-1h-2022.
[9] 2023. State of XIoT Security: 2H 2022. https://claroty.com/resources/reports/state-of-xiot-security-2h-2022.
[10] 2023. The Mirai Botnet – Threats and Mitigations. https://www.cisecurity.org/insights/blog/the-mirai-botnet-threats-and-mitigations.
[11] 2023. TP-Link Fixes Code Execution Vulnerability in End-of-Life Routers. https://threatpost.com/tp-link-fixes-code-execution-vulnerability-in-end-of-life-routers/126416/.
[12] 2023. Vulnerable SDK components lead to supply chain risks in IoT and OT environments. https://www.microsoft.com/en-us/security/blog/2022/11/22/vulnerable-sdk-components-lead-to-supply-chain-risks-in-iot-and-ot-environments/.
[13] 2024. AFLplusplus. https://github.com/AFLplusplus/AFLplusplus.
[14] Manos Antonakakis, Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *USENIX Security'17*.
[15] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *ECOOP'16*.
[16] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary search. In *FSE/ESEC'16*.
[17] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS'16*.
[18] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Ming Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, and Zhi Xue. 2022. SFuzz: Slice-based Fuzzing for Real-Time Operating Systems. In *CCS'22*.
[19] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. In *USENIX Security'21*.
[20] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware. In *DSN'18*.
[21] Kai Cheng, Tao Liu, Le Guan, Peng Liu, Hong Li, Hongsong Zhu, and Limin Sun. 2021. Finding Taint-Style Vulnerabilities in Linux-based Embedded Firmware with SSE-based Alias Analysis. *ArXiv* abs/2109.12209 (2021).
[22] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David J. Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *USENIX Security'20*.
[23] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *ASPLOS'18*.
[24] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security'13*.
[25] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Oakland'19*.
[26] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *USENIX Security'14*.
[27] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS'16*.
[28] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *CCS'16*.

[29] Xuan Feng, Xiaojing Liao, X Wang, Haining Wang, Qiang Li, Kai Yang, Hongsong Zhu, and Limin Sun. [n. d.]. Understanding and Securing Device Vulnerabilities through Automated Bug Report analysis. In *USENIX Security'19*.
[30] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *CCS'21*.
[31] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *IPSN'08*.
[32] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In *ASE'18*.
[33] Harm J. Griffioen and Christian Doerr. 2020. Examining Mirai's Battle over the Internet of Things. In *CCS'20*.
[34] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan : Efficient Multi-target Directed Greybox Fuzzing. In *Oakland'24*. https://api.semanticscholar.org/CorpusID:268386913
[35] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Oakland'12*.
[36] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *CCS'20*.
[37] Evan Johnson, Maxwell Troy Bland, Yifei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *USENIX Security'21*.
[38] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. In *TSE'02*.
[39] Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities. In *CCS'22*.
[40] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Oakland'17*.
[41] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. In *TSE'06*.
[42] Z. Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *ArXiv* abs/1801.01681 (2018).
[43] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE/ESEC'14*.
[44] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *NDSS'23*.
[45] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Inscrypt'12*.
[46] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Sekhar Jana, and Baishakhi Ray. 2020. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. *ArXiv* abs/2012.08680 (2020).
[47] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Oakland'15*.
[48] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *ACSAC'14*.
[49] Davide Pizzolotto and Katsuro Inoue. 2021. Identifying Compiler and Optimization Level in Binary Code From Multiple Architectures. *IEEE Access* 9 (2021), 163461–163475. https://api.semanticscholar.org/CorpusID:244926433
[50] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX ATC'15*.
[51] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *Oakland'20*.
[52] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal Kumar Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *ICSE'16*.
[53] Yusuke Sasaki, Tetsuo Yamamoto, Yasuhiro Hayase, and Katsuro Inoue. 2010. Finding file clones in FreeBSD Ports Collection. In *MSR'10*.
[54] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS'15*.
[55] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Oakland'16*.
[56] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2018. SoK: Sanitizing for Security. In *Oakland'18*.
[57] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S. Raj, Audrey Annika Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, Adam Doupé, Tiffany Bao, Yan Shoshitaishvili, and Ruoyu Wang. 2023. Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation. In *USENIX Security Symposium*. https://api.semanticscholar.org/CorpusID:260777808
[58] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, Christophe Hauser, and Yan Shoshitaishvili. 2022. Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs. In *USENIX Security'22*.
[59] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: jump-aware transformer for binary code similarity detection. In *ISSTA'22*.
[60] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *USENIX Security'20*.
[61] Yang Xiao, Zhengzi Xu, Weiwei Zhang, Chendong Yu, Longquan Liu, Wei Zou, Zimu Yuan, Yang Liu, Aihua Piao, and Wei Huo. 2021. VIVA: Binary Level Vulnerability Identification via Partial Signature. In *SANER'21*.
[62] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *CCS'17*.
[63] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *ISSTA'20*.
[64] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. 1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing. In *ISSTA'23*. https://api.semanticscholar.org/CorpusID:259844811
[65] Xiaokang Yin, Ruijie Cai, Yizheng Zhang, Lukai Li, Qichao Yang, and Shengli Liu. 2022. Accelerating Command Injection Vulnerability Discovery in Embedded Firmware with Static Backtracking Analysis. In *IOT'22*.
[66] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS'14*.
[67] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, et al. 2022. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in IoT firmware. In *ISSTA'22*.
[68] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security'19*.
[69] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference (Extended Version). In *USENIX Security'21*.
[70] Yaqin Zhou, Shangqing Liu, J. Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *NeurIPS'19*.

A STUDY ON VULNERABILITY REPORTS

In this study, we want to answer one question: How many vulnerability reports contain exploitation information that is required for signature extraction? First, we followed the same way described in §4.2 to collect a dataset of public firmware vulnerability reports, but only ensure one condition: it is a true IoT firmware vulnerability. In all, 332 vulnerability reports have been collected. Second, we used the method described in §3.2.1 to extract exploit inputs and security impact descriptions. We manually examined each report to ensure the correctness of the outputs. The results show that 315 reports contain exploit inputs. For the remaining 17 vulnerability reports, they either use simple manual operations to trigger the vulnerabilities or just describe the vulnerability causes/exploitation results. Further, all the reports have described the security impact. In summary, we found that most (more than 94.9%) firmware vulnerability reports have described the exploit inputs and security impact that are required for signature extraction.

B MORE DISCUSSION ON EDGE CASE

In this section, we discuss edge cases that might occur during recurring vulnerability detection. Even though we have not met false positives/negatives caused by these cases in the evaluation,

these cases could exist in other datasets and might affect recurring vulnerability detection in firmware.

Vulnerabilities in Early Input Processing Logic. FirmRec leverages constants that describe the exploit input semantics to represent the exploitation entry for detecting recurring vulnerabilities. However, there might be exceptional cases where the exploitation procedure does not involve too many constants-related input reading/validation operations. This typically happens when the vulnerability is triggered early in the input processing logic (where the input has not been assigned with a semantic context). In these cases, identifying recurring vulnerabilities would become more challenging. However, since the input processing logic of firmware is rather simple, most firmware vulnerabilities are triggered after input processing. As demonstrated in our experiments, few false negatives are caused for this reason (see §4.3).

Vulnerability Exploited after Setup. Many IoT devices provide a setup stage for user configuration before providing main services, e.g., a router setup wizard. Therefore, traditional dynamic analysis-based approaches must pass the setup stage to detect vulnerabilities in the main services. However, with the help of under-constrained symbolic execution (UCSE) technique [50, 55], FirmRec can detect the vulnerabilities in main services without such requirements. Technically, UCSE directly starts the execution at input reading locations in the main services and explores the subsequent code to detect vulnerabilities. During the code exploration, UCSE uses abstract symbols to represent uninitialized data variables, so it does not necessarily rely on the setup stage to initialize these variables.

Vulnerabilities Exploited with Nested/Encoded Inputs. IoT firmware may take nested/encoded inputs, e.g., JSON data nested in a XML file, BASE64 encoded data.

When analyzing vulnerability reports containing such nested/encoded inputs, FirmRec may fail to extract vulnerability signatures. This is because the current prototype does not implement recursive data structure parsing with different data formats, and does not attempt to recognize encoded data for decoding. So, it may fail to recognize the constants used to identify exploitation entry from the vulnerability report.

That said, FirmRec could still detect vulnerabilities that need nested/encoded inputs to exploit. This is because nested inputs are usually read at data element level and encoded inputs are often used after decoding. Thus, FirmRec could identify the input entry (at the level of data element or decoded data) for such vulnerabilities. If these input entries have similar exploitation entries to some known vulnerabilities, FirmRec can use the signatures extracted from these known vulnerabilities to detect vulnerabilities that need nested/encoded inputs.

Vulnerabilities Exploited with Different Inputs. When a vulnerability is very similar to a known vulnerability (e.g., a plain file upload vulnerability vs. an image file upload vulnerability), but requires significantly different inputs to exploit (e.g., a malformed plain file vs. a malformed image), FirmRec may fails to detect such vulnerability. This is because FirmRec is designed to reuse the exploit input data of known vulnerabilities to detect unknown recurring ones, but the new vulnerability requires new input data to exploit.