1DRep:Automatic Repair for 1-day Vulnerabilities in Reused C/C++ IoT Open-source Software Components

Weiting Cai

1DRep:Automatic Repair for 1-day Vulnerabilities in Reused C/C++ IoT Open-source Software Components

by

Weiting Cai

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
to be defended publicly on Tuesday November 5, 2024.

Student number:	5678269	
Project duration:	October 18, 2023 – November 5, 2024	
Thesis committee:	Prof. dr. Arie van Deursen,	TU Delft, Thesis advisor
	Prof. dr. Alex Voulimeneas,	TU Delft, Committee member
External member:	Prof. dr. Siqi Ma,	UNSW, External supervisor

TU Delft

Preface

Writing this thesis has been a long and challenging journey, but also a rewarding one. The widespread adoption of open-source software (OSS) in Internet of Things (IoT) projects, while accelerating software development, has exposed the systems to 1-day vulnerabilities. This motivated me to explore how to automatically repair such vulnerabilities that exist when C/C++ IoT projects reused OSS components.

I would like to express my deepest gratitude to my supervisors Prof. Arie van Deursen, Prof. Thomas Durieux and Prof. Siqi Ma, for their invaluable guidance and support throughout the course of my research. This thesis would not have been possible without them.

Although Prof. Thomas Durieux was my daily supervisor, he resigned from TU Delft on September 14th. As a result, the position of daily supervisor is not included in the Thesis Committee.

Therefore, I would like to extend my special thanks to Prof. Arie van Deursen, who kindly assumed responsibility for the final review, offering valuable feedback and numerous insights on the entire thesis, which have been both inspiring and instrumental in shaping this work.

Additionally, Prof. Siqi Ma provided guidance by proposing the thesis topic, helping me understand the vulnerabilities and repair process. She also facilitated the involvement of her PhD student, Shangzhi Xu, who assisted in building the IoT-specific dataset.

I extend my gratitude to Prof. Alex Voulimeneas for being my thesis committee members and attending the defense.

I also owe a huge thanks to my academic counsellor, Michel Rodrigues, who has been there to help me with various issues during my master's study. Additionally, I am grateful to my friends who have offered their encouragement and advice along the way.

Finally, the most significant acknowledgement is reserved for my family, whose love, unwavering support and patience have kept me grounded, especially during the most challenging moments of this journey.

Weiting Cai
Delft, November 2024

Abstract

The rapid proliferation of the Internet of Things (IoT) has introduced significant security challenges, primarily due to the widespread reuse of open-source software (OSS) components. This practice leaves IoT projects particularly vulnerable to 1-day vulnerabilities especially when developers customize the reused OSS code, rendering template patches inapplicable.

In this thesis, we propose **1DRep**, a repair tool designed to automatically detect and repair 1-day vulnerabilities in reused C/C++ IoT OSS components. First, **1DRep** integrates with the vulnerability detection tool V1SCAN to identify vulnerable code snippets in target programs. It then employs a large language model (LLM), GPT-4o, to generate and apply tailored fixes, addressing both exactly reused and modified vulnerable code without altering developer-customized functionality.

Our evaluation demonstrates that **1DRep** effectively repaired 39 out of 40 CVEs (97.5%) in 11 target vulnerable IoT projects, including 14 out of 15 modified CVEs (93.3%), and effectively fixed 81 out of 90 artificially created vulnerable reuses (90%). We constructed an IoT-specific dataset containing 1,020 C/C++ libraries which supplements an exisiting dataset Centris to enhance the detection of OSS components commonly reused in IoT projects. Additionally, we provided security reports containing customized patches for the modified CVEs by creating GitHub issues or pull requests in the affected projects.

The results indicate that **1DRep** is a promising tool for automatically repairing 1-day vulnerabilities in IoT projects, particularly in scenarios where developers' customized reuses make traditional patching techniques ineffective.

Lastly, despite the promising outcomes, limitations such as reliance on the precision of the detection model and challenges with complex vulnerabilities highlight areas for future research. Enhancing the detection mechanisms, expanding the CVE dataset, and refining repair strategies are critical steps toward improving the tool's effectiveness.

Contents

1 Introduction

1.1. Motivation and Implications

The Internet of Things (IoT) is rapidly growing, with a forecasted compound annual growth rate of 17% until 2030 [17]. It connects millions of heterogeneous devices to provide advanced intelligent services, which impacted our daily lives profoundly [28].

However, this widespread adoption has also introduced significant security challenges: the design of IoT systems often prioritizes connectivity over security, creating a blind spot that has made IoT devices lucrative and easy targets for attackers [28]. Since 2021, cybersecurity has been recognized as the top enterprise technology priority globally for companies adopting IoT software, according to recent research from IoT Analytics [18]. It was also found by IoT Analytics that in terms of end-user adoption of enterprise IoT initiatives, 84% of businesses who custom-built their IoT solutions did so because they believed that approach would better handle IT security concerns. Furthermore, they showed that of the companies who adopted off-the-shelf IoT solutions and used a systems integrator (comprising 42% of those surveyed), 100% shared that a solid IT security track record or reputation was either very important or important. This heightened focus on cybersecurity reflects the growing awareness of the risks associated with connected IoT devices, as organizations across all regions and industries prioritize securing their digital assets.

One of the most pressing issues in IoT security is the widespread use of reusable open-source software (OSS) components [28], which are often integrated into IoT devices without thorough vetting or security testing. The reuses are prevalent because reusing OSS can significantly accelerate development processes [34, 37, 45, 5]. The 2023 Open Source Security and Risk Analysis (OSSRA) report [29] revealed that 96% of 1,703 codebases across 17 industries such as IoT, Energy, Retail and eCommerce and Clean Tech, contained OSS components, illustrating the pervasive nature of reusing OSS components in modern software development.

The reliance on reusing OSS components, coupled with inadequate patch management, leaves IoT devices particularly susceptible to 1-day (or N-day) vulnerabilities [13, 14, 36, 45, 41, 28]. A 1-day vulnerability refers to a known vulnerability for which a patch is available [36, 5] but has not yet been applied. The threat is further supported by Zhao et al.'s empirical study on 1,362,906 IoT devices found that 385,060 (28.25%) were affected by at least one N-day vulnerability [48].

Even tech giants such as Apache, Oracle, VMware, and Microsoft suffered from known vulnerabilities such as Common Vulnerabilities and Exposures (CVEs), which malicious cyber actors routinely and frequently exploit, showed by the report 2022 Top Routinely Exploited Vulnerabilities [4]. Google warned that 1-day vulnerabilities are still as dangerous as 0-days, or even more so, since attackers can exploit the publicly available technical details, considering that there is often a lengthy delay before device manufacturers deploy this fix in their respective versions [30]. For example, the CVE-2022-22706 flaw in the ARM Mali GPU, patched by the vendor in January 2022, was exploited in December 2022 infecting Samsung Android devices with spyware, but it was only addressed by Google and Samsung in May and June 2023, respectively, resulting in a 17-month delay [30].

In view of the above circumstances, it is clear that, even though patches are available, effectively repairing the 1-day vulnerabilities is still a significant problem for developers due to the identified issues listed below:

1. **Increasing prevalence of the 1-day vulnerabilities across various industries:** the 2023 OSSRA report [29] found that 84% of examined codebases contained at least one known vulnerability and 48% of them contained high-risk 1-day vulnerabilities. In addition, it also showed that since 2019, among the 17 industries as mentioned above, the Retail and eCommerce sectors have experienced a 557% increase in high-risk vulnerabilities and the IoT vertical has seen a 130% rise in such vulnerabilities, with 53% of examined applications in this sector containing high-risk issues this year.
2. **Lack cybersecurity skills:** Since most developers are not skilled in cybersecurity [20, 19, 46, 22], it is error-prone and time-consuming [1, 50] for them to identify and repair a vulnerability manually.
3. **Customizations on reused code:** Developers often customize reused OSS code to fit the specific needs of their programs [36, 34, 5]. As a result, simply updating the reused OSS components to the latest version may not be feasible, as it could lead to the loss of their customizations or cause context mismatches.

In-view of the above issues, automatically detecting and repairing 1-day vulnerabilities in reused IoT OSS components can significantly help developers and users.

1.2. Research Gap and Problem Statement

To address the problem of 1-day vulnerabilities in IoT systems, in this thesis we will explore the use of Automated Program Repair (APR) techniques. Various traditional APR approaches exist [9, 24] and can be categorized as search-based, template-based, constraint-based and learning-based tools. Among these techniques, template-based tools, which was regarded as state-of-the-art, use handcrafted or mined repair templates to fix vulnerable code. However, they cannot generalize to patterns and bugs not included in the predefined templates due to the lack of continuous learning capability [38, 9]. On the other hand, they often overlook defect classes that are complex, specialized, or not widely applicable [9].

To address the limitations of traditional template-based APR methods, which lack the capacity for continuous learning and adaptation, learning-based approaches utilizing Large Language Models (LLMs) or Neural Machine Translation (NMT) have been proposed [3, 16, 11, 38]. However, NMT-based APR tools may suffer from noisy training datasets [12] and may not generalize to bug fix types not represented in their training data [38]. This limitation arises because these tools rely heavily on training data built by searching open-source repositories for commits that fix bugs.

Recently, LLM-based methods have gained favor for APR, as they can automatically learn features from known vulnerabilities, demonstrating excellent outcomes in the vulnerability repair task [38, 49]. AlphaRepair [39], which can be considered as the current state-of-the-art, directly predicts correct code based on the context information around the vulnerable code, proposing the first cloze-style (or infilling-style) APR using an LLM.

However, a significant problem with these approaches is that they focus only on the exact vulnerable lines or functions reported by vulnerability detection tools, ignoring reused code that has been modified by developers to satisfy their customization needs, which might still be vulnerable [6].

Our repair tool, **1DRep**, is designed to automatically generate and apply tailored fixes for vulnerable reuses detected in the target program without altering developer-customized code. To achieve this, we propose developing and implementing an automated tool to identify and repair 1-day vulnerabilities caused not only by exact vulnerable code but also by modified vulnerable code—that is, code reused and customized by developers but still vulnerable.

The idea is to deploy a 1-day vulnerability detection tool on a target IoT project to identify vulnerable code snippets that have already existed in other OSS components. We then launch the LLM-based repair module to automatically generate correct code for developers based on our handcrafted instructions combined with information extracted from template security patches in the CVE dataset provided by the chosen vulnerability detection tool, V1SCAN [36]. V1SCAN is a state-of-the-art tool designed to detect 1-day vulnerabilities in reused C/C++ OSS components, classifying vulnerable functions into exactly reused, modified, and unused categories, as explained further in Chapter 2.

To automatically repair the modified vulnerable functions in the target file detected by V1SCAN, we identify two main challenges (Cs):

- **Identifying Correct Insertion Points (C1)**: Determining the correct locations to insert the patch lines within the vulnerable function in the target file, especially when the context has been changed due to customizations.
- **Adapting Patches to Changed Context (C2)**: Transforming the template patches into correct ones that fit the changed context in the modified vulnerable function, ensuring that the fix does not disrupt the customized functionality.

Addressing these challenges is crucial for the effectiveness of a APR tool. We designed a methodology to deal with them in section 4.4.1. In chapter 5, we provide an analysis of the solutions implemented to overcome these challenges and evaluates their effectiveness in real-world scenarios.

1.3. Contribution

In this thesis, we have the following research contributions:

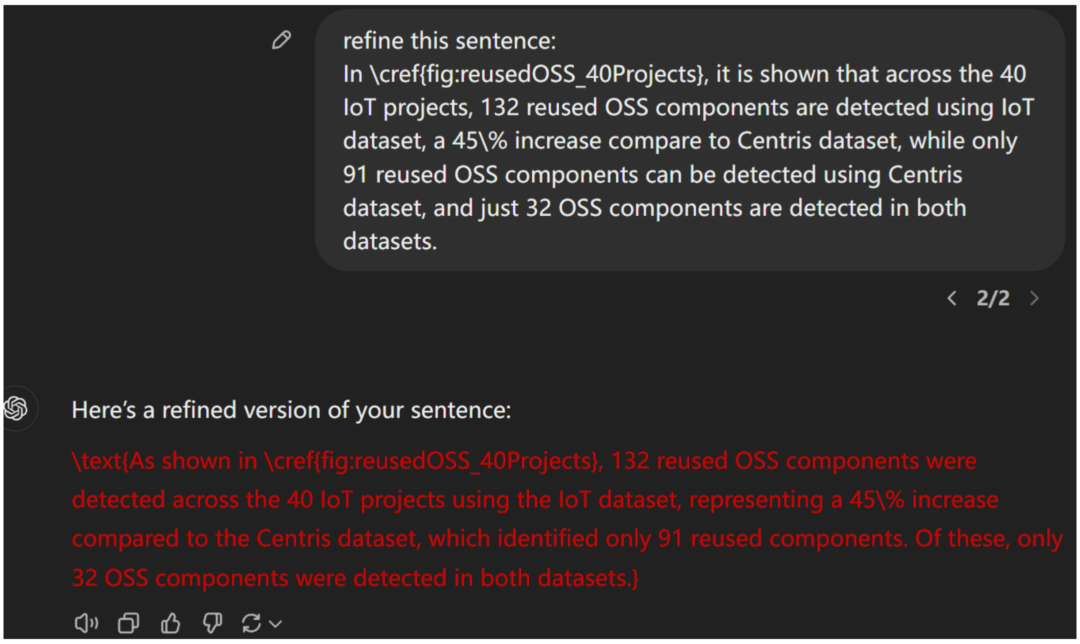
- We developed a tool, **1DRep**, to automatically repair 1-day vulnerabilities arising from the reuse of C/C++ IoT OSS components. These vulnerabilities were detected using the vulnerability detection tool V1SCAN [36]. The details are illustrated in chapter 4.
- We constructed a dataset, **IoT-1000**, containing 1,020 IOT-specific libraries for identifying more OSS components that might be reused in open-source C/C++ IOT projects. This dataset was built with the team from University of New South Wales(UNSW) as detailed in section 4.6. Its effectiveness is evaluated in section 5.1.2.
- In section 5.2, we listed the 40 IoT C/C++ projects on GitHub that were scanned to identify 1-day vulnerabilities in real-world IoT applications. In addition, we investigated the complexity of the 15 modified CVEs in section 5.2.1.
- In section 5.3, we evaluated the effectiveness of V1SCAN based on 40 detected CVEs from the target IoT projects.
- Targeting 40 IoT projects, the repair tool **1DRep** successfully repaired 39 out of 40 detected CVEs (97.5%), including 14 out of 15 modified CVEs (93.3%) as detailed in section 5.4.
- **1DRep** effectively repaired 81 out of 90 artificially created vulnerable reuses (90%). These reuses were derived from 10 selected CVEs and were modified in 9 levels of complexity as shown in section 5.5.2.
- We provided security reports with patches for the 15 modified CVEs in 5 IOT projects on GitHub by creating pull requests or opening issues with necessary patch information as illustrated in section 5.6.

1.4. Research Objectives

To validate the effectiveness of our proposed tool, **1DRep**, we conducted a evaluation focusing on both the vulnerability detection phase using V1SCAN and the automated repair process. Our evaluation addressed the following key research questions:

- **RQ1. Vulnerability Detection Performance**: In section 5.3, we assessed how well V1SCAN detects and classifies vulnerabilities in 40 target IoT projects.
- **RQ2. Repair Effectiveness**: We evaluated the effectiveness of our repair tool in addressing the exact and modified vulnerable reuses respectively in section 5.4.
- **RQ3. Repair Capability**: We explored the tool's performance on both detected vulnerabilities and hand-crafted artificial vulnerabilities in section 5.5.

The evaluation highlights the strengths of **1DRep** in repairing 1-day vulnerabilities, particularly in scenarios where code has been modified by developers. It also points to areas for future improvement, such as enhancing vulnerability detection for modified vulnerable code.


refine this sentence:
In \cref{fig:reusedOSS_40Projects}, it is shown that across the 40 IoT projects, 132 reused OSS components are detected using IoT dataset, a 45\% increase compare to Centris dataset, while only 91 reused OSS components can be detected using Centris dataset, and just 32 OSS components are detected in both datasets.

2/2

Here's a refined version of your sentence:

\text{As shown in \cref{fig:reusedOSS_40Projects}, 132 reused OSS components were detected across the 40 IoT projects using the IoT dataset, representing a 45\% increase compared to the Centris dataset, which identified only 91 reused components. Of these, only 32 OSS components were detected in both datasets.}

Figure 1.1: Refine Writing with GPT

1.5. Structure

The remainder of this thesis is organized as follows:

- **chapter 2: Background.** This chapter provides a comprehensive overview of the fundamental concepts and context necessary to understand the research presented in this thesis. It covers topics such as 1-day vulnerabilities and an introduction to the vulnerability detection tool V1SCAN.
- **chapter 3: Related Works.** In this chapter, we delve into existing research and developments in the field of automated vulnerability detection and repair.
- **chapter 4: 1DRep: The APR Tool.** This chapter illustrates the design details of the APR tool 1DRep.
- **chapter 5: Evaluation.** This chapter describes the methodology used to evaluate the performance and effectiveness of both V1SCAN and our repair tool, 1DRep.
- **chapter 6: Discussion.** In this chapter, we discuss the implications of our findings and address the challenges encountered during the development and evaluation of our repair tool.
- **chapter 7: Threats to Validity.** This chapter identifies possible factors or issues that may have influenced the results of this research.
- **chapter 8: Conclusion.** The final chapter summarizes the key findings and contributions of the thesis.

1.6. Writing process

We employed GPT-4o as a tool to enhance the clarity and coherence of our thesis writing. Specifically, we initially drafted the sentences ourselves and then utilized GPT-4o to suggest improvements. After receiving the suggestions, we carefully reviewed and selectively applied them to ensure they aligned with the intended meaning. An example of this interaction is illustrated in figure 1.1.

2 Background

This chapter provides background information on the detection and repair of 1-day vulnerabilities in reused OSS components. It begins by defining key terminology in section 2.1, followed by an introduction on the reused code classification. The chapter then introduces V1SCAN in section 2.2, a state-of-the-art tool for vulnerability detection, and illustrates its functionality through a motivating example in section 2.3.

2.1. Terminology

To ensure clarity and consistency throughout this report, the following terms are defined:

- **OSS reuse:** The practice of reusing all or part of OSS functions, for example, by copying and pasting code from third-party OSS projects [36, 34, 5].
- **OSS component:** An entire OSS package or specific functions within the OSS project that are reused in a target program [34, 36].
- **1-day (or N-day) vulnerability:** A vulnerability that is known and has a patch available, but the patch has not yet been applied [36, 5]. Due to OSS reuse, these vulnerabilities can exist in other software, potentially compromising the security of the systems.
- **Common Vulnerabilities and Exposures (CVEs):** A standardized, unique identifier assigned to known security vulnerabilities. Managed by the MITRE Corporation with authority delegated to CVE Numbering Authorities (CNAs), CVE provides a reference-method for known information-security vulnerabilities and exposures, facilitating easier sharing and management of vulnerability among security professionals [10].

2.1.1. Reused code classification

As mentioned before, developers can reuse OSS with code or structural modifications [36, 34, 5]. This might be a possible reason why recent studies [47, 42, 43] have shown that developers are prone to postponing the updating of outdated OSS components in applications. Such delays occur even when these OSS components contain critical vulnerabilities that could pose significant risks to mobile devices or users.

Developers may be disinclined to update their libraries for two reasons:

1. Newer versions of libraries, including updates for security patches, may introduce breaking changes.
2. The update might block the customized functionality of the adjusted library.

Therefore, we follow the same reused code classification approach defined in V1SCAN [36], as explained below in section 2.2, in order to provide developers more information of what changes will the repair bring to help them make decisions:

- **Exactly reused:** OSS code, for instance functions or entire files, is reused without any code changes.
- **Modified:** OSS code is reused with code modifications.
- **Unused:** OSS code is not reused at all.

2.1.2. Vulnerable code classification

Also, V1SCAN classifies the code locations where vulnerabilities exist and considered four code locations: (1) structure, (2) macro, (3) variable, and (4) function. Our tool focuses on repairing vulnerabilities within functions. This is because we assume modifications are more likely in reused functions, and such modifications are harder to repair as they require considering more contextual information. Furthermore, if we can repair such vulnerabilities, we believe that **1DRep** can be easily extended to deal with vulnerabilities in the other three locations.

The code classification facilitates the identification of the nature of reused code, thereby informing appropriate strategies for vulnerability repair.

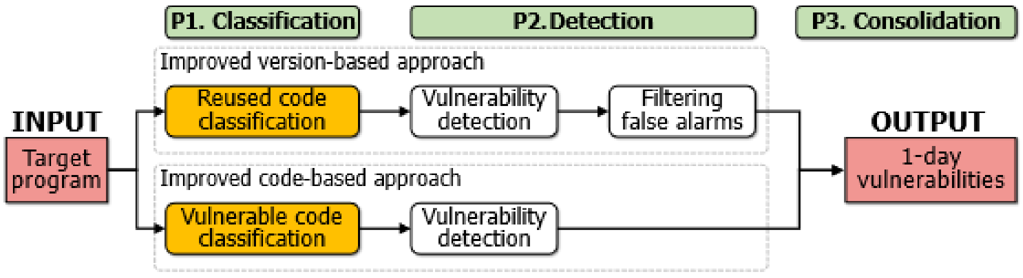


Figure 2: High-level overview of V1SCAN.

Figure 2.1: High-level overview of V1SCAN (The same figure copied from V1SCAN [36])

2.2. 1-day Vulnerability Detection Tool: V1SCAN

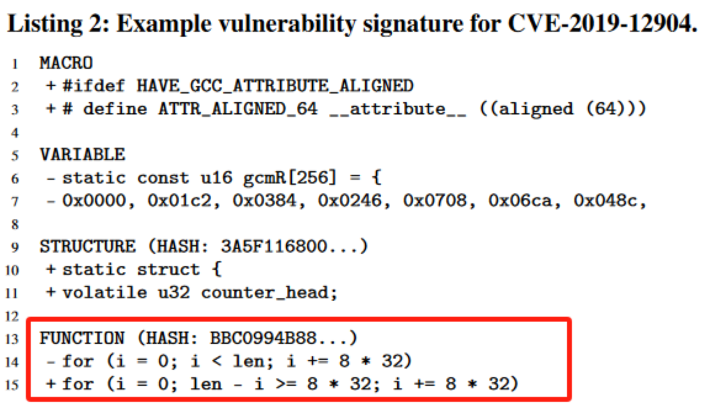
In this section, we introduce briefly the vulnerability detection tool V1SCAN [33] provided on GitHub, which is the prerequisite for our repair tool. Its design details are in the paper [36]. V1SCAN is a state-of-the-art tool that can detect and classify 1-day Vulnerabilities in reused C/C++ OSS components.

It integrates both version-based and code-based methodologies to effectively identify reused and vulnerable code segments in software projects. The tool operates through three main phases as depicted in figure 2.1.

1. **Classification (P1):** V1SCAN classifies reused code from OSS components and vulnerable code locations within the target program using code classification techniques. The detected vulnerabilities are classified into three categories: exactly reused, modified and unused as explained in section 2.1.1.
2. **Detection (P2):** It detects vulnerabilities by applying enhanced version- and code-based strategies. A key feature in this phase is the use of Locality Sensitive Hashing (LSH) in the code-based approach, which efficiently handles code modifications by hashing similar code snippets into the same "buckets" with high probability [31]. LSH utilizes a distance function (Φ) and a cutoff threshold (θ) to categorize code pairs as identical, similar, or different (the output of Φ is an integer):
   - **Identical:** If $\Phi(f_1, f_2) = 0$, f_1 and f_2 are identical.
   - **Similar:** If $0 < \Phi(f_1, f_2) \leq \theta$, f_1 and f_2 are similar.
   - **Different:** If $\Phi(f_1, f_2) > \theta$, f_1 and f_2 are different.
3. **Consolidation (P3):** V1SCAN reviews the detection outcomes from both approaches to verify the vulnerabilities in the target program.

Since we are interested in repairing the modified vulnerabilities, we just introduce briefly the core step of V1SCAN, which is the code-based approach in P2. At this step, vulnerability signature generations are generated first, which are then utilized to detect vulnerabilities.

Vulnerability signature generation. V1SCAN generates vulnerability signatures for each collected CVE patches, which are used detect propagated vulnerabilities. All the code lines added and deleted in the security patch were stored. The LSH hash value of functions is also stored. An example vulnerability signature is shown in figure 2.2. The hash value of the entire function is used for reducing false alarms in vulnerability detection.



Listing 2: Example vulnerability signature for CVE-2019-12904.

```
MACRO
 + #ifdef HAVE_GCC_ATTRIBUTE_ALIGNED
 + # define ATTR_ALIGNED_64 __attribute__ ((aligned (64)))

VARIABLE
 - static const u16 gcmR[256] = {
 - 0x0000, 0x01c2, 0x0384, 0x0246, 0x0708, 0x06ca, 0x048c,

STRUCTURE (HASH: 3A5F116800...)
 + static struct {
 + volatile u32 counter_head;

FUNCTION (HASH: BBC0994B88...)
 - for (i = 0; i < len; i += 8 * 32)
 + for (i = 0; len - i >= 8 * 32; i += 8 * 32)
```

Figure 2.2: The same picture copied from V1SCAN to illustrate the vulnerability signature. (We focus on vulnerability within functions)

Then, V1SCAN can detect and classify modified vulnerabilities through the two-step process below:

S1. Hash comparison (exactly reused vulnerability detection): Initially, V1SCAN identifies potential vulnerabilities in the target program by matching hash values of its functions with those listed in known vulnerability signatures. If an exact match is found, V1SCAN confirms the vulnerability's presence in the target program. Otherwise, if a hash value is found similar, V1SCAN proceeds to the next step, line comparison, to determine vulnerability propagation. (paraphrased the step in the V1SCAN paper)

S2. Line comparison (modified vulnerability detection): When a similar function is detected between the vulnerability signature and the target program, the number of times deleted (added) code lines appear in the target function is checked to determine if the target program contains a similar vulnerable function. If the numbers are the same as those appear in the vulnerable function, the target function is considered as a vulnerable function and classified as modified.

Two key datasets are utilized and how they are used is explained below:

- **OSS Dataset (Centris dataset [34])**: Consists of all versions of 10,241 popular C/C++ OSS projects on GitHub, available in the Centris-public repository [32]. V1SCAN took advantage of Centris because it can precisely identify OSS components that have been modified and its source code and dataset are publicly available. Using Centris, V1SCAN extracted the names of OSS components included in the target program. Note that this dataset is only concerned with component identification, and does not directly participate in vulnerability detection.
- **CVE Dataset**: Contains 4,612 C/C++ security patches collected from the National Vulnerability Database (NVD), available in the V1SCAN-public repository [33]. This CVE dataset consists of the vulnerabilities and patches that V1SCAN can detect, which can be utilized to generate vulnerability signatures as introduced before.

In summary, V1SCAN utilizes the Centris dataset to identify OSS components within target programs and the CVE dataset to detect known vulnerabilities by comparing the generated vulnerability signatures.

However, there are limitations to these datasets. The CVE dataset was found incomplete. The OSS dataset, while extensive, does not cover all OSS components commonly reused in IoT projects, so we build a new OSS dataset, **IoT-1000**, as a contribution detailed in section 4.6, aiming to provide more IoT-specific libraries so that V1SCAN might detect more vulnerabilities. More analysis about the two datasets are described in section 5.1.1

2.3. Motivating Example

To illustrate the challenges in repairing vulnerabilities in reused OSS components, we need to know the distinction between exactly reused and modified vulnerable functions.

For exactly reused vulnerable functions, it is easy to repair by just applying the template patch since the same vulnerable functions are detected in the target file.

However, it is challenging to repair the modified vulnerable functions in the target file. For these functions, if we simply apply the template security patch to the target vulnerable functions by deleting the vulnerable functions and inserting the patch functions, the customized functionality made by the developers on the target function might be lost. It is also infeasible if we try to repair by deleting vulnerable lines and inserting patch lines based on the template security patch, because the context in the modified vulnerable functions might be different in the target file. For example, listing 2.1 shows the template patch snippet for CVE-2020-5235 in FastBee. It shows that the vulnerable line (line 12) and the patch line (line 13) both use ***(uint8_t**)iter->pData**. However, as shown in listing 2.2, which presents the differences between the vulnerable file and the target file, a similar vulnerable line (line 13) in the target file uses ***(char**)iter->pData**. In this case, the vulnerability might not be repaired successfully by simply applying the template security patch.

Therefore, to address such scenarios, we utilized GPT-4o [25] to generate patches that can fit the modified code context in the target file. More details are presented in section 4.4.3.

Listing 2.1: The template patch snippet for CVE-2020-5235 in FastBee

```
//air780e/csdk/luatos-soc-2022/thirdparty/nanopb
--- ./template_vulnerable_pb_decode.c
+++ ./template_patch_pb_decode.c
@@ -636,14 +636,14 @@
size_t *size = (size_t*)iter->pSize;
void *pItem;

- (*size)++;
- if (!allocate_field(stream, iter->pData, iter->pos->data_size, *size))
+ if (!allocate_field(stream, iter->pData, iter->pos->data_size, (size_t)
    ↪ (*size + 1)))
    return false;
- pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size - 1);
+ pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size);
+ (*size)++;
initialize_pointer_field(pItem, iter);
return func(stream, iter->pos, pItem);
```

Listing 2.2: The diff between vulnerable file the target file for CVE-2020-5235 in FastBee

```
//air780e/csdk/luatos-soc-2022/thirdparty/nanopb
--- ./template_vulnerable_pb_decode.c
+++ ./target_pb_decode.c
@@ -626,14 +636,14 @@
- size_t *size = (size_t*)iter->pSize;
+ pb_size_t *size = (pb_size_t*)iter->pSize;
void *pItem;

+ if (*size == PB_SIZE_MAX)
+     PB_RETURN_ERROR(stream, "too many array entries");

- pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size - 1);
+ pItem = *(char**)iter->pData + iter->pos->data_size * (*size - 1);
initialize_pointer_field(pItem, iter);
return func(stream, iter->pos, pItem);
```

3 Related Work

In this chapter, we discuss the related work in areas pertinent to our research, focusing on methods for detecting 1-day vulnerabilities in reused OSS components, and on LLMs for automated vulnerability repair.

3.1. 1-day Vulnerability Detection

Identifying and mitigating 1-day vulnerabilities in software systems, especially those arising from reused OSS components, is a critical challenge in software security. The detection of such vulnerabilities involves two key aspects: software composition analysis and vulnerable code detection.

3.1.1. Software Composition Analysis

Software composition analysis (SCA) involves identifying the third-party OSS components reused in a target program. Accurate identification of these components is essential for vulnerability detection and compliance management. Several approaches have been proposed to detect OSS components in software systems.

CENTRIS [34], proposed by Woo et al., aims to identify modified OSS components by focusing on the unique functions. It uses code segmentation and redundancy elimination techniques to significantly reduce false positives.

OSSPolice [7], introduced by Duan et al., detects the reuse of vulnerable OSS versions in Android apps by comparing similarities between binaries.

LibD [15], developed by Li et al., identifies third-party libraries in Android apps by hashing features. When name-based obfuscation is present, it performs better when handling multi-package third-party libraries, resulting in higher precision without sacrificing scalability.

ATVHunter [44], developed by Zhan et al., utilizes Control Flow Graphs to accurately identify exact versions of third-party libraries. It can confirm whether the target program has any 1-day vulnerabilities by identifying the particular library versions.

While these methods have advanced the field of SCA, they often face challenges when dealing with modified OSS components. Modifications made by developers to fit their specific needs can hinder the accurate identification of reused components, potentially leading to false alarms or missed vulnerabilities.

3.1.2. Vulnerable Code Detection

Vulnerable code detection aims to discover code segments in the target program that are susceptible to known vulnerabilities. This is particularly challenging when the code has been modified from its original form.

VUDDY [13], developed by Kim et al., focuses on scalable detection of vulnerable code clones in large software systems. VUDDY employs a function-level granularity approach, combined with a length-filtering technique, to optimize the efficiency of signature comparisons.

MOVERY [35], developed by Woo et al., is a method for precisely detecting modified vulnerable code clones. It improves upon previous techniques by accounting for code modifications that can obscure traditional clone detection methods.

MVP [40], introduced by Xiao et al., focuses on detecting recurring vulnerabilities even when the vulnerable code has undergone syntax changes. MVP utilizes program slicing techniques to generate vulnerability signatures and match them against unpatched target functions.

While these approaches have made significant contributions to vulnerable code detection, they may not fully address the detection of vulnerabilities in modified OSS components reused in target programs. The challenges arise from significant code changes due to modifications, which can lead to false negatives in detection.

3.2. LLMs for Vulnerability Repair

Recent advancements in LLMs have significantly impacted the field of APR, particularly for repairing security vulnerabilities. Traditional APR approaches, which often rely on generate-and-validate cycles [23], may struggle with context-awareness and scalability.

LLMs such as OpenAI's GPT-4o [25] have demonstrated strong capabilities in understanding and generating code, making them suitable for generating security patches. These models can interpret the context of the code and the nature of the vulnerability to suggest appropriate fixes.

AlphaRepair [39], introduced by Tufano et al., is an approach that uses a cloze-style (infilling) method for APR by leveraging a pre-trained LLM. It directly predicts the correct code based on the context around the vulnerable code, effectively repairing vulnerabilities by filling in the missing or corrected code.

LLMs offer the advantage of performing zero-shot or few-shot learning, allowing them to repair code with minimal examples or instructions. For example, models like Codex [2, 26] and CodeBERT [8, 21] have shown strong performance in generating code completions and repairs, even without extensive task-specific training.

However, while LLMs have shown great potential, they also face limitations. One challenge is that they may not always produce correct or secure code, especially in complex scenarios or when the context is insufficient.

Our work builds upon these advancements by integrating LLMs with a vulnerability detection tool specifically designed for 1-day vulnerabilities in reused OSS components. By combining vulnerability detection with context-aware repair suggestions generated by LLMs, we aim to address the challenges of repairing vulnerabilities in modified code while preserving developers' customizations.

4 1DRep: The APR Tool

4.1. Introduction

In this chapter, we introduce our APR tool **1DRep** that is designed to detect and repair 1-day vulnerabilities in software projects by leveraging template security patches, and the capabilities of the LLM GPT-4o. The primary goal of **1DRep** is to streamline the process of identifying 1-day vulnerabilities in reused C/C++ OSS components and to automate the repair process, thereby reducing the window of exposure to known vulnerabilities.

We begin by providing an overview of the **1DRep** approach, outlining its key components and workflow in section 4.2. We then delve into the specific methodologies employed for function-based repair in section 4.3 and code-based repair in section 4.4, which address exactly reused and modified vulnerable code, respectively. We discuss the modifications made to the existing vulnerability detection tool V1SCAN to enhance its functionality for our purposes in section 4.5.

4.2. Overview of 1DRep

Figure 4.1 provides a high-level overview of **1DRep**'s pipeline. The process begins by deploying the vulnerability detection model V1SCAN [36] on the target programs to identify potential 1-day vulnerabilities. The detected vulnerabilities are then classified into two categories: **exactly reused** and **modified**, as described in section 2.1.1. Based on this classification, we employ two distinct repair strategies to address the vulnerabilities.

For exactly reused vulnerabilities, we apply a function-based repair method that involves replacing the vulnerable functions in the target files with the corresponding patched functions from the template security patches. For modified vulnerabilities, where the code has been altered from its original form, we utilize a code-based repair method. This method first extracts the necessary context from both the target file and the template security patch. Then it combines the context with a predefined instruction as the prompt to query a LLM (e.g., GPT-4o) to generate a customized patch that fits the modified context in the target file. Note that the correctness of generated patches is not evaluated automatically; instead, each patch is manually reviewed.

4.3. Function-based repair

The function-based repair method addresses exactly reused vulnerabilities detected in the target files, as shown in step 2 in the figure 4.1. In this scenario, the vulnerable functions in the target file are identical to those in the template vulnerable code. Therefore, we can repair the vulnerabilities by directly replacing the vulnerable functions in the target file with the corresponding patched functions from the template security patch.

It is important to preserve the order of function declarations and definitions in C and C++ to avoid compilation errors due to undefined functions being called before they are declared. Therefore, we parse the functions in both the target file and the patch file to determine the correct insertion points for the patched functions.

To achieve this, we perform the following steps:

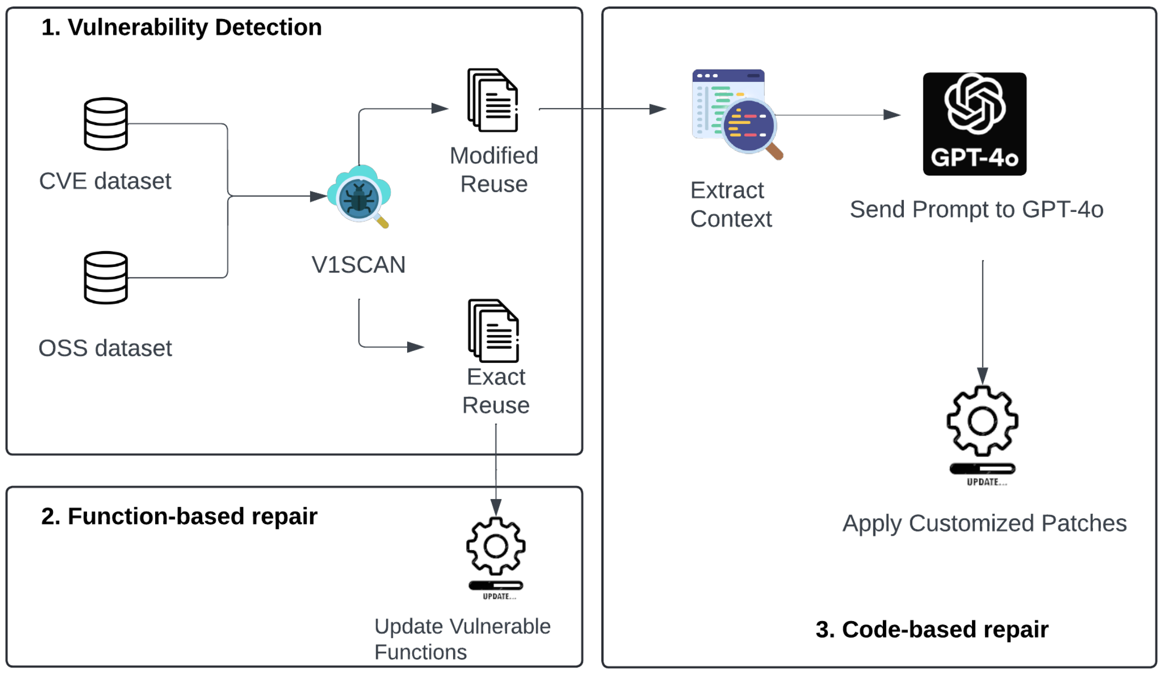


Figure 4.1: Workflow of the APR tool: Step 1 - Detect and classify vulnerable functions in target projects using the vulnerability detection tool V1SCAN. Step 2 - Exactly reused vulnerable functions are fixed by applying template security patches, preserving the context of the target file. Step 3 - For modified reused code, extracted context combined with predefined instructions are sent to GPT-4o for repair suggestions.

1. **Locate the Vulnerable Functions**: Identify and locate the vulnerable functions in the target file by matching their function signatures with those in the template vulnerable code.
2. **Delete the Vulnerable Functions**: Remove the located vulnerable functions from the target file.
3. **Insert the Patched Functions**: Insert the patched functions from the template security patch into the target file at the appropriate locations, ensuring the correct order of function declarations and definitions.

Algorithm 1 outlines the steps involved in the function-based repair process.

Algorithm 1: Function-based Repair

```
Input: $Vul\_Funcs$: List of vulnerable function signatures from the template vulnerable file.
       $Patch\_Funcs$: Corresponding patched functions from the template patch file.
       $Target\_File$: The target file containing the vulnerable functions.
Output: $Repaired\_File$: The target file with vulnerabilities repaired.
1  begin
2      foreach $func \in Vul\_Funcs$ do
3          $loc \leftarrow$ find_function_location($Target\_File, func$); if $loc$ exists then
4              delete_function($Target\_File, loc$);
5          end
6      end
7      foreach $patch\_func \in Patch\_Funcs$ do
8          $insert\_loc \leftarrow$ determine_insertion_point($Target\_File, patch\_func$);
           insert_function($Target\_File, insert\_loc, patch\_func$);
9      end
10     $Repaired\_File \leftarrow Target\_File$;
11     return $Repaired\_File$;
12 end
```

4.4. Code-based repair

For modified vulnerabilities, where the code in the target file has been altered from the original vulnerable code, the function-based repair method is insufficient. As shown in Step 3 of Figure 4.1 , we address these cases using a code-based repair method that leverages GPT-4o to generate customized patches that fit the modified context.

4.4.1. Methodology

The code-based repair process involves the following steps:

1. **Extracting context for the prompt (elaborated in section 4.4.2)**
   - **Extract Code Blocks of Template Patch With Their Context**
   - **Identify Similar Vulnerable Code in Target File**
2. **Prepare Prompt for GPT-4o**: Collect all the extracted information at step 1 and combine it with a predefined instruction to construct a prompt for GPT-4o, aiming to obtain a repair suggestion that can fit the modified context in the target file. The details are presented in section 4.4.3.
3. **Apply the Suggested Patch**: Generate the patched file by applying the repair suggestions to the target file while preserve the target file so that developers can choose not to accept the generated patch. Additionally, two more things are provided here to help developers manually assess the correctness of patched file: (1) an security report, which are detailed in section 5.6, and (2) the template security patches, which are extracted from the CVE dataset of V1SCAN and output into a separate folder. We believe that the two things can effectively inform developers, enabling them to place greater trust in the tool and to better decide whether to accept the generated patch.

To ensure consistent responses, we set the temperature of GPT-4o to 0. Specifically, we used the model version **GPT-4o-2024-05-13**.

4.4.2. Extracting Context and Crafting the Prompt

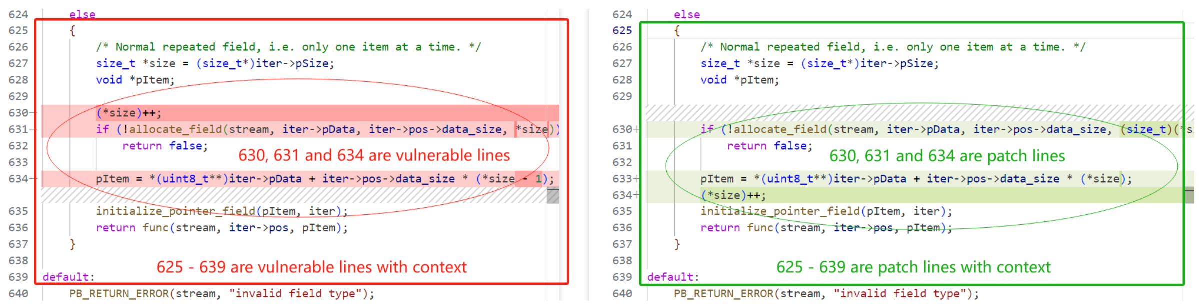
This section elaborates how we extract necessary context for crafting the prompt in two steps and the Algorithm 2 is shown below.

Algorithm 2: Extracting context to craft the prompt for GPT-4o

```
Input: $vul\_file$: The vulnerable file containing the vulnerable lines.
       $patch\_file$: The patch file containing the lines of code intended to repair the
       vulnerabilities in the $vul\_file$.
       $target\_file$: Target file containing modified vulnerable code.
Output: $code\_snippets\_tobePatched$: Vulnerable code snippets with context lines around them
        in the $target\_file$ that might need customized patches suggested by GPT-4o.
        $removed\_lines$ : Vulnerable lines with line numbers extracted from the $vul\_file$.
        $vul\_with\_context$ : Vulnerable lines with line numbers and context lines around them
        extracted from the $vul\_file$.
        $added\_lines$ : Patch lines with line numbers extracted from the $patch\_file$.
        $patch\_with\_context$ : Patch lines with line numbers and context lines around them
        extracted from the $patch\_file$.
1 begin
2     Step 1: Extract Code Blocks of Template Patch With Their Context
3     $removed\_lines, added\_lines \leftarrow$ extract_template_patch($vul\_file, patch\_file$);
4     $vul\_with\_context, patch\_with\_context \leftarrow$ extract_context($vul\_file, patch\_file$);
5     Step 2: Identify Similar Code in Target File
6     $code\_snippets\_tobePatched \leftarrow$ find_similar_code($target\_file, vul\_with\_context,$
       $patch\_with\_context$);
7     return $code\_snippets\_tobePatched, removed\_lines, added\_lines,$
8     $vul\_with\_context, patch\_with\_context$;
9 end
```

In Algorithm 2, we start by performing a diff between the template vulnerable file (vul_file) and the template security patch file ($patch\_file$). This diff operation yields two lists: $removed\_lines$ and $added\_lines$, which are the vulnerable/patch lines with line numbers respectively.

We then add context around these lines to form $vul\_with\_context$ and $patch\_with\_context$. We experimentally set context size as 5, so the context comprises 5 lines preceding and 5 lines succeeding each code block, facilitating the identification of analogous code segments in the vulnerable target function during step two. An example of the context lines is shown in figure 4.2.



```
624         else
625         {
626             /* Normal repeated field, i.e. only one item at a time. */
627             size_t *size = (size_t*)iter->pSize;
628             void *pItem;
629
630-            (*size)++;
631-            if (!allocate_field(stream, iter->pData, iter->pos->data_size, *size))
632                 return false;
633
634-            pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size - 1);

635             initialize_pointer_field(pItem, iter);
636             return func(stream, iter->pos, pItem);
637         }
638
639     default:
640         PB_RETURN_ERROR(stream, "invalid field type");
```

630, 631 and 634 are vulnerable lines
625 - 639 are vulnerable lines with context

```
624         else
625         {
626             /* Normal repeated field, i.e. only one item at a time. */
627             size_t *size = (size_t*)iter->pSize;
628             void *pItem;
629

630+            if (!allocate_field(stream, iter->pData, iter->pos->data_size, (size_t)(*s
631                 return false;
632
633+            pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size);
634+            (*size)++;
635             initialize_pointer_field(pItem, iter);
636             return func(stream, iter->pos, pItem);
637         }
638
639     default:
640         PB_RETURN_ERROR(stream, "invalid field type");
```

630, 631 and 634 are patch lines
625 - 639 are patch lines with context

Figure 4.2: Example picture of the context lines.

Next, we analyze the $target\_file$ to find code snippets that are similar to the vulnerable code blocks extracted from vul_file. We use a sliding window approach in combination with the *SequenceMatcher* class of the *difflib* module [27] to compare segments of the target file with the vulnerable code plus context. Here we experimentally set the sliding window size as 16, so the sliding window comprises of 16 lines of code during the similarity identification process. If a segment in the target file has a similarity ratio above a predefined threshold of the *SequenceMatcher*, we consider it a match and add it to $code\_snippets\_tobePatched$. By doing so, we can collect all potentially vulnerable code blocks and code blocks that might require a patch into $code\_snippets\_tobePatched$.

However, it is possible that no matches are found. When this occurs, no patch will be generated. This situation can occur under an extremely strict condition: when the target file still contains vulnerable lines but the surrounding context has changed significantly. Since such cases are rare (i.e., not observed in our datasets) and too complex to repair automatically, we decide that this issue can be mitigated by the additional two things as mentioned at the step 3 in section 4.4.1, which can assist the developers repair such files manually.

Finally, all the output of the Algorithm 2 is utilized as part of the prompt for GPT-4o, aiming to generate customized patches that fit the modified context of the target file. The prompt and its response is illustrated in the following section.

4.4.3. Prompt and response of GPT-4o

The prompt provided to GPT-4o is a predefined instruction combined with the output in section 4.4.2. An example of the prompt structure is shown in listing 4.1 and figure 4.3. Note that both contain exactly the same prompt: figure 4.3 visually explains the code injection, while listing 4.1 is provided for reproducibility, allowing the reader to copy and paste the prompt.

For our running example (section 2.3), after providing the prompt, GPT-4o generates the repair suggestion for a vulnerable target file in FastBee containing the CVE-2020-5235 as shown in listing 4.2. From this lisitng, we can see that the repair suggestion provided the line range (at line 1) for inserting the patch lines in the correct location. Specifically, the line numbers are extracted by utilizing regular expression to locate the insertion points. After our manual inspection, the insertion points are correct, which implies the C1 (Identifying Correct Insertion Points) as identified in section 1.2, is addressed.

The diff result between the vulnerable target file and the patched file is shown in listing 4.3. Also, the template patch is placed right after the diff result to for comparison in listing 4.4.

We combined the the two listings as the figure 4.4 with remarks to better illustrate why the generated patch is correct that can fit the customized context. From this figure, we can see that customized expressions are retained as **pb_size_t** at line 5 and 12 and ***(char**)** at line 15 in the generated patch (listing 4.3), although they are **size_t** at line 5 and 10 and ***(uint8_t**)** at line 13 in the template patch (listing 4.4). Therefore, since the customized expressions are retained and other parts are the same as the template patch by observing the diffs, we decide the generated patch is a correct customized patch. This shows that the GPT-4o can indeed take into account the possible changes on the context by following part of the prompt (lines 9-12 of the listing 4.1), which addresses the C2 (Adapting Patches to Changed Context) as identified in section 1.2.

Design of the prompt. Here we provide a step-by-step breakdown of how the GPT-4o might take into account the given instructions, which are designed to augment its ability to generate a correct patch that can fit customized context:

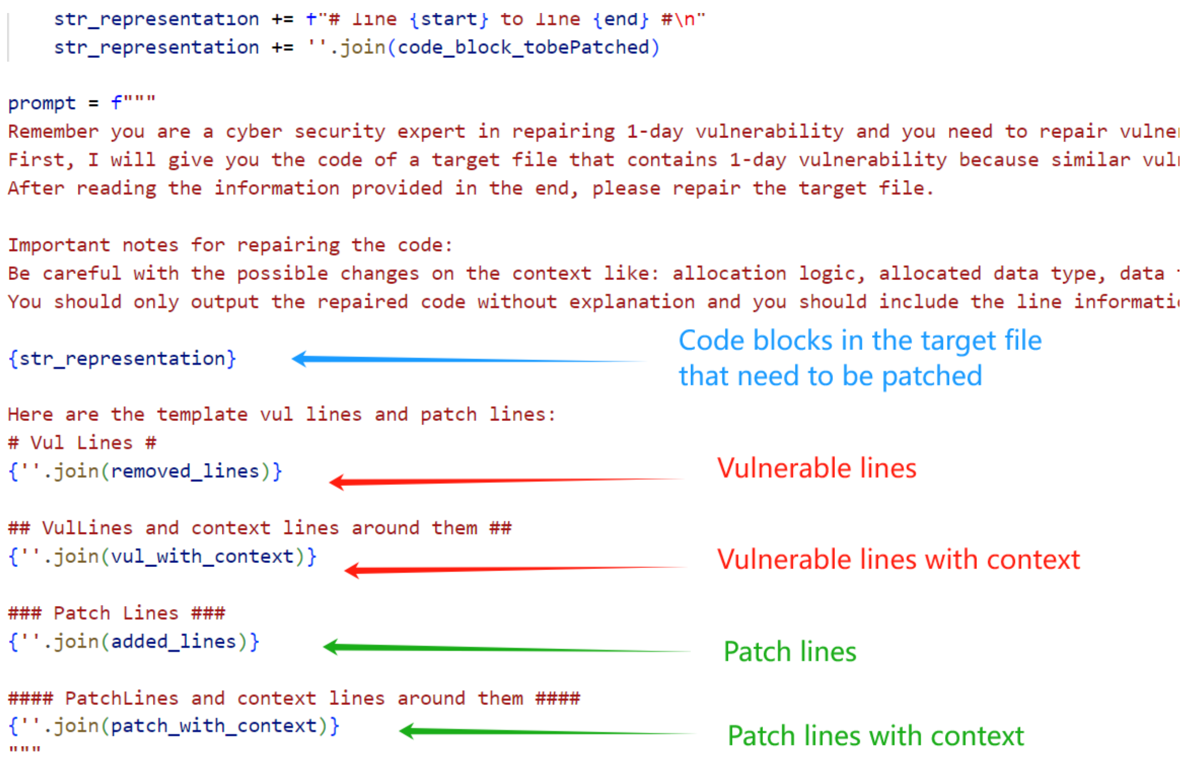
1. **Role Setting and Context Narrowing (lines 1-2).** We assign GPT-4o the role “a cyber security expert in 1-day vulnerability repair.” to help it focus on a specific knowledge domain. Specifically, we aim to make it focus less on general programming principles and more on remediation of known or similar vulnerabilities.
2. **Task Description (lines 3-6).** This part introduces the specific task: GPT-4o will receive the code of a “target file” containing a 1-day vulnerability due to similarities with a “template vulnerable file.” The prompt indicates that the model needs to examine these similarities and apply a repair to the target file based on the instructions provided below.
3. **Repair Instructions (The important notes at lines 8-14).** By lines 8-14, GPT-4o is given two specific instructions for making the repair:

    - **Context Preservation (lines 9-12).** Here, GPT-4o is instructed to “Be careful with the possible changes on the context” and “ensure the context/functionality remains correct.” This is critical because it reminds GPT-4o to carefully generate a patch without disrupting any customized functionality. Furthermore, it requires GPT-4o to focus on details, minimizing the risk of oversight. Note that the listed possible changes do not cover all kinds of modifications in the real world projects.
    - **Output Requirements (lines 13-14).** The two lines instruct GPT-4o to omit explanatory text or extra details, producing a concise response that includes the patched code and the line numbers only. This facilitates automatically applying the patch in the target file.

4. **Contextual code injection (lines 16-31).** By code injection, this part first provides GPT-4o with the code snippets that need to be patched. Then it provides GPT-4o with vulnerable lines and surrounding context, as well as patch lines with their context, guiding the model to recognize vulnerability and patch patterns to construct a compatible repair.

Effectiveness of the repair instructions. Here we examines the effectiveness of the repair instructions. When we submitted the prompt, which was identical to that for the vulnerable target file introduced in this section but without the context preservation instruction (lines 9–12), to GPT-4o, the response was the same as the template patch. Therefore, the generated patch is not correct because it does not fit the context in the target file, overlooking the customizations that developers made. From this case, we conclude that that C2 will not be addressed without the instruction specifying context preservation.

Furthermore, when we submitted the same prompt but without the output requirements instruction (lines 13–14), the response became excessively verbose, with no line information. From this case, we conclude that C1 will not addressed without the instruction specifying output requirements.



```
    str_representation += f"# line {start} to line {end} #\n"
    str_representation += ''.join(code_block_tobePatched)

prompt = f"""
Remember you are a cyber security expert in repairing 1-day vulnerability and you need to repair vulne
First, I will give you the code of a target file that contains 1-day vulnerability because similar vul
After reading the information provided in the end, please repair the target file.

Important notes for repairing the code:
Be careful with the possible changes on the context like: allocation logic, allocated data type, data
You should only output the repaired code without explanation and you should include the line informati

{str_representation}

Here are the template vul lines and patch lines:
# Vul Lines #
{''.join(removed_lines)}

## VulLines and context lines around them ##
{''.join(vul_with_context)}

### Patch Lines ###
{''.join(added_lines)}

#### PatchLines and context lines around them ####
{''.join(patch_with_context)}
"""
```

Code blocks in the target file that need to be patched
Vulnerable lines
Vulnerable lines with context
Patch lines
Patch lines with context

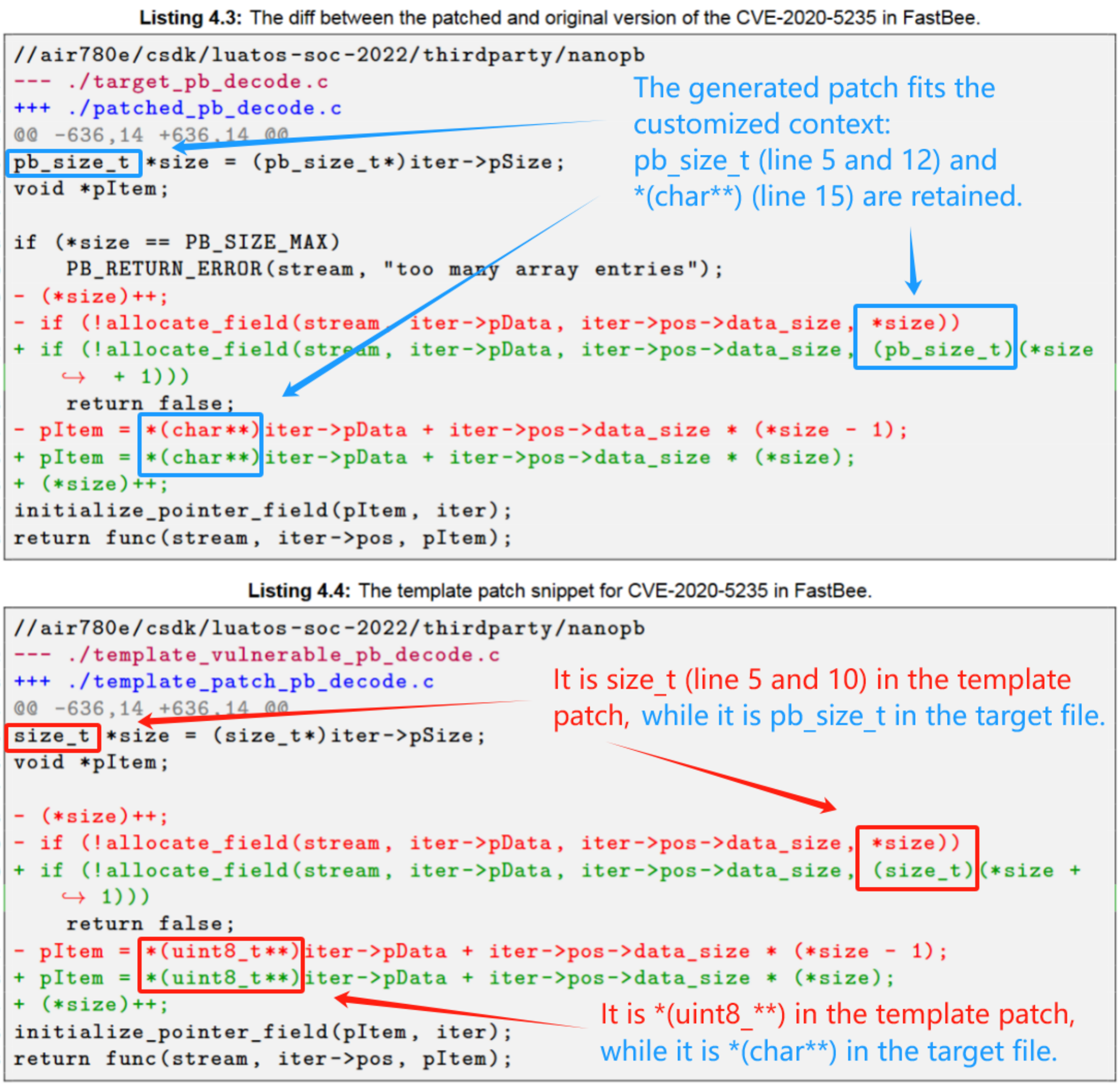
Figure 4.3: Example picture of the prompt.

Listing 4.1: Prompt sent to GPT-4o

```
Remember you are a cyber security expert in repairing 1-day vulnerability and you
need to repair vulnerability following my instructions below:
First, I will give you the code of a target file that contains 1-day vulnerability
because similar vulnerable functions from the template vulnerable file are
detected in the target file.
After reading the information provided in the end, please repair the target file.

Important notes for repairing the code:
Be careful with the possible changes on the context like: allocation logic,
allocated data type, data type, the functions and the calling logic etc.,
since YOU MUST ensure the context/functionality remains correct after
repairing the target file by deleting and adding lines.
You should only output the repaired code without explanation and you should
include the line information to indicate where the patch blocks are.

Code snippets of the function containing potential vulnerable lines in the target
file is given below:
{code_snippets_tobePatched}

Here are the template vul lines and patch lines:
# Vul Lines #
{''.join(removed_lines)}

## VulLines and context lines around them ##
{''.join(vul_with_context)}

### Patch Lines ###
{''.join(added_lines)}

#### PatchLines and context lines around them ####
{''.join(patch_with_context)}
```

Listing 4.2: Repair Suggestion generated by GPT-4o for CVE-2020-5235 in FastBee.

```
###### line 632 to line 652 ######
                return status;
            }
            else
            {
                /* Normal repeated field, i.e. only one item at a time. */
                pb_size_t *size = (pb_size_t*)iter->pSize;
                void *pItem;

                if (*size == PB_SIZE_MAX)
                    PB_RETURN_ERROR(stream, "too many array entries");

                if (!allocate_field(stream, iter->pData, iter->pos->data_size, (
                    ↪ pb_size_t)(*size + 1)))
                    return false;

                pItem = *(char**)iter->pData + iter->pos->data_size * (*size);
                (*size)++;
                initialize_pointer_field(pItem, iter);
                return func(stream, iter->pos, pItem);
            }

        default:
```



Listing 4.3: The diff between the patched and original version of the CVE-2020-5235 in FastBee.

```
1  //air780e/csdk/luatos-soc-2022/thirdparty/nanopb
2  --- ./target_pb_decode.c
3  +++ ./patched_pb_decode.c
4  @@ -636,14 +636,14 @@
5  pb_size_t *size = (pb_size_t*)iter->pSize;
6  void *pItem;
7
8  if (*size == PB_SIZE_MAX)
9      PB_RETURN_ERROR(stream, "too many array entries");
10 - (*size)++;
11 - if (!allocate_field(stream, iter->pData, iter->pos->data_size, *size))
12 + if (!allocate_field(stream, iter->pData, iter->pos->data_size, (pb_size_t)(*size
    ↪ + 1)))
13     return false;
14 - pItem = *(char**)iter->pData + iter->pos->data_size * (*size - 1);
15 + pItem = *(char**)iter->pData + iter->pos->data_size * (*size);
16 + (*size)++;
17 initialize_pointer_field(pItem, iter);
18 return func(stream, iter->pos, pItem);
```

The generated patch fits the customized context: pb_size_t (line 5 and 12) and *(char**) (line 15) are retained.

Listing 4.4: The template patch snippet for CVE-2020-5235 in FastBee.

```
1  //air780e/csdk/luatos-soc-2022/thirdparty/nanopb
2  --- ./template_vulnerable_pb_decode.c
3  +++ ./template_patch_pb_decode.c
4  @@ -636,14 +636,14 @@
5  size_t *size = (size_t*)iter->pSize;
6  void *pItem;
7
8  - (*size)++;
9  - if (!allocate_field(stream, iter->pData, iter->pos->data_size, *size))
10 + if (!allocate_field(stream, iter->pData, iter->pos->data_size, (size_t)(*size +
    ↪ 1)))
11     return false;
12 - pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size - 1);
13 + pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size);
14 + (*size)++;
15 initialize_pointer_field(pItem, iter);
16 return func(stream, iter->pos, pItem);
```

It is size_t (line 5 and 10) in the template patch, while it is pb_size_t in the target file.

It is *(uint8_**) in the template patch, while it is *(char**) in the target file.

Figure 4.4: Example of why the generated patch is correct that can fit the customized context.

Listing 4.3: The diff between the patched and original version of the CVE-2020-5235 in FastBee.

```
//air780e/csdk/luatos-soc-2022/thirdparty/nanopb
--- ./target_pb_decode.c
+++ ./patched_pb_decode.c
@@ -636,14 +636,14 @@
pb_size_t *size = (pb_size_t*)iter->pSize;
void *pItem;

if (*size == PB_SIZE_MAX)
    PB_RETURN_ERROR(stream, "too many array entries");
- (*size)++;
- if (!allocate_field(stream, iter->pData, iter->pos->data_size, *size))
+ if (!allocate_field(stream, iter->pData, iter->pos->data_size, (pb_size_t)(*size
    ↪  + 1)))
    return false;
- pItem = *(char**)iter->pData + iter->pos->data_size * (*size - 1);
+ pItem = *(char**)iter->pData + iter->pos->data_size * (*size);
+ (*size)++;
initialize_pointer_field(pItem, iter);
return func(stream, iter->pos, pItem);
```

Listing 4.4: The template patch snippet for CVE-2020-5235 in FastBee.

```
//air780e/csdk/luatos-soc-2022/thirdparty/nanopb
--- ./template_vulnerable_pb_decode.c
+++ ./template_patch_pb_decode.c
@@ -636,14 +636,14 @@
size_t *size = (size_t*)iter->pSize;
void *pItem;

- (*size)++;
- if (!allocate_field(stream, iter->pData, iter->pos->data_size, *size))
+ if (!allocate_field(stream, iter->pData, iter->pos->data_size, (size_t)(*size +
    ↪ 1)))
    return false;
- pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size - 1);
+ pItem = *(uint8_t**)iter->pData + iter->pos->data_size * (*size);
+ (*size)++;
initialize_pointer_field(pItem, iter);
return func(stream, iter->pos, pItem);
```

4.5. Modification on V1SCAN

To integrate V1SCAN effectively into 1DRep's pipeline, we needed to modify the existing vulnerability detection tool V1SCAN [33]. The original V1SCAN prototype reports the presence of specific CVEs in the target program but does not provide the exact locations of the vulnerabilities within the code.

We extended V1SCAN to:

- **Output Vulnerability Locations:** Modify V1SCAN to provide precise locations (file names and line numbers) of the detected vulnerabilities in the target file. Also, the vulnerable target file and template security patches are moved into a directory for the repair process.
- **Integration with 1DRep:** Ensure that the output format of V1SCAN aligns with the requirements of 1DRep's repair mechanisms, facilitating seamless data flow between detection and repair stages.

4.6. The new OSS dataset IoT-1000

To identify more C/C++ IoT OSS components, we built a new OSS dataset called **IoT-1000**, containing 1,020 C/C++ libraries. This contribution is significant as it complements the Centris dataset by improving the coverage of OSS components relevant to IoT as evaluated in section 5.1.2, which addresses the limitation of the Centris dataset presented in section 5.1.1.

Construction process. The first step was done by a PhD student from UNSW, Shangzhi Xu: he collected all prevalent repositories from GitHub, OpenWRT, stm32duino, awesome-cpp, awesome-c, and mongoose-os-libs in C/C++, and then conducted a selection to choose the specific OSS components. Then we did the second step: filter the repositories via keywords. The keywords were crafted manually by us selecting which are more likely to occur in the tag and title of the IoT projects, and 19,057 libraries were left. The third step is done by Shangzhi Xu, who further filtered the libraries by identifying the dependent libraries and only retaining the parent OSS components and 1,872 IoT-specific libraries were left. Finally, we randomly selected 1,020 libraries from the list to form the **IoT-1000** dataset. Our aim was to investigate whether a smaller dataset could provide significantly better results than the Centris dataset for detecting reused OSS components in IoT projects. Additionally, using a smaller dataset saves considerable time, as utilizing Centris-public [32], which is built from source code, is time-consuming and resource-intensive. Furthermore, since we found that the **IoT-1000** dataset significantly improves the detection of reused OSS components in IoT projects, we decided not to add more data for further experiments, as detailed in section 5.1.1.

The statistics for the **IoT-1000** and Centris datasets are shown in table 4.1. For the Centris dataset, the number of stars (#stars) is not provided in the Centris paper [34]. All **IoT-1000** results were obtained by sending requests to the GitHub API. For the #versions in **IoT-1000**, there are notably many versions across the 1,020 GitHub repositories. This is possibly because Centris extracted versions by tags, while we counted these by following a series of requests to GitHub as outlined below:

- **Step 1:** Attempt to get the number of tags. If found, return that number.
- **Step 2:** If no tags, attempt to get the number of releases. If found, return that number.
- **Step 3:** If no releases, fetch the default branch name.
- **Step 4:** Get the number of commits on the default branch.
- **Step 5:** If all else fails, set versions to 1.

Dataset	#Repositories	#Lines of code	#Versions	#Stars	#Shared repositories between them
IoT-1000	1020	892484140	174428	2184149	415
Centris	10241	80388355577	229326	Not given	

Table 4.1: Statistics of Centris dataset and IoT-1000

5 Evaluation

This chapter presents a comprehensive evaluation of our automated repair tool, **1DRep**, and the vulnerability detection tool, V1SCAN [33], upon which our tool relies. We aim to assess the performance and effectiveness of V1SCAN in detecting and classifying 1-day vulnerabilities in IoT projects, as well as the capability of our repair tool in addressing these vulnerabilities.

Before evaluating V1SCAN and our repair tool, we introduce the datasets used for vulnerability detection. Recognizing that the first step of V1SCAN involves uncovering OSS components within the target program—and given our focus on 1-day vulnerabilities in reused C/C++ IoT OSS components—we constructed a new IoT-specific dataset and combined it with an existing OSS dataset. This combined dataset, utilized by V1SCAN to detect vulnerabilities in IoT projects, is detailed in Section 5.1. Then the target IoT projects and complexity of the detected vulnerabilities are presented in section 5.2.

To structure this evaluation, we pose specific research questions, outline our methodology for dataset creation and analysis, and present the results corresponding to each research question. Our evaluation is guided by the following research questions:

- **RQ1. Vulnerability Detection Performance:** *How precisely does V1SCAN detect and classify vulnerabilities?* (section 5.3)
- **RQ2. Repair Effectiveness:** *How effective is our repair tool in addressing the detected vulnerabilities?*(section 5.4)

To explore the limits of **1DRep**'s repair capability, we generated 90 artificial vulnerable cases by manually modifying existing CVEs and tested the tool on these cases answering the following RQ3:

- **RQ3. Repair Capability:** *How does the repair tool perform across our hand-crafted vulnerabilities?* (section 5.5)

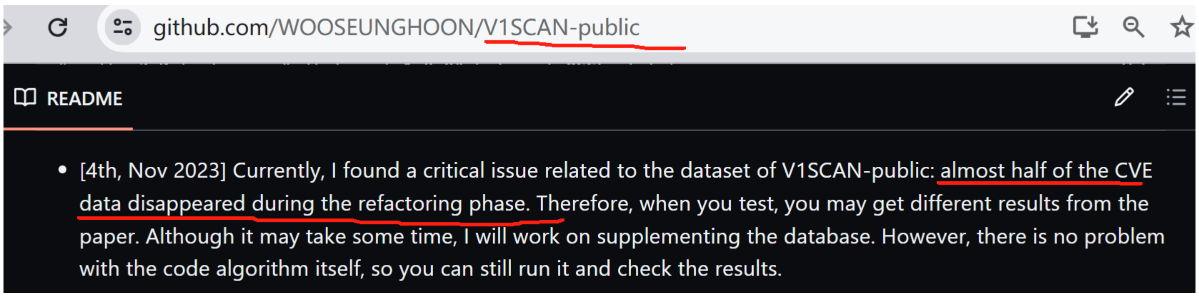
Finally, in section 5.6, we present the security reports submitted to developers with the expectation of receiving feedback.

5.1. Datasets for V1SCAN

5.1.1. OSS dataset and CVE dataset

In order to deploy the vulnerability detection tool V1SCAN [36] to detect and classify vulnerable functions in target programs, the first step is to get an OSS dataset for identifying OSS components and a CVE dataset for classifying and detecting the vulnerable codes. Fortunately, the two datasets are available, which are the same as those described in the paper of V1SCAN [36]. More specifically, the OSS dataset utilized by V1SCAN (Centris dataset) consisting of all versions of the 10,241 popular C/C++ OSS projects on GitHub, is available in the GitHub repository Centris-public [32], and the CVE dataset containing 4,612 C/C++ security patches from the NVD is available in the GitHub repository V1SCAN-public [33].

However, we found two limitations for the two datasets summarized below:


github.com/WOOSEUNGHOON/V1SCAN-public

README

- [4th, Nov 2023] Currently, I found a critical issue related to the dataset of V1SCAN-public: almost half of the CVE data disappeared during the refactoring phase. Therefore, when you test, you may get different results from the paper. Although it may take some time, I will work on supplementing the database. However, there is no problem with the code algorithm itself, so you can still run it and check the results.

Figure 5.1: CVE dataset disappeared description in the V1SCAN-public repository on GitHub

- **Incomplete CVE Dataset:** When we did some experiments using the prototype of V1SCAN [33], we obtained different results compared to those reported in their paper. After we reported the issues to the authors of V1SCAN through emails, we got a reply that: "almost half of the CVE data disappeared during the refactoring phase." as noted in the figure 5.1.
- **Limited Applicability of the Centris Dataset for IoT Projects:** Although the Centris dataset contains many popular C/C++ OSS projects on GitHub, it is still possible that just a few of them are reused in IoT projects. Our experimental findings suggest that this dataset is not particularly effective for identifying OSS components in IoT projects as illustrated in section 5.1.2.

For the first limitation, given that creating a complete CVE dataset is time-consuming and the dataset only impacts the detection phase, we opted to continue to use the incomplete version, although this might be one of the reasons why the detection performance is unsatisfying (see section 5.3) regarding to the precision metric.

For the second limitation, we built the new dataset **IoT-1000**, which complements the Centris dataset, as detailed in section 4.6.

5.1.2. Effectiveness of the new IoT dataset IoT-1000

As shown in table 4.1, we found only 415 OSS components are common across both datasets, suggesting that our new OSS dataset, **IoT-1000**, indeed provides more IoT-specific libraries. Then we tested the practical effectiveness of **IoT-1000** by applying Centris [34], which is used for component identification, to 40 C/C++ IoT projects as shown in figure 5.2 and detailed in section 5.2. We utilize Centris because it can precisely identify OSS components that have been modified and its source code and dataset are publicly available (the same reason as V1SCAN provided as described in section 2.2). By using Centris, we can extract the names of OSS components included in the target program.

Here we show how many reused OSS components can be detected respectively by using the Centris dataset and **IoT-1000** dataset respectively. As shown in figure 5.2, 132 reused OSS components were detected across the 40 IoT projects using the IoT dataset, representing a 45% increase compared to the Centris dataset, which identified only 91 reused components. Additionally, the figure shows that only 32 identified OSS components overlap, suggesting minimal overlap in their detection capabilities in IoT projects. This result demonstrates that we successfully expanded the dataset to identify C/C++ OSS components within target IoT programs. Although the Centris dataset includes 10 times more libraries and 100 times more #lines of code than our dataset, as shown in table 4.1, our **IoT-1000** dataset can detect significantly more reused OSS components in IoT projects.

To provide more details for the 11 projects that contain at least one detected CVE by V1SCAN, we show the number of reused OSS components identified by using the Centris and IoT dataset respectively in figure 5.3 and their distribution in figure 5.4. Among the 11 projects, V1SCAN detects 30 additional OSS components with the IoT dataset, a 46.9% increase compared to the Centris dataset. Note that we did not count the number of false positives (FPs), since it is time-consuming and our focus is building a tool can repair vulnerabilities automatically.

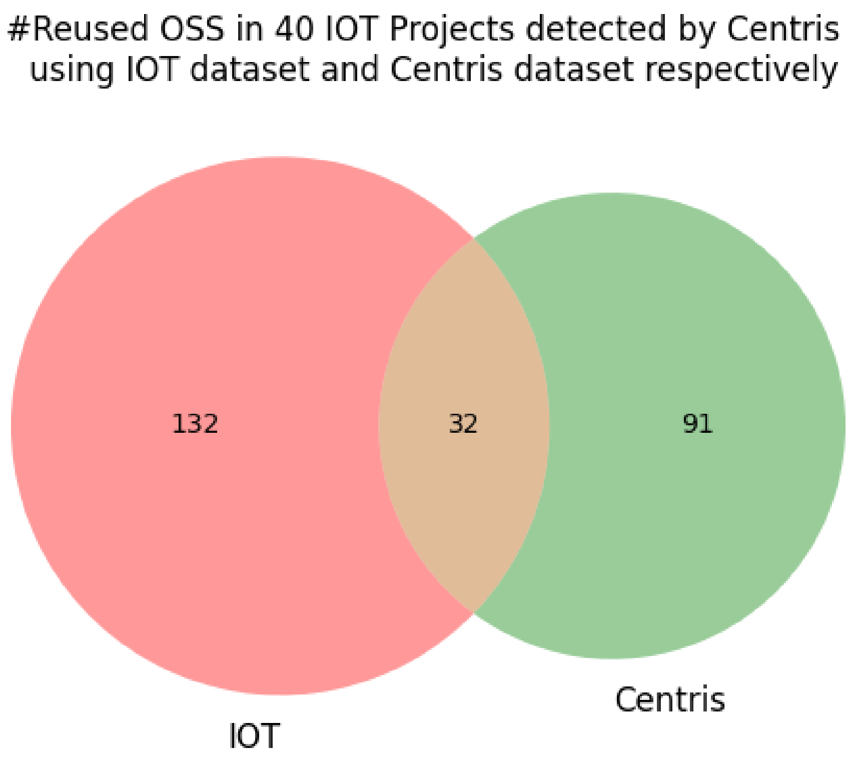


Figure 5.2: The number of reused OSS across 40 IoT projects

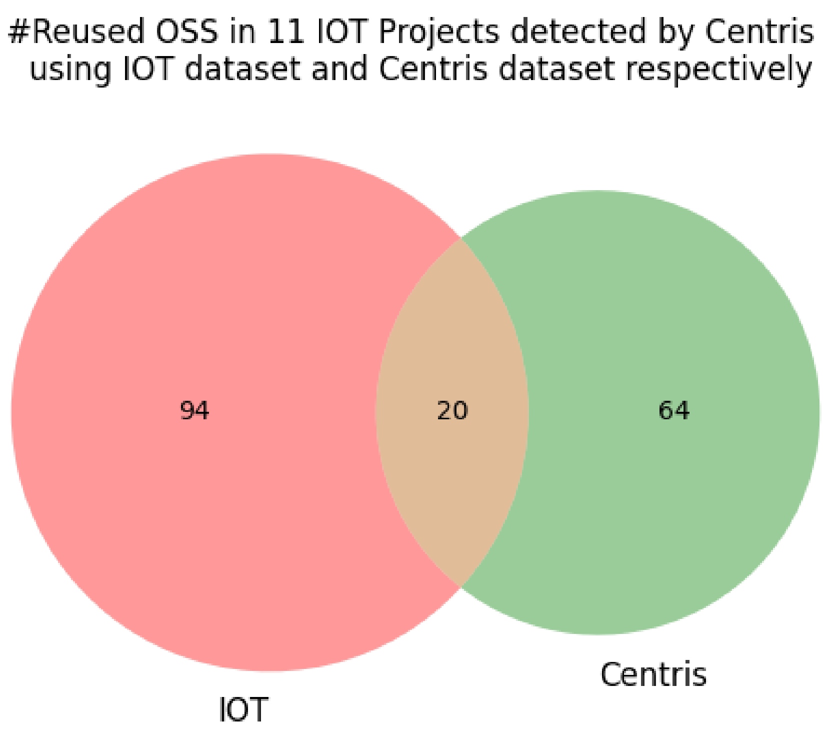


Figure 5.3: The number of reused OSS across 11 vulnerable IoT projects

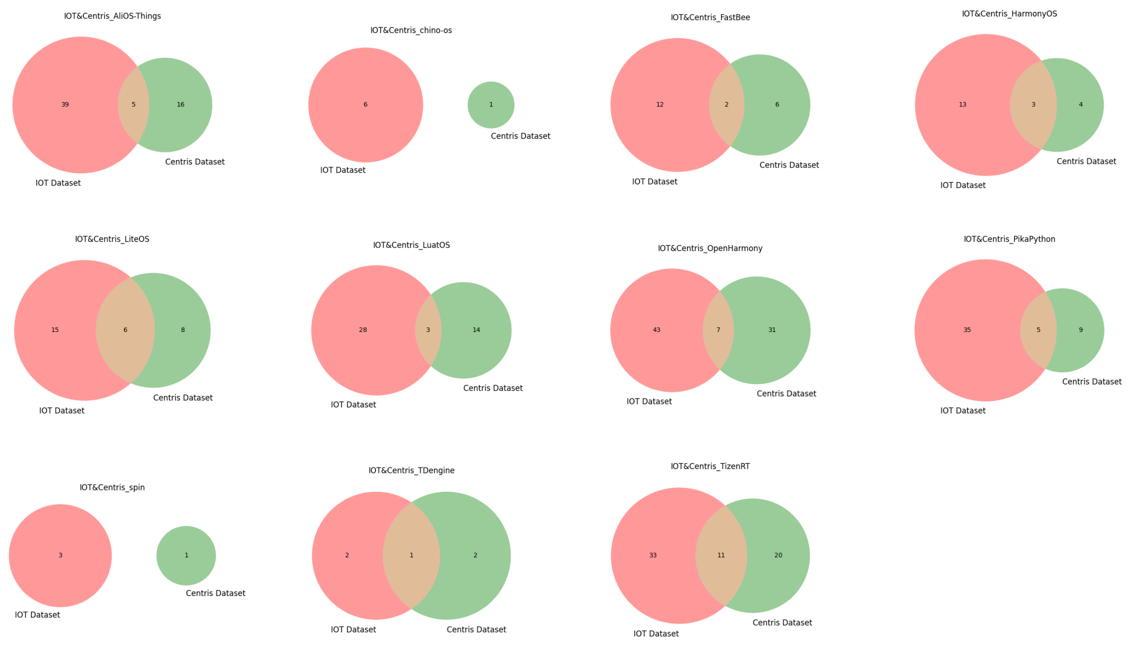


Figure 5.4: Distribution of the number of reused OSS across 11 IoT projects that contain at least one CVE

5.2. Target IoT projects and complexity of the detected vulnerabilities

To collect 1-day vulnerabilities in reused C/C++ IoT OSS components without loss of generality, we randomly picked 40 IoT C/C++ projects, which are the same ones used in section 5.1.2, as input for V1SCAN based on the stars of the repos on GitHub that have IoT tag and their main languages are C or C++. We present all the 40 projects with their commit SHAs in the table 5.1. The commit SHAs indicate the versions of the projects that are used to detect vulnerabilities by V1SCAN. Only 11 of the target projects detected by V1SCAN that contain CVEs are summarized in the table 5.2.

#	Project Name	Commit SHA
1	SIDN/spin	a5185b76c5f39220657ffaedbb1ce7724bce820f
2	openLuat/LuatOS	d4aaca0a011228ab4ed908fdfca1be5010618c6e
3	Samsung/TizenRT	1601a36b77dc7bdee54f3b72876213b6b56c2e8e
4	alibaba/AliOS-Things	a99f20706f9c666903a12a205edce13263b1fadb
5	LiteOS/LiteOS	2f8fdf9c444339262f03f61f6652af2d8a4b4255
6	pikasTech/PikaPython	d8cc0b046552c14eff8a9d84feda01c7b04a6d2c
7	chino-os/chino-os	e4b444983a059231636b81dc5f33206e16a859c2
8	taosdata/TDengine	9186035c014694d83e70ea8058c7d083142d7f4b
9	Awesome-HarmonyOS/HarmonyOS	5d2008e97ea2547dfb3d0b4c9409953e1bf633fc
10	fenwii/OpenHarmony	b5c261646c019f9b7dcd9f0b5aba6b7d452391eb
11	kerwincui/FastBee	3d44f4674c6eaecf12232ae96bb5256cb63a5403
12	RIOT-OS/RIOT	6a59642a1eb74af3671c904b1754acc3a824b0b5
13	arendst/Tasmota	95f7d33c206e6b56346ca529972e2f89df1f417a
14	zephyrproject-rtos/zephyr	ddb147d0a45df2b52af1065a49770566847ab975
15	armink/FlashDB	24305a9c76e735e89ad0d973a09f161edbc4279e
16	timmbogner/Farm-Data-Relay-System	d3cefe9851804bf2ac04c55f2bf51db1261a2b07
17	openairproject/sensor-esp32	374e28ed91679678e267c937e63ae3dcc64d7818
18	sysprogs/PicoHTTPServer	b116dd821a208dc3f41d6041d933136e97ff1310
19	dontsovcmc/waterius	fdfecc5f93b94369dfadd142dccf43055aac3f38
20	gmag11/EnigmaIOT	113c4a72dc44ef6915a97bf68cdef2824c01cb3a
21	halfgaar/FlashMQ	cd631dfab01fce4041b17929841dacb64e833ceb
22	phodal/smart-home	c221303473985207d0f010eb590e35bebfc4d3c8
23	metriful/sensor	e1ce8a0b99acbbded89027702b3aa6e3e5c10629
24	anklimov/lighthub	242db1552e0e6aa5b251a6468ae1c97f79387b28
25	timescale/timescaledb	ecf6beae5db1fb0cef93230e46dc7faf305cb5ef
26	cesanta/mongoose	be5d8b1d4f3b196ae0ee8e66525e743ed587d35f
27	supergreenlab/SuperGreenOS	13b2c711086fee2a62390bbb0e3abbda094c8ea0
28	SMotlaq/open-watch	d742a95125eeb9999e190d81bcfb1ffff49144aa
29	lupyuen/stm32bluepill-mynewt-sensor	c52dacb486bdc50b31bedf161a316ee61ca71cf9
30	EDI-Systems/M7M01_Eukaron	9ffc6cc82b3b773b5b18d98b828b014ecec2b5cb
31	RT-Thread/IoT_Camera	1d4eccb6468556a79ebae1b311bf7bd0817ba3b6
32	SRA-VJTI/Wall-E	803db2d46313b534094a5ee3eb455927051151fb
33	ARMmbed/mbed-os	baf6a3022a328b91713e03fd88f65126a9a53f01
34	lithiumice/XTerminal	bbcbd4887dfad404889f73c51f8315bd9c2166bd
35	oosmos/oosmos	2a308310deb9727ae5a638795d2f35fe298d8729
36	At-EC/At-RTOS	15ae539784cbd1d3a83e2f08c4cf5c4d7e6a71e8
37	ljalves/hfeasy	ae1b59fb6a4fac6fac5c8fc47a1cdf20d8e34f2e
38	SmingHub/Sming	ffb1b3a336a1a49cd97681720a7bf4b23821b3ca
39	alf45tar/PedalinoMini	7fa5ecb1f9bf3023889be7917817361f56aecba4
40	younghyunjo/esp32-homekit	fb7392db400e7ed218936400f844426422ce9d1f

Table 5.1: Target projects with their commit hashes

#	Project Name	#CVE	#OSS	#Stars	#C/C++ Lines
1	SIDN/spin	1	4	76	538 517
2	openLuat/LuatOS	1	45	444	367 457 255
3	Samsung/TizenRT	13	64	557	143 865 023
4	alibaba/AliOS-Things	14	60	4551	230 374 199
5	LiteOS/LiteOS	3	29	4770	78 490 628
6	pikasTech/PikaPython	2	49	1435	349 786 612
7	chino-os/chino-os	1	7	147	3 445 133
8	taosdata/TDengine	1	5	22 906	22 469 533
9	Awesome-HarmonyOS/HarmonyOS	5	20	19 168	33 470 470
10	fenwii/OpenHarmony	18	81	785	53 769 724
11	kerwincui/FastBee	3	4	1381	9 513 145
Total	11 vulnerable IoT projects	62	368	56 220	1 293 180 239

Table 5.2: Overview of the metrics of the 11 projects with at least one detected CVE.

5.2.1. Complexity of the vulnerabilities detected in target IoT projects

Since repairing modified vulnerabilities is considerably more complex, we focused this section on analyzing the true positive (TP) modified vulnerabilities used to evaluate the effectiveness of 1DRep, as detailed in section 5.4. Specifically, we examined all 15 TP modified vulnerabilities, which include the 10 “modified only” TPs and the 5 “exactly reused & modified”” TPs, detailed in section 5.3. Upon examining the 15 TP modified vulnerabilities in detail, we classified them into three categories based on the repair effort required:

- **Old version:** Target file contains a vulnerability that can be repaired by updating it to the latest version, because the target file is reused without any extra functionality.
- **Unchanged context:** Target file contains a vulnerability that can be repaired by adding/deleting vulnerable lines and patch lines based on the template security patch, since the context does not change even if some modifications on the vulnerable functions still exist.
- **Changed context:** Target file contains a vulnerability that cannot be repaired directly by using the template patch lines, because the context is changed in the target file. Therefore, a transformed patch suggested by GPT-4o is required for this case.

The distribution of the three kinds of vulnerabilities among the 15 TP modified vulnerabilities is presented in the table 5.3, from which we can see that 14 out of the 15 vulnerabilities (93.3%) can still be repaired by directly applying the template patch to the target file or updating the target file to the latest version. Two special cases are identified here: one is CVE-2019-16910, which is classified as “old version”” but is not repaired successfully. Another one is the single “changed context”” vulnerability, CVE-2020-5235, which requires code transformations on the template patch due to changed context in the target file. This vulnerability is correctly repaired. More analysis about repairing the two special cases is presented in section 5.4.

However, this does not imply that only this single “changed context” vulnerability should proceed to Step 3 of **1DRep** (code-based approach). Automatically categorizing vulnerabilities that require contextual code-based repair is challenging and often impractical, as it demands a deep contextual understanding. Therefore, as a practical approach, any vulnerability classified as ”modified” should be repaired through the code-based approach, which can save developers significant time in understanding contexts.

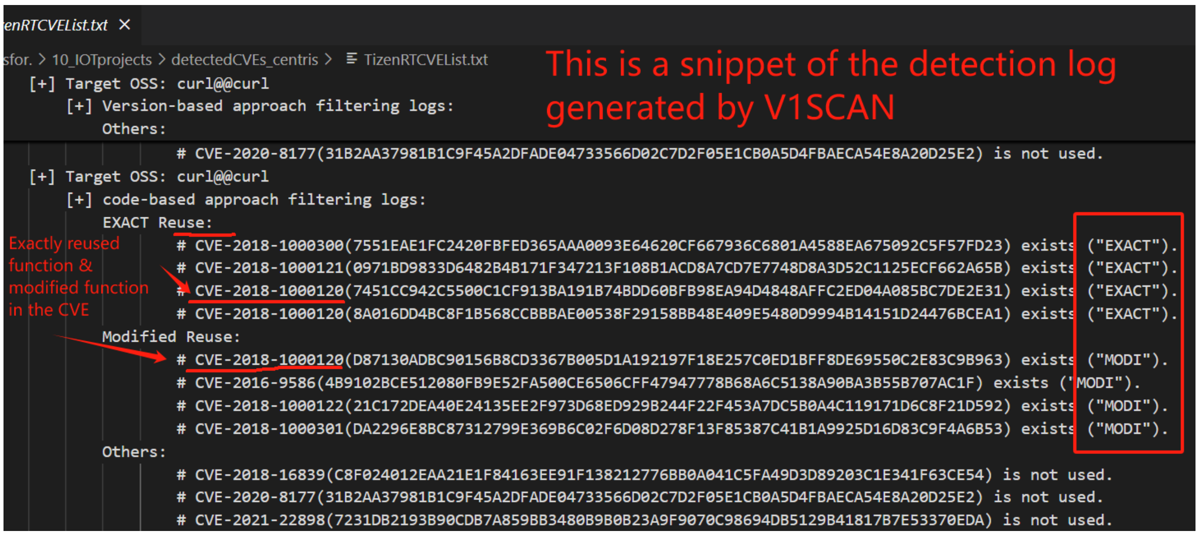
	#Old version	#Unchanged context	#Changed context
15 TP modified vulnerabilities	7	7	1

Table 5.3: Vulnerability complexity distribution among 15 detected modified vulnerabilities

Table 5.4: Detection results of V1SCAN for different vulnerability types.

Vulnerability type	**CVEs***	**V1SCAN**		
		#TP	**#FP**	**P†**
Exactly reused only	25	25	0	1.00
Modified only	32	10	22	0.313
Exactly reused & Modified	5	5	0	1.00
Total	**62**	**40**	**22**	**0.645**

CVEs*: Total number of CVEs detected by V1SCAN, P†: Precision


enRTCVEList.txt ×
sfor. > 10_IOTprojects > detectedCVEs_centris > TizenRTCVEList.txt
This is a snippet of the detection log generated by V1SCAN
Exactly reused function & modified function in the CVE

```
  [+] Target OSS: curl@@curl
      [+] Version-based approach filtering logs:
          Others:
                  # CVE-2020-8177(31B2AA37981B1C9F45A2DFADE04733566D02C7D2F05E1CB0A5D4FBAECA54E8A20D25E2) is not used.
  [+] Target OSS: curl@@curl
      [+] code-based approach filtering logs:
          EXACT Reuse:
                  # CVE-2018-1000300(7551EAE1FC2420FBFED365AAA0093E64620CF667936C6801A4588EA675092C5F57FD23) exists ("EXACT").
                  # CVE-2018-1000121(0971BD9833D6482B4B171F347213F108B1ACD8A7CD7E7748D8A3D52C1125ECF662A65B) exists ("EXACT").
                  # CVE-2018-1000120(7451CC942C5500C1CF913BA191B74BDD60BFB98EA94D4848AFFC2ED04A085BC7DE2E31) exists ("EXACT").
                  # CVE-2018-1000120(8A016DD4BC8F1B568CCBBBAE00538F29158BB48E409E5480D9994B14151D24476BCEA1) exists ("EXACT").
          Modified Reuse:
                  # CVE-2018-1000120(D87130ADBC90156B8CD3367B005D1A192197F18E257C0ED1BFF8DE69550C2E83C9B963) exists ("MODI").
                  # CVE-2016-9586(4B9102BCE512080FB9E52FA500CE6506CFF47947778B68A6C5138A90BA3B55B707AC1F) exists ("MODI").
                  # CVE-2018-1000122(21C172DEA40E24135EE2F973D68ED929B244F22F453A7DC5B0A4C119171D6C8F21D592) exists ("MODI").
                  # CVE-2018-1000301(DA2296E8BC87312799E369B6C02F6D08D278F13F85387C41B1A9925D16D83C9F4A6B53) exists ("MODI").
          Others:
                  # CVE-2018-16839(C8F024012EAA21E1F84163EE91F138212776BB0A041C5FA49D3D89203C1E341F63CE54) is not used.
                  # CVE-2020-8177(31B2AA37981B1C9F45A2DFADE04733566D02C7D2F05E1CB0A5D4FBAECA54E8A20D25E2) is not used.
                  # CVE-2021-22898(7231DB2193B90CDB7A859BB3480B9B0B23A9F9070C98694DB5129B41817B7E53370EDA) is not used.
```

Figure 5.5: A snippet of the V1SCAN detection log as an example for TizenRT using the Centris dataset.

5.3. RQ1. Vulnerability Detection Performance: How precisely does V1SCAN detect and classify vulnerabilities in the 40 target IoT projects?

In this section, we aim to evaluate the overall performance of V1SCAN in detecting and classifying vulnerabilities across our 40 target IoT projects.

Methodology. To evaluate the performance of V1SCAN, we used the following three metrics: **true positive (TP), false positive (FP), precision (P = #TP/(#TP + #FP))**. We do not use **recall (R = #TP/(#TP + #FN))** as an evaluation metric, because we do not know how many vulnerabilities would exist in a target program as ground truth without using other vulnerability detection tools and our main goal is to build an automatic repair tool. The TPs and FPs were determined by manual analysis: we first got the detection report of V1SCAN (an example is shown in figure 5.5) and then vulnerabilities are examined by referring to (1) the target vulnerable code, (2) the security patch, (3) the NVD description. Note that V1SCAN can detect vulnerabilities in functions in a target program and a vulnerability might contain more than 1 vulnerable functions. Therefore, a target program might contain a vulnerability that is classified as both exactly reused and modified. This occurs when the target program exactly reused some vulnerable functions while modifying others. In the log snippet shown in figure 5.5, V1SCAN reported that the target project TizenRT reused the curl library and contained the vulnerability CVE-2018-1000120 in both exact reuse and modified reuse forms, since TizenRT reused two vulnerable functions reused exactly and one modified.

Results. From the results presented in the table 5.4, we can see that among the 62 vulnerabilities detected in our 40 target IoT projects, 40 are TPs and 22 are FPs, resulting in a precision rate of 64.5%. Furthermore, 25 are classified as exactly reused, 32 as modified, and 5 as both exactly reused and modified.

Answer to RQ1. As shown in table 5.4, V1SCAN has a perfect precision (1.00) for the 25 exactly reused vulnerabilities, while the precision for modified vulnerabilities is low: only 10 out of 32 are TPs (0.313). However, by comparing this result with the results reported in the paper of V1SCAN [36], we found a huge inconsistency: V1SCAN detected 137 TPs and 6 FPs across 10 target programs, so their precision is 0.96. This highlights the need for improvements in detection algorithms for modified code.

Analysis. The lower precision observed is primarily due to FPs that were reported as vulnerabilities, despite having already been patched. According to V1SCAN's algorithm, these should have been filtered out. Only a few of them are due to the inevitable shortcoming in the line comparison part of its algorithm.

To understand why patched ones are not filtered out, we checked the code of V1SCAN prototype provided on GitHub [33] and found that some bugs existed in their algorithm. We had this conclusion by reading their paper [36], which was inconsistent with the implementation in the provided prototype. More specifically, the algorithm for filtering out FPs failed to compare all pairs of patch functions and target functions to decide whether target function is vulnerable or not. Instead, it only compares the one patch function to the target function to get the similarity score.

5.4. RQ2. Repair Effectiveness: How effective is our repair tool in addressing the detected vulnerabilities?

In this section, we aim to assess how well our repair tool, **1DRep**, can resolve vulnerabilities detected in the IoT projects.

Methodology. We evaluated the effectiveness of our repair tool by assessing the number of complete fixes achieved for the detected TP vulnerabilities categorized by their type. A fix is considered complete only if it correctly addresses all the vulnerable functions without changing the customizations made by developers. Other partial fixes or fixes that change the customizations are not regarded as successful.

Results. Note that the vulnerability types presented in table 5.5 are the TP vulnerabilities presented in table 5.4. Here, TP modified vulnerabilities include "modified only" TPs and "exactly reused & modified"" TPs.

As shown in table 5.5, **1DRep** successfully repaired all vulnerabilities except for one “modified”” TP, CVE-2019-16910 in OpenHarmony, achieving a 93.3% precision rate for ”modified” TPs and a 100% precision rate for ”exactly reused” TPs. This exception is a special case, categorized as an ”old version” type, as explained in the following analysis and detailed in Notably, it successfully repaired the special case CVE-2020-5235 found in FastBee, which is a TP modified vulnerability and categorized as “changed context””, as explained in section 5.2.1.

Analysis. Here we analyze two special cases, beginning with the “changed context”” vulnerability, CVE-2020-5235 in FastBee. It is special because successfully repairing it indicates that our tool has the ability to address C1 (identifying correct insertion points) and C2 (adapting patches to changed context), as outlined in section 1.2. Additionally, this case is rare, as we detected only one such “changed context”” instance among 40 systems. By investigating this special case, we found the changed context was located around the vulnerable/patch lines and was included within the extracted context lines. Furthermore, the patch is quite short (3 lines), resulting in a short prompt. This inclusion and short prompt likely enabled GPT-4o to recognize the surrounding context and generate a correctly transformed patch.

However, because a single case is insufficient to fully demonstrate **1DRep’s** capability, we further tested it on artificial vulnerable cases, evaluated in RQ3 as detailed in section 5.5.2.

Another special case is the failure to repair CVE-2019-16910 in OpenHarmony that can be attributed to the length of the patch. The template security patch for this CVE spans 99 vulnerable and context lines, with an additional 243 patch and context lines, totaling 437 lines and resulting in a prompt string length of 23,080 characters. This extensive length makes it challenging for GPT-4o to generate a complete and effective fix. Although this vulnerability is classified as an ”old version” type, which is relatively easy to repair manually, the prompt’s size affects the repair ability. This is why this case is also regarded as special.

Although we tried to improve the prompt by adjusting the predefined instructions, we failed to generate a correct patch for this special case. Therefore, although it is possible to generate a correct patch by providing a better structured prompt, we believe it is more possibly because when a prompt is longer, the model is highly likely to misinterpret the intent or become distracted by irrelevant details. By providing a shorter prompt, the model is much more likely to generate a correct patch. This case highlights a limitation of the tool when dealing with vulnerabilities having a lengthy patch and suggests a future improvement for dealing with such cases.

Answer to RQ2. 39 out of 40 vulnerabilities are completely repaired (97.5% precision rate) except for 1 TP modified vulnerability. This demonstrates its effectiveness in repairing both exact and modified reused 1-day vulnerabilities. Successfully repairing the CVE-2020-5235 indicates that our tool has the ability to address C1 and C2 when 1-day vulnerabilities contain limited vulnerable/patch lines. The scarcity of the “changed context”” cases prompts the need to obtain more cases as solved in section 5.5.2. However, the failure to repair CVE-2019-16910 highlights a limitation of the tool when dealing with vulnerabilities having a lengthy patch and suggests a future improvement for dealing with such cases.

Vulnerability type	#Completely Patched	#Vulnerabilities	Success Rate
TP Exactly reused	25	25	100%
TP Modified	14	15	93.33%
Total	39	40	97.5%

Table 5.5: Repair results of **1DRep** for the detected vulnerable reuses

5.5. RQ3: Repair Capability: How does 1DRep perform among the artificial vulnerable reuses?

To explore the limits of **1DRep**’s repair capability, we test it on the artificial vulnerable cases generated by modifying existing template vulnerable files in the CVE dataset. This is because it is hard to find actual cases that are of the type “changed context”” in the wild (only one in 40 systems).

Before we present the evaluation in section 5.5.2, we explain why and how we generate the artificial cases in the following subsection.

5.5.1. Generating artificial vulnerable reuses

Motivation and Characteristics of The Reuses. To thoroughly evaluate the limits of **1DRep's** repair capability, in handling modified vulnerable code reuses, we artificially generated such cases. These artificial reuses serve as test cases to assess **1DRep**'s ability to generate appropriate customized patches under varying levels of complexity.

The artificially generated vulnerable reuses mimic special real-world scenarios where template security patches become inapplicable due to developers' modifications that change the context. In this situation, repairing is more challenging when vulnerable target files cannot be repaired merely by substituting the vulnerable lines with template patch lines, as these templates do not align with the modified context in the target files.

We focus on these challenging scenarios for several reasons:

1. **Limited Availability of Special Cases:** Out of 15 TP modified vulnerable reuses detected, only two qualify as special cases: one is the "changed context"" case, while another one is the "old version"" case (briefly introduced in section 5.2.1 and in-depth analysis is in section 5.4). This scarcity necessitates the creation of additional artificial cases. Notably, we focus solely on handling the "changed context" cases, acknowledging that addressing another type—where the patch is excessively lengthy—presents greater challenges and is left for future work.
2. **Necessity for Customized Patches:** The cases that require customized patches generated by our tool, **1DRep**, are critical for evaluating the tool's unique capabilities.
3. **Complexity:** Due to the changed context, these cases are more challenging to repair. This complexity makes them ideal candidates for thoroughly exploring the limits of **1DRep's** repair capability.

The artificial vulnerable reuses we generated have the following characteristics:

- **Based on Template Vulnerabilities with Limited Context:** Through experimentation, we found that when the context for patching a vulnerability is too extensive, GPT-4o tends to produce incorrect or incomplete patches, regardless of prompt modifications. To mitigate this, we focus on vulnerabilities with a smaller context (within 20 lines of code). We generate artificial cases by randomly modifying template vulnerable files that contain a limited number of vulnerable and patch lines.
- **Proximity to Vulnerable/Patch Lines.** Modifications are made directly on or near the vulnerable or patch lines. Such targeted changes increase the likelihood that template security patches become inapplicable, thereby necessitating the use of our tool, 1DRep.
- **Various Modification Types:** In real-world scenarios, developers modify code based on specific purposes and the role of target lines within their programs. Given the lack of discernible patterns from the limited number of existing vulnerable reuses, we assume that all types of modifications are possible. For simplification, we focus on the following modification types: changing variable names, changing called function names, altering if conditions, modifying returned error values, changing comments, and altering function calls. Changes in this context can be add, remove, or modify.
- **Vulnerability Persistence:** The goal of the progression is to ensure that the vulnerability remains, but in a context where the patch can no longer be applied directly. As the complexity increases, the persistence of the vulnerability is made more ambiguous or disguised by the modifications, meaning that the repair tool must perform more sophisticated analysis to identify the vulnerability and apply a customized patch.
- **Increasing Complexity:** We progressively increase the complexity of the modifications. As the number of modifications increases, we assume that the complexity of the repair task increases correspondingly.

Methodology Our methodology comprises two steps: first, we select vulnerabilities from the CVE dataset provided by [33], focusing on those with a limited number of vulnerable and patch lines.

Then, we manually modify the selected vulnerable code. This process involves introducing various changes, such as renaming variables, altering function calls, or modifying control flow statements. To facilitate a systematic evaluation of complexity, we adopt an incremental approach to modifications. For instance, a case may be categorized based on the number of changes made: a Level 1 (L1) case involves one change, while a L9 case entails nine changes. This framework enables us to create a diverse set of test cases, progressively increasing complexity while ensuring that the original template patch remains inapplicable and that the underlying vulnerability persists.

This allows us to explore the impact of various modifications on the effectiveness of our repair tool.

Example of Artificially Generated Vulnerable Reuse Here we provide a detailed example of how we construct an artificial vulnerable case for the CVE-2020-8094 (we only provide Listings for L1, L3, L7 and L9):

- **L1 (1 change)**: Based on the template vulnerable file, a single modification, such as renaming a variable (e.g., renaming `input_len` to `data_length` as shown in listing 5.1).
- **L2 (2 changes)**: Based on L1, adds a second modification, such as changing a function call (e.g., modifying `BestEffortAbort` to `EmergencyAbort`).
- **L3 (3 changes)**: Based on L2, introduces a new condition to an existing control flow statement (e.g., adding a check for `output == nullptr` as shown in listing 5.2).
- **L4 (4 changes)**: Based on L3, modifies the error message string (e.g., appending “critical failure” to the error message).
- **L5 (5 changes)**: Based on L4, changes the return value of a variable (e.g., altering `result = 0` to `result = -1`).
- **L6 (6 changes)**: Based on L5, adds or modifies comments explaining critical logic (e.g., adding a comment about checking output validity).
- **L7 (7 changes)**: Based on L6, alters the function signature by changing the type of the `input` variable (e.g., from `const char*` to `const void*` as shown in listing 5.3).
- **L8 (8 changes)**: Based on L7, modifies additional logic, such as changing how `tmp_output_len` is calculated (e.g., `tmp_output_len = *output_len / 2`).
- **L9 (9 changes)**: Based on L8, modifies more complex structures, such as changing the type of the `output` parameter (e.g., from `char **output` to `char *output[]` as shown in listing 5.4).

Note that it does not matter if the 9 mutated cases do not compile, as the primary focus of these artificial cases is to explore the limits of the tool’s repair capability.

Listing 5.1: The diff between the L1 modified vulnerable reuse and original version of the CVE-2020-8094

```
--- ./OLD##CVE-2020-8904##0##google@@asylo##ecalls.cc
+++ ./level1_modified_ecalls.c

 // Invokes the enclave restoring entry-point. Returns a non-zero error
    ↪ code on
 // failure.
-int ecall_restore(const char *input, uint64_t input_len, char **output,
+int ecall_restore(const char *input, uint64_t data_length, char **output,
                   uint64_t *output_len) {
   if (!asylo::primitives::TrustedPrimitives::IsOutsideEnclave(input,
-                                                              input_len))
   ↪  {
+                                                              data_length
   ↪ )) {
     asylo::primitives::TrustedPrimitives::BestEffortAbort(
         "ecall_restore: input found to not be in untrusted memory.");
   }
```

Listing 5.2: The diff between the L3 modified vulnerable reuse and original version of the CVE-2020-8094

```
--- ./OLD##CVE-2020-8904##0##google@@asylo##ecalls.cc
+++ ./level3_modified_ecalls.c

@@ -59,12 +59,13 @@

 // Invokes the enclave restoring entry-point. Returns a non-zero error
    ↪ code on
 // failure.
-int ecall_restore(const char *input, uint64_t input_len, char **output,
+int ecall_restore(const char *input, uint64_t data_length, char **output,
                   uint64_t *output_len) {
   if (!asylo::primitives::TrustedPrimitives::IsOutsideEnclave(input,
-                                                              input_len))
   ↪  {
-    asylo::primitives::TrustedPrimitives::BestEffortAbort(
-        "ecall_restore: input found to not be in untrusted memory.");
+                                                              data_length
   ↪ ) ||
+      output == nullptr) {
+    asylo::primitives::TrustedPrimitives::EmergencyAbort(
+        "ecall_restore: input or output is invalid.");
   }
   int result = 0;
   size_t tmp_output_len;
```

Listing 5.3: The diff between the L7 modified vulnerable reuse and original version of the CVE-2020-8094

```
--- ./OLD##CVE-2020-8904##0##google@@asylo##ecalls.cc
+++ ./level7_modified_ecalls.c

@@ -59,14 +59,16 @@

 // Invokes the enclave restoring entry-point. Returns a non-zero error
    ↪ code on
 // failure.
-int ecall_restore(const char *input, uint64_t input_len, char **output,
+int ecall_restore(const void *data, uint64_t data_length, char **output,
                   uint64_t *output_len) {
-  if (!asylo::primitives::TrustedPrimitives::IsOutsideEnclave(input,
-                                                              input_len))
   ↪ {
-    asylo::primitives::TrustedPrimitives::BestEffortAbort(
-        "ecall_restore: input found to not be in untrusted memory.");
+  // Critical check for output validity
+  if (!asylo::primitives::TrustedPrimitives::IsOutsideEnclave(data,
+                                                              data_length
   ↪ ) ||
+      output == nullptr) {
+    asylo::primitives::TrustedPrimitives::EmergencyAbort(
+        "Critical failure: input or output is invalid.");
   }
-  int result = 0;
+  int result = -1;
   size_t tmp_output_len;
   try {
     result = asylo::Restore(input, static_cast<size_t>(input_len), output
        ↪ ,
```

Listing 5.4: The diff between the L9 modified vulnerable reuse and original version of the CVE-2020-8094

```
--- ./OLD##CVE-2020-8904##0##google@@asylo##ecalls.cc
+++ ./level9_modified_ecalls.c
@@ -59,15 +59,17 @@

 // Invokes the enclave restoring entry-point. Returns a non-zero error
    ↪ code on
 // failure.
-int ecall_restore(const char *input, uint64_t input_len, char **output,
+int ecall_restore(const void *data, uint64_t data_length, char *output[],
                   uint64_t *output_len) {
-  if (!asylo::primitives::TrustedPrimitives::IsOutsideEnclave(input,
-                                                              input_len))
   ↪  {
-    asylo::primitives::TrustedPrimitives::BestEffortAbort(
-        "ecall_restore: input found to not be in untrusted memory.");
+  // Critical check for output validity
+  if (!asylo::primitives::TrustedPrimitives::IsOutsideEnclave(data,
+                                                              data_length
   ↪ ) ||
+      output == nullptr) {
+    asylo::primitives::TrustedPrimitives::EmergencyAbort(
+        "Critical failure: input or output is invalid.");
   }
-  int result = 0;
-  size_t tmp_output_len;
+  int result = -1;
+  size_t tmp_output_len = *output_len / 2;
   try {
     result = asylo::Restore(input, static_cast<size_t>(input_len), output
        ↪ ,
                             &tmp_output_len);
```

	L1	L2	L3	L4	L5	L6	L7	L8	L9
CVE-2020-8904	o	o	o	o	o	o	o	o	o
CVE-2020-8905	o	o	o	o	o	o	o	o	o
CVE-2021-3450	o	o	o	o	o	o	o	o	o
CVE-2021-32055	x	x	x	x	x	x	x	x	x
CVE-2021-37690	o	o	o	o	o	o	o	o	o
CVE-2021-41099	o	o	o	o	o	o	o	o	o
CVE-2022-36879	o	o	o	o	o	o	o	o	o
CVE-2022-34927	o	o	o	o	o	o	o	o	o
CVE-2019-16161	o	o	o	o	o	o	o	o	o
CVE-2019-19061	o	o	o	o	o	o	o	o	o
#Complete/Total Patches	9/10	9/10	9/10	9/10	9/10	9/10	9/10	9/10	9/10

Table 5.6: Repair results for the artificial vulnerable reuses modified in 9 levels based on 10 CVEs. ‘x’ represents that the patch generated by **1DRep** is incomplete, while ‘o’ represents completely patched.

5.5.2. Evaluation

Methodology. To assess the limits of 1DRep’s repair capability, we evaluated its performance on 90 artificially generated vulnerable reuses, which were created by modifying 10 CVEs across 9 complexity levels as described in section 5.5.1. These artificial cases were designed to progressively challenge the tool’s ability to recognize and repair modified vulnerabilities. We follow the same criteria as previously described in section 5.4 (the RQ2) to determine the correctness of generated patches: a fix is considered complete only if it correctly addresses all the vulnerable functions without changing the customized functionalities. Other partial fixes or fixes that change the customizations are not regarded as successful. Note that all patches are assessed manually for correctness, without testing their compilability, as the primary focus of these artificial cases is to explore the limits of the tool’s repair capability.

Results. As shown in table 5.6, **1DRep** generated 81 complete patches for the 90 artificial vulnerable reuses, achieving a 90% precision rate. (‘x’ represents that the patch generated by **1DRep** is incomplete, while ‘o’ represents completely patched.) Remarkably, **1DRep** failed to repair the 9 artificial vulnerable reuses derived from CVE-2021-32055. The template patch of this CVE is visualized in figure 5.6 by diffing. Next, the diff results obtained by comparing the L1, L6, and L9 cases with the template vulnerable file (CVE-2021-32055) are visualized in figure 5.7, figure 5.8 and figure 5.9, respectively. Finally, we only visualized the patch results for the L1 and L6 in figure 5.10 and figure 5.11 respectively by diffing. This is because the program failed to identify a similar vulnerable code in the L8 and L9 cases as explained further in the following analysis.

Additionally, **1DRep** fails to repair the L6 artificial vulnerable reuse of CVE-2020-8905.

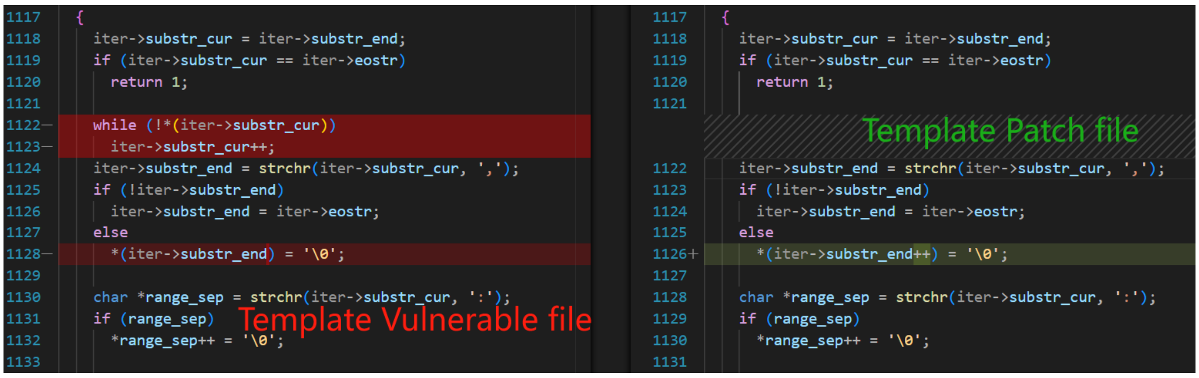
Analysis. We examined all generated patches for the L1 through L9 cases to investigate the reasons for failure. For the L1 and L2 cases, generated patches for them were identical and both failed to remove the vulnerable code. They only provided a partial fix by correctly addressing one patch line in both cases as shown in figure 5.10. We found this failure occurred because changes to variable names in the vulnerable lines led to incomplete context being extracted for the prompt. As a result, **1DRep** failed to fully recognize that the original vulnerability persisted. This finding highlights that the accuracy of **1DRep** is highly sensitive to the precision of the extracted context. While adjusting parameters like context size and sliding window size may offer partial solutions, the two cases suggest a need for future work to develop a more robust code extraction technique to ensure that generated prompts contain complete information necessary to repair the vulnerability.

For the L3 through L7 cases, all the identified insertion points and the generated patches were identical and entirely incorrect as shown in figure 5.11. In the L8 and L9 cases, no patches were generated because the program failed to identify any similar vulnerable code lines and erroneously concluded that no patch was necessary. This issue arises because, while the target files still contain vulnerable code, the surrounding context has changed significantly—a result of the “no match”” condition previously discussed in section 4.4.2).

The findings on the 7 cases highlights that the complexity of modifications can significantly affect the repair capability, prompting the need for handling of more complex modifications as future works.

The failure at level 6 of CVE-2020-8905 highlights a significant issue: the accuracy of GPT-4o's repair suggestions is not stable and might vary considerably as the complexity of the modifications increases.

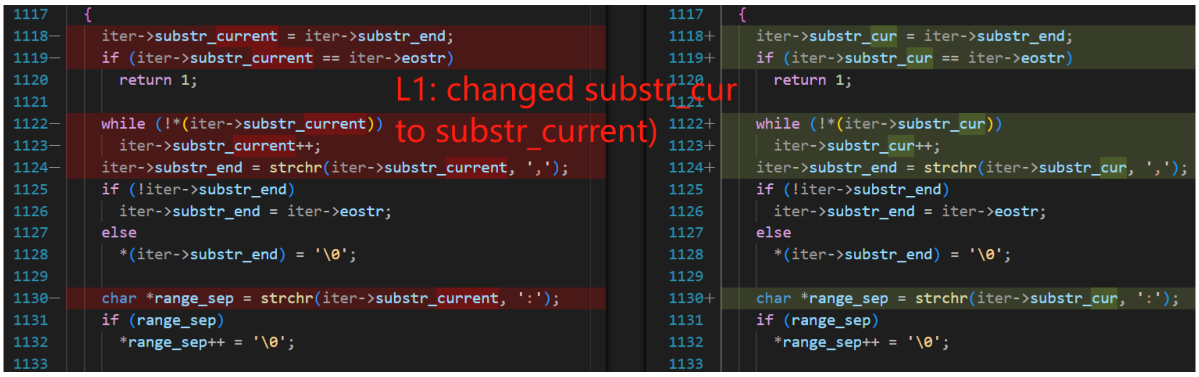
Answer to RQ3. As shown in table 5.6, **1DRep** successfully repaired 81 out of 90 artificial vulnerable cases (90%), which are generated by modifying 10 CVEs in 9 levels progressively as described in section 5.5.1. However, the failure with CVE-2021-32055 highlights the need for better context extraction and handling of more complex modifications, pointing to potential areas for future improvement.



```
{
  iter->substr_cur = iter->substr_end;
  if (iter->substr_cur == iter->eostr)
    return 1;

  while (!*(iter->substr_cur))
    iter->substr_cur++;
  iter->substr_end = strchr(iter->substr_cur, ',');
  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end) = '\0';

  char *range_sep = strchr(iter->substr_cur, ':');
  if (range_sep)
    *range_sep++ = '\0';
```

```
{
  iter->substr_cur = iter->substr_end;
  if (iter->substr_cur == iter->eostr)
    return 1;

  iter->substr_end = strchr(iter->substr_cur, ',');
  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end++) = '\0';

  char *range_sep = strchr(iter->substr_cur, ':');
  if (range_sep)
    *range_sep++ = '\0';
```

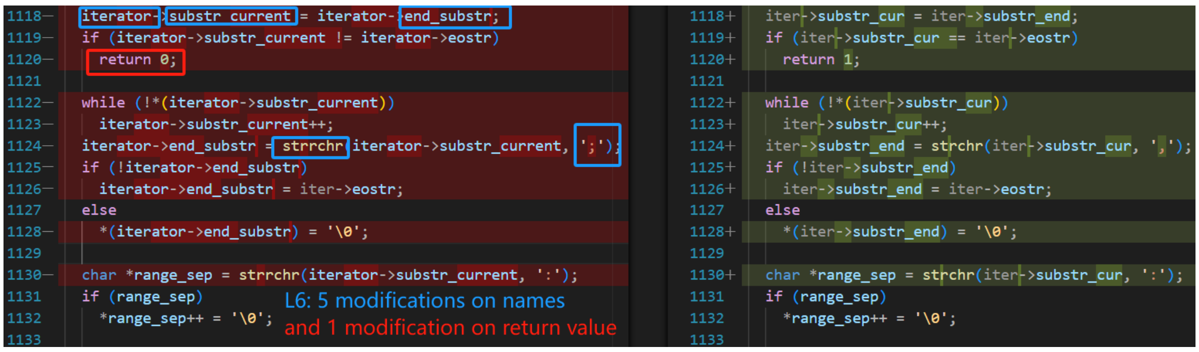
Figure 5.6: The template patch for CVE-2021-32055. (All generated patches failed to repair the 9 artificial cases derived from this CVE)



```
{
  iter->substr_current = iter->substr_end;
  if (iter->substr_current == iter->eostr)
    return 1;

  while (!*(iter->substr_current))
    iter->substr_current++;
  iter->substr_end = strchr(iter->substr_current, ',');
  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end) = '\0';

  char *range_sep = strchr(iter->substr_current, ':');
  if (range_sep)
    *range_sep++ = '\0';
```

```
{
  iter->substr_cur = iter->substr_end;
  if (iter->substr_cur == iter->eostr)
    return 1;

  while (!*(iter->substr_cur))
    iter->substr_cur++;
  iter->substr_end = strchr(iter->substr_cur, ',');
  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end) = '\0';

  char *range_sep = strchr(iter->substr_cur, ':');
  if (range_sep)
    *range_sep++ = '\0';
```

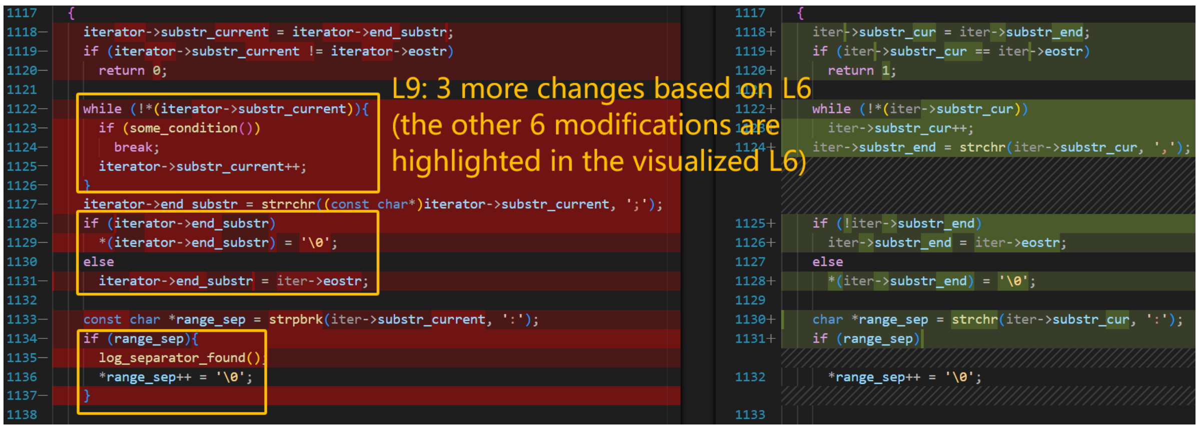
Figure 5.7: The diff result showing changes between the L1 case and the CVE-2021-32055.


L6: 5 modifications on names
and 1 modification on return value

```
  iterator->substr_current = iterator->end_substr;
  if (iterator->substr_current != iterator->eostr)
    return 0;

  while (!*(iterator->substr_current))
    iterator->substr_current++;
  iterator->end_substr = strrchr(iterator->substr_current, ';');
  if (!iterator->end_substr)
    iterator->end_substr = iter->eostr;
  else
    *(iterator->end_substr) = '\0';

  char *range_sep = strrchr(iterator->substr_current, ':');
  if (range_sep)
    *range_sep++ = '\0';
```

```
  iter->substr_cur = iter->substr_end;
  if (iter->substr_cur == iter->eostr)
    return 1;

  while (!*(iter->substr_cur))
    iter->substr_cur++;
  iter->substr_end = strchr(iter->substr_cur, ',');
  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end) = '\0';

  char *range_sep = strchr(iter->substr_cur, ':');
  if (range_sep)
    *range_sep++ = '\0';
```

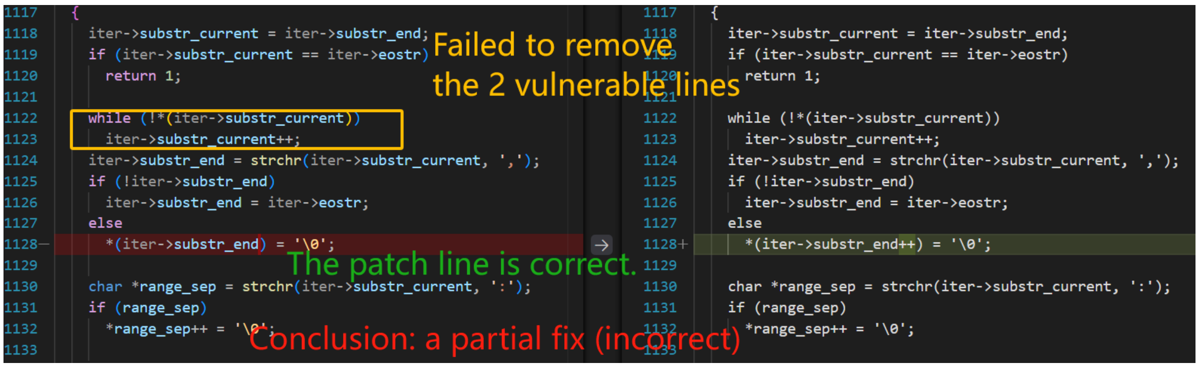
Figure 5.8: The diff result showing changes between the L6 case and the CVE-2021-32055.


L9: 3 more changes based on L6
(the other 6 modifications are
highlighted in the visualized L6)

```
{
  iterator->substr_current = iterator->end_substr;
  if (iterator->substr_current != iterator->eostr)
    return 0;

  while (!*(iterator->substr_current)){
    if (some_condition())
      break;
    iterator->substr_current++;
  }
  iterator->end_substr = strrchr((const char*)iterator->substr_current, ';');
  if (iterator->end_substr)
    *(iterator->end_substr) = '\0';
  else
    iterator->end_substr = iter->eostr;

  const char *range_sep = strpbrk(iter->substr_current, ':');
  if (range_sep){
    log_separator_found();
    *range_sep++ = '\0';
  }

```

```
{
  iter->substr_cur = iter->substr_end;
  if (iter->substr_cur == iter->eostr)
    return 1;

  while (!*(iter->substr_cur))
    iter->substr_cur++;
  iter->substr_end = strchr(iter->substr_cur, ',');

  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end) = '\0';

  char *range_sep = strchr(iter->substr_cur, ':');
  if (range_sep)

    *range_sep++ = '\0';

```

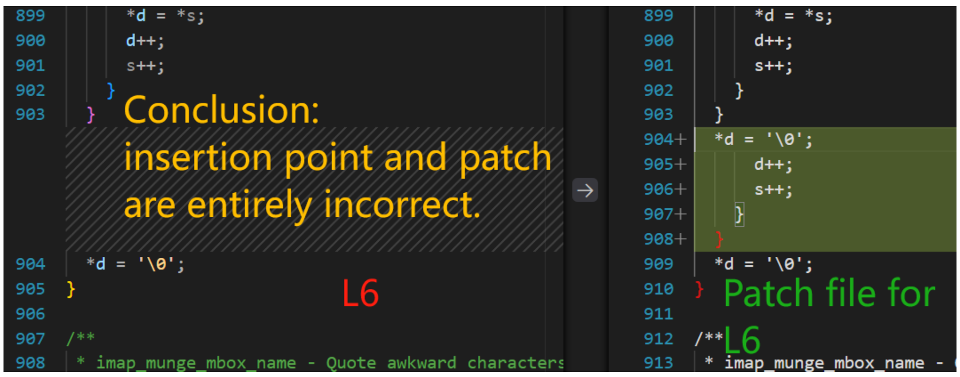
Figure 5.9: The diff result showing changes between the L9 case and the CVE-2021-32055 (the visualized L6 is shown in figure 5.8).


Failed to remove
the 2 vulnerable lines
The patch line is correct.
Conclusion: a partial fix (incorrect)

```
{
  iter->substr_current = iter->substr_end;
  if (iter->substr_current == iter->eostr)
    return 1;

  while (!*(iter->substr_current))
    iter->substr_current++;
  iter->substr_end = strchr(iter->substr_current, ',');
  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end) = '\0';

  char *range_sep = strchr(iter->substr_current, ':');
  if (range_sep)
    *range_sep++ = '\0';

```

```
{
  iter->substr_current = iter->substr_end;
  if (iter->substr_current == iter->eostr)
    return 1;

  while (!*(iter->substr_current))
    iter->substr_current++;
  iter->substr_end = strchr(iter->substr_current, ',');
  if (!iter->substr_end)
    iter->substr_end = iter->eostr;
  else
    *(iter->substr_end++) = '\0';

  char *range_sep = strchr(iter->substr_current, ':');
  if (range_sep)
    *range_sep++ = '\0';

```

Figure 5.10: The diff result showing changes between the L1 case and the generated patch for it.



```
      *d = *s;
      d++;
      s++;
    }
  }
  *d = '\0';
}

/**
 * imap_munge_mbox_name - Quote awkward characters
```

```
      *d = *s;
      d++;
      s++;
    }
  }
  *d = '\0';
      d++;
      s++;
    }
  }
  *d = '\0';
}

/**
 * imap_munge_mbox_name -
```

Figure 5.11: The diff result showing changes between the L6 case and the generated patch for it.

5.6. Security Reports

To assess how developers might respond to the proposed repair suggestions, we provided security reports for all 15 TP modified CVEs across 5 IoT projects on GitHub. These reports were submitted by either creating pull requests or opening issues, providing the necessary information. The CVEs, corresponding IoT projects, and links to the associated issues or pull requests are detailed in table 5.7. In conducting our assessment of developer responses to proposed security repairs, we initially opted to report vulnerabilities as issues. This decision was influenced by the presence of a structured bug report template available in the AliOS-Things project, which provided a standardized format for conveying bug-related information. Therefore, we assumed that the project maintainers might prefer this format. Then to maintain uniformity in reporting and ensure that each project received the vulnerability report in a consistent manner, we continued creating issues across all subsequent projects. Additionally, we assumed that creating issues typically invite a more open discussion about possible solutions without assuming the patch provided is final.

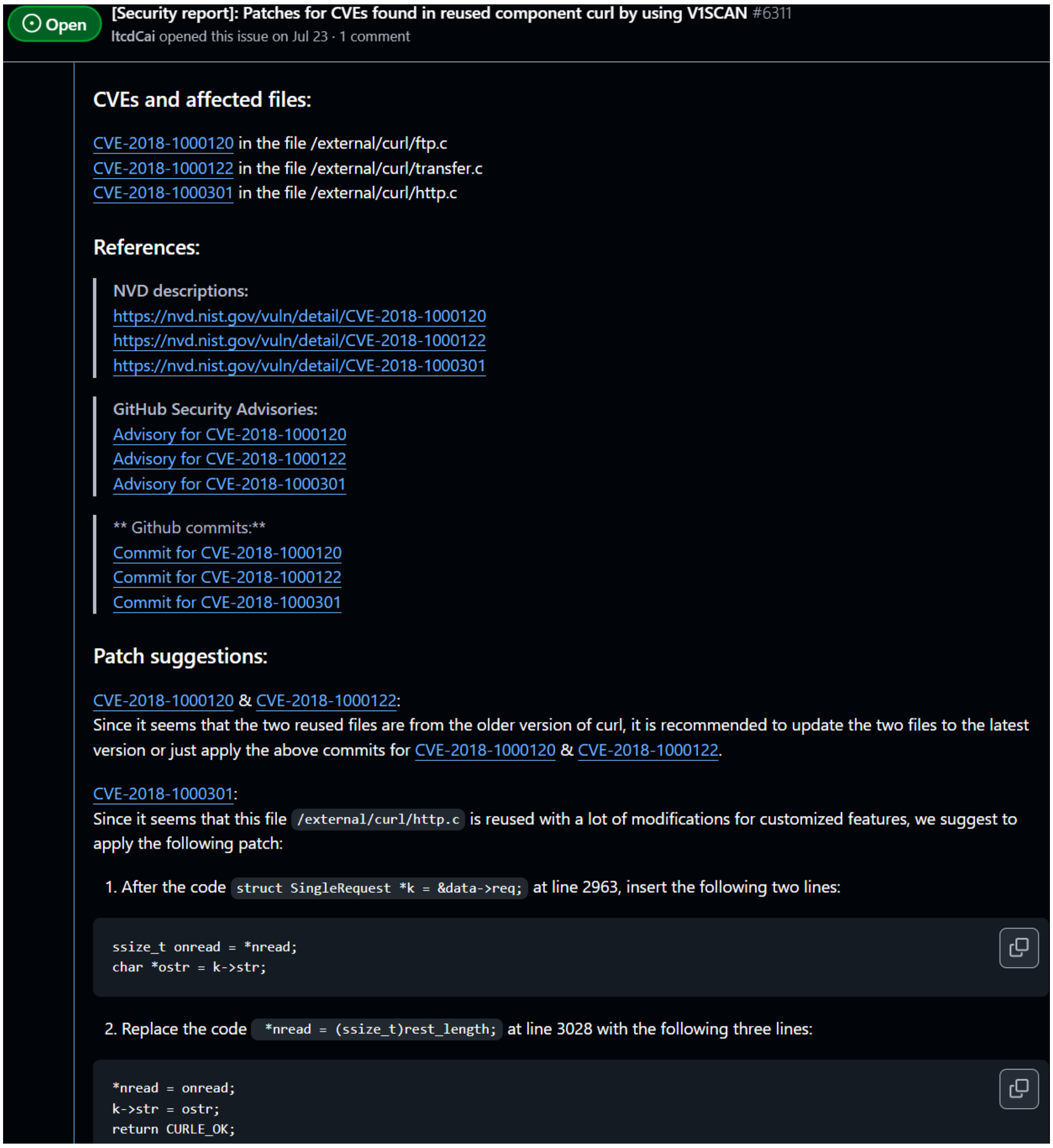
For the HarmonyOS project, however, an issue reporting option was unavailable, which necessitated submitting a pull request as the only viable reporting method.

Notably, four of the five projects containing the 15 TP-modified CVEs are coincidentally Chinese. To better capture the developers' attention and encourage a quicker response, we provided reports in Chinese for these projects. For the single project in English, TizenRT, we correspondingly wrote the report in English.

As illustrated in figure 5.12, an example of a security report, we included comprehensive details for the developers including (1) the NVD description, (2) GitHub Security Advisories, (3) GitHub commits and (4) customized patch suggestions to fit the context in their project.

However, despite creating reports, we received no responses from any of the IoT projects except for TizenRT, which assigned a developer to investigate the issues. We believe there are three primary reasons for this:

- **Prioritization of issues:** Open-source projects often prioritize feature development over security concerns, especially if those vulnerabilities are perceived as having a low or non-immediate impact on the system's functionality. If the vulnerabilities were not seen as critical, the developers might deprioritize addressing them.
- **Low level of maintenance:** Some IoT open-source projects may have low levels of active maintenance, so the reports might not receive attention.
- **Security expertise and dependencies:** Some IoT projects may lack contributors with expertise in security, making it difficult for them to quickly assess and apply the suggested patches. Additionally, since the developers reused third-party OSS components, they may prefer to wait for upstream fixes rather than implement custom patches themselves.



Open [Security report]: Patches for CVEs found in reused component curl by using V1SCAN #6311
ltcdCai opened this issue on Jul 23 · 1 comment

CVEs and affected files:

CVE-2018-1000120 in the file /external/curl/ftp.c
CVE-2018-1000122 in the file /external/curl/transfer.c
CVE-2018-1000301 in the file /external/curl/http.c

References:

> NVD descriptions:
> https://nvd.nist.gov/vuln/detail/CVE-2018-1000120
> https://nvd.nist.gov/vuln/detail/CVE-2018-1000122
> https://nvd.nist.gov/vuln/detail/CVE-2018-1000301

> GitHub Security Advisories:
> Advisory for CVE-2018-1000120
> Advisory for CVE-2018-1000122
> Advisory for CVE-2018-1000301

> ** Github commits:**
> Commit for CVE-2018-1000120
> Commit for CVE-2018-1000122
> Commit for CVE-2018-1000301

Patch suggestions:

CVE-2018-1000120 & CVE-2018-1000122:
Since it seems that the two reused files are from the older version of curl, it is recommended to update the two files to the latest version or just apply the above commits for CVE-2018-1000120 & CVE-2018-1000122.

CVE-2018-1000301:
Since it seems that this file `/external/curl/http.c` is reused with a lot of modifications for customized features, we suggest to apply the following patch:

1. After the code `struct SingleRequest *k = &data->req;` at line 2963, insert the following two lines:

```
ssize_t onread = *nread;
char *ostr = k->str;
```

2. Replace the code `*nread = (ssize_t)rest_length;` at line 3028 with the following three lines:

```
*nread = onread;
k->str = ostr;
return CURLE_OK;
```

Figure 5.12: Example of a security report for TizenRT.

Project	CVE ID	Links
AliOS-Things	CVE-2019-13616 CVE-2020-8177 CVE-2020-8169 CVE-2019-14906	https://github.com/alibaba/AliOS-Things/issues/2018 https://github.com/alibaba/AliOS-Things/issues/2019 https://github.com/alibaba/AliOS-Things/issues/2020 https://github.com/alibaba/AliOS-Things/issues/2021
TizenRT	CVE-2020-26243 CVE-2020-5235 CVE-2018-1000120 CVE-2018-1000122 CVE-2018-1000301	https://github.com/Samsung/TizenRT/issues/6311 https://github.com/Samsung/TizenRT/issues/6312
HarmonyOS	CVE-2018-9988	https://github.com/Awesome-HarmonyOS/HarmonyOS/pull/120
OpenHarmony	CVE-2018-9988 CVE-2019-16910 CVE-2021-3711 CVE-2021-22901	https://github.com/fenwii/OpenHarmony/issues/5 https://github.com/fenwii/OpenHarmony/issues/4 https://github.com/fenwii/OpenHarmony/issues/3
FaseBee	CVE-2020-5235	https://github.com/kerwincui/FastBee/issues/17

Table 5.7: The security reports for the 15 TP modified CVEs in 5 IoT projects.

6 Discussion

In this chapter, we discuss the significance and limitations of **1DRep** and explore potential future work that can enhance its effectiveness and applicability.

6.1. Significance of the Repair Tool 1DRep

The development of **1DRep** addresses a critical need in software security by automating the detection and repair of 1-day vulnerabilities in reused C/C++ IoT OSS components.

6.1.1. Handling Modified Reused Code

A key challenge in repairing vulnerabilities in reused code is handling cases where the reused code has been modified by developers, rendering standard patches ineffective. **1DRep** addresses this challenge by employing a code-based repair strategy that leverages GPT-4o to generate context-aware patches. This approach enables the tool to repair modified code that traditional patching methods would fail to address, thus expanding the scope and effectiveness of automated vulnerability repair.

6.1.2. Contribution to IoT Security

Many IoT devices run on software that includes OSS components, which may not receive timely updates. **1DRep** specifically explored this domain by detecting and repairing 40 IoT projects, showing the ability of automating the repair of vulnerabilities in reused C/C++ IoT OSS components. This contribution shows the potential to mitigate the risks associated with the proliferation of vulnerable IoT devices.

6.2. Limitations

While our research presents a novel approach to automated vulnerability repair, there are inherent limitations that must be acknowledged.

6.2.1. Applicability to Short Patches

Our tool is currently most effective for vulnerabilities requiring short patches. This limitation arises from the challenges associated with processing and generating repairs for longer code segments, both in terms of context extraction and the capabilities of the GPT-4o. Complex vulnerabilities involving extensive code changes are not yet adequately addressed.

6.2.2. Dependence on V1SCAN's Performance

The efficacy of our repair tool is directly linked to the performance of V1SCAN. The low precision rate of V1SCAN in detecting vulnerabilities in IoT projects, limits the overall effectiveness of our approach. This dependence also means that undetected vulnerabilities cannot be repaired by our tool.

6.2.3. V1SCAN's limitation

As described in section 5.1 and section 5.3, the CVE dataset of V1SCAN is incomplete, and some bugs existed in their algorithm resulted the unexpected low performance of V1SCAN. Therefore, future works can be improving the detection tool and CVE dataset.

6.2.4. Assumptions on Code Customizations

Our approach assumes that developers make only minimal customizations to reused code. However, in practice, developers may significantly modify third-party code, affecting the applicability of our repair strategies. The artificial cases with customizations used in our evaluation are based on assumptions and may not fully represent real-world scenarios.

6.2.5. Limited Detected Dataset for Evaluation

Due to the small number of CVEs detected and the limited number of modified vulnerabilities (only 15), our evaluation may not capture the full spectrum of potential vulnerabilities and repair scenarios. As a result, the accuracy and generalizability of our tool may decrease when applied to a broader set of vulnerabilities.

6.2.6. Limitations of Code-Based Repair

Our code-based repair approach has specific limitations:

- **Dependency on GPT-4o Quality**: The success of the repair depends on the LLM's ability to generate correct and secure code.
- **Complex Code Structures**: Highly complex or obfuscated code may not yield high similarity scores, potentially missing some vulnerable code blocks.
- **Context Size Limitations**: Large code contexts may exceed the input limitations of GPT-4o, necessitating context reduction techniques.

6.3. Future work of the repair tool

While our repair tool demonstrates promising results in repairing 1-day vulnerabilities, there are several areas where further research and development can enhance its capabilities.

Repairing Complex Vulnerabilities. Currently, our tool is most effective in scenarios where the vulnerabilities and corresponding patches involve relatively short code segments. However, real-world vulnerabilities can often be complex, involving extensive code changes across multiple functions or modules.

To address this limitation, future work can focus on:

- **Handling Long Contexts:** Enhancing the tool's ability to process and repair vulnerabilities that require patches involving longer code segments. This may involve optimizing prompts for LLMs to handle larger inputs or repairing the vulnerabilities step by step by prompting the LLM multiple times.
- **Advanced Context Extraction:** Employing more advanced methods for extracting relevant code snippets could enhance the quality of the information provided to the LLM, leading to better repair suggestions even in complex scenarios. This could involve Semantic Code Analysis and Abstract Syntax Trees.

Benchmarking Different LLMs. Our current implementation only utilizes GPT-4o, but it might be useful to compare GPT-4o with other state-of-the-art models to evaluate their effectiveness in code repair tasks. This is because exploring other LLMs might provide insights into performance differences and potential improvements.

Better detection model. Since the precision of V1SCAN is low due to some bugs in its implementation and half of its CVE dataset disappeared, future efforts could be on developing a better detection tool that can detect 1-day vulnerabilities in reused OSS components or improving V1SCAN's performance by completing/expanding the CVE dataset, fixing its bugs and improving its mechanisms.

Expanding the CVE Dataset. To improve vulnerability detection, it is essential to have a comprehensive dataset. Future efforts could focus on building a larger CVE dataset particularly those relevant to IoT projects, to improve the performance of the detection tool.

Enhancing Prompt Engineering for GPT-4o. Optimizing how we interact with GPT-4o can lead to better repair suggestions. Therefore, developing more effective prompt structures might help in generating better patches. Also, since some vulnerabilities might contain too many vulnerable/patch lines, it might be helpful to adjust prompts based on the complexity or characteristics of the code being repaired.

Addressing Non-Function Vulnerabilities. While our tool currently focuses on repairing vulnerabilities within functions, V1SCAN can also detect vulnerabilities in structures, macros, and variables. Future development could expand the repair capabilities to address these types of vulnerabilities, thereby increasing the tool's overall utility.

Leveraging Mutation Testing Techniques. While our study utilizes artificially generated vulnerable reuses to evaluate the limits of **1DRep**'s repair capabilities, these synthetic changes closely align with the principles of mutation testing. Mutation testing is a well-established technique in software testing that involves introducing small, systematic modifications (mutations) to a program's source code to assess the effectiveness of test suites. This technique can generate a diverse set of challenging scenarios, providing a more robust evaluation of 1DRep's repair capacity. This is because manually generating synthetic data cannot fully capture the range of modifications found in real-world scenarios; the process is both time-consuming and prone to errors, making it challenging to create a sufficiently comprehensive dataset. Therefore, future work could leverage mutation testing techniques to:

1. **Systematic Generation of Mutations:** Future work could involve applying standard mutation operators—such as statement deletion, variable renaming, conditional negation, and control flow alterations—to generate a comprehensive set of mutated code samples. This systematic approach ensures a wide coverage of potential code variations that may occur in real-world software development.
2. **Guiding Targeted Improvements on LLM**: Certain mutations may introduce complexities that challenge the LLM's (currently GPT-4o) ability to generate appropriate patches. Identifying these challenges would provide insight into the LLM's limitations with specific code patterns.

   Recognizing patterns in LLM failures could allow adjustments to prompt strategies, enabling clearer, more targeted instructions to improve repair effectiveness.

   Furthermore, if certain types of mutations consistently cause issues, it might indicate a need to fine-tune the LLM with additional training data that includes these challenging code scenarios.

Developing Systematic Methods to Assess Patch Correctness. WWhile our current evaluation of patches generated by **1DRep** relies on manual inspection, we recognize the necessity of systematic methods to ensure comprehensive and objective assessment. Systematic evaluation is crucial for validating the effectiveness of GPT-4 in generating correct patches for modified vulnerabilities. Future works can focus on the following:

1. **Static Analysis Tools.** Static analysis tools can be employed to ensure that generated patches are syntactically correct and semantically coherent. It is also possible to use security-focused static analyzers to detect potential vulnerabilities that may have been introduced inadvertently.
2. **Standardized Evaluation Criteria:** A standardized checklist or set of criteria for reviewers can be developed to consistently assess the patches, focusing on correctness, completeness, and potential side effects.
3. **Automated Testing:** If test cases are available, **1DRep** can automatically run the test cases on both the vulnerable code and patched code to verify that the vulnerability is resolved. Additionally, this can ensure the generated patch does not introduce new issues.

7 Threats to validity

In this chapter, we discuss potential threats to the validity of our study.

7.1. Internal Threats to Validity

The following issues could affect the accuracy of our findings:

Limited Number of Detected Modified Vulnerabilities. The small number of CVEs detected, particularly the low number of true positive modified vulnerabilities (just 15), poses a threat to the internal validity of our evaluation. This limited sample size may not be representative of the diversity of vulnerabilities present in IoT projects, and as such, our findings may not generalize well to other contexts.

Artificial Cases. The artificially generated cases with customizations are generated manually based on our assumptions and selected CVEs rather than real-world data, which may not accurately reflect actual developer practices and introduce subjective biases or errors. Also, they may not include enough diverse or representative vulnerabilities, possibly biasing the repair effectiveness of 1DRep.

Dependency on V1SCAN and GPT-4o. If one of these tools fails, the repair process might be compromised. There is a risk in over-relying on the accuracy of V1SCAN for 1-day vulnerability detection and GPT-4o for code generation.

In addition, the assumptions we made during the experiments might not hold:

- **Short Security Patches are Prevalent**: We observed that most security patches in the CVE dataset involve short code segments, typically within a single function. We assumed this would also apply to IoT projects; however, although it is true in our 11 selected vulnerable IoT projects, with a larger dataset, the proportion of vulnerabilities requiring complex patches may increase.
- **Minimal Code Customizations**: We assumed that developers make only minor customizations to reused code. Although it is true in our experiments, in reality, customization levels might vary widely, potentially affecting the tool's effectiveness.

7.2. External Threats to Validity

Selection of IoT projects. Since we choose a small set of IoT projects (just 40) for testing, the detected vulnerabilities may not cover enough types of 1-day vulnerabilities in IoT projects, so they might not represent the vulnerabilities in general IoT projects, which potentially reduces the broader applicability of 1DRep.

The CVE dataset. It is difficult to maintain the CVE dataset in real time. Vulnerabilities are continuously being discovered, and there is often a delay between the discovery of a vulnerability and its inclusion in the CVE database. This introduces a lag in the identification and repair of emerging security issues, particularly in rapidly evolving fields like IoT.

7.2.1. Construct Validity

Our evaluation only used precision as the metric for assessing the detection tool, without considering recall due to the unknown total number of vulnerabilities in the target projects. Therefore, we might overlook situations where **1DRep** could fail to detect certain types of vulnerabilities, particularly if those vulnerabilities involve more complex modifications. Also, the increasing levels of modification might not accurately reflect real-world difficulty in fixing vulnerabilities.

8

Conclusion

In this thesis, we set out to address the challenge of automating the detection and repair of 1-day vulnerabilities in C/C++ software projects, with a particular focus on IoT projects that often rely on reused open-source software components.

To deal with this issue, we developed **1DRep**, an automated repair tool designed to detect and repair 1-day vulnerabilities by leveraging template security patches and a LLM GPT-4o. Our approach relies on the vulnerability detection tool V1SCAN. We apply V1SCAN to target projects to identify vulnerabilities, classifying them into two types: exactly reused and modified vulnerabilities. For exactly reused vulnerabilities, we use a version-based repair approach, while for modified vulnerabilities, we implement a code-based repair strategy, which heavily replies on GPT-4o. Finally, we provided security reports containing patches for the detected CVEs in the 11 IoT projects by creating GitHub issues or pull requests.

Our evaluation of **1DRep** demonstrates its effectiveness by achieving a 97.5% (39 out of 40) success rate in our 11 target vulnerable IoT projects. The tool shows particular strength in handling vulnerabilities involving short code segments, which are common in security patches. Moreover, by utilizing GPT-4o, **1DRep** is capable of generating patches for modified context, addressing scenarios where traditional patching methods fall short.

Despite the promising results, limitations such as reliance on the precision of the detection model and challenges with complex vulnerabilities highlight areas for future research. Enhancements to the underlying detection mechanisms, expanding the CVE dataset, and refining the code-based repair approach are critical steps to improve the tool's effectiveness.

In conclusion, **1DRep** is a promising APR tool for 1-day vulnerabilities in reused C/C++ IoT OSS components. By continuing to develop and refine such tools, we can better protect software systems against known vulnerabilities.

References

[1] Sicong Cao et al. “MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1456–1468. isbn: 9781450392211. doi: 10.1145/3510003.3510219. url: https://doi.org/10.1145/3510003.3510219.

[2] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. url: https://arxiv.org/abs/2107.03374.

[3] Z. Chen et al. “SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair”. In: *IEEE Transactions on Software Engineering* 47.09 (Sept. 2021), pp. 1943–1959. issn: 1939-3520. doi: 10.1109/TSE.2019.2940179.

[4] cisa.gov. *2022 Top Routinely Exploited Vulnerabilities*. 2023. url: https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-215a (visited on 01/16/2024).

[5] Ruian Duan et al. “Identifying Open-Source License Violation and 1-Day Security Risk at Large Scale”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2169–2185. isbn: 9781450349468. doi: 10.1145/3133956.3134048. url: https://doi.org/10.1145/3133956.3134048.

[6] Ruian Duan et al. “Identifying Open-Source License Violation and 1-day Security Risk at Large Scale”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017). url: https://api.semanticscholar.org/CorpusID:7402387.

[7] Ruian Duan et al. “Identifying Open-Source License Violation and 1-day Security Risk at Large Scale”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2169–2185. isbn: 9781450349468. doi: 10.1145/3133956.3134048. url: https://doi.org/10.1145/3133956.3134048.

[8] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. url: https://arxiv.org/abs/2002.08155.

[9] Kai Huang et al. “A Survey on Automated Program Repair Techniques”. In: *arXiv preprint arXiv:2303.18184* (2023).

[10] Imperva. *What is the Common Vulnerabilities and Exposures (CVE) Glossary*. 2023. url: https://www.imperva.com/learn/application-security/cve-cvss-vulnerability/#:~:text=CVE%20stands%20for%20Common%20Vulnerabilities,threat%20level%20of%20a%20vulnerability. (visited on 09/29/2024).

[11] Nan Jiang, Thibaud Lutellier, and Lin Tan. “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021. doi: 10.1109/icse43902.2021.00107. url: http://dx.doi.org/10.1109/ICSE43902.2021.00107.

[12] Yanjie Jiang et al. “Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 686–698. doi: 10.1109/ICSE43902.2021.00069.

[13] S. Kim et al. “VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2017, pp. 595–614. doi: 10.1109/SP.2017.62. url: https://doi.ieeecomputersociety.org/10.1109/SP.2017.62.

[14] S. Kwon et al. “OCTOPOCS: Automatic Verification of Propagated Vulnerable Code Using Reformed Proofs of Concept”. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2021, pp. 174–185. doi: 10.1109/DSN48987.2021.00032. url: https://doi.ieeecomputersociety.org/10.1109/DSN48987.2021.00032.

[15] Menghao Li et al. “LibD: scalable and precise third-party library detection in android markets”. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 335–346. isbn: 9781538638682. doi: 10.1109/ICSE.2017.38. url: https://doi.org/10.1109/ICSE.2017.38.

[16] Yi Li, Shaohua Wang, and Tien N. Nguyen. “DLFix: Context-based Code Transformation Learning for Automated Program Repair”. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 602–614.

[17] Knud Lasse Lueth. *State of IoT Spring 2024: 10 emerging IoT trends driving market growth*. 2024. url: https://iot-analytics.com/state-of-iot-10-emerging-iot-trends-driving-market-growth/ (visited on 08/16/2024).

[18] Knud Lasse Lueth. *Top 5 enterprise technology priorities: AI on the rise, but cybersecurity remains on top*. 2024. url: https://iot-analytics.com/top-5-enterprise-technology-priorities/ (visited on 08/16/2024).

[19] Siqi Ma et al. “CDRep: Automatic Repair of Cryptographic Misuses in Android Applications”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS ’16. Xi’an, China: Association for Computing Machinery, 2016, pp. 711–722. isbn: 9781450342339. doi: 10.1145/2897845.2897896. url: https://doi.org/10.1145/2897845.2897896.

[20] Siqi Ma et al. “VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples”. In: *European Symposium on Research in Computer Security*. 2017. url: https://api.semanticscholar.org/CorpusID:12983722.

[21] Ehsan Mashhadi and Hadi Hemmati. *Applying CodeBERT for Automated Program Repair of Java Simple Bugs*. 2021. arXiv: 2103.11626 [cs.SE]. url: https://arxiv.org/abs/2103.11626.

[22] Na Meng et al. “Secure Coding Practices in Java: Challenges and Vulnerabilities”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 372–383. doi: 10.1145/3180155.3180201.

[23] Martin Monperrus. “Automatic Software Repair: A Bibliography”. In: *ACM Computing Surveys* 51.1 (Jan. 2018), pp. 1–24. issn: 1557-7341. doi: 10.1145/3105906. url: http://dx.doi.org/10.1145/3105906.

[24] Martin Monperrus. “The living review on automated program repair”. PhD thesis. HAL Archives Ouvertes, 2018.

[25] OpenAI. *GPT-4o*. Accessed: 2024-10-09. 2024. url: https://openai.com/gpt-4.

[26] Julian Aron Prenner and Romain Robbes. *Automatic Program Repair with OpenAI’s Codex: Evaluating QuixBugs*. 2021. arXiv: 2111.03922 [cs.SE]. url: https://arxiv.org/abs/2111.03922.

[27] Python. *DifflibSequenceMatcher*. 2023. url: https://docs.python.org/3/library/difflib.html (visited on 06/10/2024).

[28] Yuba Raj Siwakoti et al. “Advances in IoT Security: Vulnerabilities, Enabled Criminal Services, Attacks, and Countermeasures”. In: *IEEE Internet of Things Journal* 10.13 (2023), pp. 11224–11239. doi: 10.1109/JIOT.2023.3252594.

[29] Inc. Synopsys. *2023 Open Source Security and Risk Analysis Report*. 2023. url: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html (visited on 12/30/2023).

[30] Bill Toulas. *Google: Android patch gap makes n-days as dangerous as zero-days*. 2023. url: https://www.bleepingcomputer.com/news/security/google-android-patch-gap-makes-n-days-as-dangerous-as-zero-days/ (visited on 01/16/2024).

[31] Wikipedia. *Locality-sensitive hashing*. 2023. url: https://en.wikipedia.org/wiki/Locality-sensitive_hashing (visited on 10/25/2024).

[32] Seunghoon Woo. *Centris-public*. 2024. url: https://github.com/WOOSEUNGHOON/Centris-public (visited on 08/01/2024).

[33] Seunghoon Woo. *V1SCAN-public*. 2023. url: https://github.com/WOOSEUNGHOON/V1SCAN-public (visited on 12/31/2023).

[34] Seunghoon Woo et al. "Centris: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse". In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE '21. Madrid, Spain: IEEE Press, 2021, pp. 860–872. isbn: 9781450390859. doi: 10.1109/ICSE43902.2021.00083. url: https://doi.org/10.1109/ICSE43902.2021.00083.

[35] Seunghoon Woo et al. "MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3037–3053. isbn: 978-1-939133-31-1. url: https://www.usenix.org/conference/usenixsecurity22/presentation/woo.

[36] Seunghoon Woo et al. "V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6541–6556. isbn: 978-1-939133-37-3. url: https://www.usenix.org/conference/usenixsecurity23/presentation/woo.

[37] Martin Woodward. *Octoverse 2022: 10 years of tracking open source*. 2022. url: https://github.blog/2022-11-17-octoverse-2022-10-years-of-tracking-open-source/ (visited on 12/30/2023).

[38] Chunqiu Steven Xia and Lingming Zhang. *Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT*. 2023. arXiv: 2304.00385 [cs.SE]. url: https://arxiv.org/abs/2304.00385.

[39] Chunqiu Steven Xia and Lingming Zhang. "Less training, more repairing please: revisiting automated program repair via zero-shot learning". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 959–971. isbn: 9781450394130. doi: 10.1145/3540250.3549101. url: https://doi.org/10.1145/3540250.3549101.

[40] Yang Xiao et al. "MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC'20. USA: USENIX Association, 2020. isbn: 978-1-939133-17-5.

[41] Songtao Yang et al. "1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 867–879. isbn: 9798400702211. doi: 10.1145/3597926.3598102. url: https://doi.org/10.1145/3597926.3598102.

[42] Tatsuhiko Yasumatsu et al. "Understanding the Responsiveness of Mobile App Developers to Software Library Updates". In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. CODASPY '19. Richardson, Texas, USA: Association for Computing Machinery, 2019, pp. 13–24. isbn: 9781450360999. doi: 10.1145/3292006.3300020. url: https://doi.org/10.1145/3292006.3300020.

[43] Xian Zhan et al. "ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications". In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE '21. Madrid, Spain: IEEE Press, 2021, pp. 1695–1707. isbn: 9781450390859. doi: 10.1109/ICSE43902.2021.00150. url: https://doi.org/10.1109/ICSE43902.2021.00150.

[44] Xian Zhan et al. “ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications”. In: *Proceedings of the 43rd International Conference on Software Engineering*. ICSE ’21. Madrid, Spain: IEEE Press, 2021, pp. 1695–1707. isbn: 9781450390859. doi: 10.1109/ICSE43902.2021.00150. url: https://doi.org/10.1109/ICSE43902.2021.00150.

[45] Xian Zhan et al. “Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review”. In: *IEEE Transactions on Software Engineering* 48.10 (2022), pp. 4181–4213. doi: 10.1109/TSE.2021.3114381.

[46] Ying Zhang et al. “Example-Based Vulnerability Detection and Repair in Java Code”. In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ICPC ’22. Virtual Event: Association for Computing Machinery, 2022, pp. 190–201. isbn: 9781450392983. doi: 10.1145/3524610.3527895. url: https://doi.org/10.1145/3524610.3527895.

[47] Zicheng Zhang et al. “An Empirical Study of Potentially Malicious Third-Party Libraries in Android Apps”. In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec ’20. Linz, Austria: Association for Computing Machinery, 2020, pp. 144–154. isbn: 9781450380065. doi: 10.1145/3395351.3399346. url: https://doi.org/10.1145/3395351.3399346.

[48] Binbin Zhao et al. “A Large-Scale Empirical Study on the Vulnerability of Deployed IoT Devices”. In: *IEEE Transactions on Dependable and Secure Computing* 19.3 (2022), pp. 1826–1840. doi: 10.1109/TDSC.2020.3037908.

[49] Xin Zhou et al. *Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead*. 2024. arXiv: 2404.02525 [cs.SE]. url: https://arxiv.org/abs/2404.02525.

[50] Zhou Zhou et al. “SPVF: security property assisted vulnerability fixing via attention-based models”. In: *Empirical Softw. Engg.* 27.7 (Dec. 2022). issn: 1382-3256. doi: 10.1007/s10664-022-10216-4. url: https://doi.org/10.1007/s10664-022-10216-4.