PredicateFix: Repairing Static Analysis Alerts with Bridging Predicates

Yuan-An Xiao
Key Lab of HCST (PKU), MOE
Beijing, China
SCS, Peking University
Beijing, China
xiaoyuanan@pku.edu.cn

Weixuan Wang
Key Lab of HCST (PKU), MOE
Beijing, China
SCS, Peking University
Beijing, China
wangweixvan@gmail.com

Dong Liu
ZTE Coporation
Chengdu, China
liu.dong3@zte.com.cn

Junwei Zhou
ZTE Coporation
Chengdu, China
zhou.junwei2@zte.com.cn

Shengyu Cheng
ZTE Coporation
Chengdu, China
cheng.shengyu@zte.com.cn

Yingfei Xiong[*]
Key Lab of HCST (PKU), MOE
Beijing, China
SCS, Peking University
Beijing, China
xiongyf@pku.edu.cn

Abstract

Fixing static analysis alerts in source code with Large Language Models (LLMs) is becoming increasingly popular. However, LLMs often hallucinate and perform poorly for complex and less common alerts. Retrieval-augmented generation (RAG) techniques aim to solve this problem by providing the model with a relevant example, but existing approaches face the challenge of unsatisfactory quality of such examples.

To address this challenge, we utilize the predicates in the analysis rule, which serve as a bridge between the alert and relevant code snippets within a clean code corpus, called key examples. Based on this insight, we propose an algorithm to retrieve key examples for an alert automatically, and build PredicateFix as a RAG pipeline to fix alerts from two static code analyzers: CodeQL and GoInsight. Evaluation with multiple LLMs shows that PredicateFix increases the number of correct repairs by 27.1% ~ 69.3%, significantly outperforming other baseline RAG approaches.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Program analysis.*

Keywords

Automated Program Repair, Retrieval-Augmented Generation, Software Analysis

ACM Reference Format:
Yuan-An Xiao, Weixuan Wang, Dong Liu, Junwei Zhou, Shengyu Cheng, and Yingfei Xiong. 2026. PredicateFix: Repairing Static Analysis Alerts with Bridging Predicates. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3744916.3773159

1 Introduction

Fixing software bugs is a time-consuming job, and many automated program repair (APR) approaches have been proposed in recent decades [8, 14, 23, 28, 57, 66]. Among different types of APR approaches, repairing static analysis alerts [2, 15, 25, 50, 63] is of particular interest because static analyzers are widely used in the industry and dealing with their output has become a burden. Companies have integrated APR tools into their developing routine to fix alerts flagged by code analyzers such as Infer and FindBugs [27, 39].

Recently, LLMs have shown promising results in various coding tasks [24, 52, 56], and many existing studies have used LLMs to fix static analysis alerts [4, 25]. While LLMs are good at fixing many straightforward and common alerts (e.g., unused variables and null pointer exceptions), they suffer from the problem of hallucination and may generate incorrect results for complicated and less common alerts (e.g., uncommon cases of API misuse and complex security vulnerabilities). In particular, we observe that fixing many alerts requires domain-specific or even project-specific knowledge, such as invoking a custom sanitizer or passing a specific parameter. Such knowledge may be absent from the training set of LLMs.

Retrieval-augmented generation (RAG) is a popular technique to improve the performance of LLMs on unfamiliar topics by guiding LLMs with relevant *examples*. Many existing learning-based APR approaches guide the LLM with historical patches [25, 26, 53] or similar code snippets [55, 64]. However, the quality of such examples limits their effectiveness. It is challenging to find examples highly relevant to the expected patch, particularly for less common alerts where the LLM does not have enough knowledge.

We address this challenge with a novel RAG approach that retrieves example code snippets to guide the LLM to generate a patch. The core problem is how we can know that an example code snippet contains the required knowledge to guide the fix, i.e., a *key example*. We utilize the definition of static analysis rules to address this challenge. Our novel insight is that some predicates in the analysis rule, i.e. *bridging predicates*, can serve as a bridge between the alert

[*] Corresponding author.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773159

and key examples. We will illustrate later that a code snippet is a key example if, by negating a bridging predicate, the alert to fix disappears and a new alert on the example code snippet appears. In other words, the key example contains the code change to fix the alert, as identified by the bridging predicate. We propose an algorithm to identify bridging predicates and key examples.

Based on the above insight, we build a RAG pipeline that collects a corpus of clean code, retrieves key examples in the corpus, and then prompts the LLM with the examples to enhance its repair capability. We implemented this approach to fix security vulnerabilities as reported by CodeQL [10], a popular Datalog-based static code analyzer, and GoInsight, an internal static code analyzer in ZTE corporation. We evaluated PredicateFix with different LLMs, and the experiment result shows that it can increase the effectiveness of these LLMs by a large number (27.1% ∼ 69.3%), significantly outperforming other baseline RAG techniques.

The contributions of this paper are summarized as follows:

- A novel approach to identify key examples for program repair that utilizes the internal state (predicates) of the static analyzer.
- An automatic RAG pipeline based on the above approach, which significantly increases the effectiveness of multiple LLMs, as our experimental evaluation shows.
- The cross-language dataset consisting of security vulnerabilities in CVE that CodeQL can detect, which is useful in future studies.

The remainder of this paper is organized as follows. Section 2 introduces a vulnerability detected by static analysis as a running example to motivate our approach. Section 3 illustrates the concept of predicates and bridging predicates, proposes an algorithm to automatically identify key examples based on them, and then describes PredicateFix – an end-to-end RAG pipeline that utilizes this algorithm. Section 4 describes the implementation of PredicateFix on CodeQL and GoInsight. Section 5 presents the experimental setup and results of our evaluation. Section 6 compares our approach to related work. Section 7 concludes this paper.

2 Motivation and Approach Overview

2.1 The Running Example

In Figure 1, we use the security vulnerability CVE-2020-13946 [49] in the Apache Cassandra database software to motivate our approach. The vulnerability lies in a Java method that misconfigures an `RMIConnectorServer` object to allow any type of object for credentials, leaving the chance of executing malicious code through deserialization. The CodeQL code analyzer [10] can detect this with the rule `InsecureRmiJmxEnvironmentConfiguration.ql`. It reports an alert to the user about this vulnerability, as shown below:

[Name] InsecureRmiJmxAuthenticationEnvironment
[Severity] error
[Description] This query detects if a JMX/RMI server is created with a potentially dangerous environment, which could lead to code execution through insecure deserialization.

Figure 1 also shows the developer's patch for this vulnerability, where the deleted lines are colored red and the added lines are colored green. The patch adds a configuration `"jmx.remote.rmi.server.credential.types"` to the variable `env` that will allow only a string or an array of strings as the credential, preventing the chance of code execution through insecure deserialization.

The developer's patch indicates that fixing this alert requires knowledge of the specific configuration. Although the configuration is from the official Java SE platform, its usage is sparse in the public domain. On GitHub, searching for this configuration key only returns 87 Java code snippets and 1 Pull Request. As a result, even recent LLMs such as GPT-4o [42] and Claude-3.5-Sonnet [1] cannot correctly fix this vulnerability.

The need for such uncommon knowledge is unfortunately common when fixing static analysis alerts. As our evaluation will show, in our benchmark based on 6027 vulnerabilities in the CVE dataset, over 60% of the alert types appear only once, indicating that they are not frequent in the public domain. Based on the situation in ZTE Corporation, it is also common that an organization or a project defines its own coding rules involving organization- or project-specific knowledge, such as calling a customized sanitizer. LLMs will not learn such knowledge from public training sets.

To enhance the effectiveness of LLMs in this scenario, researchers have proposed various prompt engineering or RAG approaches to provide the model with additional knowledge. However, none of them works for this running example, which we will elaborate on later. In the following subsections, we go through these existing approaches, analyze why they fail, and propose our approach.

2.2 Existing Prompt Engineering Approaches

Many existing LLM-based program repair approaches [51, 57, 62] prompt the LLM with the natural language description of the alert and the context of the code (typically captured as multiple lines above and below the bug location). However, the vulnerability description in the running example only explains the root cause but does not provide the correct configuration that is critical to the repair. As a result, LLMs still fail to fix this vulnerability with such prompts. For example, GPT-4o would incorrectly remove the original configuration and break the code:

[System]
You are a helpful programming assistant to fix security vulnerabilities in Java code. You are given a code snippet and a description of the vulnerability. You should briefly describe how to fix the vulnerability, and finally output the modification.
[User]
Vulnerability description: *(alert name)*: *(alert description)*
Code snippet: *(location of the alert with 20 lines of code around)*
[Assistant]
To fix the vulnerability, we need to ensure that the environment map used to create the `RMIConnectorServer` does not include any potentially dangerous attributes that could lead to insecure deserialization. Specifically, we should avoid using `RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE` directly and instead use a more secure configuration.
……

```
- Map<String, ?> env = Collections.singletonMap(
- RMIConnectorServer.RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE,
- serverFactory);
+ Map<String, Object> env = new HashMap<>();
```

```
  RMIServerSocketFactory serverFactory = new RMIServerSocketFactoryImpl();
- Map<String, ?> env = Collections.singletonMap(RMIConnectorServer.RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE, serverFactory);
+ Map<String, Object> env = new HashMap<>();
+ env.put(RMIConnectorServer.RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE, serverFactory);
+ env.put("jmx.remote.rmi.server.credential.types", new String[] { String[].class.getName(), String.class.getName() });
  ......
  jmxServer = new RMIConnectorServer(url, env, server, ManagementFactory.getPlatformMBeanServer());
```

Figure 1: The running example (CVE-2020-13946) and the developer's patch.

A possible improvement of the above basic approach is to provide more information, such as the query documentation [51]. However, as mentioned, different projects often have different requirements in fixing alerts, e.g., by calling project-specific sanitizers or adhering to project-specific code format, and it is difficult to cover them in a per-rule documentation. In this example, GPT-4o follows the instruction in the CodeQL documentation of this rule, which is valid only for Java 10+ programs. It generates `env.put(RMIConnectorServer.CREDENTIALS_FILTER_PATTERN, "java.lang.String;!*");`, which is not the correct configuration for this project that targets Java 8+.

2.3 Existing RAG Approaches

Retrieval-Augmented Generation (RAG) is a popular technique to enhance LLM by extracting task-related prompts within a corpus of examples. For program repair tasks, a common source of examples is historical patches [25, 26, 53]. To build such a corpus, approaches typically collect a large set of commits from open-source repositories, pick out potential patches with keywords such as "fix" and "bug" in commit messages, and run the patches through the static analyzer to determine the alert type that they fix.

However, high-quality historical patches for static analysis alerts are difficult to find [22, 61]. For the running example, there are only 87 code snippets and 1 Pull Request that contain the required configuration key on GitHub. The Pull Request is also a refactor irrelevant to the fix. Therefore, it is nearly impossible to find a patch containing the configuration key without a priori knowledge of its name. As a result, the corpus is often filled with irrelevant changes that dismiss the alert by coincidence (e.g., deleting the function or refactoring to another API), which do not help.

Another possible approach is to search for code snippets similar to the vulnerable code. This approach is widely used in RAG-based [55, 64] and traditional APR approaches [23, 58] that do not use LLMs. As identified in existing work [16], BM-25 is a suitable metric to identify similar code snippets for LLMs. However, the precision is often unsatisfactory in our experiment: Although the retrieved code snippets are similar to the vulnerable code, they still lack the key ingredient to fix the alert. For example, below is the retrieved snippet with the top BM-25 similarity to the vulnerable code in the running example:

```
......
final MBeanServer jmxServer =
  ManagementFactory.getPlatformMBeanServer();
try {
  jmxServer.unregisterMBean(new ObjectName(this.getMBeanName()));
}
......
```

We see that while some parts (e.g., variable name `jmxServer` and method name `getPlatformMBeanServer`) in this retrieved snippet match the code in Figure 1, it is semantically irrelevant to the fix, and thus LLMs still fail when prompted with this snippet.

2.4 Utilizing the Analysis Rule

The fundamental flaw of the above prompt engineering and RAG baselines is that they rely on high-quality knowledge in the prompts or the retrieved information. Since LLMs generally work well with familiar tasks, and high-quality knowledge is sparse for the other less common tasks (otherwise, such knowledge will be crawled into the training set and the LLM should be familiar with them), these baselines cannot complement the LLM when it fails. A good approach should provide knowledge that is both new to the model and useful to the fix.

An idea to provide such knowledge is to provide the analysis rule. The rule defines whether the analyzer reports an alert or not, so it should include useful knowledge to fix the alert. In the running example, we can see that the rule includes the exact key name of the configuration (highlighted in pink) inside the `putsCredentialtypesKey` predicate, which will dismiss this alert when a correct configuration is present:

```
module SafeFlowConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    putsCredentialtypesKey(source.asExpr()) }
  ......
  private predicate putsCredentialtypesKey(Expr qualifier) {
    exists(MapPutCall put |
      put.getKey().(CompileTimeConstantExpr).getStringValue() = [
         "jmx.remote.rmi.server.credential.types",
          "jmx.remote.rmi.server.credentials.filter.pattern"
      ] or ......
    | put.getQualifier() = qualifier and
      put.getMethod().(MapMethod).getReceiverKeyType()
        instanceof TypeString and
      put.getMethod().(MapMethod).getReceiverValueType()
        instanceof TypeObject ) } }
module SafeFlow = DataFlow::Global<SafeFlowConfig>;
......
from Call c, Expr envArg
where (isRmiOrJmxServerCreateConstructor(c.getCallee())
    or isRmiOrJmxServerCreateMethod(c.getCallee()))
  and envArg = c.getArgument(1)
  and not SafeFlow::flowToExpr(envArg)
select c, getRmiResult(envArg), envArg, envArg.toString()
```

However, directly prompting the model with the analysis rule does not work either. In the running example, GPT-4o overrides the whole authentication mechanism and leaves it incomplete with a placeholder comment. We recognize three aspects that cause directly prompting with analysis rules suboptimal:

- **Incompleteness:** The analysis rule does not necessarily contain all the ingredients needed to generate a fix. For example, the predicate `putsCredentialtypesKey` only contains the name of the configuration key, but not its value. This is reasonable because a program with this configuration key is usually safe. However, without knowing a correct configuration value, we cannot form a patch from scratch.
- **Identification:** The analysis rule is a long piece of code. In this example, the rule has 89 lines of code and imports more definitions from a shared library. Inlining these definitions leads to even longer code. LLMs may struggle to identify the useful part in such a long input.
- **Understanding:** Static analyzers are only a small and special part of all software on the Internet, and analysis rules are rare in the wild. Therefore, understanding analysis rules is harder for LLMs than understanding regular code snippets.

2.5 Our Approach: Retrieving Examples through the Analysis Rule

To overcome the above three problems, our approach returns to the RAG paradigm and retrieves a code snippet that demonstrates how to fix the alert for the LLM. In our running example, we would retrieve the following snippet.

```
......
  env.put(JMXConnectorServer.AUTHENTICATOR,
    new JMXPluggableAuthenticatorWrapper(env)); }
env.put("jmx.remote.rmi.server.credential.types", new String[]
  {String[].class.getName(), String.class.getName()});
return env;
......
```

Prompting the model with this snippet avoids all the three problems above: (1) it contains complete ingredients, not only the configuration key but also the correct value corresponding to that key; (2) the snippet is short and easy to identify the key part; (3) LLMs are well trained to understand code examples in the same programming language. As a result, models such as GPT-4o or even GPT-4o-mini can generate the correct patch given this snippet.

Now, the problem is how to retrieve such a code snippet. In particular, how do we ensure that the code snippet contains the ingredients needed for fixing, i.e., a *key example*. Since the analysis rule contains some useful knowledge for the fix, we utilize the analysis rule to identify key examples. Concretely, we use a predicate in the analysis rule to relate the alert and the key examples. We call such a predicate a *bridging predicate* as it bridges the gap between the alert and the key examples. Predicates are basic units for determining properties of code fragments, and widely exist in different static analyzers implemented in either logic or imperative programming languages (discussed more in Section 3.1), e.g., predicate is an explicit language construct in CodeQL.

Given a reported alert a and a code snippet s from a corpus of clean code, we identify s as a potential key example for a if there exists a predicate p in the analysis rule, such that the following three conditions hold.

Condition 1: Predicate p matches a fragment in s, i.e., p evaluates to true on that fragment.

Condition 2: If we negate p in the rule and scan the vulnerable code again, the original alert a should disappear.

Condition 3: If we negate p in the rule and scan the code containing s again, a new alert of the same type should appear.

The first condition ensures that the example contains ingredients relevant to the analysis rule. In the running example, the `env.put()` statement setting the configuration key `"jmx.remote.rmi.server.credential.types"` is considered as a potential ingredient, because it is matched by the predicate `putsCredentialtypesKey`.

The second condition ensures that the example contains ingredients relevant to the current alert because the predicate matching the example is proven crucial for the alert to appear. For example, if the rule reports an alert when either predicate A or B is true, and the actual alert instance is based on the predicate A, examples matched by the predicate B are irrelevant to fixing this instance, and will be ruled out by this condition.

The third condition ensures that the example as a whole is relevant to the current alert. Due to the incompleteness of the analysis, predicates that comply with the first two conditions may match code snippets that contain only some part of the relevant ingredients, but are not relevant as a whole. In the running example, using only the first two conditions could identify a snippet that sets the configuration key in another scenario where the configuration value cannot be used in `RMIConnectorServer`. This condition excludes such examples by ensuring that they also include other parts relevant to the alert (e.g., the `new RMIConnectorServer()` statement; otherwise the alert would not appear).

From another perspective, these conditions come from the idea that if s is the intended code change between the vulnerable and patched code, we could theoretically fix the alert by "adding" s to the code under repair, or introduce an alert by "subtracting" s from the clean code. Due to the differences between the code under repair and the code in the corpus (which may have different variable names and code structures), we could not easily add or subtract a code snippet. However, we can emulate its effect by negating the corresponding predicate p. Therefore, these conditions indicate that s is a key example under the abstraction of p.

Based on the three conditions, we develop an efficient retrieval algorithm and an automatic RAG pipeline for program repair, as will be discussed in the next section.

3 Approach

3.1 Predicates in the Analysis Rule

PredicateFix assumes a static analyzer is built upon *predicates* – expressions in an analysis rule that return a Boolean value based on some fragments of the input program. We show that such a concept universally exists in analyzers implemented in either logic or imperative programming languages.

Many static analyzers are implemented in a logic programming language where the concept of predicates naturally exists. For example, Doop [5] uses Datalog, a popular logic programming language; CodeQL [10] uses its extended version of Datalog. Below, we give a toy example in Datalog for checking null pointer exceptions (NPEs).

```
isNull(V, L) :- assignStmt(V, "null", L).
isNull(V, L) :- isNull(V, L0), controlFlowTo(L0, L), !nullGuard(V, L).
nullGuard(V, L) :- assignStmt(V, E, L), constructorCall(E).
nullGuard(V, L) :- assertStmt(V, "!=", "null").
hasAlert(L) :- methodCall(V, _, L), isNull(V, L).
```

In a typical Datalog rule for program analysis, there are primitive predicates for matching code snippets in the source code. In this example, `assignStmt`, `assertStmt`, `methodCall`, `constructorCall`, and `controlFlowTo` are primitive predicates. When a code snippet is matched, a fact is generated to represent this match. For example, predicate `assignStmt` matches all assignment statements in the code, and a fact `assignStmt(V, E, L)` is generated for each match to represent an assignment statement at the location `L` that assigns the value of expression `E` to the variable `V`.

Upon a set of primitive facts, an analysis rule generates new facts with inference rules. In the above example, each statement is an inference rule, indicating that the fact on the left-hand side can be generated when all the facts on the right-hand side are present. `isNull(V, L)` determines whether the variable `V` can be null in the location `L`, which holds when either `V` is explicitly assigned to `null` (line 1) or is previously null in `L0` and doesn't reach a null guard at `L` (line 2); another predicate `nullGuard(V, L)` determines whether location `L` can be considered as a null guard of the variable `V`, which holds when it assigns `V` to a object constructor call `E` (line 3) or it asserts the `V` not to be `null` (line 4). Finally, the `hasAlert` predicate prescribes the main outcome of the analysis: If a statement `L` is a method call `V.anymethod()` where `V` can be null, location `L` should have an NPE problem. The Datalog engine continues to generate facts using the inference rules until a fixed point is reached. Finally, an alert is reported at location `L` for each instance of `hasAlert(L)`. Note that using a location parameter in the predicate is common in analyzers based on logic programming to facilitate error reporting. For ease of implementation, we consider that a predicate matches a code snippet if there exists a fact of the predicate containing a location parameter that points to the code snippet.

Besides logic programming, some static analyzers express analysis rules in the form of imperative programs that loop through each part of the code and check for its properties. For example, Infer [7] is implemented in OCaml, and gosec [46] is implemented in Golang. Below is the analysis rule in gosec checking the SSRF vulnerability:

```
func (r *ssrf) Match(n ast.Node, c *gosec.Context) (*issue.Issue) {
  if node := r.ContainsPkgCallExpr(n, c, false); node != nil {
    if r.ResolveVar(node, c) {
      return c.NewIssue(...) } }
  return nil }
func (r *ssrf) ResolveVar(n *ast.CallExpr, c *gosec.Context) bool {
  if len(n.Args) > 0 {
    arg := n.Args[0]
    if ident, ok := arg.(*ast.Ident); ok {
      obj := c.Info.ObjectOf(ident)
      if _, ok := obj.(*types.Var); ok {
        scope := c.Pkg.Scope()
        if scope != nil && scope.Lookup(ident.Name) != nil {
          return true }
        if !gosec.TryResolve(ident, c) {
          return true } } } }
  return false }
```

The main procedure of gosec parses the code under analysis and calls the `Match` function against each AST node. This function then checks for many properties, each with a Boolean `if` condition (highlighted in pink). If all conditions are met, it returns with a `NewIssue` call, indicating a reported alert.

For such analyzers, we consider each Boolean expression used in the `if` statement as a predicate, and the code fragment upon which the expression depends as the matched code. The matched code can be identified by a dynamic dependency analysis, or by utilizing the pattern of the analysis rule. For example, the parameter `n` in the above example can be directly used as the matched code for the subsequent `if` conditions.

Algorithm 1 Identifying key examples

Require: Target codebase t, analysis rule r, clean code corpus C.
Ensure: key examples E.

```
 1: $P \leftarrow \text{getPredicates}(r)$
 2: $P \leftarrow \{p \in P \mid \text{checkCond1}(p) \land \text{checkCond2}(p, t)\}$
 3: $E \leftarrow \varnothing$
 4: for each $c \in C, p \in P$ do
 5:     for each $s \in \text{getMatches}(p, c)$ do
 6:         if $\text{checkCond3}(p, s, c)$ then
 7:             $E \leftarrow E \cup \{\text{expandContext}(s)\}$
 8:         end if
 9:     end for
10: end for
```

3.2 Matching Key Examples

The three conditions in Section 2.5 give us a basic way to determine whether a code snippet is a potential key example or not, but do not allow us to efficiently identify all potential key examples from a large code corpus. Here we propose Algorithm 1 as an efficient way of identifying key examples. It takes a target codebase t, which has an alert detected by the analysis rule r, and also a clean code corpus C with codebases free of alerts detected by r.

The algorithm first initializes a set of possible bridging predicates P as all the predicates appearing in the analysis rule r. Then it filters P with the first two conditions: a bridging predicate should match code snippets, and negating it should dismiss the alert. Then, the loop on line 4 loops through all possible bridging predicates and codebases in the corpus to find possible key examples. We test the third condition for each matched code snippet, and if a snippet passes the test, we expand it to a full key example (by adding several lines of context around the code) and add it to the final result E. In case the static analyzer fails to run due to implementation-level limitations, we skip to the next predicate/example.

We can see that instead of looping through all possible code snippets and checking each condition individually for each snippet, Algorithm 1 checks the three conditions in a particular order (Condition 1, then Condition 2, and finally Condition 3) to optimize efficiency: Checking Condition 1 only requires to parse the argument list of each predicate, which does not depend on the corpus and takes nearly no time; checking Condition 2 requires to re-run the analysis on the code under repair, but does not depend on the corpus; checking Condition 3 requires to run the analysis for each project in the corpus, which is computationally heavy if there are a large number of projects, so we put it at the last step.

Since there are many different ways to write a rule, the three conditions are heuristic and cannot guarantee that all identified examples are key to the fix. For example, the rule may capture two distinct patterns for an alert, where the buggy code falls into one pattern, the example falls into the other pattern, and the bridging

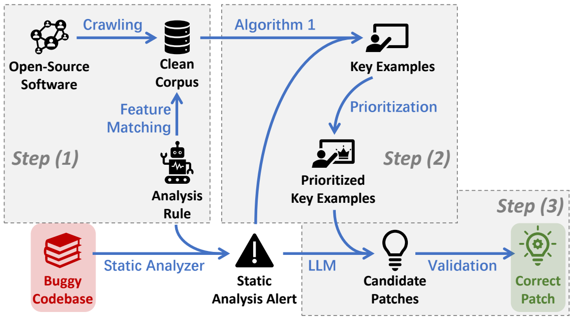

Figure 2: The RAG pipeline of PredicateFix.

predicate is used in both patterns. The prioritization step that we will describe in Section 3.3.2 is designed to deal with such cases.

3.3 The RAG Pipeline

Finally, we discuss how to build a RAG-based program repair tool, PredicateFix, based on Algorithm 1 that finds key examples.

The overview of PredicateFix is shown in Figure 2. Given that the analysis rule reports an alert on the target codebase, PredicateFix takes three major steps to fill the gap between the alert and the fix: (1) It collects a corpus of clean code based on various sources; (2) it runs Algorithm 1 to get key examples from the corpus, and then prioritizes key examples to form the prompt with the most relevant key examples; (3) it calls the LLM to obtain candidate fixes and validates each fix against the analysis rule. In the following, we discuss each important part of this pipeline categorized by its step.

3.3.1 Corpus Collection. Any RAG approach requires a corpus from which the examples are retrieved. The completeness of the corpus is essential in the whole approach, because there is no way to retrieve a relevant example in case it is absent in the corpus. We build a corpus containing mostly clean code from three sources.

- The first source is **open-source software repositories** in the same programming language as the target codebase. We can assume that their code quality is good enough, but they may lack key examples using some less common APIs or features.
- To augment the corpus when the analysis rule is less common, we additionally search for **repositories matching string literals** in the analysis rule or code examples in the documentation of the analysis. For example, when fixing the running example as described in Section 2, we recognize the literal string `"jmx.remote.rmi.server.credential.types"` in the analysis rule and search for relevant code repositories.
- We can also add **other parts of the target codebase** or other **user-specified additional codebases** to the corpus. This is because the user may have some distinct code styles and patterns, and we may get a relevant example in these codebases.

Since the corpus only depends on the analysis rule instead of the alert, users could practically collect and cache the corpus for each analysis rule before repair. This can improve the efficiency of PredicateFix by offloading some analysis time.

3.3.2 Key Example Prioritization. According to Gao et al. [16], more examples may not always lead to better LLM performance, and their experiment has shown that four examples are enough for bug fixing. We take a gradual approach to minimize the cost and response time of PredicateFix: It iteratively calls the LLM, starting with no example (same as the Basic baseline in Section 2.2), and picking the next example if the LLM does not repair successfully. It stops when a patch fixing the alert has been found or all of the first four examples have failed for each source of the corpus.

We apply several prioritization rules to the key examples from the list returned by Algorithm 1, as shown below.

- **Similarity heuristic:** We calculate the BM-25 similarity between each key example and the code context of the alert. The examples with the highest similarity are ranked at the top. Since BM-25 requires the code to be tokenized, we use the BPE tokenizer from CodeLlama [45], which is trained with source codes and can handle out-of-vocabulary identifier names.
- **Count heuristic:** We count the number of occurrences of each bridging predicate in the key examples list. If a bridging predicate has too many occurrences, we drop key examples matched by this predicate or rank them at the bottom. The rationale for this heuristic rule is that such predicates are typically not specific enough to be useful (e.g., matching all constructor calls or all assignment statements).
- **Code hierarchy heuristic:** If the file containing a key example or its bridging predicate is located in a library path, we drop the example or rank it at the bottom. This is because such examples are not project-specific (for key examples) or rule-specific (for bridging predicates), so they are less likely to be fix ingredients.

These rules improve the quality of key examples and the efficiency when there are false positives (irrelevant predicates/examples) matched by the three conditions in Section 2.5. For example, when fixing `XssThroughDom.ql` alerts in CodeQL, a predicate matches all `length` field access for string variables. Therefore, all `xxx.length` expressions would be incorrectly considered as key examples. These false positives will be ruled out by the count heuristic.

3.3.3 Interfacing with LLMs. Large language models receive input and give output in the form of natural language conversation, so we need to organize relevant information to fix the alert in a natural language format ("prompt"), and instruct the model to respond with the patch in a format that PredicateFix can automatically parse and validate. The quality of the input/output formats may influence their effectiveness.

Our prompt format generally follows the OpenAI guide [47] and contains the task description, the output formatting instruction, the natural language description of the alert, the context of the code to repair, and retrieved key examples. The task description and the output formatting instruction are in the system prompt, and other task-specific contexts are in the user prompt, as shown below.

[System]
You are a helpful programming assistant to fix security vulnerabilities in *(language)* code. You are given a code snippet and a description of the vulnerability. You should briefly describe how to fix the vulnerability, and finally output the modification in JSON.

In the JSON, for each modification, use `old_line` to mark the exact line to modify, and `new_line` as the modified line. Example format:

~~~
```json
[{ "old_line": "int␣x␣=␣1;",
   "new_line": "int␣x␣=␣1;␣x++;" }]
```
~~~

[User]
Vulnerability description: *(alert name)*: *(alert description)*
Code snippet: *(location of the alert with 20 lines of code around)*
Below code snippet is a safe example. You can use it if helpful.
(the key example with 6 lines of code around)

Note that the output format instructed in the prompt is different from the standard GNU diff format. This is because models often generate invalid diffs [35], such as generating wrong line numbers (`@@ -1,1 +1,1 @@`) and forgetting to put a space at the beginning of each unmodified line. As a result, these diffs cannot be parsed by machines. In contrast, recent models are fine-tuned to output valid JSON, so this output format leads to better effectiveness.

After receiving the response from the LLM, we extract the JSON code block containing the patch and then apply each of the line modifications. The patched codebase is then re-analyzed by the same analysis rule, and if the alert disappears, the fix is considered successful and provided to the user as the final patch. Note that a *successful* patch that passes this check may not be the *correct* one wanted by the user. For example, the LLM often generates placeholder or function-breaking codes. Human inspection is necessary to determine the correctness of patches.

4 Implementation

4.1 On Logic Programming Analyzers

We have implemented PredicateFix to fix the security vulnerabilities reported by the CodeQL [10] static analyzer, a popular static analyzer integrated in GitHub [18] and used by many open-source projects such as Chromium [19]. We chose this static analyzer as our target because it shows the ability to detect non-trivial security vulnerabilities across different programming languages, while many other static analyzers only contain a few types of analyses or only report trivial issues such as unused variables. The rules of CodeQL are also open source, which facilitates our implementation.

Corpus selection: Section 3.3.1 discusses different sources of the clean code corpus. For the first source (popular software), we search for the top 1000 repositories in the programming language of the target codebase, and then clone them from GitHub. We also include other repositories in the benchmark in the corpus. For the second source (matching string features), we use the code search feature in GitHub and clone the top 3 repositories. This hyperparameter is determined by our experience with a few rules: A good key example is generally within the first few search results if the keyword is specific enough (e.g., `jmx.remote.rmi.server.credential.types`); otherwise, the keyword is likely too general (e.g., `toString`) so keeping more results will not improve data quality. We do not assign other additional codebases as the third source. To avoid possible data leakage in the corpus, we drop the code from the corpus if it precisely matches the developer's patched code.

Example prioritization parameters: Section 3.3.2 introduces three heuristic rules to filter and prioritize key examples. For the count heuristic, we remove a key example if the number of occurrences of the corresponding bridging predicate is greater than 20. For the code hierarchy heuristic, we remove bridging predicates located in basic language definition rules such as `Expr.qll`. For the similarity heuristic, we pick the top four key examples with the highest BM-25 similarity, following the existing paper [16].

4.2 On Imperative Analyzers

To evaluate whether our approach works for static analyzers implemented in different programming paradigms (i.e., logic or imperative), we applied PredicateFix on GoInsight, an internal Golang code scanner in ZTE Corporation. GoInsight is an imperative static analyzer with 91 analysis rules written in Golang. The implementation of this analyzer is similar to gosec [46], as we have discussed in Section 3.1. The analysis rules are customized to be more accurate and cover more types of issues than the built-in rules in gosec.

Predicate extraction: Algorithm 1 requires a mechanism to obtain and negate predicates behind an analysis rule. For this purpose, we modified the source code of GoInsight to add a trace statement to each Boolean calculation, such as an `if` condition. We can then negate a specific predicate corresponding to a trace ID in the source code. This allows PredicateFix to control the execution process of the analyzer, including obtaining and negating these predicates.

5 Evaluation

In this section, we empirically evaluate PredicateFix with the following research questions.

RQ1. **Effectiveness:** Can PredicateFix increase the effectiveness of LLMs in fixing static analysis alerts?
RQ2. **Comparison with other techniques:** Is PredicateFix better than other possible retrieval-augmented generation (RAG) and prompt engineering techniques?
RQ3. **Generalization:** How does the effectiveness of PredicateFix generalize to another static analyzer?

5.1 Experimental Setup

5.1.1 The Benchmark. **RQ1-2** evaluate the effectiveness of fixing security vulnerabilities reported by CodeQL [17].

We first constructed a dataset of security vulnerabilities. There are multiple existing datasets, including BigVul [13], CrossVul [41], Vul4J [6], and Project KB [43], all of which are based on the Common Vulnerabilities and Exposures (CVE) dataset. We merged these datasets into a cross-language dataset containing 6027 CVE entries with the Git repository of the source code and the exact commit ID of the developer's patch.

Then, we ran all of the built-in analysis rules in CodeQL against all entries in Java, JavaScript, Python, or Ruby. We chose only these four languages because analyzing programs in other languages (e.g., C++ and Go) requires an environment to compile and build the codebase, which cannot be set up automatically. For each entry, we compare the analysis results before and after the the developer's patch commit. If one or more alerts are eliminated, the commit

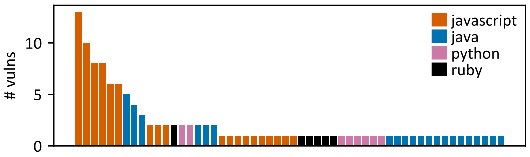

Figure 3: Distribution of 117 CVE vulnerabilities detected by 53 CodeQL rules.

Table 1: Distribution of 416 alerts detected by 91 GoInsight rules. 17 discarded alerts are not suitable for automatic repair.

Issue Type	#Rules	#Alerts (+Discarded)
BUG.*	36	142 (+11)
PERFORMANCE.*	39	179 (+6)
SECURE.*	16	78 (+0)
Σ Total	91	399 (+17)

is marked as fixing that CodeQL alert and included in the benchmark. We further manually inspected these commits and discarded 13 commits that made an irrelevant change and unintentionally caused the alert to disappear (e.g., by refactoring to another API not covered by CodeQL). The final benchmark for our experiment contains 117 verified vulnerabilities that CodeQL can recognize with the corresponding patches from developers. It includes 67 vulnerabilities in JavaScript, 33 in Java, 10 in Python, and 7 in Ruby.

These 117 vulnerabilities are detected by 53 CodeQL rules following a long-tail distribution, as shown in Figure 3. We can clearly see the distinction between different types of vulnerabilities: While the most common rule (`XssThroughDom.ql`) flagged 13 JavaScript vulnerabilities, many other vulnerabilities are less common, including 34 rules that recognize only one vulnerability in the whole CVE benchmark, taking up 64% of rules or 30% of alerts.

RQ3 evaluates the generalization of our approach by applying PredicateFix on GoInsight, as discussed in Section 4.2. GoInsight is an imperative code analyzer that covers different types of issues, as shown in Table 1. Among the 91 rules in total, 36 rules check for logical bugs in the program, such as missing switch cases and dereferencing null pointers; 39 rules check for performance issues, such as inefficient string comparison and unused assignment statements; 16 rules check for security vulnerabilities, such as using weak hash algorithms and possible integer overflows during conversion.

To build the benchmark of RQ3, we retrieved the list of Golang repositories with the most stars on GitHub. For each repository, we then ran GoInsight against it and collected alerts with the corresponding alert-triggering code snippets for each rule. False alarms are skipped in this process through manual inspection. We kept at most five alerts per rule to avoid flooding the benchmark with too many simple alerts. The process continued until each rule had at least one alert. The collected test suite contains 416 verified alerts in total: 5 alerts for each of 78 rules, 1-4 alerts for each of the remaining 13 rules. We further discarded 17 alerts that are not suitable for an automatic repair (e.g., a rule warns about empty loop bodies, but such bodies are intentionally left empty in some code, so the intended repair is not well-defined). Finally, we got 399 remaining alerts as the benchmark for RQ3, covering 346 projects.

5.1.2 *Language Models.*

For **RQ1-2**, our evaluation included a variety of LLMs from different vendors:

- GPT-4o (2024-08-06) and GPT-4o-mini (2024-07-18) from OpenAI [42];
- DeepSeek-V3 (0324) and DeepSeek-R1 (0528) from DeepSeek [11];
- Claude-3.5-Sonnet (20241022) from Anthropic [1];
- Qwen2 from Alibaba Group [59].

The LLMs cover both commercial and open-weight models from different vendors, and also cover both best-performing and cost-efficient models. Therefore, we believe that the selection can represent the recent development of LLM at the time the experiment is conducted (in January 2025).

Following existing papers [16, 31] and OpenAI's guide [47], the temperature parameter is set to zero for all models, which is optimal for coding tasks and helps reproduce the experiment.

For **RQ3**, we used the internal LLM deployed in ZTE Corporation, which powers various code intelligence functions for the employees.

5.1.3 *Baselines Approaches.*

For **RQ1** and **RQ3**, we compare PredicateFix against the Basic baseline approach as described in Section 2.2. The Basic approach prompts the model with the alert name, its natural language description, and the code context, without additional RAG techniques. Therefore, it can be estimated as the intrinsic effectiveness of that LLM. PredicateFix will be empirically helpful if its effectiveness is greater than the Basic baseline.

For **RQ2**, we consider three possible RAG techniques, as listed below. The prompts used in all these baselines include the alert details (as in the "Basic" baseline) and additional retrieved information.

- "History" (described in Section 2.3), which retrieves the historical patch identified in the wild;
- "Similar" (described in Section 2.3), which retrieves the most relevant code snippets with BM-25 similarity;
- "Rule Src" (described in Section 2.4), which retrieves the source code of the analysis rule, truncated to 1000 tokens;

We also consider two better prompt engineering techniques:

- "Rule Expl", which enhances "Rule Src" by asking the model to first explain the source code in natural language, and then fix the bug given the explanation;
- "Docs" (described in Section 2.2), which enhances "Basic" by providing the full CodeQL documentation (i.e., the `.qhelp` file) of the alert.

We compare the increase in effectiveness of PredicateFix with each of the five techniques above.

5.1.4 *The Metric.*

We consider a repair to be *successful* if the model output contains a valid patch, and the alert from the static analyzer disappears after applying the patch. However, a successful repair might not be *correct*: a patch deleting the entire vulnerable functionality certainly causes the alert to disappear (thus successful), but it is highly unlikely to be the repair expected by the developer (thus incorrect). Human inspection is necessary to distinguish correct repairs from plausible repairs.

Table 2: Number of correct (successful) repairs against 117 CVE vulnerabilities with various LLMs.

Model	Basic Baseline	With PredicateFix	Improvement
GPT-4o	59 (70)	75 (92)	27.1% (31.4%)
GPT-4o-mini	49 (60)	66 (84)	34.7% (40.0%)
DeepSeek-V3	47 (52)	76 (88)	61.7% (69.2%)
DeepSeek-R1	51 (58)	74 (84)	45.1% (44.8%)
Claude-3.5-Sonnet	56 (62)	81 (91)	44.6% (46.8%)
Qwen2	46 (55)	73 (85)	58.7% (54.5%)

Therefore, for **RQ1** and **RQ3**, we manually inspected the patches to flag successful patches that are incomplete code, break existing functionalities or introduce new issues to be incorreect. After this process, we report the number and percentage of correct repairs, which is a more suitable metric than successful repairs.

Only for **RQ2**, we cannot perform a human inspection because the total number of patches is large (up to 117 alerts × 6 models × 5 new baselines). Therefore, we only report the number of successful repairs in RQ2. We argue that: (1) Manual inspection on RQ1 will show that 85% successful repairs are correct, so the precision of the successful metric will be adequate for a qualitative finding given the results in RQ2; (2) it is common in existing publications [37, 44] to rely on an automatic metric or to inspect only on a small set of patches when a full human inspection is impossible;

The evaluation does not discuss the time used to fix the vulnerability because the response time of LLMs is highly unstable depending on the service load. Since the static analysis result and intermediate data can be cached for the corpus, most of the work in PredicateFix can be done in advance once per rule. Therefore, the impact of PredicateFix on time usage can be minimal (within one minute per bug in most cases if properly optimized).

The evaluation also does not discuss the costs of calling LLMs because a key example is generally a few lines of code consisting of at most several hundreds of tokens. Therefore, the token cost is not a severe limitation of PredicateFix or any other baseline. In fact, the whole experiment costs us less than $50 to fix 117 alerts with the 6 LLMs and 7 approaches ($0.01 per alert on average).

5.2 RQ1: Effectiveness

To evaluate the effectiveness of PredicateFix, we count the number of successful and correct repairs with PredicateFix against the Basic baseline for each LLM. The results are shown in Table 2.

We can see that for all six LLMs, PredicateFix improves the number of correct repairs by 27.1% to 61.7%, or successful repairs by 31.4% to 69.2%. This result indicates that PredicateFix steadily increases the effectiveness of LLMs in the program repair task.

From the numbers in Table 2, PredicateFix increases the number of successful patches by 167, while increasing the number of correct patches by only 137. The difference between these two numbers (30 patches) indicates the cases where we generate a plausible (successful but incorrect) patch. Overall, PredicateFix slightly increases the plausible rate among successful patches from 13.7% to 15.0%. However, Since the increase in correct patches is significant, we believe that PredicateFix is overall beneficial to users.

With the help of PredicateFix, the best model (Claude-3.5-Sonnet) can correctly repair 81/117 = **69.2%** bugs in this benchmark; even the worst model (Qwen2) can now repair 73/117 = **62.4%** bugs, significantly better than any LLM in the Basic baseline. Therefore, PredicateFix opens an opportunity for program repair users to switch to a faster and cheaper model while the effectiveness is even increased. For example, GPT-4o-mini with PredicateFix correctly fixes 7 more bugs than GPT-4o with basic prompt engineering.

During the manual inspection of patches, we found PredicateFix especially helpful in two scenarios: (1) Some vulnerabilities have multiple directions to fix, with varying difficulties. LLMs may stick to a difficult direction and fail to generate a fully correct patch in the end. With multiple key examples, LLMs can be better aware of other possible directions. (2) When the fix involves a particular configuration, LLMs may be unsure about the exact configuration to add, leaving a placeholder for the user to fill in (e.g., `// implement access control here`). With relevant key examples, LLMs can better generate the complete configuration.

Another interesting observation is that the performance of Deep Seek-R1 is on par with DeepSeek-V3, contradicting the common belief that a reasoning model should be better than the base model. A possible explanation is that the bottleneck for the current task is the awareness of specific knowledge instead of the reasoning ability. If the model is not aware of an important fix ingredient for a less common alert type, it cannot correctly fix the alert anyway, with or without reasoning.

Finding 1:
PredicateFix can significantly improve the effectiveness of LLMs to fix static analyzer alerts, increasing the number of correct repairs by a large number (27.1% ∼ 61.7%). The increase is observed in all six LLMs studied in this RQ.

5.3 RQ2: Comparison with Other Techniques

To compare PredicateFix with other possible RAG and prompt engineering techniques, we implemented the other five baselines described above and compared the number of successful repairs. We further break down the number into each programming language to measure the steadiness of the improvement.

From Figure 4, we can see that the five new baselines (labeled in dark red) cannot steadily increase the effectiveness over Basic. Their increases are small and vary across different LLMs, and can even be negative for some models (e.g., GPT-4o-mini and Qwen2). We speculate that these models have a weaker ability to judge the relevance of the provided RAG information, and they may be misled when the information is irrelevant to the correct fix. In contrast, PredicateFix provides the models with high-quality key examples and thus significantly outperforms the baselines. As a result, PredicateFix generates unique fixes that all baselines fail, including the running example discussed in Section 2.1. For each LLM in Figure 4, the number of unique fixes is 4, 5, 4, 3, 2, and 7.

Furthermore, the effectiveness increase of PredicateFix is steady in different programming languages and alerts, indicating that it helps the LLM to be more effective even for common alerts. This

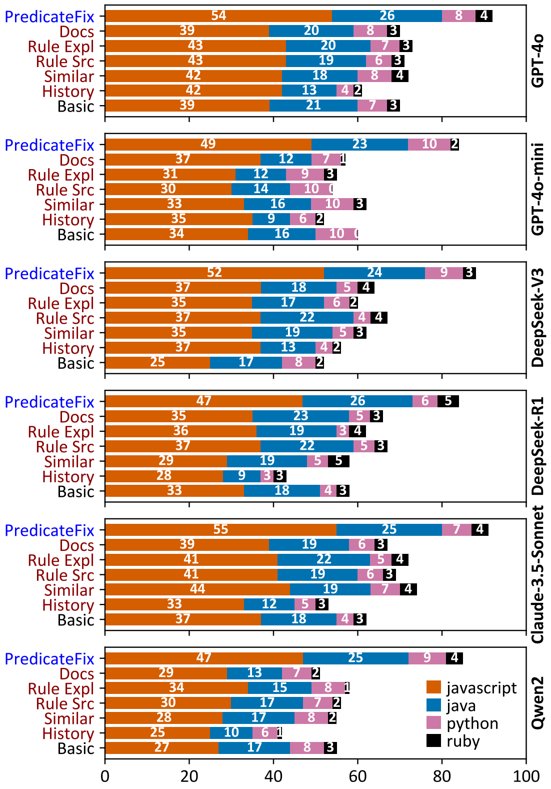

Figure 4: Number of successful repairs against 117 CVE vulnerabilities between PredicateFix and baselines.

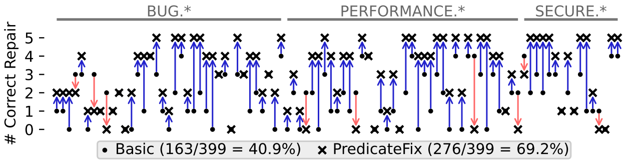

Figure 5: Correct repairs for each GoInsight alert type. Blue arrows indicate that PredicateFix increases the number of correct repairs. Red arrows indicate a decrease.

can be seen as a by-product of the proposed RAG pipeline: a variety of examples can instruct the model to think in multiple aspects, hence further increasing its effectiveness on tasks it already has relevant knowledge about.

Finding 2:
PredicateFix can steadily increase the effectiveness of repair over the Basic baseline, with multiple unique fixes. In comparison, the other five RAG and prompt engineering baselines have a weaker or even negative increase for some LLMs.

Table 3: Number of correct (successful) repairs against additional 11 recent CVE vulnerabilities.

Model	PredicateFix	Docs	Rule Expl	Rule Src	Similar	History	Basic
3.5-Turbo	5 (6)	3 (6)	3 (4)	3 (4)	2 (4)	3 (3)	2 (3)
4o-mini	7 (8)	5 (6)	5 (6)	6 (7)	4 (4)	7 (7)	5 (6)

5.4 RQ3: Generalization

To evaluate the generalization of PredicateFix beyond fixing vulnerabilities detected by CodeQL, we applied it to GoInsight to form a repair system for general Golang bugs. We also combined GoInsight with basic prompt engineering as the Basic baseline. We then counted the number of correct repairs of these two systems against the benchmark described in Section 5.1.1.

On the Basic baseline, the system has generated **163** correct repairs in total. Breaking down into categories, the number consists of 59 correct repairs for `BUG.*` alerts, 69 for `PERFORMANCE.*` alerts, and 35 for `SECURE.*` alerts. During manual inspection, we found that LLM often hallucinates while repairing, possibly because Golang is a new programming language. For example, to fix the `BUG.NO.SWITCH.CASE` alert, we need to add the missing cases in a `switch` clause, and the missing cases to add are already instructed by the static analyzer. The repair should be an easy task, but in one case, the LLM ignores the instruction and determines that a `break` statement should be added at the end of each switch case. It may have confused the Golang syntax with C/C++, where a switch case must end with a `break` statement to avoid the fall-through behavior. Golang does not have the fall-through behavior, so the LLM fixed a hallucinated bug that does not exist.

With PredicateFix, the number of correct repairs increases to **276**, which consists of 100 correct repairs for `BUG`, 121 for `PERFORMANCE`, and 55 for `SECURE`. The system now correctly fixes **69.2%** of all 399 bugs in the benchmark (was **40.9%** with Basic), with a relative improvement of 69.3%. Figure 5 further visualizes the increase in correct repairs by each alert type. We can see that PredicateFix increases the number of correct repairs in most cases. In many cases including the `BUG.NO.SWITCH.CASE` alert discussed above, the LLM follows the right path to repair and no longer hallucinates when a key example is given. This demonstrates the generalization and practical utility of PredicateFix in a real-world scenario.

Finding 3:
PredicateFix generalizes to GoInsight, an imperative static analyzer for Golang, with a 69.3% increase in the number of correct repairs over the Basic baseline.

5.5 Threats to Validity

5.5.1 Internal. The main internal threat to validity comes from potential data leakage in datasets. To avoid this threat, we drop exact matches of the patched code in our clean code corpus, as described in Section 4. However, there may still be data leakage through LLMs, since some experiment subjects may be used for their training, and we cannot control it. To learn the effect of this threat, we conducted a small additional experiment.

Setup: For the LLM, we used GPT-3.5-Turbo (version 0125, knowledge cut-off in September 2021) and GPT-4o-mini (knowledge cut-off in October 2023). For the benchmark, we used ReposVul [54], which includes C/C++/Java/Python CVE vulnerabilities up to 2023. Following the same process of RQ1-2, we ran CodeQL analysis for recent Java and Python CVE entries between September 2021 and October 2023. As a result, we collected 381 vulnerability-fixing commits, 11 of which fixed a CodeQL alert. We used these 11 cases between the knowledge cut-off of the two LLMs as the benchmark.

Result: The numbers of correct and successful repairs of PredicateFix and other baselines are shown in Table 3. Although the sample size (11 cases) is insufficient to measure the exact percentage of improvement, we can roughly see that PredicateFix increases the number of correct repairs over Basic by at least 2, regardless of whether the knowledge cutoff date is before (i.e., GPT-3.5-Turbo) or after (i.e., GPT-4o-mini) these vulnerabilities. Therefore, this threat should have a limited chance of affecting previous findings.

5.5.2 External. The main external threat to validity comes from the generalization of our approach: can PredicateFix show similar effectiveness in other programming languages, LLMs, experiment subjects, or static analyzers? To address the former two threats, our evaluation has used six LLMs from different vendors with different sizes, and RQ2 also has a breakdown of effectiveness in each programming language. For the latter two threats, RQ3 performs an additional evaluation on GoInsight, an imperative Golang static analyzer, where PredicateFix shows a non-trivial effectiveness increase. These experiments reduced the threat of generalization.

6 Related Work

6.1 Program Repair Targeting Static Analysis

PredicateFix shares similar objectives with many works aimed at automatically repairing alerts identified through static analysis alerts. Liu et al. [32] analyzed the distribution of bug-fix pairs for FindBugs violations, and proposed a neural network to automatically identify fix patterns through historical patches. Tools such as Avatar [33], Phoenix [3], and Getafix [2] can cluster and generalize fix patterns for different alert types and use them to generate new patches. These approaches relied on a large corpus of historical patches for learning fix rules. Section 2.3 and the unsatisfactory effectiveness of the "History" baseline in RQ2 have shown that finding only a few high-quality historical patches is already challenging, not to mention building a corpus of historical patches. In contrast, PredicateFix requires only clean code examples and expands the search scope to find high-quality examples with bridging predicates.

Some approaches focus on specific error types, such as memory errors [15, 20, 29, 30], API misuse [9], and missing sanitizers or guards [22]. These approaches are tightly integrated with one or a few analysis rules and are tailored to fix alerts on only these rules. Approaches such as SpongeBugs [38] and Sorald [12] fix multiple types of alerts by manually specifying a rule for each type. PredicateFix, however, applies to general static analyzers without manual efforts on each alert type.

Furthermore, some symbolic approaches leverage static analysis information for repair. Senx [21] uses symbolic execution to patch a program to satisfy the user-specified safety properties. SymlogRepair [36] encodes candidate patches as Datalog values and uses constraint solving to efficiently find patches that satisfy Datalog rules. EffFix [63] uses incorrectness separation logic to group patches according to their semantic effects and measure how close a patch is to fixing the bug. These approaches require precisely specifying the repair condition, and face the scalability issue of symbolic methods on a large codebase. In contrast, PredicateFix can be applied to a wider scope of alerts and programs where symbolic approaches are infeasible. Therefore, our approach is complementary to these approaches in handling more alerts and programs.

There is growing interest in deep learning methods for program repair. TFix [4] fine-tuned a T5 model to fix ESLint violations. DrRepair [60] trained a graph neural network on graphs that connect source codes with diagnostic messages to model the reasoning behind program analysis. Compared with them, PredicateFix does not require a training set of historical patches. As discussed above, it is difficult to collect historical patches for many alert types, and thus it is difficult to build a high-quality training set.

More recently, LLMs have shown great potential in automated program repair [56]. Approaches such as CORE [51] and DeepVulGuard [48] prompt the LLM with information such as the issue description. The "Basic" and "Docs" baselines in RQ2 of our evaluation have shown that using only such information in the prompt has limited effectiveness in improving LLMs.

6.2 Retrieval-Augmented Program Repair

The performance of large language models can be improved through in-context learning of relevant knowledge [16, 62]. This motivates some APR approaches using RAG techniques.

FitRepair [55] exploits the plastic surgery hypothesis by searching for example code snippets in the same project. Ring [26] selects examples based on the similarity of the error message. RAPGen [53] accounts for both lexical and semantic code matching. Cedar [40] retrieves code based on embedding similarity or frequency analysis. InferFix [25] and VulAdvisor [61] retrieve historical patches to enhance the prompt. VulMaster [65] incorporates code syntax trees and CWE expert knowledge in prompt design and uses the Fusion-in-Decoder technique to deal with the context length limit. T-RAP [34] uses edit scripts as templates to retrieve similar fixes. Unlike these RAG approaches, PredicateFix considers the static analyzer as a white box and takes advantage of the predicate information in the retrieval process, leading to high-quality key examples and a better effectiveness than the "Similar" baseline as shown in RQ2.

7 Conclusion

This paper presented PredicateFix, a retrieval-augmented generation (RAG) approach that fixes static analysis alerts in program code. The novel insight behind our approach is to identify and utilize bridging predicates from the analysis rule and key examples in a clean code corpus. Based on this insight, we proposed an algorithm to automatically retrieve key examples as the source of demonstration. In this way, we can provide relevant knowledge to the large language model and thus increase its effectiveness, particularly on less common alerts where the model often hallucinates.

We implemented and evaluated PredicateFix on multiple static analyzers, programming languages, and LLMs. The evaluation confirmed that PredicateFix can increase effectiveness by 27.1% to 69.3%, significantly outperforming other baseline RAG and prompt engineering techniques.

The artifacts of this paper are available at FigShare: https://doi.org/10.6084/m9.figshare.26956228.

Acknowledgments

This work was sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, the National Natural Science Foundation of China (NSFC) under Grant No. W2411051, and the ZTE Industry-University-Institute Cooperation Funds under Grant No. HC-CN-20210319008.

References

[1] Anthropic. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–27. https://doi.org/10.1145/3360585

[3] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. ACM, Tallinn, Estonia, 613–624. https://doi.org/10.1145/3338906.3338952

[4] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: learning to fix coding errors with a text-to-text transformer. In *Proceedings of the 38th International Conference on Machine Learning (ICML '21)*, Vol. 139. PMLR, Virtual Event, 780–791. https://proceedings.mlr.press/v139/berabi21a.html

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09)*. ACM, Orlando, FL, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[6] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. ACM, Pittsburgh, Pennsylvania, 464–468. https://doi.org/10.1145/3524842.3528482

[7] Cristiano Calcagno and Dino Distefano. 2011. Infer: an automatic program verifier for memory safety of C programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Vol. 6617. Springer, Berlin, Heidelberg, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[8] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (Sept. 2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[9] Juan Alfredo Cruz-Carlon, Mahsa Varshosaz, Claire Le Goues, and Andrzej Wąsowski. 2022. Patching locking bugs statically with Crayons. *ACM Transactions on Software Engineering and Methodology* 32, 3 (Aug. 2022), 1–28. https://doi.org/10.1145/3548684

[10] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2008. .QL: object-oriented queries made easy. In *Generative and Transformational Techniques in Software Engineering II (GTTSE 2007)*, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.). Springer, Berlin, Heidelberg, 78–133. https://doi.org/10.1007/978-3-540-88643-3_3

[11] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[12] Khashayar Etemadi, Nicolas Harrand, Simon Larsén, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikström, and Martin Monperrus. 2023. Sorald: automatic patch suggestions for SonarQube static analysis violations. *IEEE Transactions on Dependable and Secure Computing* 20, 4 (July 2023), 2794–2810. https://doi.org/10.1109/TDSC.2022.3167316

[13] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. ACM, Seoul, Republic of Korea, 508–512. https://doi.org/10.1145/3379597.3387501

[14] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO '09)*. ACM, Montrea, QC, Canada, 947–954. https://doi.org/10.1145/1569901.1570031

[15] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for C programs. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Vol. 1. IEEE, Florence, Italy, 459–470. https://doi.org/10.1109/ICSE.2015.64

[16] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. 2023. What makes good in-context demonstrations for code intelligence tasks with LLMs?. In *Proceedings of the 38th International Conference on Automated Software Engineering (ASE '23)*. IEEE, Luxembourg, Luxembourg, 761–773. arXiv:2304.07575 [cs]

[17] GitHub. 2024. CodeQL: the libraries and queries that power security researchers around the world, as well as code scanning in GitHub Advanced Security. https://github.com/github/codeql

[18] GitHub. 2025. About code scanning. https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning

[19] Julia Hansbrough. 2024. Finding Bugs in Chrome with CodeQL. https://bughunters.google.com/blog/5085111480877056/finding-bugs-in-chrome-with-codeql

[20] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. ACM, Seoul, Republic of Korea, 271–283. https://doi.org/10.1145/3377811.3380323

[21] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '19)*. IEEE, San Francisco, CA, USA, 539–554. https://doi.org/10.1109/SP.2019.00071

[22] Naman Jain, Shubham Gandhi, Atharv Sonwane, Aditya Kanade, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2023. StaticFixer: From Static Analysis to Static Repair. arXiv:2307.12465 [cs.SE] https://arxiv.org/abs/2307.12465

[23] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '18)*. ACM, Amsterdam, Netherlands, 298–309. https://doi.org/10.1145/3213846.3213871

[24] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE, Melbourne, Australia, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[25] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: end-to-end program repair with LLMs

over retrieval-augmented prompts. arXiv:2303.07263 [cs]

[26] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: multilingual program repair with LLMs. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 4 (June 2023), 5131–5140. https://doi.org/10.1609/aaai.v37i4.25642

[27] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. 2021. On the introduction of automatic program repair in Bloomberg. *IEEE Software* 38, 4 (July 2021), 43–51. https://doi.org/10.1109/MS.2021.3071086

[28] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, Vol. 1. IEEE, Suita, 213–224. https://doi.org/10.1109/SANER.2016.76

[29] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: static analysis-based repair of memory deallocation errors for C. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. ACM, Lake Buena Vista, FL, USA, 95–106. https://doi.org/10.1145/3236024.3236079

[30] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: repairing Java null pointer exceptions without tests. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, Pittsburgh, Pennsylvania, 1532–1544. https://doi.org/10.1145/3510003.3510186

[31] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. LLM-assisted static analysis for detecting security vulnerabilities. arXiv:2405.17238 [cs]

[32] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2021. Mining fix patterns for FindBugs violations. *IEEE Transactions on Software Engineering* 47, 1 (Jan. 2021), 165–188. https://doi.org/10.1109/TSE.2018.2884955

[33] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. IEEE, Hangzhou, China, 1–12. https://doi.org/10.1109/SANER.2019.8667970

[34] Pei Liu, Bo Lin, Yihao Qin, Cheng Weng, and Liqian Chen. 2024. T-RAP: a template-guided retrieval-augmented vulnerability patch generation approach. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware (Internetware '24)*. ACM, Macau, China, 105–114. https://doi.org/10.1145/3671016.3672506

[35] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. MarsCode Agent: AI-native Automated Bug Fixing. https://doi.org/10.48550/arXiv.2409.00899 arXiv:2409.00899 [cs]

[36] Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. 2023. Program repair guided by Datalog-defined static analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. ACM, San Francisco, CA, USA, 1216–1228. https://doi.org/10.1145/3611643.3616363

[37] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[38] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. 2019. Automatically generating fix suggestions in response to static code analysis warnings. In *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM '19)*. IEEE, Cleveland, OH, USA, 34–44. https://doi.org/10.1109/SCAM.2019.00013

[39] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: automated end-to-end repair at scale. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE, Montreal, QC, Canada, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[40] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE, Melbourne, Australia, 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[41] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. ACM, Athens, Greece, 1565–1569. https://doi.org/10.1145/3468264.3473122

[42] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/

[43] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE, Montreal, QC, Canada, 383–387. https://doi.org/10.1109/MSR.2019.00064

[44] Daniel Ramos, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2024. BatFix: Repairing language model-based transpilation. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–29.

[45] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: open foundation models for code. arXiv:2308.12950 [cs]

[46] securego. 2025. gosec - Go Security Checker. https://github.com/securego/gosec

[47] Jessica Shieh. 2024. Best practices for prompt engineering with OpenAI API. https://help.openai.com/en/articles/6654000

[48] Benjamin Steenhoek, Kalpathy Sivaraman, Renata Saldivar Gonzalez, Yevhen Mohylevskyy, Roshanak Zilouchian Moghaddam, and Wei Le. 2024. Closing the Gap: A User Study on the Real-world Usefulness of AI-powered Vulnerability Detection & Repair in the IDE. *arXiv preprint arXiv:2412.14306* (2024).

[49] The MITRE Corporation. 2020. CVE-2020-13946. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13946

[50] Rijnard Van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, Gothenburg, Sweden, 151–162. https://doi.org/10.1145/3180155.3180250

[51] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues Using LLMs. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 789–811. https://doi.org/10.1145/3643762

[52] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (April 2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208

[53] Weishi Wang, Yue Wang, Shafiq Joty, and Steven C.H. Hoi. 2023. RAP-Gen: retrieval-augmented patch generation with CodeT5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. ACM, San Francisco, CA, USA, 146–158. https://doi.org/10.1145/3611643.3616256

[54] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 472–483.

[55] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The plastic surgery hypothesis in the era of large language models. In *Proceedings of the 38th International Conference on Automated Software Engineering (ASE '23)*. IEEE, Luxembourg, Luxembourg, 522–534. https://doi.org/10.1109/ASE56229.2023.00047

[56] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE, Melbourne, Australia, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[57] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the conversation going: fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385 [cs]

[58] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE 17')*. IEEE, Urbana, IL, USA, 660–670. https://doi.org/10.1109/ASE.2017.8115676

[59] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. 2024. Qwen2 Technical Report. arXiv:2407.10671 [cs.CL] https://arxiv.org/abs/2407.10671

[60] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *Proceedings of the 37th International Conference on Machine Learning (ICML '20)*, Vol. 119. PMLR, Vienna, Austria, 10799–10808. https://proceedings.mlr.press/v119/yasunaga20a.html

[61] Jian Zhang, Chong Wang, Anran Li, Wenhan Wang, Tianlin Li, and Yang Liu. 2024. VulAdvisor: Natural Language Suggestion Generation for Software Vulnerability Repair. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1932–1944.

[62] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2024. A critical review of large language model on software engineering: an example from ChatGPT and automated program repair. arXiv:2310.08879 [cs]

[63] Yuntong Zhang, Andreea Costea, Ridwan Shariffdeen, Davin McCall, and Abhik Roychoudhury. 2023. Patch space exploration using static analysis feedback. arXiv:2308.00294 [cs]

[64] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. ACM, Vienna, Austria, 1592–1604. https://doi.org/10.1145/3650212.3680384

[65] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. 2024. Out of sight, out of mind: better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the 46th International Conference on Software Engineering (ICSE '24)*. ACM, Lisbon, Portugal, 1–13. https://doi.org/10.1145/3597503.3639222

[66] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. ACM, Athens, Greece, 341–353. https://doi.org/10.1145/3468264.3468544