VULTERMINATOR: Bringing Back Template-Based Automated Repair for Fixing Java Vulnerabilities

Quang-Cuong Bui
Institute of Software Security
Hamburg University of Technology
Hamburg, Germany
cuong.bui@tuhh.de

Emanuele Iannone
Institute of Software Security
Hamburg University of Technology
Hamburg, Germany
emanuele.iannone@tuhh.de

Riccardo Scandariato
Institute of Software Security
Hamburg University of Technology
Hamburg, Germany
riccardo.scandariato@tuhh.de

***Abstract*—Mainstream techniques for Automated Vulnerability Repair (AVR) lean heavily on Large Language Models (LLMs) and treat the vulnerability repair as a code translation task. Yet, their effectiveness is limited due to the complex nature of vulnerability fixes and, possibly, the lack of training datasets in the Java programming language. On the other hand, template-based Automated Program Repair (APR) remains a popular way to fix general bugs. However, only a few approaches have ever employed vulnerability-specific fix templates. This paper introduces VULTERMINATOR, a novel repair approach for Java vulnerabilities that leverages both *heuristic-based* and *data-driven* fix templates. The former are specialized for certain vulnerability types, such as XML External Entity (XXE) injection, which can be more easily patched with predefined heuristics. The latter aim to repair a broader class of vulnerabilities by generating common patch templates with *masks*, which are later filled by a fine-tuned Masked Language Model (MLM). In this paper, we introduce a total of eleven fix templates distilled from real-world Java patches and evaluate VULTERMINATOR on 106 vulnerabilities with test cases from Vul4J+, as well as on 169 *unseen* vulnerabilities from a newly curated dataset called Vul4JL. VULTERMINATOR achieves the best overall repair performance, outperforming the state-of-the-art approaches by 7% on Vul4J+ and 27% on Vul4JL, as confirmed by manual inspection. VULTERMINATOR managed to fix 10 vulnerabilities in Vul4J+ and 16 in Vul4JL that no other approach could do, mainly due to the contribution of heuristic-based templates.**

***Index Terms*—Automated Vulnerability Repair, Automated Program Repair, Large Language Models**

I. INTRODUCTION

As of October 2025, the National Vulnerability Database (NVD) has disclosed 37,998 new security vulnerabilities in the year 2025 alone, accounting for 12.13% of the entire database. Due to the large volume of new vulnerabilities emerging in recent years, researchers have proposed Automated Vulnerability Repair (AVR) approaches to help developers in fixing software vulnerabilities [1], [2], [3], [4], [5], [6], [7], [8]. Modern techniques often leverage Large Language Models (LLMs) to formulate the repair task as a sequence-to-sequence code generation problem. Namely, they take the vulnerable code as inputs along with additional pieces of information (e.g., the CWE [4]) to generate the allegedly vulnerability-free version. However, the performance of the current *data-hungry* AVR approaches varies across different programming languages, due to constraints related to the availability of large and high-quality training datasets [9]. Indeed, most of them focus on C/C++ programs, largely due to the availability of vulnerability datasets like Big-Vul [10] and PrimeVul [11], whereas only a few target other popular programming languages with less training data, such as Java [12], [7].

On the other hand, template-based Automated Program Repair (APR) has been gaining ground, with numerous mature techniques developed by the research community over the last decade. Notably, TBar [13] is considered the baseline for traditional template-based APR for Java general bugs. Recently, researchers proposed new LLM-based APR approaches guided by the fix templates, which significantly boost repair performance in fixing general bugs [14], [15], [16]. From a general perspective, these techniques first select the fix templates to generate an intermediate patch, which is then fed to transformer models in order to produce final patches via either mask prediction or sequence-based transformation. Just like bugs, vulnerabilities exhibit recurring fix patterns [17], [18], which have the potential to be exploited for AVR.

Therefore, we propose a novel repair technique called **VULTERMINATOR**, which employs both *heuristic-based* and *data-driven* fix templates to repair Java vulnerabilities. The former are specialized for certain vulnerability types, while the latter aim to repair a broader class of vulnerabilities by generating common patch templates with *masks*, which are later filled in by a fine-tuned Masked Language Model (MLM). The design of VULTERMINATOR is motivated by the existence of recurring fix patterns observed in vulnerability patches, which can simplify the repair process in several cases. This intuition is illustrated by the motivating examples in Figure 1. CVE-2017-5662 (an XXE vulnerability) is addressed by configuring `SAXParserFactory` to disable its "external-general-entities" and "external-parameter-entities" features. Common patches for XXE vulnerabilities are often bound to specific Java XML Processing APIs and can be generated using a set of predefined heuristic-based fix templates. Instead, CVE-2013-4378 (an XSS vulnerability) adds a call to the sanitizing method `htmlEncodeButNotSpace`. Sanitization applies not only to XSS but also to other types of vulnerabilities, such as SQL Injection and Command Injection. In this case, data-driven fix templates can be used to first generate the template code with masks (e.g., `<mask0>(remoteAddr)` to signal

```
// Human patch of CVE-2017-5662 (XXE vulnerability)
try {
  saxParser = saxFactory.newSAXParser();
  parser = saxParser.getXMLReader();
  ...
+ parser.setFeature("http://xml.org/sax/features/" +
+    "external-general-entities", false);
+ parser.setFeature("http://xml.org/sax/features/" +
+    "external-parameter-entities", false);
catch (SAXException e) {
  e.printStackTrace();
}

// Human patch of CVE-2013-4378 (XSS vulnerability)
- write(remoteAddr);
+ write(htmlEncodeButNotSpace(remoteAddr));
```

Fig. 1: Patches for CVE-2017-5662 and CVE-2013-4378.

that a sanitization method must be added). Then, a deep-learning model is used to predict the concrete name of the sanitization method via masked language modeling, leveraging the vulnerability information and the surrounding context.

We defined eleven fix templates—six heuristic-based and five data-driven—derived from the findings in the ExtraVul dataset [17], and implemented them into VULTERMINATOR to repair Java security vulnerabilities. The MLM that fills the masks has been fine-tuned on a dataset of 4,147 change pattern instances mined with the COMING tool [19] from 16,466 real-world vulnerability patches. During the mask prediction, some elements of the source code of the belonging project are also provided in the model input, serving as additional contexts.

To evaluate the repair effectiveness of VULTERMINATOR, we compare it with five state-of-the-art APR and AVR techniques, as well as two leading LLMs widely used in program repair studies, on 106 real-world Java vulnerabilities with available vulnerability-witnessing tests (a.k.a. Proof-of-Vulnerability tests) from the Vul4J+ dataset [20], which is an extension of the popular Vul4J dataset [21]. The candidate patches are evaluated against the existing project test suite, which includes vulnerability-witnessing tests that confirm the removal of the vulnerability. The plausible patches were also manually verified to ascertain correctness. Then, to assess the generalizability of VULTERMINATOR, we also perform an extended evaluation on Vul4JL, a dataset of 169 vulnerabilities that we prepared in this study. The experimental results indicate that VULTERMINATOR achieves the best vulnerability repair performance in both datasets, correctly fixing 31 vulnerabilities in Vul4J+ and 47 in Vul4JL. This outperforms the best-performing baseline (Gemini-2.5 Pro) by 6.9% and 27.02%, respectively. During further investigations, we found that the heuristic-based templates have the greatest impact on the success of VULTERMINATOR. Although data-driven templates generally contributed less than heuristic-based ones, several of them demonstrate great repair effectiveness on Vul4JL. For example, the data-driven fix template *SanitizeInput* repairs 2.5× more vulnerabilities in Vul4JL than in Vul4J+.

In summary, the paper makes the following contributions:

- *Novel AVR Technique.* We introduce VULTERMINATOR, a novel technique that leverages both heuristic-based and data-driven fix templates to repair Java vulnerabilities. The latter are aided by a deep-learning model to fill missing parts in the generated patches.
- *New Vulnerability Fix Dataset.* We collect a new dataset of 16,466 Java vulnerability-fixing commits (containing 62,286 source file pairs), which is curated from open-source repositories on GitHub.
- *Extensive Evaluation.* We conduct extensive evaluations of our proposed approach and the baselines on two datasets: Vul4J+, containing 106 vulnerabilities with runnable vulnerability-witnessing tests, and Vul4JL, containing 169 vulnerabilities that *are not seen* by fix templates. We also perform an investigation to reveal the contributions of individual fix templates to the overall repair effectiveness of VULTERMINATOR.

II. VULTERMINATOR

A. Overview

In this section, we present VULTERMINATOR, a novel repair technique for Java vulnerabilities that leverages both *heuristic-based* and *data-driven* fix templates. A heuristic-based fix template consists of a set of predefined transform rules at the abstract syntax tree (AST) level that patch the vulnerable code. A data-driven fix template, instead, consists of an intermediate representation (i.e., the template code) of the intended patch and contains *masks*, which are later filled using a large language model (LLM). The code tokens that the model generates for these masks are called *fix ingredients*—e.g., a method name or a conditional expression.

The main workflow of VULTERMINATOR is depicted in Figure 2. VULTERMINATOR accepts as inputs the source code of the vulnerable project and vulnerability information, such as vulnerable locations and the CWE ID (if available). First, in the **Fix Template Selection** step (detailed in Section II-D), the project's source code is transformed into ASTs, on which the vulnerable ASTs are identified. Then VULTERMINATOR selects the suitable heuristic-based and data-driven templates for the vulnerable ASTs from the *fix template database* (see Section II-D). Next, during the **Patch Generation** (detailed in Section II-E), the selected fix templates perform the code mutations to generate the patch candidates. Finally, the generated patch candidates are validated by the set of vulnerability-witnessing tests and the regression test suite in the **Patch Validation** phase (detailed in Section II-F).

In this work, we identified a total of eleven fix templates for common Java security vulnerabilities and implemented them in VULTERMINATOR. The templates formalize the repair patterns previously mined from the ExtraVul dataset [17]. In total, we designed six heuristic-based templates and five data-driven templates. Some of which have *variants*, i.e., ways to implement the template, as described in Sections II-B and II-C.

B. Heuristic-based Fix Templates

FT1. Prevent XXE Vulnerabilities. XML External Entity (XXE) injection vulnerabilities (CWE-611) are usually caused by an XML parser that is not properly configured. This vulnerability type is rather popular among the Java open-source

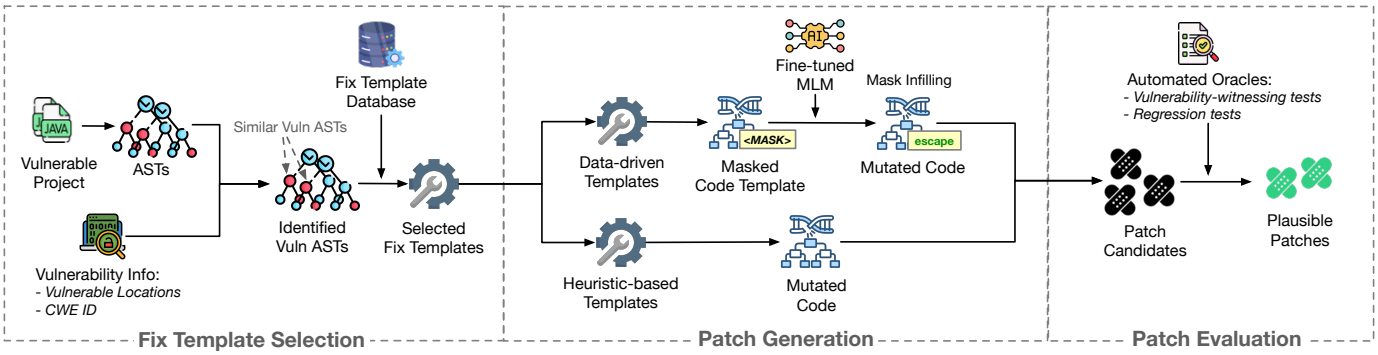


Fig. 2: Overall workflow of VULTERMINATOR.

projects [17]. Nevertheless, it is relatively easy to address by configuring the XML parsers to disallow custom document type definitions (DTDs). Following the guidelines by OWASP XXE Prevention Cheatsheet [22], VULTERMINATOR supports three variants of fixes for six common XML parsers in Java. The following code snippet shows the three possible fixes when `DocumentBuilderFactory` is used. While **FT1.1** is considered the main defense by disallowing all DTDs, **FT1.2** and **FT1.3**, or the combination of them, offer more flexibility by disabling only external entities and external DTDs, allowing internal DTDs to be loaded if required. Due to space constraints, the list applied for other kinds of XML parsers is provided in the replication package [23].

```
DocumentBuilderFactory dbf =
↪  DocumentBuilderFactory.newInstance();
+ try {
FT1.1: + dbf.setFeature("...disallow-doctype-decl", true);
FT1.2: + dbf.setFeature("...general-entities", false);
       + dbf.setFeature("...parameter-entities", false);
FT1.3: + dbf.setFeature("...load-external-dtd", false);
+ } catch (ParserConfigurationException e) {/* Handling */}
```

FT2. Instantiate Secure Random Generator. Standard Pseudo-Random Number Generators (PRNGs), such as `java.util.Random`, rely on statistical algorithms to generate the next number, which is predictable. A simple solution is to use a cryptographic PRNG algorithm, such as in `java.security.SecureRandom` instead. This ensures the output generated is cryptographically secure and impossible to guess.

```
- Random ran = new Random();
+ Random ran = new SecureRandom();
```

FT3. Instantiate SnakeYaml Parser with Secure Configuration. `SnakeYaml` is a Java library widely used for parsing data stored in YAML files. When instantiated with the default configuration, a `SnakeYAML` parser is vulnerable to Deserialization of Untrusted Data (CWE-502). In particular, it allows the instantiation of arbitrary classes from untrusted YAML sources, which can lead to Remote Code Execution attacks. This critical vulnerability in `SnakeYaml` is officially disclosed via the CVE-2022-1471 [24]. To resolve this, the `SafeConstructor` class can be used to disable the deserialization of any class. We defined two variants of the fix templates: If no argument is passed to the parser constructor, **FT3.1** adds `new SafeConstructor()`, otherwise, **FT3.2** analyzes the given arguments `options` and possibly instantiates extra required arguments such as `Representer` and `DumperOptions` to fit the correct parser constructor.

```
FT3.1: - Yaml yaml = new Yaml();
       + Yaml yaml = new Yaml(new SafeConstructor());
FT3.2: - Yaml yaml = new Yaml(options);
       + Yaml yaml = new Yaml(new SafeConstructor(),
                              arg1, arg2,..., options);
```

FT4. Prevent Path Traversal Vulnerabilities. Path traversal vulnerabilities (CWE-22) are caused by an insufficient validation of untrusted input, which allows an attacker to access or write to arbitrary files or directories on the file system. We defined three variants for fixing this. For full path traversal, **FT4.1** can be used to reject file names input containing bad characters, i.e., the two dots "..", and the slashes "/", "\". For partial traversal, **FT4.2** ensures the newly created file path always begins with the base directory's path. Note that the paths need to be normalized and validated using the proper Java File API. Specifically, the method `getCanonicalPath`, which was widely used to detect path traversal, is no longer considered sufficient—as shown in CVE-2022-31159 [25]. This check should be done via the new Path API from the Java NIO package, such as `toPath().normalize()` instead. The fix template **FT4.3** helps improve deprecated fixes that rely on `getCanonicalPath`.

```
// Full Path Traversal
File file = new File(name);
FT4.1: + if (name.contains("..") || name.contains("/")
       +     || name.contains("\\")) {
       +   throw new IOException("Vul Prevented!");
       + }
// Partial Path Traversal
File file = new File(dir, name);
FT4.2: + if (!file.toPath().normalize()
       +     .startsWith(dir.toPath().normalize())) {
       +   throw new IOException("Vul Prevented!");
       + }
FT4.3: - if (!file.getCanonicalPath()
       -     .startsWith(dir.getCanonicalPath())) {
       + if (!file.toPath().normalize()
       +     .startsWith(dir.toPath().normalize())) {
       +   throw new IOException("Vul Prevented!");
       + }
```

FT5. Prevent Insecure Temporary File Creation. By default, the use of `File.createTempFile` to create temporary files is insecure in many Linux/Unix systems [26]. Indeed, the temporary file is created in the shared directory `/tmp` with default access permissions that are world-readable, allowing

unauthorized parties to access sensitive data. It is recommended to use the `Files.createTempFile` from the Java NIO package to create temporary files with more restrictive access permissions [27].

```
- File tempFile = File.createTempFile(prefix, suffix);
+ File tempFile = Files.createTempFile(prefix,
↪  suffix).toFile();
```

FT6. Prevent the Exposure of Sensitive Data. In many cases, sensitive data is exposed to unauthorized parties. This is caused either accidentally by developers during the debugging process or by a lack of proper authentication checks. To mitigate this vulnerability, **FT6.1** and **FT6.2** remove sensitive data from being written to the public endpoints of WebUI, API, or Logging System (e.g., `rsp.getWriter().write()`, `logger.info()`).

```
// public_writer() writes to public or to logs
FT6.1: - public_writer(var1 + sen_data + var2 +...);
       + public_writer(var1 + var2 +...);
FT6.2: - public_writer(sen_data1 + sen_data2 +...);
```

C. Data-driven Fix Templates

In contrast to heuristic-based templates, which target specific vulnerability types caused by improper use of Java APIs (e.g., XML parsers and XXE), data-driven fix templates are more generic and can address a broader range of vulnerabilities. However, this also results in a large search space when it comes to selecting the right fix ingredients, e.g., due to the variety of libraries and frameworks that can be used in projects. To address this, we used a Masked Language Model (MLM) model to assist in identifying the correct fix ingredients for the templates.

Table I summarizes our five data-driven fix templates. **FT7** sanitizes untrusted inputs to help mitigate injection vulnerabilities such as Cross-Site Scripting and SQL Injection, **FT8** breaks the infinite loop that can lead to Denial-of-Service attacks, and **FT9** adds checks to ensure code execution occurs under proper permissions. The last two templates (**FT10** and **FT11**) are commonly used in both vulnerability and general bug fixes [17], [28]. They prevent the program execution from reaching a vulnerable state by throwing exceptions or returning errors. For each fix template, the "Template Code" column in Table I shows the fixed code with *masks*. Some templates may have two *masks*, for example, **FT10** uses <*mask0*> as a placeholder for the precondition expression and <*mask1*> for the thrown exception type. The "Fix Ingredients" column shows the potential fix ingredients and the scopes where they are retrieved. These fix ingredients are added to the MLM's input through in-line comments to guide the model in sampling the right tokens. For example, since **FT7.1** aims to sanitize untrusted inputs (mostly stored as `String`), VULTERMINATOR retrieves all the *string-handling* methods that are static and exist in the codebase and libraries as potential ingredients. We did not constrain the MLM to forcefully select an ingredient from the suggested list, as that list may be incomplete.

To build a model capable of predicting fix ingredients, we first construct a dataset of Java vulnerability fixes that fit the templates in Table I. We then replace the key code components corresponding to each fix template with *masks* and fine-tune a pre-trained Masked Language Model to predict them as target fix ingredients. We selected UNIXCODER [29] for a handful of reasons: (1) UniXCoder was pre-trained of several programming languages, including Java, (2) it has been pre-trained with an MLM task, making it ideal for our case; and (3) it has demonstrated strong capabilities in the program repair domain [14], [30], [31], making it promising also for the vulnerability repair task as well.

Dataset of Vulnerability Fixes. We searched vulnerability-fixing commits in popular Java repositories on GitHub using a set of vulnerability- and patch-related keywords. We followed these steps: **Step 1.** We used SEART GitHub Search Engine [32] to obtain the most popular Java projects on GitHub. We set our search criteria as: (i) the repository must contain Java code, (ii) it should have at least 1,000 commits and at least 100 stars, and (iii) its most recent commit should fall between January 1st, 2014, and June 30, 2025. We ended up with 3,322 repositories that met these criteria. **Step 2.** We then downloaded all the repositories and used `git-vuln-finder` [33], a lightweight tool that finds vulnerability-relevant commits based on matching the commit messages to predefined security-related keywords proposed by Zhou et al. [34]. We obtained 96,058 security-related commits. **Step 3.** Next, we retained only the commits whose message started with vulnerability-fixing verbs, e.g., *fix*, *prevent*, *disallow*, *validate* (the full list can be found in our replication package [23]). The set of verbs was extracted from the commit messages in Project KB [35], which is a dataset of curated Java vulnerability fixes. Then, we *discarded* the commits that: (i) were related to Android and Games; (ii) contained *javadoc*, *typo*, and *lint* in their messages, as these commits do not fix any vulnerabilities but, rather, only change comments and typos; and (iii) existed in our testing datasets (see Section III-A) to avoid overfitting. After this process, 16,466 commits were selected. **Step 4.** We analyzed the changes in the commits and retained only those related to Java source files. We stored both the vulnerable and fixed versions for each file. In total, we end up with 62,286 Java file pairs. **Step 5.** We used the COMING tool [19] to mine the fix template instances from the file pairs. We specified the eight variants for our five data-driven fix templates via XML files, as required by COMING. For a file pair, only the fix templates corresponding to the keywords previously matched by `git-vuln-finder` in the their belong commit are mined (e.g., if "xss" is matched in the commit message, then only **FT7** is mined, the full mapping list is provided in our replication package [23]). Ultimately, we collected 4,147 repair instances (COMING's specific format) that are representative of our data-driven fix templates.

Model Fine-tuning. We prepared the input for fine-tuning the UNIXCODER model. Given each repair instance obtained by COMING, we obtain information about the matched fix template, and the corresponding code change found in the vulnerability patch, along with its method. Similar to existing LLM-based repair works [14], [16], [34], we treat the

TABLE I: List of Data-driven Fix Templates.

ID	Template Name	Fix Ingredients	Template Code

FT7	SANITIZEINPUT	*mask0*: Static method takes a `String` as a parameter and returns a `String`
		Scope: Project and External Libs
			FT7.1: `- method(unstrusted_input);`
			`+ method(<mask0>(unstrusted_input));`
			FT7.2: `- method(unstrusted_input);`
			`+ method(unstrusted_input.<mask0>());`

FT8	BREAKINFINITELOOP	*mask0*: Atomic Conditional Expr.
		Scope: Current File, Related Files
			FT8.1: `+ break;`
			FT8.2: `+ if (<mask0>) {`
			`+ break;`
			`+ }`

FT9	CHECKPERMISSION	*mask0*: Method Calls
		mask1: Atomic Conditional Expr.
		Scope: Current File, Related Files
			FT9.1: `+ <mask0>();`
			FT9.2: `+ if (<mask1>) {`
			`+ <mask0>();`
			`+ }`

FT10	ADDIF_THROW	*mask0*: Atomic Conditional Expr.
		mask1: Declared Exception Types
		Scope: Current File, Related Files
			`+ if (<mask0>) {`
			`+ throw new <mask1>("Vul prevented!");`
			`+ }`

FT11	ADDIF_RETURN	*mask0*: Atomic Conditional Expr.
		mask1: Variable Names, `false`, `-1`
		Scope: Current File, Related Files
			`+ if (<mask0>) {`
			`+ return <mask1>;`
			`+ }`

whole method code as the main source input for the model. Depending on the fix template, we mask out the code tokens at specific locations to generate code templates. We set the masked tokens as the ground truth fix ingredients that the model is expected to generate. Moreover, the model is given an additional piece of information, i.e., potential fix ingredients mined from the project hosting the vulnerable method (see Table I). This aims to address a common issue of transformer models in program repair, i.e., the tendency to generate patches that are not customized for the project codebase, as reported in a recent study [36]. For example, the fix may introduce a new method call for input sanitization that does not exist in the codebase, causing compilation errors. Providing the relevant code tokens in the input can help the model to prioritize and reuse them, hence generating codebase-friendly patches. We extend the model's special tokens to include `<mask0>` and `<mask1>` to prevent the masks from being split during tokenization (performed using Byte Pair Encoding). Figure 3 shows an entry of our fine-tuning dataset.

```
// ----------- Source -----------
// FIX_TEMPLATE: SanitizeInput
// FIX_INGREDIENTS: HTMLUtils decodeString urlEncode
↪  escapeHtml trimWhitespace getStringForJS...
private void writeSession(String input) {
  write(<mask0>(input));
}
// ----------- Target -----------
<mask0>HTMLUtils.escapeHtml
```

Fig. 3: Input example given to the model for fine-tuning.

We fine-tune the model with 30 epochs and set the learning rate as $5e^{-5}$. We use *cross-entropy loss* for fine-tuning and also employ a validation set (10% of the dataset) to check the model's performance after each epoch. We select the checkpoint with the highest BLEU score [37].

D. Fix Template Selection

For the given vulnerable ASTs, VULTERMINATOR checks the most suitable fix templates from the *fix template database*. Similar to TBar [13] and GAMMA [14], VULTERMINATOR adopts an AST matching approach. Specifically, for each fix template, VULTERMINATOR checks if the required context for the fix template (e.g., node type) matches the existing context in the vulnerable ASTs to determine if the fix template should be selected. For example, if the vulnerability is located within a loop (e.g., parent AST node type is `WhileStatement`), the fix template **FT8** (*BreakInfiniteLoop*) should be applied to address the potential infinite loop vulnerabilities. Multiple fix templates may be applicable to the same vulnerable AST; in such cases, the selection is determined by their precedence within the fix template database. Moreover, if the vulnerability type (i.e., CWE-ID) is available, VULTERMINATOR prioritizes the corresponding fix template for that vulnerability type during template selection. For example, if the vulnerability type is given as CWE-611, the fix template **FT1** (*Prevent XXE Vulnerabilities*) will be prioritized in the first place to generate XXE patches. This strategy helps VULTERMINATOR produce the correct patches more quickly by selecting the proper fix template.

Template Selection for Multi-location Vulnerability. VULTERMINATOR can also fix vulnerabilities that require similar fixes in multiple locations. This is based on the nature of some vulnerability fixes where developers try to fix the same vulnerability in multiple places [38], such as in the patch of CVE-2017-1000207 [39]). To this end, we first utilize the SRC2ABS tool [40] to abstract the code at vulnerable locations, then rebuild the vulnerable ASTs and run Algorithm 1 to compute their similarities. Specifically, we count how many nodes in the left-hand side AST have the same types as their corresponding nodes in the right-hand side AST (the node name is also taken into account if the node is a `MethodInvocation`). The resulting count is then normalized to a similarity score ranging from 0 to 1. We consider two ASTs similar if they have their similarity score is equal to or greater than a threshold of 0.9.

Algorithm 1: Similarity Computation for ASTs

Input : A pair of AST nodes (N_1, N_2)
Output: Normalized similarity score in range $[0, 1]$

Function `computeSimilarity`(N_1, N_2)
  $score, maxScore \leftarrow 0$;
  $minHeight \leftarrow \min(N_1.height, N_2.height)$;
  $maxScore \leftarrow maxScore + minHeight$;
  if $N_1.type = N_2.type$ **then**
    $score \leftarrow score + minHeight$;
  if $N_1.isLeaf() \wedge N_2.isLeaf() \vee N_1.type = MethodInvocation \wedge N_1.type = N_2.type$ **then**
    $maxScore \leftarrow maxScore + minHeight$;
    if $N_1.name = N_2.name$ **then**
      $score \leftarrow score + minHeight$;
  else if $N_1.isNonLeaf() \wedge N_2.isNonLeaf()$ **then**
    for $i \leftarrow 0$ **to** $minChildrenSize(N_1, N_2)$ **do**
      $computeSimilarity(N_1.child[i], N_2.child[i])$;
  return $score/maxScore$;

E. Patch Generation

VULTERMINATOR applies the selected templates to generate patch candidates on the vulnerable locations. During the patch generation phase, heuristic-based templates traverse the ASTs of the given vulnerable statements to find suspicious code expressions (e.g., the `new SnakeYaml()` object instantiation expression in **FT3**). They then apply the predefined set of code mutations on these code components to generate the concrete patch candidates. The ingredients used for synthesizing the patches are usually collected from the vulnerable statements themselves. Instead, data-driven templates require a more complex process to generate patches. First, these templates are used to add the masked tokens to the vulnerable method. The masked code is extracted along with its context method, and then the whole method is fed as the input to the model. VULTERMINATOR also adds the fix template name and potential ingredient list at the beginning of the model input in a comment (akin to Figure 3). The list of fix ingredients retrieved varies according to the selected fix template (see Table I). VULTERMINATOR employs the beam search strategy to generate values for the masked tokens. Specifically, the beam size parameter $\mathcal{N}$ is set to predict the top-$\mathcal{N}$ code tokens with the highest probabilities. They are later integrated into the code template to generate patch candidates, which are subsequently validated for correctness.

F. Patch Validation

To validate the patches, VULTERMINATOR follows the common workflow used in existing APR and AVR studies to evaluate the generated patches [17], [41], [42]. In particular, for each patch candidate generated by the fix templates, VULTERMINATOR first attempts to install the patch into the originating project and then recompiles it to quickly filter out candidate patches that cause compilation errors. If the compilation succeeds, VULTERMINATOR runs any available vulnerability-witnessing tests and the whole test suite. The patches that pass all the tests are said to be *plausible* and are ultimately returned. In our study, the plausible patches undergo an additional, manual correctness assessment (see Section III-C).

III. VALIDATION METHODOLOGY

To evaluate VULTERMINATOR's effectiveness in repairing vulnerabilities, we formulate the following research questions:

- ***RQ1:*** *How well does* VULTERMINATOR *fix Java vulnerabilities compared to the state-of-the-art AVR and APR approaches?*
- ***RQ2:*** *How does* VULTERMINATOR *perform in fixing vulnerabilities of different types?*
- ***RQ3:*** *To what extent are the vulnerabilities fixed by* VULTERMINATOR *unique compared to the state-of-the-art AVR and APR approaches?*
- ***RQ4:*** *How does each fix template contribute to the overall repair performance of* VULTERMINATOR*?*

A. Evaluation Datasets

To evaluate the repair performance of VULTERMINATOR, we first use 106 real-world vulnerabilities with the reproducible *vulnerability-witnessing tests* (a.k.a. Proof-of-Vulnerability test) from the **Vul4J+** dataset [20]. The tests serve as an automated oracle to assess the plausibility of patch candidates generated by the repair tools. Each vulnerability in Vul4J+ has one or more associated witnessing tests. Vul4J+ is an extended version of the original Vul4J dataset (that contained 79 vulnerabilities), which has been widely used in AVR studies since its release [17], [4], [43], [16], [36]. To the best of our knowledge, Vul4J+ is the dataset with the largest number of reproducible real-world Java vulnerabilities to date.

The fix templates employed by VULTERMINATOR have been extracted from the ExtraVul dataset [17], which, like Vul4J+, is also related to Vul4J. This means that the templates might have an advantage for some of the vulnerabilities in Vul4J+. Therefore, we created a new dataset, called **Vul4JL**, which contains vulnerabilities "unseen" by our fix templates. To this end, we followed the methodology described in the original Vul4J paper [21] to collect new Java vulnerabilities from REEF [44] and ReposVul [45] datasets (however, we did not create the vulnerability-witnessing tests, as discussed later in Section III-C). We then excluded the vulnerabilities that are also present in Vul4J+ and ExtraVul, as well as those whose patches modified more than seven lines—as they are likely to contain changes irrelevant to the fix. As a result, Vul4JL contains 169 real-world vulnerabilities, mostly containing vulnerabilities disclosed between 2019 and 2023. The size of VuL4JL is comparable to those used in the literature for Java vulnerability repair. Indeed, the largest Java dataset employed so far contains 150 mutated vulnerabilities derived from 50 original vulnerabilities [43].

B. Baselines

We compared our approach against the state-of-the-art APR and AVR techniques, as well as the leading LLMs. We selected three APR tools: TBar [13], GAMMA [14], and NTR [16].

TBar is a template-based approach for traditional APR techniques that achieves the best performance in repairing Java vulnerabilities [17]. GAMMA and NTR are recent state-of-the-art learning-based APR approaches guided by fix templates. Moreover, we selected two LLMs that are widely used in program repair studies [46], [47], [48], i.e., GPT-4o [49] and Gemini-2.5 Pro [50]. GPT-4o is the successor of Codex—the best-performing LLM for repairing Java vulnerabilities as reported in a recent study [43], while Gemini-2.5 Pro [50] is the leading LLM excelling at complex reasoning and coding tasks. Additionally, we also involved two state-of-the-art repair techniques specialized for fixing Java vulnerabilities, i.e., VulRepair$_{NTR}$ [16] (a replicated version of the original VulRepair [2] by Huang et al. [16]) and VulMaster [4]. Unfortunately, we were unable to execute them on our datasets due to the lack of runnable artifacts for their experiments on Java code. The best we could do was to extract the repair performance reported in previous studies [4], [16]. Such performance originates from a dataset of 35 single-hunk Java vulnerabilities, which is also a subset of Vul4J+. In total, we compare VULTERMINATOR against five different repair tools (three of which are reproducible) and two LLMs.

The tools and models were run under *perfect fault localization*, i.e., the exact vulnerable locations are provided in the input code to repair. This is the preferred setting among APR and AVR studies since it eliminates biases introduced by differences in fault localization across techniques [51], [43], [17]. Moreover, the the CWE ID and name are also provided to the tools that can accept it. In our case, this applied to GPT-4o and Gemini-2.5 Pro prompts.

We ran GAMMA and NTR with a beam size of 100, which is similar to the number used in their original studies [14], [16]. Note that the number of patch candidates can be much larger. For example, NTR employs ten fix templates and generates 100 patches for each template, resulting in a total of 1,000 patches. Due to computational costs, we queried GPT-4o and Gemini-2.5 Pro ten times with the same prompt to generate ten patches for each vulnerability (the prompt template is reported in our replication package [23]). To make a fair comparison with all the baselines, we set the beam size for VULTERMINATOR to 10.

C. Evaluation Protocol

We use two common metrics used in APR studies [42], [41], i.e., the number of *correct patches* and the number of *plausible patches*. For Vul4J+, we let the tools generate all the patches, then run the vulnerability-witnessing tests and the whole test suite against the patches until we find the first plausible patches that pass all tests. We then manually inspect these patches to identify those that correctly fix the vulnerabilities while preserving the project's functionalities. For Vul4JL, since no tests are available to screen the generated patches, we cannot assess the plausibility. We acknowledge that the *exact match* metric [2] is often used to evaluate generated patches statically. However, it primarily relies on syntactic similarity and may miss the semantically correct patches that have different syntax compared to the ground truth. Therefore, we resorted to manual inspection once more. However, in the context of this study, it was not feasible to inspect all generated candidate patches due to their large number—for example, NTR can produce up to 1,000 patches for a single vulnerability. Therefore, we selected only the *first three patches* yielded by each tool. In total, we assessed the correctness for 169×6×3 = 3,042 patches. Supported by the related work, we argue that considering only the *top three fix suggestions* is reasonable in practice [52]. We acknowledge that this choice may affect the performance of some tools, including our own VULTERMINATOR, but it ensures that all techniques are treated equally in this study.

IV. EVALUATION RESULTS

A. RQ1: General Repair Effectiveness

Since we have repair results from VulMaster and VulRepair$_{NTR}$ only for 35 single-hunk vulnerabilities from Vul4J+, we first show the results of all the evaluated techniques solely on this subset. This is summarized in Table II. Overall, we observe that VULTERMINATOR (17 successful repairs) outperforms LLMs (up to 15 repairs), which in turn outperform state-of-the-art repair tools (up to 11 repairs). VULTERMINATOR also performs better than the other tools that specialize in vulnerability repair.

Table III shows the repair results on the whole Vul4J+ and Vul4JL datasets (including multi-hunk vulnerabilities) for the approaches that we could re-run. On the Vul4J+ dataset, VULTERMINATOR outperforms all the baselines with 31 correctly fixed vulnerabilities (90.6% success rate). Performance-wise, LLMs represent the second tier, while TBar and GAMMA remain as the lower performers, which is not surprising as these tools ar not specialized for vulnerabilities.

On Vul4JL, the number of plausible patches is not reported since there are no vulnerability-witnessing tests. Overall, we observed the same trend as for Vul4J+. Specifically, VULTERMINATOR achieves the best repair performance with 47 correctly fixed (28% success rate). Gemini-2.5 Pro and GPT-4o follow with 37 (22%) and 28 (17%) vulnerabilities fixed.

Figures 4 and 5 report two vulnerabilities from Vul4JL that only VULTERMINATOR was able to fix correctly, concerning CVE-2019-10077 (an XSS vulnerability) and CVE-2022-4878 (a path traversal vulnerability), respectively. In both of the examples, we observe that the patches made by GPT-4o and Gemini-2.5 Pro understand the vulnerability context, however, they fail to complete the fixes or generated obsolete patches. In particular, for the XSS vulnerability, they suggest invoking the `escapeHtml` and `TextUtil.encodeForHTML`, which are not present in the codebase. On the other hand, VULTERMINATOR suggests a set of fix ingredients retrieved from the affected project, and then UniXcoder model (the masked language model) chooses the right one, which is `TextUtil.escapeHTMLEntities`. For the path traversal vulnerability, the patches generated by these LLMs are deemed insecure (see FT4 in Section II-B for details), whereas our tool leverages up-to-date knowledge to produce a secure fix.

TABLE II: Repair results on 35 single-hunk vulnerabilities (a subset of Vul4J+).
*The repair results of VulMaster and VulRepair$_{NTR}$ are extracted from previous works [4], [16].

Tools/Models	*Template-guided APR*			*SOTA AVR*		*LLMs*		*Our Tool*
	TBar	**GAMMA**	**NTR**	**VulMaster**	**VulRepair$_{NTR}$**	**GPT-4o**	**Gemini-2.5 Pro**	**VulTerminator**
#Correct/#Plausible	5/8	6/9	11/14	9/-*	4/10*	13/14	15/17	17/19

TABLE III: Repair results on the Vul4J+ and Vul4JL datasets.

Tools/Models	**Vul4J+** (106 Vulns)	**Vul4JL** (169 Vulns)
	#Correct/#Plausible	#Correct
TBar	5/11	1
GAMMA	6/10	2
NTR	16/20	7
GPT-4o	17/22	28
Gemini-2.5 Pro	29/32	37
VulTerminator	31/35	47

```
// GPT-4o's patch
- Object[] args = { extWiki };
+ Object[] args = { escapeHtml(extWiki) };

// Gemini-2.5 Pro's patch
- Object[] args = { extWiki };
+ Object[] args = { TextUtil.encodeForHTML(extWiki) };

// VulTerminator's patch, identical to developer's patch
- Object[] args = { extWiki };
+ Object[] args = { TextUtil.escapeHTMLEntities(extWiki) };
```

Fig. 4: Patches generated for CVE-2019-10077.

B. RQ2: Repair Effectiveness by Vulnerability Type

We performed a further investigation into the vulnerability types (CWE) that can be fixed by VulTerminator. To this end, as shown in Table IV, we zoom into the top ten most frequent CWEs in Vul4JL and show the proportion of correct repairs for VulTerminator, GPT-4o and Gemini-2.5 Pro (the best performing approaches). Nine out of ten vulnerability categories also appear in the latest OWASP Top 10 Web Application Security Risks (2021) [53], which represent the most common and impactful web-based vulnerability types in practice. This shows how the Vul4JL dataset reflects Java vulnerabilities in the real world reasonably well.

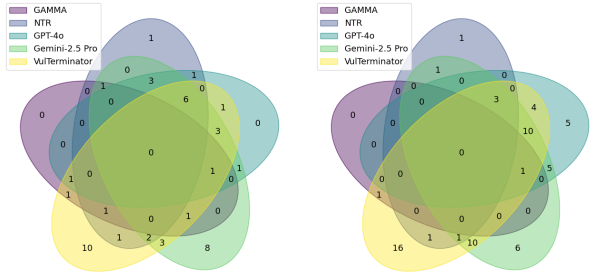
Overall, VulTerminator was able to fix 36.46% of the top ten vulnerability types Vul4JL, while GPT-4o and Gemini-2.5 Pro fixed 13.54% and 22.92%, respectively. In particular, we find that VulTerminator stand out for CWE-22 *(Path Traversal)* and CWE-611 *(XXE)*, with high success rates of 68.75% and 81.75%, respectively. In these categories, GPT-4o and Gemini-2.5 Pro are outperformed by a wide margin. VulTerminator demonstrates comparable repair performance with the LLMs for CWE-79, CWE-502, and CWE-918; however, it also shows inferior performance for CWE-89 and CWE-203—though these types only had a few vulnerabilities. From a broader perspective, we can see that the LLMs have a more balanced repair performance across the top ten CWEs.

```
// Developer's patch
+ destDir = destDir.toPath().normalize().toFile();

// Patch generated by GPT-4o and Gemini-2.5 Pro
+ String destDirPath = destDir.getCanonicalPath();
+ String destFilePath = file.getCanonicalPath();
+ if (!destFilePath().startsWith(destDirPath)) {
+   throw new IOException("Exception message");
+ }

// VulTerminator's patch, correctly fix the vulnerability
+ if (!file.toPath().normalize()
+     .startsWith(destDir.toPath().normalize())) {
+   throw new IOException("Exception message");
+ }
```

Fig. 5: Patches generated for CVE-2022-4878.



(a) In Vul4J+ (106 Vuls) (b) In Vul4JL (169 Vuls)

Fig. 6: Overlaps of vulnerabilities fixed by different approaches.

C. RQ3: Analysis of Complementarity

To investigate the extent to which VulTerminator complements the existing repair approaches (i.e., its degree of uniqueness), we calculate the number of overlapping fixed vulnerabilities in Vul4J+ and Vul4L. Figure 6 shows the overlaps with Venn diagrams. For the sake of readability, we report only the overlap between VulTerminator and the four best repair approaches, i.e., GAMMA, NTR, GPT-4o, and Gemini-2.5 Pro. The figure shows that **VulTerminator was able to fix the largest number of *unique* vulnerabilities in both datasets**, i.e., 10 in Vul4J+ and 16 in Vul4JL. The runner-up was Gemini-2.5 Pro, which uniquely fixes 8 vulnerabilities in Vul4J+ and 6 in Vul4JL. GPT-4o repairs 5 unique vulnerabilities in Vul4JL and none in Vul4J+. NTR has 1 unique vulnerability per dataset, while GAMMA has none.

In Figure 7, we further analyzed the overlapping fixed vulnerabilities by focusing on VulTerminator, Gemini-2.5 Pro, and GPT-4o. In particular, we broke down the type of fix templates (heuristic-based vs. data-driven) that VulTerminator employed to repair the vulnerabilities. This helped to understand which part of VulTerminator contributed to its complementarity with other approaches. The figure shows

TABLE IV: Repair performance on different types of vulnerabilities in the Vul4JL dataset.

The CWEs are ranked according to their frequency in the dataset. **OWASP10*** denotes whether the CWE is listed in the ten most critical Web Application Security Risks for 2021, as published by OWASP [53].

Rank	CWE ID	CWE Name	Count	OWASP10*	GPT-4o	Gemini-2.5 Pro	VulTerminator
1	CWE-79	Cross-site Scripting (XSS)	20	✓	20%	15%	20%
2	CWE-22	Path Traversal	16	✓	12.5%	18.75%	68.75%
3	CWE-502	Deserialization of Untrusted Data	16	✓	6.25%	18.75%	18.75%
4	CWE-611	XML External Entity Injection (XXE)	11	✓	27.27%	36.36%	81.82%
5	CWE-200	Exposure of Sensitive Information	8	✓	0%	0%	100%
6	CWE-918	Server-Side Request Forgery (SSRF)	6	✓	0%	16.67%	16.67%
7	CWE-20	Improper Input Validation	5	✓	0%	0%	20%
8	CWE-668	Exposure of Resource to Wrong Sphere	5	✓	20%	80%	100%
9	CWE-89	SQL Injection (SQLi)	5	✓	20%	40%	0%
10	CWE-203	Observable Discrepancy	4		25%	50%	0%
		Total (Top 10)	**96**	-	13.54%	22.92%	36.46%
		Total (Whole dataset)	**169**	-	16.57%	21.89%	27.81%

that, **heuristic-based fix templates are responsible for 74% of the fixes in Vul4J+, and 85% in Vul4JL**. Furthermore, most of the unique repairs by VulTerminator are also due to the heuristic-based fix templates.

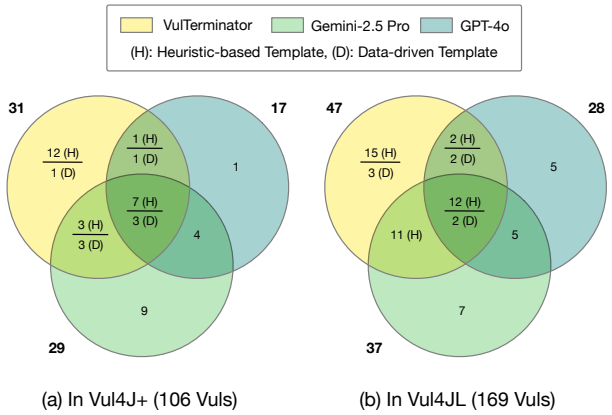
We also examined the vulnerabilities Gemini-2.5 Pro and GPT-4o fixed that VulTerminator could not. We found that many of them could potentially be fixed with the data-driven templates. However, the correct fix ingredients could not be inferred from the codebase and, hence, VulTerminator was unable to synthesize them. For example, VulTerminator is potentially able to fix CVE-2021-4296 (XSS vulnerability) by invoking a sanitization method on the untrusted input before it is included in HTML content; however, this method call is not available in the codebase. GPT-4o successfully repairs this vulnerability by creating a new method definition to escape common special HTML characters.

In conclusion, the results suggest that VulTerminator could be integrated with other approaches, such as Gemini-2.5 Pro, to repair more Java vulnerabilities or to obtain confirmation of each technique's results.

D. RQ4: Fix Template Contribution

Lastly, we shed more light on the contributions of the individual fix template of VulTerminator. We zoomed into the 78 vulnerabilities correctly repaired by VulTerminator in Vul4J+ and Vul4JL, and we counted the number of fixed vulnerabilities for each fix template. Figure 8 shows the distribution of the fix templates used for repairing the vulnerabilities in the two datasets. Note that we do not include results of **FT9** since this fix template could not generate any correct patches.

Overall, ten out of the eleven defined fix templates could fix at least one vulnerability. Among them, **FT1** (*Prevent XXE Vulnerabilities*) produced correct patches for 21 vulnerabilities, achieving the best repair performance, followed by **FT4** (*Prevent Path Traversal Vulnerabilities*) with 17 correctly fixed vulnerabilities. Both **FT1** and **FT4** are heuristic-based.



(a) In Vul4J+ (106 Vuls) (b) In Vul4JL (169 Vuls)

Fig. 7: Overlaps of vulnerabilities fixed by VulTerminator, Gemini-2.5 Pro, and GPT-4o. Focus on heuristic-based (H) vs data-driven (D) fix templates.

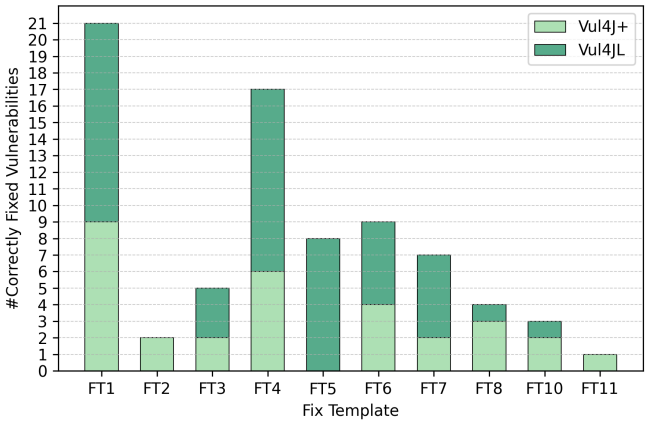


Fig. 8: Contributions of individual fix templates to overall repair performance of VulTerminator.

Moreover, **FT6** (*Prevent the Exposure of Sensitive Data*) and **FT5** (*Prevent Insecure Temporary File Creation*), which also belong to heuristic-based templates, produced correct

patches for nine and eight vulnerabilities, respectively. This indicates that heuristic-based templates dominate and contribute to the most correct repairs in VULTERMINATOR. Among data-driven fix templates, **FT7** (*SanitizeInput*) fixed the largest number of vulnerabilities. Interestingly, this fix template performs 2.5× better on Vul4JL than Vul4J+ (i.e., five vulnerabilities in Vul4JL vs. two in Vul4J+). This indicates that although **FT7** does not achieve performance comparable to the heuristic-based templates, it shows its great repair effectiveness in real-world cases. The two least effective fix templates in our study are **FT2** (*Instantiate Secure Random Generator*) and **FT11** (*AddIf_Return*), which only fix two and one vulnerabilities in Vul4J+, respectively. The number of vulnerabilities related to PRNGs, which **FT2** targets, is relatively small (only four in total across both datasets). In contrast, the vulnerabilities that **FT11** failed to repair often required complex fix ingredients that could not be found in the codebase.

V. RELATED WORK

Automated Program Repair (APR) has gained ground in software engineering research since the seminal work GenProg [54]. Among APR families, template-based APR becomes a popular approach to fix general bugs, by utilizing the fix templates derived manually from human knowledge [55], [56], [57] or mined from code repositories [58], [59], [60], [61]. TBar [13] is considered the baseline for this family by summarizing 35 fix templates from the previous studies. Recently, LLMs have brought significant improvements to template-based APR with state-of-the-art repair techniques such as GAMMA [14], TENNURE [15], and NTR [16]. These learning-based approaches first use fix templates to generate code with masks, and then leverage LLMs to fill them.

On the other hand, Automated Vulnerability Repair (AVR) remains an evolving area of research, as only a few works focus on repairing vulnerabilities. Le Goues et al. [62] improved GenProg and evaluated it on several vulnerabilities, including one linked to a CVE record (CVE-2011-1148). Huang et al. [63] introduced Senx, consisting of a set of rules based on safety properties designed to generate patches for memory-based vulnerabilities. Abadi et al. [64] proposed an approach that fixes injection vulnerabilities by adding sanitizers in the vulnerable code. VuRLE [65] and SEADER [66] infer the transformations *from examples* to fix vulnerabilities in Java bytecode binaries. While these approaches improved repair performance for specific vulnerabilities, their generalizability to broader vulnerability types remains unclear. Recently, several promising deep learning–based AVR tools have been proposed for repairing vulnerabilities, such as VRepair [1], SeqTrans [7], VulRepair [2], SPVF [3], and VulMaster [4]. These AVR approaches treat vulnerability repair as a sequence transformation-based code generation task, where the goal is to generate patches for the vulnerable function as input using LLMs. Still, their repair performance remains limited by the scarcity of large, high-quality training datasets.

Unlike existing AVR techniques, VULTERMINATOR adopts ideas from template-based APR to offer a trade-off solution for fixing vulnerabilities, which is implemented through heuristic-based and data-driven fix templates.

VI. THREATS TO VALIDITY

The main threat to external validity may stem from the choice of our evaluation datasets. We selected Vul4J+ as it is the largest collection of Java vulnerabilities with runnable vulnerability-witnessing tests, which are fundamental for assessing the plausibility of the generated patches. VULTERMINATOR inferred its fix templates from ExtraVul [17], which overlaps with Vul4J+ on 79 vulnerabilities. Therefore, the results obtained on Vul4J+ might be biased. Therefore, we also introduced a novel dataset, Vul4JL, containing 169 new Java vulnerabilities extracted from REEF [44] and ReposVul [45] that were not used to infer the fix templates in VULTERMINATOR. The vulnerabilities in Vul4JL are representative of real-world scenarios, specifically, the nine most frequent vulnerability types in Vul4JL correspond to entries in the 2021 OWASP Top 10 [53]. The experiments show that the trends in repair results are consistent between Vul4J+ and Vul4JL. Notably, several of our fix templates even achieve better performance on the unseen dataset. A potential threat to internal validity arises from the manual validation of patch correctness, where human error may occur. To minimize this risk, two authors double-checked all patches, and a patch was deemed correct only if both authors agreed on its correctness. VULTERMINATOR was designed for Java, so its performance may not generalize to other programming languages.

VII. CONCLUSION

This paper presents VULTERMINATOR, a novel vulnerability repair approach that leverages eleven heuristic-based and data-driven fix templates. To the best of our knowledge, this is the first work to propose and apply security-specific fix templates for repairing Java vulnerabilities with a wide array of vulnerability types. Our experiments demonstrate that VULTERMINATOR achieves state-of-the-art performance. Further analysis reveals that heuristic-based fix templates contribute the most to the success of our approach, while the data-driven fix templates, which employ a fine-tuned LLM, are capable of repairing a broader range of vulnerabilities, complementing those addressed by the heuristic-based templates.

DATA AVAILABILITY

The implementation of VULTERMINATOR is open-sourced at: **https://github.com/tuhh-softsec/VulTerminator**. We also provide a replication package [23], which contains all scripts required to rerun the experiments, as well as the raw and processed data used in this study.

ACKNOWLEDGEMENTS

This work was partially supported by EU-funded project Sec4AI4Sec (grant no. 101120393).

REFERENCES

[1] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.

[2] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 935–947.

[3] Z. Zhou, L. Bo, X. Wu, X. Sun, T. Zhang, B. Li, J. Zhang, and S. Cao, "Spvf: security property assisted vulnerability fixing via attention-based models," *Empirical Software Engineering*, vol. 27, no. 7, p. 171, 2022.

[4] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[5] M. Fu, V. Nguyen, C. Tantithamthavorn, D. Phung, and T. Le, "Vision transformer inspired automated vulnerability repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 3, pp. 1–29, 2024.

[6] Y. Kim, S. Shin, H. Kim, and J. Yoon, "Logs in, patches out: Automated vulnerability repair via {Tree-of-Thought}{LLM} analysis," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 4401–4419.

[7] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, "Seqtrans: Automatic vulnerability fix via sequence to sequence learning," *IEEE Transactions on Software Engineering*, 2022.

[8] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. Yao, and N. Meng, "Example-based vulnerability detection and repair in java code," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 190–201.

[9] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 121–133.

[10] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th international conference on mining software repositories*, 2020, pp. 508–512.

[11] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?" *arXiv preprint arXiv:2403.18624*, 2024.

[12] "2025 stack overflow developer survey: Most popular programming, scripting, and markup languages," https://survey.stackoverflow.co/2025/technology#1-programming-scripting-and-markup-languages, accessed: 2025-09-25.

[13] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[14] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 535–547.

[15] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, "Template-based neural program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1456–1468.

[16] K. Huang, J. Zhang, X. Meng, and Y. Liu, "Template-guided program repair in the era of large language models," in *Proceedings of the 47th International Conference on Software Engineering, ICSE*, 2024, pp. 367–379.

[17] Q.-C. Bui, R. Paramitha, D.-L. Vu, F. Massacci, and R. Scandariato, "Apr4vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities," *Empirical software engineering*, vol. 29, no. 1, p. 18, 2024.

[18] G. Canfora, A. Di Sorbo, S. Forootani, M. Martinez, and C. A. Visaggio, "Patchworking: Exploring the code changes induced by vulnerability fixing activities," *Information and Software Technology*, vol. 142, p. 106745, 2022.

[19] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 79–82.

[20] E. Iannone, Q.-C. Bui, M. Isztin, B. Bogenfürst, G. Antal, P. Hegedűs, and R. Scandariato, "Vul4j+: A dataset of vulnerabilities for automated vulnerability repair," Sep. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13752193

[21] Q. C. Bui, R. Scandariato, and N. E. D. Ferreyra, "Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques," in *International Conference on Mining Software Repositories (MSR)*, 2022.

[22] "Owasp xml external entity prevention cheat sheet," https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html, accessed: 2025-09-25.

[23] "Replication package for our paper," Figshare, 2025. [Online]. Available: https://doi.org/10.6084/m9.figshare.30369793

[24] "Cve-2022-1471," https://nvd.nist.gov/vuln/detail/cve-2022-1471, accessed: 2025-09-25.

[25] "Cve-2022-31159," https://nvd.nist.gov/vuln/detail/cve-2022-31159, accessed: 2025-09-25.

[26] "Cwe-378: Creation of temporary file with insecure permissions," https://cwe.mitre.org/data/definitions/378.html, accessed: 2025-09-25.

[27] "Api documentation of files.createtempfile," https://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html, accessed: 2025-09-25.

[28] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 130–140.

[29] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[30] K. Huang, J. Zhang, X. Bao, X. Wang, and Y. Liu, "Comprehensive fine-tuning large language models of code for automated program repair," *IEEE Transactions on Software Engineering*, 2025.

[31] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[32] "Seart github search engine," https://seart-ghs.si.usi.ch/, accessed: 2025-09-25.

[33] "Git vuln finder," https://github.com/cve-search/git-vuln-finder, accessed: 2025-09-25.

[34] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 914–919.

[35] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[36] B. Steenhoek, K. Sivaraman, R. S. Gonzalez, Y. Mohylevskyy, R. Z. Moghaddam, and W. Le, " Closing the Gap: A User Study on the Real-world Usefulness of AI-powered Vulnerability Detection & Repair in the IDE ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 1–13.

[37] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318.

[38] H. W. Alomari, C. Vendome, and H. Gyawali, " A Slicing-Based Approach for Detecting and Patching Vulnerable Code Clones ," in *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2025, pp. 60–72.

[39] "Patch for cve-2017-1000207," https://github.com/swagger-api/swagger-parser/commit/4c65843, accessed: 2025-09-25.

[40] "src2abs: a tool that abstracts java source code," https://github.com/micheletufano/src2abs, accessed: 2025-09-25.

[41] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman, "Evaluating automatic program repair capabilities to repair api misuses," *IEEE Transactions on Software Engineering*, 2021.

[42] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 615–627. [Online]. Available: https://doi.org/10.1145/3377811.3380338

[43] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1282–1294.

[44] C. Wang, Z. Li, Y. Pena, S. Gao, S. Chen, S. Wang, C. Gao, and M. R. Lyu, "Reef: A framework for collecting real-world vulnerabilities and fixes," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1952–1962.

[45] X. Wang, R. Hu, C. Gao, X.-C. Wen, Y. Chen, and Q. Liao, "Reposvul: A repository-level high-quality vulnerability dataset," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 472–483.

[46] J. Zhang, K. Huang, J. Zhang, Y. Liu, and C. Chen, "Repair ingredients are all you need: Improving large language model-based program repair via repair ingredients search," *arXiv preprint arXiv:2506.23100*, 2025.

[47] H. Hu, Y. Shang, G. Xu, C. He, and Q. Zhang, "Can gpt-o1 kill all bugs? an evaluation of gpt-family llms on quixbugs," in *2025 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2025, pp. 11–18.

[48] W. Wang, W. Ma, Q. Hu, Y. Zhang, J. Sun, B. Wu, Y. Liu, G. Xu, and L. Jiang, "Vulnrepaireval: An exploit-based evaluation framework for assessing large language model vulnerability repair capabilities," *arXiv preprint arXiv:2509.03331*, 2025.

[49] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[50] G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, N. Sachdeva, I. Dhillon, M. Blistein, O. Ram, D. Zhang, E. Rosen *et al.*, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," *arXiv preprint arXiv:2507.06261*, 2025.

[51] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.

[52] A. Papotti, R. Paramitha, and F. Massacci, "On the acceptance by code reviewers of candidate security patches suggested by automated program repair tools," *Empirical Software Engineering*, vol. 29, no. 5, p. 132, 2024.

[53] "Owasp top ten 2021," https://owasp.org/Top10/, accessed: 2025-09-25.

[54] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[55] K. Pan, S. Kim, and E. J. Whitehead Jr, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[56] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[57] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.

[58] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.

[59] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[60] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.

[61] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

[62] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[63] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 539–554.

[64] A. Abadi, R. Ettinger, Y. A. Feldman, and M. Shomrat, "Automatically fixing security vulnerabilities in java code," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011, pp. 3–4.

[65] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 229–246.

[66] Y. Zhang, M. Kabir, Y. Xiao, N. Meng *et al.*, "Data-driven vulnerability detection and repair in java code," *arXiv preprint arXiv:2102.06994*, 2021.