A Match Made in Heaven? AI-driven Matching of Vulnerabilities and Security Unit Tests

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

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

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

Abstract

Software vulnerabilities are often detected via taint analysis, penetration testing, or fuzzing. They are also found via *unit tests* that exercise security-sensitive behavior with specific inputs, called *vulnerability-witnessing tests*. Generative AI models could help developers in writing them, but they require many examples to learn from, which are currently scarce. This paper introduces VuTeCo, an AI-driven framework for collecting examples of vulnerability-witnessing tests from Java repositories. VuTeCo carries out two tasks: (1) The "*Finding*" task to determine whether a unit test case is security-related, and (2) the "*Matching*" task to relate a test case to the vulnerability it witnesses. VuTeCo addresses the *Finding* task with UniXcoder, achieving an $F_{0.5}$ score of 0.73 and a precision of 0.83 on a test set of unit tests from Vul4J. The *Matching* task is addressed using DeepSeek Coder, achieving an $F_{0.5}$ score of 0.65 and a precision of 0.75 on a test set of pairs of unit tests and vulnerabilities from Vul4J. VuTeCo has been used in the wild on 427 Java projects and 1,238 vulnerabilities, obtaining 224 test cases confirmed to be security-related and 35 tests correctly matched to 29 vulnerabilities. The validated tests were collected in a new dataset called Test4Vul. VuTeCo lays the foundation for large-scale retrieval of vulnerability-witnessing tests, enabling future AI models to better understand and generate security unit tests.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; **Software libraries and repositories**; • **Security and privacy → Software security engineering**.

Keywords

Mining Software Repositories, Vulnerability-witnessing Tests, Security Testing, Unit Testing, Language Models

ACM Reference Format:
Emanuele Iannone, Quang-Cuong Bui, and Riccardo Scandariato. 2025. A Match Made in Heaven? AI-driven Matching of Vulnerabilities and Security Unit Tests. In *Proceedings of 23rd International Conference on Mining Software Repositories (MSR 2026).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

MSR 2026, Rio De Janeiro, Brazil

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software vulnerabilities are often detected using techniques such as taint analysis, penetration testing, or fuzzing [4, 33, 36, 58], which test a complete application and are usually not integrated into the development workflow. The "shift-left" paradigm encourages a *test-first* approach, in which the test infrastructure to detect vulnerabilities starts at the unit level, akin to how bugs are found [20].

Security unit tests trigger vulnerabilities with crafted payloads and use assertions to confirm they are really present [14, 16, 45]. Listing 1 shows a security unit test for a path traversal vulnerability (CWE-22) affecting Apache JSPWiki, which is localized in the `getForwardPage()` method and is disclosed via CVE-2019-0225. Lines 3–4 show the official patch for this vulnerability. If the test method `testNastyDoPost()` is executed on the vulnerable version of `getForwardPage()`, it will fail because the focal method does not return the main wiki page as intended, given the crafted payload at Line 9. Hence, the failed assertion at Line 16 confirms the presence of the vulnerability. A test case behaving like this is called **vulnerability-witnessing test** [32] (a.k.a. Proof of Vulnerability, PoV [8, 50]), or simply *"witnessing test"* throughout this paper.

Real-world examples of witnessing tests are collected in Vul4J [8], the only Java dataset containing manually-validated unit tests (108 in total) matched with 79 vulnerabilities affecting 51 Java projects. The process that led to the creation of Vul4J mainly consisted of building the projects and running their test. This encountered two key problems. First, building the projects largely failed due to compile errors and missing dependencies, requiring extensive, tentative manual fixes that did not always succeed. Second, the selected test suites included the tests that existed just after the vulnerability-fixing commit; however, they may not include the witnessing tests, since those could be added in later commits. Among the 899 vulnerabilities inspected, the authors could only reproduce 79 (~9%). Consequently, this strategy for retrieving vulnerability-witnessing tests based on dynamic execution is often unsuccessful (like finding a *needle in a haystack*) and effort-consuming.

Thus, the software security research community is in dire need of more samples of vulnerability witnessing tests. Some compelling use cases for such data are: (i) Understanding the nature of witnessing tests, (ii) training AI models that support the generation of security tests [32, 66], and (iii) validating the techniques that produce security patches [8, 18, 37, 44]. In this paper, we present **VuTeCo** (**VU**lnerability **TE**st **CO**llector), a fully-static AI-driven framework that retrieves vulnerability-witnessing tests in Java test suites. VuTeCo addresses two tasks: (1) The "*Finding*" task to determine whether a test case is security-related, and (2) the "*Matching*" task to pair a test case to the specific vulnerability it is witnessing.

```
// Fix to CVE-2019-0225
public String getForwardPage(HttpServletRequest request) {
- return request.getPathInfo();
+ return "Wiki.jsp";
}
// Vulnerability-witnessing test
@Test
public void testNastyDoPost() throws Exception {
  MockHttpServletRequest req = new MockHttpServletRequest("/JSPWiki","/wiki/Edit.jsp");
  WikiServlet wikiServlet = new WikiServlet();
  MockServletConfig config = new MockServletConfig();
  config.setServletContext(new MockServletContext("/JSPWiki"));
  wikiServlet.init(config);
  wikiServlet.doPost(req, new MockHttpServletResponse());
  wikiServlet.destroy();
  Assertions.assertEquals("/Wiki.jsp?page=Main&", req.getForwardUrl());
}
```

Listing 1: Official fix for CVE-2019-0225 and its related witnessing test in Apache JSPWiki.

VuTeCo has been evaluated in a hold-out set of Vul4J, where it achieved 0.73 $F_{0.5}$ score and 0.83 precision in the *Finding* task. In the *Matching* task, VuTeCo scored 0.65 $F_{0.5}$ score and 0.75 precision. Afterwards, VuTeCo was used in a large-scale mining campaign on GitHub (427 projects), where it retrieved 224 confirmed security-related test cases and 35 tests correctly matched to 29 vulnerabilities. Thus, VuTeCo provides valuable support for large-scale retrieval of security unit test examples for many downstream applications.

In summary, this paper: (1) Introduces VuTeCo, the **first ever framework** to retrieve vulnerability-witnessing tests in Java repositories; (2) extensively evaluates several AI model types to find the right configuration for VuTeCo; and (3) releases **Test4Vul**, a **dataset** containing 259 confirmed security-related unit tests found and matched by VuTeCo in the wild.

2 The VuTeCo Framework

2.1 Overview

VuTeCo supports two tasks: *Finding* and *Matching*. The ***Finding*** task accepts a unit test case, i.e., a JUnit test method, and tells if it is security-related (*"Security"*) or if its nature is unclear (*"Unclear"*). The ***Matching*** task accepts a unit test case and a textual description (in English) of a known vulnerability and determines if the test case witnesses that vulnerability (*"Matched"*) or not (*"Not-Matched"*). Therefore, both tasks are modeled as *binary classification* problems, which can be run independently from one another (namely, the *Matching* does not require a prior run of the *Finding* task). The two tasks are further detailed in Sections 2.2 and 2.3.

Figure 1 depicts the functioning of VuTeCo. Besides the two AI models, VuTeCo also includes a tool that automatically retrieves JUnit test methods from a given Git repository and vulnerability descriptions from CVE identifiers (see Section 2.4 for details).

2.2 The *Finding* Task

The upper part of Figure 2 illustrates the architecture of the model responsible for the *Finding* task, which is a neural network based on UniXcoder [21]. We hereby refer to this model as *Finder*. The choice of UniXcoder resulted from the experimentation designed in Section 3.1 and reported in Section 4.1. We selected the checkpoint *unixcoder-base* from Hugging Face [43], which has been

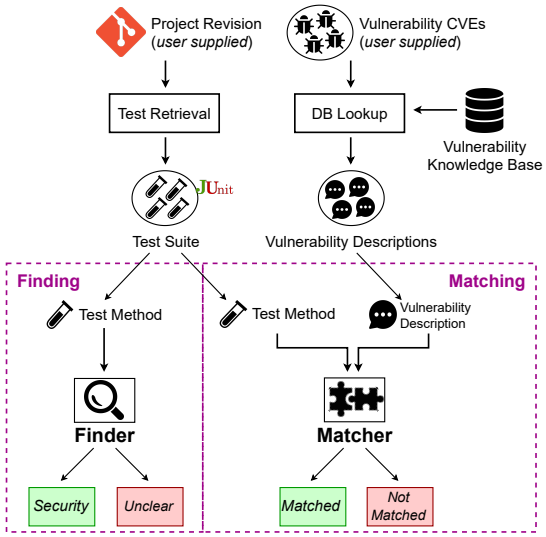

Figure 1: Graphical overview of VuTeCo.

pre-trained on multiple representations of source code, including Abstract Syntax Trees for better understanding. It experienced six different programming languages, including Java, as well as natural language from code comments (often English), making it suitable for understanding the content of JUnit test methods.

Before feeding the input to the model, the test method is stripped of newline characters and consecutive whitespace (including tabs), reducing it to a single line. The resulting string is tokenized with Byte-Pair Encoding using a vocabulary fitted on CodeSearchNet [28], and then sent to the UniXcoder input layer in "encoder-only" mode, i.e., by prepending the `[CLS]` token and the special token `[Enc]` to the input (up to 512 encoded tokens). To obtain the sentence embedding, which represents the entire test method, we perform mean pooling over all token embeddings, as in UniXcoder's official implementation [42], yielding a 768-dimensional representation. On top of this, we added a classification head with two linear layers (with the GELU activation function [23]) of 512 and 128 neurons, respectively, plus an output layer that returns the probability of belonging to the positive class.

The whole model (UniXcoder and the classification head) was trained for ten epochs on a dataset of *"Security"* and *"Unclear"* unit test methods using Weighted Binary Cross-entropy loss to mitigate the large class imbalance. Additional details on the training setting are reported in Section 3.1.

2.3 The *Matching* Task

The lower part of Figure 2 illustrates the architecture of the model responsible for the *Matching* task, which uses DeepSeek Coder [22]. We hereby refer to this model as *Matcher*. The choice of DeepSeek Coder resulted from the experimentation designed in Section 3.1

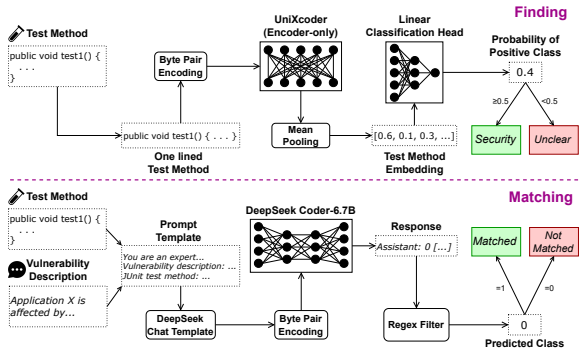

Figure 2: Inner working of *Finding* and *Matching* in VuTeCo.

and reported in Section 4.1. DeepSeek Coder is a chat-based generative large language model specialized in code-related tasks, including *code understanding*. We selected the checkpoint *deepseek-coder-6.7b-instruct* from Hugging Face [52], which has been pre-trained in three sessions: Two self-supervised sessions (next-token prediction and fill-in-the-middle) to learn to understand individual files, and one supervised session to learn to follow instructions from prompts (i.e., instruction-tuning) [22]. In total, DeepSeek Coder experienced 87 programming languages, including English text from code comments, GitHub Markdown, and StackExchange. For all the said reasons, this model can understand the content of JUnit test methods and vulnerability descriptions.

VuTeCo inserts the vulnerability description and the JUnit test method code (truncated if exceeding the model's maximum length, set to 4,096 tokens) in a natural language prompt as follows:

DeepSeek Coder Prompt Template

[System] You are an expert in unit testing and security testing. Given the following vulnerability description and JUnit test method (it might be truncated if too long), answer with 1 if the test case is likely to identify the described vulnerability in the code under test, or 0 if it is not. Answer with only the number, with no explanation.
[User] Vulnerability Description:
`{vulnerability_description}`
JUnit Test Method:
`{method_signature_and_body}`
[Assistant]

The final wording of the system prompt was developed with assistance from ChatGPT (GPT-4o) in May 2025, which helped us refine our initial version (in the replication package [30]) to better suit an LLM. The placeholders [System], [User], and [Assistant] are not part of the prompt and only indicate the three roles recognized by DeepSeek Coder. This prompt is further transformed by applying the DeepSeek Coder's chat template, which starts with the system prompt, then adds '`### Instruction:`' to initiate the user part, and '`### Response:`' to start the assistant part. The resulting text is tokenized with Byte-Pair Encoding.

At test and inference time, the [Assistant] part is intentionally left empty to induce the model to respond to the system+user prompt, whereas during training, the [Assistant] part contains the ground truth (expected) response—according to the standard practice to fine-tune generative LLMs. According to the system prompt, the responses are enforced to provide the classification with just 0 or 1. Hence, the responses are post-processed with a regular expression to extract the first digit encountered (disregarding any additional text), mapping 0 to *"Not-Matched"* and 1 to *"Matched"*.

We followed a two-step approach to train the DeepSeek Coder model. First, we ran a pre-training session for eight epochs on a small dataset of *"Matched"* and *"Not-Matched"* pairs involving solely vulnerability-witnessing test methods (i.e., excluding non-security tests), and a fine-tuning session for five epochs on the complete dataset of *"Matched"* and *"Not-Matched"* pairs involving non-security tests as well (more details in Section 3.1.3). Both training steps were modeled as a *Causal Language Modeling* task, computing the Cross Entropy loss only on the [Assistant] part, and employed LoRA (Low-Rank Adaptation) optimization [26]. The two training datasets were oversampled with SPAT [65], a tool that creates semantically equivalent clones of Java methods. We used it to generate further examples of vulnerability-witnessing tests, thereby increasing the number of pairs. We ran it five times to generate diverse examples (removing any resulting duplicates). More details on the training setting are reported in Section 3.1. To ensure that the same pair will always be classified in the same way, VuTeCo uses *greedy decoding* during inference, i.e., at each step the model selects the single most probable next token.

2.4 Tool-assisted Input Retrieval

VuTeCo is assisted by a *tool* that collects and prepares the input (upper part of Figure 1) for the two AI models (the core of VuTeCo). The tool accepts a revision hash (i.e., a commit) of any Git-based repository (remote URL) from the user. This will be cloned locally, and all its Java files parsed (using `javalang` [9]). Then, it marks as a *"test case"* any **method** with the following properties:

(1) it is annotated with `@Test` (JUnit 4 and 5) or its class extends TestCase or any subclass of it (for JUnit 3);
(2) it is not overriding a method defined in class TestCase, like `run()` or `getName()` (for JUnit 3);
(3) it is not a "lifecycle method", i.e., annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, or `@AfterEach`;
(4) it returns `void` if not annotated with `@TestFactory`;
(5) it is not `abstract`, `static` or `private`;
(6) its class is not `abstract`.

Such heuristics were designed based on how the JUnit guide describes a test case [59] and the Javadoc of JUnit beyond version 3.

Additionally, in the *Matching* task, the tool also accepts a set of CVE identifiers from the user, which are used to look up an internal *knowledge base* of 1,992 Java vulnerabilities to retrieve their descriptions. We prepared this catalog by aggregating and de-duplicating three established datasets of Java vulnerabilities, i.e., ProjectKB [56], REEF [62], and ReposVul [63]. The catalog also stores additional useful metadata, such as CWE (Common Weakness Enumeration) and known fix commits. If a CVE identifier

is not found in the catalog, the tool can retrieve the missing data online using the *Vulnerability-Lookup* API [13].

Afterward, the tool sends the retrieved data to the model responsible for the requested task (*Finding* or *Matching*). In the *Finding* case, each collected test case is sent to the *Finder* model to assess its relevance to security. In the *Matching* case, the test cases are paired with the vulnerability descriptions in all possible ways (i.e., Cartesian product), and each is sent to the *Matcher* model to assess whether the test case witnesses the paired vulnerability.

3 Evaluation Design

We conducted a *two-phase* evaluation. First, we performed an **experimental evaluation** to search for the best AI model to carry out the *Finding* and *Matching* tasks. This involved testing several AI model types on a hold-out set originating from Vul4J—the same source also used for their training. We involved *six* main models: Three based on pre-trained transformers, CodeBERT [17], CodeT5+ [64], and UniXcoder [21]; and three generative large language models, CodeLlama [54], DeepSeek Coder [22], and Qwen2.5-Coder [27]. We hereby distinguish the two groups as *code representation models* (CRMs) and *large language models* (LLMs), respectively. The CRMs transform the input into fixed-length embeddings and process them with a linear classification head to obtain the predicted classes. The LLMs, instead, are prompted to analyze the input and provide a binary response directly. We downloaded from Hugging Face the snapshots: *codebert-base*, *codet5p-220m*, *unixcoder-base*, *CodeLlama-7b-Instruct-hf*, *deepseek-coder-6.7b-instruct*, *Qwen2.5-Coder-7B-Instruct*. This evaluation aimed to assess the suitability of such models for the *Finding* and *Matching* tasks using validated data. For the *Finding* task, the six models were trained and tested on a collection of test cases extracted from projects in Vul4J. In the *Matching* task, the six models were trained and tested using the same set of test cases, but also paired with vulnerability descriptions that affected the corresponding projects. Thus, we carried out a total of 12 distinct train-test sessions.

In the second phase, we performed a complementary analysis that examined VuTeCo's performance **in the wild**, i.e., running the best AI models (already mentioned in Section 2) resulting from the previous evaluation on a large set of Java projects outside Vul4J. Accordingly, we formulated the following research questions:

> **RQ$_1$.** *What are the **best AI models** for finding security unit tests and matching them with the right vulnerability?*

> **RQ$_2$.** *How well can VuTeCo find security unit tests **in the wild** and match them with the right vulnerability?*

3.1 Experimental Evaluation (RQ$_1$)

3.1.1 Data Selection and Preprocessing. The primary source of data for this experimental evaluation was Vul4J [8], as it is the only known source with validated JUnit test methods matched with the vulnerabilities they witness. At the time of this paper writing, Vul4J had 79 reproducible vulnerabilities across 51 Java projects. From these projects, we checked out their patched revisions (i.e., the project versions in which the vulnerability has been fixed and where the witnessing tests have been found) and collected all their test suites using the heuristic described in Section 2.4. We fetched metadata—including descriptions—of the 79 vulnerabilities appearing in Vul4J using the VuTeCo's internal knowledge base (Section 2.4), resulting in 76 vulnerabilities with metadata (three were reported through inaccessible bug trackers rather than CVE).

For the *Finding* task, we considered all the 108 JUnit test methods reported Vul4J as confirmed examples of witnessing tests, labeling them as *"Security"* (the positive class). To create the negative class (*"Unclear"*), we used all the remaining non-duplicated JUnit test methods mined from the 51 projects in Vul4J, totaling 39,542. We note that the negative class for the *Finding* task comprises test cases lacking sufficient evidence of their security relevance, rather than being tests that are entirely unrelated to security (hence the name *"Unclear"*). Then, we split the dataset using stratified sampling (i.e., keeping the same class distribution), creating a training set TR_F (70%), a development set DE_F (10%), and a test set TE_F (20%).

For the *Matching* task, we created pairs of test cases and the descriptions of the witnessed vulnerabilities in Vul4J. For example, the test method `testSendingStringMessage()` was paired with the vulnerability CVE-2015-0263 (according to the dataset entry VUL4J-3). Due to the three vulnerabilities discarded previously, the total number of valid pairs was 105, which formed the positive class (*"Matched"*). To create the negative class (*"Not-Matched"*), we paired all test methods in Vul4J with unrelated vulnerabilities affecting the same project, forming 84,506 invalid pairs. Just as for the *Finding* task, we split this dataset with stratified sampling, creating a training set TR_M (70%), a development set DE_M (10%), and a test set TE_M (20%).

Depending on the model type, we prepared the input in different ways. For the CRMs, the input test methods in both *Finding* and *Matching* tasks were flattened into a single line, and any multiple occurrences of whitespace were replaced with a single one—indeed, we observed that removing them could improve performance beyond reducing the input length. In the *Matching* task, the description was placed in the test method's Javadoc, exploiting the familiarity that the underlying encoder models have with source code with documentation. Regarding the LLMs, the prompt for the *Matching* task was the same as the one presented in Section 2.3, while for the *Finding* task, we used a similar one that omits the vulnerability description. The difference lies in the central part: *'Given the following JUnit test method [...] answer with 1 if the test case is likely to identify a vulnerability in the code under test'*. In both tasks, we did not preprocess the test method code (as done, instead, with the CRMs), but truncated it if it exceeded the model's maximum length (4,096 tokens).

3.1.2 Performance Indicators. Both the *Finding* and *Matching* tasks were modeled as binary classification problems. Hence, we relied on the traditional metrics to measure the goodness of binary predictions, i.e., *precision* (Pr), *recall* (Re), *F1 score* [5, 51]. We also selected the *Matthews's correlation coefficient* (MCC) [40] as a key indicator due to its reliability in imbalanced problems (like these two), and we reported the absolute number of positive classifications (i.e., True Positives and False Positives) to contextualize all the metrics.

VuTeCo's primary goal is to **maximize the number of correct findings while minimizing incorrect ones**. The aim is to develop an approach that identifies examples of witnessing tests in software repositories, addressing the challenges outlined in Section 1. Given

this circumstance, we include the $F_{0.5}$ *score* in the analysis, which is a varied version of the F_1 score where twice as much weight is given to precision compared to recall [60]. We prioritized this score in the experimental evaluation because it quantifies the trade-off between precision and recall and aligns with VuTeCo's requirements. Additional metrics, such as the AUC-ROC, are reported in the replication package [30].

3.1.3 Model Configuration. We identified key **factors** that we hypothesized could affect the performance of the AI models. In Section 4.1, we report the best-performing configuration for each AI model (based on the $F_{0.5}$ score on the test set).

Factor: Data Augmentation. In both tasks, we addressed class imbalance in the respective training sets. We experimented with three mechanisms: (1) **JavaTransformer** (JT) [53], a tool that transforms a Java method (including test methods) into semantically-equivalent clones by applying nine semantic-preserving transformation rules, like renaming variables or adding random logging statements; (2) **SPAT** [65], another tool creates semantically-equivalent clones of Java methods with 18 transformation rules akin to JavaTransformer; (3) **Bootstrapping** (BS), a resampling technique creating exact copies of instances in the minority class (a.k.a. *random oversampling*). We ran JavaTransformer (JT) and SPAT five times, as they could generate additional semantically equivalent clones due to random variable names they can synthesize. For bootstrapping (BS), instead, we set the target imbalance ratio to 0.25, i.e., re-sampling until the minority instances were 25% of the total. We also experimented with a fourth case in which the training data were not augmented.

Factor: Loss Function (CRMs only). In both tasks, we controlled the loss function for training the CRMs. We experimented with the standard binary cross-entropy and its **weighted** version, i.e., where we supplied the weights of the two classes to penalize more the misclassifications made on the minority class (either *"Security"* or *"Matched"*) [67]. The weights were computed from the training set using the `compute_class_weight()` function of `scikit-learn`. Regarding the LLMs, we could only use the standard cross-entropy loss as their training is modeled as a *causal language modeling* task, where the prediction target is a sequence of tokens, rather than one of two possible classes.

Factor: Decomposition of *Matching* (CRMs only). In the standard case, training a model for the *Matching* task consisted of fitting it on TR_M. However, we also modeled this task with a different approach for the three CRMs, i.e., by employing two *sub-models* (of the same architecture, e.g., two UniXcoder models). The first sub-model aims to determine whether the test method is likely to witness a vulnerability—therefore, it reuses the best configurations resulting from the experimentation made for the *Finding* task. The second sub-model implements a "simplified" version of the ordinary *Matching* task by assuming that the input test method is always security-related. Hence, its three datasets (training, development, and testing) are made by discarding the pairs involving non-witnessing test methods from TR_M, DE_M, and TE_M. We refer to these altered datasets as TR'_M, DE'_M, and TE'_M, totaling 105 *"Matched"* pairs and 7,665 *"Not-Matched"* pairs. We separately searched for the optimal configuration for the second sub-model on this simplified *Matching* task. We did not employ any decomposition for the LLMs due to the very slow inference time of invoking two models per prediction.

The outputs of the two sub-models are **integrated** into a single, final judgment to address the "real" *Matching* task. This integration happened using three approaches. (1) **"Meta"** aggregates the logits from both sub-models using a linear layer that outputs a single logit, learning a weighted combination of their predictions. (2) **"Fuse"** concatenates the final hidden states (embeddings) from both sub-models, projects them into a 64-dimensional latent space, and then applies another linear layer to output the final logit. (3) **"Mask"** returns the logit from the second sub-model only if the first sub-model predicted that the test method is likely to witness a vulnerability ($P(\text{"Security"}) \geq 0.5$); otherwise, the logit from the first sub-model is selected as the final output.

Factor: Training Mode for *Matching*. Regardless of whether the *Matching* task is performed by a single model or two sub-models, the availability of the dataset TR'_M has enabled experimentation with three distinct training modes. In the **pre-train** case (PT), the training happens solely on TR'_M if the *Matching* task is carried out with one model; otherwise, the two sub-models are just trained separately on TR_F and TR'_M, respectively. **Fine-tune** (FT) trains the full model directly on TR_M, regardless of whether the *Matching* task is decomposed. **Full-train** (PT-FT) employs both *pre-train* and *fine-tune* strategies, sequentially. These three training modes were tested for all six AI models involved.

Hyperparameter Optimization. Since the number of factors for the *Finding* task was fewer than for the *Matching* task, we could run additional experiments by optimizing some *hyperparameters* to further improve performance, selecting the variants with the highest $F_{0.5}$ score and the lowest loss on the development set. We optimized the "intensity" of augmentation mechanisms, i.e., the number of times the JavaTransformer or SPAT were run during oversampling (5 and 15) and the target imbalance ratio for bootstrapping (0.25 and 0.50). The two layers of the CRMs' classification head were set to 512 and 128 in size; however, for the *Finding* task, we also optimized their size by searching in the space $\{512, 256\}$ and $\{128, 64\}$, respectively.

3.1.4 Model Implementation and Training. The models have been implemented with PyTorch and trained with the transformers API. For all training sessions, the weights were updated using the AdamW optimizer [38], with a learning rate of 10^{-5} for CRMs, and $2 \cdot 10^{-4}$ for LLMs, both decaying linearly. The CRMs were trained for 10 epochs, while the LLMs were trained with LoRA (rank 16) [26] for five epochs. After each epoch, the model was evaluated on the development set. Upon completion of training, the checkpoint with the highest $F_{0.5}$ score was selected, using the lowest loss as a tiebreaker if necessary. The classification heads of the CRMs used a dropout probability of 0.1 between every linear layer to reduce the risk of overfitting [25].

The experiment was conducted on a server equipped with an NVIDIA Tesla A100 Core GPU and an Intel Xeon Platinum 8352V CPU. Counting all combinations of factors (the details are in the replication package [30]), we ran 36 train-testing sessions for the *Finding* task, of which 24 were for the CRMs and 12 were for the LLMs. On average, each CRM session took 4 hours, whereas each LLM session took 50 hours. For the *Matching* task, we ran a total of

324 train-testing sessions. For the CRMs, we had 72 sessions from the case without decomposition into two sub-models, while the other 216 sessions were from the case with two sub-models. For the LLMs, we had 36 sessions. On average, each CRM session took 5 hours, whereas each LLM session took 60 hours.

3.1.5 Baseline Approaches. Our study is the first to address the problem of finding security test cases and matching them to vulnerabilities (to the best of our knowledge); thus, there are no direct competitors. Nevertheless, we developed **three** *baseline approaches* for the *Finding* task and **four** for the *Matching* task, relying on principles different from those of the CRMs and LLMs. We experimented with multiple configurations for each baseline. In Section 4.1, we report only the best-performing configuration for each approach (based on the $F_{0.5}$ score on the test set).

Baselines for *Finding*. The approach *GrepFind* checks if the test method code contains one or more security-specific keywords, such as *'secur'*, *'cve'*, *'xss'*; if so, it is flagged as *"Security"*. We used the set of keywords defined by Zhou and Sharma [69] and further expanded with additional ones (the full list is in the replication package [30]). In essence, this approach behaves like the `grep -e` command. We conducted experiments varying the required number of matches from 1 to 5, testing each condition with and without our extended list, totaling 10 configurations. The *VocabFind* approach, instead, fits a vocabulary of terms on the test methods in the training set TR_F. Then, it checks if the terms of the input test method adhere to the fitted vocabulary (i.e., the test method shares a similar vocabulary with the known test cases). If the number of matched terms exceeds the threshold N, the input test method is flagged as *"Security"*. The terms were extracted in two ways: (i) using YAKE, which picks the K most relevant terms from a text in an unsupervised manner [10], and (ii) retrieving all the identifiers (i.e., variable and method names) in the test case, as they likely indicate the purpose of the test and are not mixed with other irrelevant language-specific keywords. We refer to these two flavors as $VocabFind_{YAKE}$ and $VocabFind_{Iden}$, respectively. We experimented with N=[1..10] (step of 1) for both flavors. For $VocabFind_{YAKE}$ we experimented with K=[5..30] (step of 5), while for $VocabFind_{Iden}$ we experimented with honoring `camelCase` and `snake_case` notations (e.g., whether identifiers like `user_Password` must be split into `user` and `password`), totaling 80 configurations.

Baselines for *Matching*. The *GrepMatch* approach checks if the test method code matches one or more terms appearing in the paired vulnerability description; if so, the pair is flagged as *"Matched"*. The terms are vulnerability descriptions obtained via word-level tokenization and removal of English stop words. We varied the required number of hits from 1 to 5. The *SimMatch* approach checks the *similarity* between the test method code and the vulnerability description and flags the pair as *"Linked"* if the similarity score surpasses an arbitrary threshold T. We made two different implementations. $SimMatch_{YAKE}$ extracts the keyword sets from both inputs using YAKE [10] and compares them using the Jaccard index. $SimMatch_{CRM}$ uses one of three CRMs, i.e., CodeBERT [17], CodeT5+ [64], and UniXcoder [43], to create embeddings for both inputs and compare them using cosine similarity. For $SimMatch_{YAKE}$ we experimented with T=[0.01..0.05] (step of 0.01) and K=[5..30] (step of 5). For $SimMatch_{CRM}$ we experimented with five similarity thresholds depending on the embedding models: T=[0.91..0.95] (step of 0.01) for CodeBERT, for CodeT5+ T=[0.7..0.9] (step of 0.05), and for UniXcoder T=[0.3..0.5] (step of 0.05). A total of 45 configurations have been evaluated. Lastly, we developed *FixCommits*, which inspects the fix commits of a vulnerability v, collects the set of added or modified test methods TM, and flags the pairs $(t, v), \forall t \in TM$ as *"Matched"*. Any other pair is considered as *"Not-Matched"*. This approach relies on the assumption that developers create unit tests alongside the patches to demonstrate that the vulnerability has been successfully fixed. We note that there is currently insufficient evidence to support that this core assumption always holds, as developers may not create tests when fixing vulnerabilities [8]. For example, in CVE-2010-0684 of ActiveMQ none of the three fix commits (`2895197`, `fed39c3`, and `9dc43f3`) added any test. Besides, *FixCommits* can only be run when the fix commit is known, which is not the case for all vulnerabilities disclosed on CVE—while VuTeCo can be used regardless of the fix commit. For these reasons, *FixCommits* is treated as another heuristic method rather than a ground truth.

3.2 In-the-wild Evaluation (RQ_2)

The evaluation in the wild aimed to assess the practical usefulness of VuTeCo, i.e., whether it can find new security-related test cases and match them with the right vulnerability. We prepared VuTeCo according to the configuration described in Section 2, which resulted from the experimental evaluation (Section 3.1). This time, we retrained and exported the best models for each task by merging the test sets into their corresponding training sets, since the test sets were no longer needed at this stage.

The dataset comprised open-source Java projects and their vulnerabilities that we selected from VuTeCo's internal knowledge base of 1,992 vulnerabilities (introduced in Section 2). First, we discarded 60 vulnerabilities that were also part of Vul4J; indeed, they had already been used to train VuTeCo, and we already know their witnessing tests. We discarded 19 more vulnerabilities that had no CVE descriptions, preventing us from running the *Matching* task. Then, we assessed the accessibility of their related projects, discarding those without a remote URL or that were no longer accessible, which resulted in the removal of an additional 61 vulnerabilities. We manually reviewed all remaining repositories to remove duplicates (e.g., mirrored repositories hosted outside GitHub) and those containing testing libraries (e.g., JUnit), since these are not used during deployment. This step eliminated 21 duplicate repositories and 2 testing projects. Since we had no guidance on when exactly the witnessing tests were added, we selected the latest revision of each repository (i.e., the HEAD of the respective default branches) on May 21, 2025—indeed, even after a vulnerability has been fixed, any witnessing tests that developers wrote might still be part of the test suites. Therefore, the input given to VuTeCo for both tasks was a project's repository (remote URL) and its latest revision. Besides, for the *Matching* task, VuTeCo also requires a list of historical vulnerabilities that affected it (from the knowledge base).

At this point, we applied the test case retrieval heuristics (Section 2.1) to ensure the projects had valid test suites. This revealed 388 projects without any tests that VuTeCo could process. After all these steps, we obtained 427 projects and 1,238 vulnerabilities.

The 427 projects had a total of 1,105,491 test cases, which were given to VuTeCo for the *Finding* task. The *Finder* model required 3 days to classify all test cases. All valid combinations of test cases and vulnerabilities resulted in 5,451,212 pairs. The *Matcher* model required 36 days to process and classify all of them.

We assessed the correctness of VuTeCo in both tasks by measuring its *precision*, i.e., the proportion of positive classifications that are correct. We extracted all the test cases flagged as *"Security"* (*Finding*) and the pairs of test cases and vulnerabilities flagged as *"Matched"* (*Matching*). From this evaluation, we disregarded the 108 vulnerability-witnessing test cases that also appear in Vul4J—as they have been used to train VuTeCo before its deployment in the wild. We then involved two independent researchers with experience in software security and unit testing. They were instructed to indicate whether a classification was correct, so that we could compute the true and false positive predictions. We used Cohen's Kappa score [12, 41] to measure the agreement between the two inspectors. The two inspectors discussed all cases of disagreement and reached a consensus to resolve them. We could not measure recall because it required an infeasible labeling session involving more than one million test methods before launching VuTeCo.

4 Evaluation Results

4.1 Experimental Results (RQ_1)

4.1.1 Finding Task. Table 1 reports the performance of the six evaluated models and the three baseline approaches in their best configurations for the *Finding* task on TE_F extracted from Vul4J (the other configurations are in the replication package [30]). The best model was UniXcoder [21], achieving an $F_{0.5}$ score of 0.73, supported by a high precision of 0.83. This means that it can often recognize real vulnerability-witnessing tests, with minimal false positives (just two). In other words, the tests that it flags are very likely to be security-related. The good performance of UniXcoder was also confirmed by the high MCC score of 0.61, indicating a strong positive correlation with the ground truth (following the same interpretation of Pearson's correlation coefficient [49]). UniXcoder identified 10 out of 21 vulnerability-witnessing tests in TE_F, resulting in an unremarkable recall score of 0.48. Still, this was the highest recall across all tested models. UniXcoder had slightly lower precision than Qwen2.5-Coder, which was 0.88; however, this was due to a single false positive.

As also reported in Section 2, the best configuration of UniXcoder is trained using the Weighted Binary Cross-entropy loss to handle the large class imbalance. Given this, augmenting the training set was unnecessary (in fact, it was even counterproductive). Compared to the configuration using the standard Binary Cross-entropy, the improvement in $F_{0.5}$ score was 23%, from 0.56 to 0.73. Thus, we observed that the weighted loss brought the expected benefit. On the contrary, the data augmentation techniques did not significantly contribute to improving performance—only SPAT [65] marginally enhanced the $F_{0.5}$ score when the standard loss function was used. The same happened for the runner-up model, i.e., Qwen2.5-Coder, whose best configuration (with a 0.66 $F_{0.5}$ score) also adopted the weighted loss and no data augmentation. From a broader perspective, we observed that CRMs performed comparably to LLMs. However, given their faster training and inference times, CRMs are the best choice for this task.

Table 1: Performance of the six main AI models and three baselines (best configurations) for the *Finding* task.

Approach		Performance						
		Pr	**Re**	**F_1**	**$F_{0.5}$**	**MCC**	**TP**	**FP**
CRM	CodeBERT	0.78	0.34	0.47	0.61	0.51	7	2
	CodeT5+	0.70	0.33	0.45	0.57	0.48	7	3
	UniXcoder	0.83	**0.48**	**0.61**	**0.73**	**0.63**	**10**	2
LLM	CodeLlama	0.69	0.43	0.53	0.62	0.54	9	4
	DeepSeek Coder	0.69	0.43	0.53	0.62	0.54	9	4
	Qwen2.5-Coder	**0.88**	0.33	0.48	0.66	0.54	7	**1**
Baseline	*GrepFind*	0.01	0.24	0.03	0.02	0.05	5	375
	$VocabFind_{YAKE}$	0.08	0.05	0.06	0.07	0.06	1	11
	$VocabFind_{Iden}$	0.02	0.10	0.03	0.02	0.04	2	114

Among the baseline approaches, none achieved satisfactory results. *GrepFind* successfully classified just five witnessing tests in the right class, despite the wide set of keywords that we prepared. This suggests that developers frequently employ non-obvious terms when writing security-specific test cases. In any case, this resulted in 375 false positives, rendering it unhelpful. Even after fitting a vocabulary of terms from the training set using $VocabFind_{YAKE}$ and $VocabFind_{Iden}$, we still experienced poor performance. To improve these approaches, a more curated list of security-related keywords inferred from real-world examples is required. However, achieving this requires a larger collection of vulnerability-witnessing test cases, which is actually the primary goal of this work.

4.1.2 Matching Task. Table 2 reports the performance of the six evaluated models and the three baseline approaches in their best configurations for the *Matching* task on TE_M extracted from Vul4J (the other configurations are in the replication package [30]). The best model was DeepSeek Coder [22], achieving an $F_{0.5}$ score of 0.65, supported by a high precision of 0.75. This means that it can often validate pairs of test cases and vulnerability descriptions, with minimal false positives (just three). In other words, the matches that it returns are very likely to be valid. This is also confirmed by the MCC score of 0.57, indicating a strong positive correlation with the ground truth [49].

Unlike the *Finding* task, the best model belongs to the LLM group. This DeepSeek Coder model was trained in "full-train" mode (PT-FT), which made a pre-training on TR'_M (the simplified *Matching* task) for eight epochs and then the fine-tuning on TR_M (the regular *Matching* task) for five epochs. Both training datasets were augmented with SPAT [65] (ran five times). This confirmed that the model benefited from the two-phase training. Indeed, comparing this version with the configuration that only performed fine-tuning (FT) on TR_M, the $F_{0.5}$ score went from 0.58 to 0.65, due to an improved precision (from 0.64 to 0.75) and the same recall. The other LLMs, by contrast, achieved their best performance with only fine-tuning (i.e., pre-training actually lowered their scores). The effectiveness of two-phase training appears to vary with the specific model, making it an impactful factor. Furthermore, all LLMs showed consistent benefits from data augmentation. This is likely

Table 2: Performance of the six main AI models and four baselines (best configurations) for the *Matching* task.

Approach		Performance						
		Pr	Re	F_1	$F_{0.5}$	MCC	TP	FP
CRM	CodeBERT	0.60	0.29	0.39	0.49	0.41	6	4
	CodeT5+	0.71	0.23	0.36	0.51	0.41	5	2
	UniXcoder	**1.00**	0.24	0.39	0.61	0.49	5	**0**
LLM	CodeLlama	0.64	0.43	0.51	0.52	**0.58**	9	5
	DeepSeek Coder	0.75	0.43	**0.55**	**0.65**	0.57	9	3
	Qwen2.5-Coder	0.86	0.29	0.43	0.61	0.50	6	1
Baseline	*GrepMatch*	0.50	0.14	0.22	0.33	0.27	3	3
	$SimMatch_{YAKE}$	0.01	0.10	0.02	0.01	0.03	2	220
	$SimMatch_{CRM}$	0.33	0.05	0.09	0.15	0.12	1	2
	FixCommits	0.30	**0.57**	0.39	0.41	0.32	**12**	29

due to the inability to employ a weighted loss function as we did with the CRMs.

Regarding the CRMs, the integrated use of two sub-models (one for the *Finding* part and one for the simplified *Matching*) was not sufficient to outperform DeepSeek Coder. Nevertheless, the two UniXcoder sub-models ranked second, with perfect precision but very low recall (0.24). The two sub-models have been integrated with the "Mask" style (described in Section 3.1.3) and were pre-trained on TR_F and TR'_M but without fine-tuning on TR_M. This configuration was significantly better than the equivalent configuration without the integration (i.e., only one UniXcoder), which had an $F_{0.5}$ score of 0.01. Upon closer inspection of the other CRMs, "Mask" was the style that benefited from pre-training alone, as all the other integration styles did not perform well without fine-tuning. This observation aligns with how "Mask" operates. With the exception of the "Mask" style, the "full-train" mode was the most effective way to train the CRMs. Thus, if faster inference is needed at the cost of recall (so, with a higher expectation of valid matches misclassified as invalid), UniXcoder is the best option.

Overall, the performance in this task is slightly lower than what was observed in the *Finding* task. This was anticipated because the *Matching* task appears more challenging: It requires the model to comprehend both the test method and the vulnerability description, which are presented in two different languages (Java and English).

Among the baseline approaches, *SimMatch* performed poorly. The $SimMatch_{CRM}$ flavor achieved greater precision, but at the cost of reducing the detection rate almost to zero (only one true positive and two false positives). Despite its simplicity, *GrepMatch* (with two required hits) achieved higher precision, i.e., 0.50. This indicates that when we searched for two terms within the vulnerability descriptions in the test case text, we could validate 50% of the true pairs. This method appears suitable as an initial, lightweight approach for quickly matching a few test cases with vulnerabilities (i.e., the "low-hanging fruit"). Unfortunately, this method also missed many cases (0.14 recall).

Overall, the best baseline approach was *FixCommits*, achieving 0.41 $F_{0.5}$ score and 0.32 MCC score. Although its performance remains lower than that of any of the six AI models, its recall score of 0.57 was the highest among all—namely, it correctly classified 12 valid pairs, albeit at the cost of several false positives (29). We shed further light on the unique contribution of this baseline and whether it can complement the findings of the best AI model (DeepSeek Coder). Hence, we selected the valid pairs only (i.e., the 21 positive instances in TE_M) and conducted an *overlap analysis* between the two approaches. We observed that they agreed on five out of 21 cases, with seven cases validated by *FixCommits* only, and four by DeepSeek Coder. Both missed five matches (false negatives). In total, their agreement—measured through the Jaccard index—on the positive classifications was 0.31, meaning that they shared the same judgment in approximately 1 out of 3 cases. Although the *intersection* of their predicted sets can reduce the number of false positives to zero, and so maximize the precision to maximum, it also drastically reduces the recall to 0.24 (-58% compared to *FixCommits*). Hence, in this case, the *union* could be more beneficial as it increases the recall to 0.76 (+33% compared to *FixCommits*), though this would also reduce precision to 0.5 (-33% compared to DeepSeek Coder). In any case, it is worth noting that this joint use is only possible when the fix commits of a vulnerability are known and accessible; otherwise, we are unable to use *FixCommits*.

> 👉 **Answer to RQ_1.** UniXcoder is the best model for finding security-related unit tests with minimal false positives, achieving 0.73 $F_{0.5}$ score, 0.83 precision, and 0.63 MCC score. DeepSeek Coder is the best model for matching unit tests and vulnerabilities with minimal false positives, achieving 0.65 $F_{0.5}$ score, 0.75 precision, and 0.57 MCC score.

4.2 In-the-wild Results (RQ_2)

4.2.1 Finding Task. As a result of the experimentation in $\mathbf{RQ_1}$, VuTeCo uses UniXcoder for the *Finding* task. After processing 1,105,491 test cases (from a total of 427 projects), VuTeCo flagged 319 test methods as *"Security"*, found in 83 projects. Among these, the manual assessment confirmed that 224 were covering security-related aspects—i.e., VuTeCo scored 0.70 precision. The inspectors agreed on 294 (92%) cases, with 0.81 Cohen's Kappa score, indicating a *strong* inter-rater agreement. Afterward, they jointly reviewed the remaining 25 cases, where they had conflicting judgments, until they reached consensus.

The precision score in the wild is not much far from the 0.83 observed during the experiment in $\mathbf{RQ_1}$—indeed, a non-negligible drop was anticipated. Thus, the transfer of the *Finder* model in the wild mostly preserved its capability. We observed that VuTeCo was misled by test cases containing *keywords* commonly associated with vulnerabilities, even though these tests did not actually uncover any security issues. For instance, a test case in Apache Struts contains terms like *'bad'*, *'pollution'*, and *'inject'*—which are often found in security vocabularies—despite having a different meaning there [3]. Besides, we observed that test cases containing constant strings such as *paths* (e.g., URLs or file paths), *shell commands*, *version numbers*, or *hashes* are more likely to be incorrectly flagged. Such problems could be addressed by introducing fine-grained semantic analyses and improved keyword matching to substantially reduce the number of false positives.

4.2.2 Matching Task. As a result of the experimentation in $\mathbf{RQ_1}$, VuTeCo uses DeepSeek Coder for the *Matching* task. After processing 5,451,212 pairs of test cases and vulnerability descriptions, VuTeCo flagged 96 pairs as *"Matched"*, involving 35 unique tests from 19 projects. Among these, the manual assessment confirmed

that 45 were correct, i.e., VuTeCo scored 0.47 precision. The inspectors agreed on 85 (89%) cases, with 0.77 Cohen's Kappa score, indicating a *substantial* inter-rater agreement. Afterward, they jointly reviewed the remaining 11 cases in which they had conflicting judgments until they reached consensus.

Unlike the *Finding* task, here we observed a larger drop in precision than the 0.75 achieved during experimentation in $\mathbf{RQ}_1$. Many of the invalid matches were due to the similarity between the terms in the test case and the CVE description (just as in the *Finding* case). This decline in precision can be attributed to two main reasons: (i) The 427 projects chosen for this assessment might have no valid match at all, whereas the 51 projects used in $\mathbf{RQ}_1$ were guaranteed to have at least one valid match; (ii) the number of vulnerabilities for each project is higher than in $\mathbf{RQ}_1$, particularly for large projects like Spring Framework or Jenkins. Therefore, there is a natural *distribution shift* that increases the likelihood of false positives. Despite these issues, the transfer of the *Matcher* model in the wild was deemed acceptable, though it requires further improvements, including better prompting and safeguards against misleading terms.

In the end, the 35 tests correctly matched with the right vulnerability were added to the new dataset Test4Vul [30] alongside the 224 security-related tests from the *Finding* task, totaling 259.

> 👉 **Answer to $\mathbf{RQ}_2$.** VuTeCo found 224 security-related test cases in the wild, i.e., 70% of all tests returned. The false positives were primarily due to security-related terms and constant strings appearing in the test code. Then, VuTeCo returned 45 correct matches, i.e., 47% of the total matches. Test code and CVE descriptions sharing a similar vocabulary led to incorrect matches.

5 Discussion

The Retrieval of Witnessing Tests. The main advantage of VuTeCo lies in its fully static nature. A dynamic assessment would require building an entire project and setting up the environment to execute all test cases, which may not always succeed due to dependency issues. The extent of the results that VuTeCo returned in the wild ($\mathbf{RQ}_2$) permits manual inspections to be done in a reasonable time. Indeed, VuTeCo greatly reduced the search space by *four orders of magnitude*, from millions to a few hundred cases. Hence, VuTeCo is designed as a lightweight assessment tool to find vulnerability-witnessing tests before proceeding to a more comprehensive dynamic assessment. Despite the positive results obtained from the experimental evaluation ($\mathbf{RQ}_1$), the retrieval of vulnerability-witnessing tests remains a *challenging task*. We further examined the errors made by VuTeCo to identify potential improvements. The main reason is attributable to the vocabulary of the test code. As observed, terms related to security and certain constant strings often lead AI models into error, likely due to their similarity to the vocabulary of the positive instances in the training sets for the *Finding* and *Matching* tasks. This problem highlighted the need for a *dedicated pre-training session* to enable the model to adapt its vocabulary to the testing and security domains. Besides, denoising the input test method to remove excessively long strings and version numbers could be beneficial. Concerning the *Matching* task, the errors were also due to the *limited understanding of the vulnerability*, caused by the too short description given by CVE. This could be mitigated by adding additional sources to fully comprehend the vulnerability, e.g., issue trackers and mailing lists.

The Usefulness of Witnessing Tests. The release of Vul4J in 2022 paved the way for numerous software security tasks. The ability of VuTeCo to find security-related tests can help expand the known body of vulnerability-witnessing tests, enabling activities like the automated generation of security unit tests [11, 19, 32]. Nevertheless, the potential applications of vulnerability-witnessing tests extend far beyond this [48]. For example, the witnessing tests can act as *proofs-of-concept* supporting the automatic generation of realistic exploits, building on advances in security test generation models [11, 19]. Besides, they can support the automated vulnerability repair (AVR) process by localizing the vulnerable statements or assessing the plausibility of a generated patch [7, 46, 55, 68]. We also envision using witnessing tests to support the retrieval of vulnerability-contributing commits [6, 24], as they can be run to triangulate when a vulnerability was introduced in a project; to the best of our knowledge, this task has not been investigated yet. Additionally, software engineers can benefit from access to a variety of example test cases. For instance, they can reuse tests from past vulnerabilities to address similar issues in their projects. The dataset Test4Vul, which we publicly release for the research community [30], is designed to achieve such foreseen applications.

The Anatomy of Witnessing Tests. The retrieval of witnessing tests is challenging mainly due to the lack of empirical knowledge on what such tests look like. To date, no study has outlined a clear profile of tests that witness vulnerabilities or, more broadly, unit tests focused on security. The *absence of characterization* of vulnerability tests entails a significant knowledge gap, especially when compared with traditional functional tests. For instance, we are unaware of the setup required before calling the vulnerable component, what assertions should examine, or the number of tests required to "cover" all relevant scenarios for a vulnerability type. The only commonality between the two test types is that they both aim to identify undesirable behaviors in the code that violate certain requirements or properties. We believe this lack of knowledge might be ascribable to the difficulty in formulating security requirements at the unit/component level (i.e., methods or classes) since they are often considered at the system level [16, 39]. Unfortunately, shedding light on these aspects requires numerous examples of witnessing tests. In fact, this study was conducted to address this shortage by providing an approach (VuTeCo) to expand the knowledge base of witnessing tests and to draw more attention to this topic. Once a line is drawn between vulnerability-witnessing tests and traditional tests, innovative solutions can be designed to help developers write more security tests.

6 Threats to Validity

We carefully addressed potential sources of data leakage that could affect the validity of VuTeCo's performance. During the experimental evaluation ($\mathbf{RQ}_1$), the two pre-training datasets for *Matching* task (see Section 3.1.3), i.e., TR_F and TR'_M, were cleaned from test methods and vulnerability descriptions appearing in the test set TE_M. During the evaluation in the wild ($\mathbf{RQ}_2$), we ignored any known vulnerability-witnessing test case from Vul4J, as all have

been used to train VuTeCo's models. We also acknowledge that the original pre-training of the CRMs and LLMs might have seen some of the test methods used in our evaluation, raising the possibility of data contamination. Nevertheless, the *Finding* and *Matching* classification tasks were different from the objectives of such pre-training, which focused on predicting missing code tokens. Namely, the models had no prior knowledge about whether a certain test method was security-related or linked to specific vulnerabilities.

The three LLMs experimented with are also available in larger variants, reaching up to ~30 billion parameters. We opted for the ~7 billion versions due to memory and computational constraints. This provided a reasonable balance between capability and efficiency [31], especially given the large dataset sizes for the two tasks and the many configurations we tested.

The 336 configurations tested in the experimental evaluation ($\mathbf{RQ}_1$) explored key factors affecting performance. While additional designs were possible, resource limitations made further testing impractical. $\mathbf{RQ}_1$ focused on the model types (CRMs and LLMs), which were hypothesized (and confirmed) to have a relevant impact.

VuTeCo has been trained on JUnit test methods from projects appearing in Vul4J. The results cannot be generalized as-is to other programming languages or testing frameworks (e.g., TestNG). We focused on Java because of Vul4J, which provides validated examples of witnessing tests. Java remains a relevant language to analyze from a security perspective, as it continues to exhibit new vulnerabilities [61]. The results could also not be extended to vulnerability types not existing in Java, e.g., memory-related vulnerabilities, or underrepresented in Vul4J, such as 'OS Command Injection' (CWE-78) that had only one test case [8]. We partially mitigated this lack of examples during training using data augmentation.

7 Related Work

Test Case Classification. No research has classified test cases to identify those related to security. Existing studies primarily focused on determining whether a test exhibits *flaky* behavior. Fatima et al. [15] presented Flakify, a data-driven approach to detect flaky tests in Java projects. Flakify leverages a pre-trained CodeBERT and a feed-forward neural network to predict whether a JUnit test method had a flaky behavior. Flakify achieved 0.79 and 0.73 F1 scores on two different experiments on FlakeFlagger dataset [2], while achieving 0.98 and 0.89 F1 score on IDoFT dataset [34], outperforming state of the art approaches. Somewhat similarly, FlakyCat exploits few-shot learning to predict the exact category of flakiness of JUnit tests [1]. FlakyCat relies on a pre-trained CodeBERT to create the embeddings of test cases and a Siamese Network to project the embeddings into a space where tests of the same flakiness category appear similar (based on cosine distance). This is enacted by the Triplet Loss function [57] during the training.

Security Unit Testing. Existing works focused on generating test cases for *third-party vulnerabilities*, i.e., those indirectly added through dependencies (e.g., libraries) rather than introduced by the project developers. Kang et al. [32] introduced Transfer to generate security test cases for Java projects affected by vulnerable library dependencies. Transfer builds on existing vulnerability-witnessing tests mined from the upstream library project and tries to generate a test case targeting the client project that recreates the same program state generated by the execution of the witnessing test in the original library. This approach uses a genetic algorithm to find a client test that "mimics" the library test's behavior. Transfer successfully generated security tests for 14 known library vulnerabilities in 23 client projects. Later, Transfer was extended by Chen et al. [11] by including a migration step, which helps ensure that generated tests from client projects are similar to the original vulnerability tests. Their tool, Vesta, outperforms Transfer by 53% in test generation effectiveness on a dataset of 30 Java vulnerabilities. More recently, Gao et al. introduced VulEUT [19], which combines static call graph analysis to find out the triggering conditions and a GPT-3.5-Turbo model to generate the concrete unit tests. VulEUT succeeded in 56/70 cases, whereas VESTA only succeeded in 45.

8 Conclusion

We presented VuTeCo, a framework for finding security-related tests in Java projects and matching them to their corresponding vulnerabilities. The experimental evaluation ($\mathbf{RQ}_1$) identified promising AI models for *Finding* and *Matching* tasks, while the assessment in the wild ($\mathbf{RQ}_2$) demonstrated the usefulness of VuTeCo. The manually-confirmed tests have been collected in a novel dataset, Test4Vul, which we have publicly released. VuTeCo is the first solution explicitly addressing the problem of finding security unit tests in software repositories, laying the foundation for future research in this area.

After extensive experimentation, we identified several ways to improve VuTeCo. The maximum input size for VuTeCo's AI models could be expanded to include *contextual information*, such as the production code (e.g., the vulnerable class or method) or a natural-language summary of the test cases. Namely, the *Matching* could use additional textual sources like security bug reports or commit messages [6, 29, 35, 47, 69] to improve its accuracy. Then, the *Matching* task could replace its "pair-wise" classification style with a "zero-shot" scheme, allowing it to evaluate a set of candidate vulnerabilities at once rather than individually, thereby reducing the overall number of predictions. Lastly, VuTeCo could include an *automated dynamic assessment* of the retrieved tests to confirm that they can trigger the matched vulnerability as expected.

Data Availability

The paper's **replication package** is available on FigShare [30]. It contains the implementation of VuTeCo (as in this paper), the dataset Test4Vul, the scripts to reproduce the experiments, and the resulting raw and processed data.

Acknowledgments

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

References

[1] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2023. FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. 140–151. doi:10.1109/AST58925.2023.00018

[2] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1572–1584. doi:10.1109/ICSE43902.2021.00140

[3] Apache Flink. 2025. Test case testArrayClassPollutionBlockedByPattern in class ParametersInterceptorTest.java. https://github.com/apache/struts/blob/f06bc517ab2bb68521293ddb61fbdae10833085b/core/src/test/java/org/apache/struts2/interceptor/parameter/ParametersInterceptorTest.java#L286. Online.

[4] Andrew Austin, Casper Holmgreen, and Laurie Williams. 2013. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology* 55, 7 (2013), 1279–1288. doi:10.1016/j.infsof.2012.11.007

[5] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. 1999. *Modern Information Retrieval*. ACM Press.

[6] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*. 2352–2364.

[7] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. 2024. APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities. *Empirical software engineering* 29, 1 (2024), 18.

[8] 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* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 464–468. doi:10.1145/3524842.3528482

[9] c2nes. 2025. javalang. https://github.com/c2nes/javalang. Online; accessed 29 July 2025.

[10] Ricardo Campos, Vítor Mangaravite, Arian Pasquali, Alípio Jorge, Célia Nunes, and Adam Jatowt. 2020. YAKE! Keyword extraction from single documents using multiple local features. *Information Sciences* 509 (2020), 257–289. doi:10.1016/j.ins.2019.09.013

[11] Zirui Chen, Xing Hu, Xin Xia, Yi Gao, Tongtong Xu, David Lo, and Xiaohu Yang. 2024. Exploiting Library Vulnerability via Migration Based Automating Test Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 228, 12 pages. doi:10.1145/3597503.3639583

[12] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[13] Computer Incident Response Center Luxembourg (CIRCL). 2025. Vulnerability-Lookup. https://cve.circl.lu/. Online; accessed 29 July 2025.

[14] Daniela Soares Cruzes, Michael Felderer, Tosin Daniel Oyetoyan, Matthias Gander, and Irdin Pekaric. 2017. How is Security Testing Done in Agile Teams? A Cross-Case Analysis of Four Software Teams. In *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, Cham, 201–216.

[15] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. 2023. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1912–1927. doi:10.1109/TSE.2022.3201209

[16] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. 2016. Chapter One - Security Testing: A Survey. Advances in Computers, Vol. 101. Elsevier, 1–51. doi:10.1016/bs.adcom.2015.11.003

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[18] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[19] Yi Gao, Xing Hu, Zirui Chen, Tongtong Xu, and Xiaohu Yang. 2025. Vulnerability-Triggering Test Case Generation from Third-Party Libraries. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. 125–135. doi:10.1109/Forge66646.2025.00021

[20] Danielle Gonzalez, Paola Peralta Perez, and Mehdi Mirakhorli. 2021. Barriers to Shift-Left Security: The Unique Pain Points of Writing Automated Tests Involving Security Controls. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '21)*. Association for Computing Machinery, Article 11, 12 pages. doi:10.1145/3475716.3475786

[21] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).

[22] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE] https://arxiv.org/abs/2401.14196

[23] Dan Hendrycks and Kevin Gimpel. 2023. Gaussian Error Linear Units (GELUs). (2023). arXiv:1606.08415 [cs.LG] https://arxiv.org/abs/1606.08415

[24] Torge Hinrichs, Emanuele Iannone, Tamás Aladics, Péter Hegedundefineds, Andrea De Lucia, Fabio Palomba, and Riccardo Scandariato. 2025. Back to the Roots: Assessing Mining Techniques for Java Vulnerability-Contributing Commits. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2025). doi:10.1145/3769105 Just Accepted.

[25] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. (2012). arXiv:1207.0580 [cs.NE] https://arxiv.org/abs/1207.0580

[26] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] https://arxiv.org/abs/2106.09685

[27] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186

[28] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 [cs.LG] https://arxiv.org/abs/1909.09436

[29] Emanuele Iannone, Giulia Sellitto, Emanuele Iaccarino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2024. Early and Realistic Exploitability Prediction of Just-Disclosed Software Vulnerabilities: How Reliable Can It Be? *ACM Trans. Softw. Eng. Methodol.* (mar 2024). doi:10.1145/3654443 Just Accepted.

[30] Iannone, Emanuele and Bui, Quang-Cuong and Scandariato, Riccardo. 2025. Paper's Online Appendix. https://figshare.com/s/44b014d85a5024358570. Online.

[31] Alexia Jolicoeur-Martineau. 2025. Less is More: Recursive Reasoning with Tiny Networks. arXiv:2510.04871 [cs.LG] https://arxiv.org/abs/2510.04871

[32] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S. Păsăreanu, and David Lo. 2022. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 276–288. doi:10.1145/3533767.3534398

[33] Arvinder Kaur and Ruchikaa Nayyar. 2020. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science* 171 (2020), 2023–2029. doi:10.1016/j.procs.2020.04.217 Third International Conference on Computing and Network Communications (CoCoNet'19).

[34] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 312–322. doi:10.1109/ICST.2019.00038

[35] Triet Huynh Minh Le, Bushra Sabir, and Muhammad Ali Babar. 2019. Automated software vulnerability assessment with concept drift. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 371–382.

[36] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 544–555. doi:10.1145/3533767.3534380

[37] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 31–42.

[38] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. (2019). arXiv:1711.05101 [cs.LG] https://arxiv.org/abs/1711.05101

[39] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2018. A Natural Language Programming Approach for Requirements-Based Security Testing. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 58–69. doi:10.1109/ISSRE.2018.00017

[40] Brian W. Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure* 405, 2 (1975), 442–451.

[41] Mary McHugh. 2012. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB* 22, 3 (Oct. 2012), 276–82. doi:10.11613/bm.2012.031

[42] Microsoft. 2025. CodeBERT. https://github.com/microsoft/CodeBERT/blob/master/UniXcoder/unixcoder.py#L80. Online; accessed 29 July 2025.

[43] Microsoft. 2025. UniXcoder-base. https://huggingface.co/microsoft/unixcoder-base. Online; accessed 29 July 2025.

[44] Mahmoud Mohammadi, Bill Chu, and Heather Richter Lipford. 2019. Automated repair of cross-site scripting vulnerabilities through unit testing. In *2019 IEEE International symposium on software reliability engineering workshops (ISSREW)*. IEEE, 370–377.

[45] Mahmoud Mohammadi, Bill Chu, Heather Richter Lipford, and Emerson Murphy-Hill. 2016. Automatic web security unit testing: XSS vulnerability detection. In *Proceedings of the 11th International Workshop on Automation of Software Test*

(Austin, Texas) *(AST '16)*. Association for Computing Machinery, New York, NY, USA, 78–84. doi:10.1145/2896921.2896929

[46] Mahmoud Mohammadi, Bill Chu, and Heather Richter Lipford. 2019. Automated Repair of Cross-Site Scripting Vulnerabilities through Unit Testing. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 370–377. doi:10.1109/ISSREW.2019.00098

[47] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E Hassan, et al. 2023. Multi-granularity detector for vulnerability fixes. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4035–4057.

[48] Zhiyuan Pan, Xing Hu, Xin Xia, Xian Zhan, David Lo, and Xiaohu Yang. 2024. PPT4J: Patch Presence Test for Java Binaries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 225, 12 pages. doi:10.1145/3597503.3639231

[49] Karl Pearson and Francis Galton. 1895. VII. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London* 58, 347-352 (1895), 240–242. arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rspl.1895.0041 doi:10.1098/rspl.1895.0041

[50] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A comparative study of automatic program repair techniques for security vulnerabilities. In *2021 IEEE 32nd international symposium on software reliability engineering (ISSRE)*. IEEE, 196–207.

[51] David M. W. Powers. 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[52] Qwen. 2025. Qwen2.5-Coder-7B. https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct. Online; accessed 29 July 2025.

[53] Md Rafiqul Islam Rabin, Nghi D.Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552. doi:10.1016/j.infsof.2021.106552

[54] 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.CL] https://arxiv.org/abs/2308.12950

[55] Zoltán Ságodi, Gábor Antal, Bence Bogenfürst, Martin Isztin, Péter Hegedundefineds, and Rudolf Ferenc. 2024. Reality Check: Assessing GPT-4 in Fixing Real-World Software Vulnerabilities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 252–261. doi:10.1145/3661167.3661207

[56] SAP. 2025. project-kb. https://github.com/SAP/project-kb. Online; accessed 29 July 2025.

[57] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. doi:10.1109/cvpr.2015.7298682

[58] Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.* 44, 3, Article 11 (jun 2012), 46 pages. doi:10.1145/2187671.2187673

[59] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, Christian Stein. 2025. JUnit 5 User Guide. https://docs.junit.org/current/user-guide/. Online; accessed 29 July 2025.

[60] C.J. Van Rijsbergen. 1979. *Information Retrieval*. Butterworths. https://books.google.de/books?id=t-pTAAAAMAAJ

[61] Veracode. 2024. State of Software Security. https://www.veracode.com/wp-content/uploads/2024/06/SOSS-Report-2024.pdf. Online; accessed 29 July 2025.

[62] Chaozheng Wang, Zongjie Li, Yun Peng, Shuzheng Gao, Sirong Chen, Shuai Wang, Cuiyun Gao, and Michael R. Lyu. 2024. REEF: A Framework for Collecting Real-World Vulnerabilities and Fixes. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 1952–1962. doi:10.1109/ASE56229.2023.00199

[63] 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* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 472–483. doi:10.1145/3639478.3647634

[64] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint* (2023).

[65] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data Augmentation by Program Transformation. *Journal of Systems and Software* 190 (2022), 111304. doi:10.1016/j.jss.2022.111304

[66] Ying Zhang, Wenjia Song, Zhengjie Ji, Danfeng, Yao, and Na Meng. 2023. How well does LLM generate security tests? (2023). arXiv:2310.00710 https://arxiv.org/abs/2310.00710

[67] Zhilu Zhang and Mert R. Sabuncu. 2018. Generalized cross entropy loss for training deep neural networks with noisy labels. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 8792–8802.

[68] 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 IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 88, 13 pages. doi:10.1145/3597503.3639222

[69] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 914–919.