



# AI4CYBER

Artificial Intelligence for next generation CYBERsecurity














Deliverable title	Deliverable ID:
<b>AI-driven self-testing and automatic error correction for robustness – Initial version</b>	D3.1
	Preparation date:
	31/01/2024
	Editor/Lead beneficiary (name/partner):
	Ákos Milánkovich / SLAB
	Internally reviewed by (name/partner):
	Ana Rosa Cavalli / MI Marek Pawlicki / ITTI

Abstract:

This deliverable outlines the initial version of two pivotal components of the AI4CYBER project: AI-driven self-testing and automatic error correction (AI4FIX) and AI-enhanced Vulnerability Identification (AI4VULN). It provides a detailed depiction of these innovative AI-powered tools, focusing on their architectural design, specifications, and the proof-of-concept tests. AI4FIX leverages advanced AI technologies like large language models to automate the correction of robustness-related weaknesses in software code, while AI4VULN employs AI-boosted symbolic execution to efficiently identify vulnerabilities in source code. The document also describes the process of creating and analysing a comprehensive dataset repository from open-source software development records, which is crucial for tracking software evolution and identifying vulnerabilities. The implementation of these models, currently a work in progress, includes initial testing with collected datasets using third-party online models.

Dissemination level		
<b>PU</b>	Public, fully open	X
<b>SEN</b>	Sensitive, limited under the conditions of the Grant Agreement	
<b>EU-R</b>	Classified R-UE/EU-R – EU RESTRICTED under the Commission Decision No 2015/444	

## AI4CYBER consortium

1	 <p><b>tecnalia</b> MEMBER OF BASQUE RESEARCH &amp; TECHNOLOGY ALLIANCE</p>	Fundación Tecnalía Research & Innovation (TECNALIA, Spain)
2		University of Western Macedonia (UOWM, Greece)
3		Montimage EURL (MI, France)
4		Thales Six GTS France SAS (TSG, France)
5	 <p><b>SEARCH-LAB</b> SECURITY EVALUATION ANALYSIS AND RESEARCH LABORATORY</p>	Search Lab (SLAB, Hungary)
6		Frontendart Szoftver KFT (FEA, Hungary)
7	 <p><b>EOS</b> EUROPEAN ORGANISATION FOR SECURITY</p>	European Organisation for Security (EOS, Belgium)
8		PDM E FC Projecto Desenvolvimento Manutencao Formacao e Consultadorialda (PDMFC, Portugal)
9		ITTI Sp. z o.o. (ITTI, Poland)
10		Public Power Corporation S. A. (PPC, Greece)
11	 <p>Hospital do Espírito Santo E.P.E.</p>	Hospital Do Espírito Santo de Evora EPE (HES, Portugal)
12		Caixabank S.A. (CXB, Spain)
13		Metamind Innovations P.C (MINDS, Greece)

## Table of contents

AI4CYBER consortium .....	2
List of figures .....	5
List of tables .....	6
Executive Summary .....	7
1 Introduction.....	8
1.1 Objective of the document .....	8
1.2 Structure of the document .....	8
1.3 Relationships with other deliverables .....	8
1.4 Contributors.....	8
1.5 Acronyms and abbreviations .....	8
1.6 Revision history.....	9
1.7 Change log.....	10
2 AI4FIX Architectural Design & Implementation – Initial Version.....	11
2.1 AI4FIX – State of the Art Analysis.....	11
2.1.1 Robustness improvements .....	11
2.1.2 Software evolution .....	13
2.1.3 AI-based program repair .....	22
2.1.4 AI-generated automated testing .....	24
2.2 AI4FIX System Requirements Analysis .....	27
2.3 AI4FIX Specification .....	28
2.3.1 System context .....	28
2.3.2 Container model .....	29
2.3.3 Component model .....	30
2.3.4 Hardware requirements .....	34
2.3.5 Software evolution .....	34
2.3.6 Code synthesis using AI and LLM.....	35
2.4 AI4FIX Communication Interfaces.....	43
2.5 AI4FIX Installation and User Guide .....	44
2.5.1 Installing dependencies .....	44
2.5.2 Packaging, publishing .....	44
2.5.3 Generate .vsix file: .....	45
2.5.4 Example of usage on a demo project .....	45
2.5.5 Example project.....	46
2.5.6 Open-source repository .....	46
2.6 AI4FIX Unit Testing .....	46
3 AI4VULN Architectural Design & Implementation – Initial Version .....	50
3.1 AI4VULN – Symbolic execution-based vulnerability detection .....	50
3.1.1 State of the Art Analysis .....	50
3.1.2 Background .....	51
3.1.3 Introduction .....	57
3.1.4 Handling finally blocks .....	62
3.1.5 Constraint solver integration .....	63
3.1.6 Low-level integration challenges .....	67
3.1.7 Integration result .....	68
3.2 AI-based vulnerability detection .....	70
3.3 AI4VULN System Requirements Analysis.....	71
3.4 AI4VULN Specification .....	71
3.4.1 System context .....	72

---

3.4.2	Container model .....	72
3.4.3	Component model .....	73
3.4.4	Prospector Dataset .....	74
3.4.5	JiraMiner dataset .....	75
3.4.6	Identifying security relevance .....	78
3.4.7	LLM-based Vulnerability detection (proof of concept) .....	84
3.5	AI4VULN Communication Interfaces .....	88
3.6	AI4VULN Installation and User Guide .....	90
3.7	AI4VULN Unit Testing .....	90
3.7.1	Validation on open-source projects .....	91
4	Conclusions .....	93
	References .....	94



## List of figures

Figure 1: System context of AI4FIX.....	29
Figure 2: Container model of AI4FIX.....	30
Figure 3: Component model of Software evolution.....	31
Figure 4: Component model of Model evolution.....	32
Figure 5: Component model of Fix and Test generation .....	33
Figure 6: isTriangleValid function.....	52
Figure 7: Symbolic execution tree .....	53
Figure 8: Control Flow Graph (CFG) of the isTriangleValid function.....	56
Figure 9: Symbolic execution tree .....	57
Figure 10: SMT-LIBv2 language.....	60
Figure 11: Try-catch with multiple resources .....	61
Figure 12: Original CFG .....	61
Figure 13: Improved CFG with resource handling .....	62
Figure 14: Try-catch-finally construct .....	62
Figure 15: Original CFG for handling finally blocks.....	63
Figure 16: Improved handling of finally blocks.....	63
Figure 17: Participants of an SMT solver competition .....	65
Figure 18: Faulty test case.....	70
Figure 19: System context of AI4VULN .....	72
Figure 20: Container model of AI4VULN .....	73
Figure 21: Component model of AI4VULN .....	74
Figure 22: Security-relevance classification .....	84
Figure 23: Scorecard for command injection results .....	87
Figure 24: Scorecard for weak randomness results.....	88
Figure 25: Lines of Code histogram of the examined Java projects .....	92

## List of tables

Table 1: Goals of software evolution approaches .....	13
Table 2: Datasets for software evolution.....	16
Table 3: Metrics for software evolution.....	17
Table 4: Models for software evolution .....	20
Table 5: Comparison of LLMs.....	23
Table 6: Benchmarks.....	23
Table 7: System requirements coverage by initial version of AI4FIX component.....	27
Table 8: Hardware requirements for the tools.....	34
Table 9: Functional validity comparison.....	36
Table 10: Result of developers' evaluation.....	37
Table 11: Test generation results .....	43
Table 12: System test case AI4FIX_001 .....	46
Table 13: System test case AI4FIX_002.....	47
Table 14: System test case AI4FIX_003.....	47
Table 15: System test case AI4FIX_004.....	48
Table 16: Integration test case AI4FIX_005 .....	48
Table 17: Results of the measurements.....	66
Table 18: Improvements in the regression tests .....	69
Table 19: System requirements coverage by initial version of AI4VULN component.....	71
Table 20: Keyword classes in JIRA .....	78
Table 21: Number of issues, which have at least one associated java file.....	79
Table 22: Metrics results .....	83
Table 23: Command injection test results .....	86
Table 24: Testing one-shot and self-reflection techniques.....	88
Table 25: System test case AI4VULN_001 .....	90
Table 26: System test case AI4VULN_002 .....	91
Table 27: The Lines of Code and Number of Classes metrics of the examined Java projects	92

## Executive Summary

This document is deliverable D3.1 AI-driven self-testing and automatic error correction for robustness – Initial version of AI4CYBER project.

The deliverable presents the outcomes of tasks T3.1 and T3.2 of WP3 “AI-driven Preparedness for Robustness”, captured in the form of two key exploitable result prototypes, AI4FIX and AI4VULN, respectively.

Task 3.1 studies the “AI-driven self-testing and automatic error correction” and led by SLAB focuses on utilizing advanced AI technologies, such as large language models, to automate the correction of robustness-related weaknesses in software code. This task involves creating a repository of datasets from various open-source software development records, spanning decades. It aims to analyze these datasets to track software evolution, identify and correct vulnerabilities - a step beyond the capabilities of current code analysis tools. The key objective is to enable automatic code corrections without relying solely on human intervention. Additionally, Task 3.1 will develop AI-based methods to generate unit tests and automatically validate the modified system, ensuring thorough re-testing post-correction. These developments are integral to the AI4FIX system, enhancing automated software maintenance and quality assurance."

Task 3.2 deals with the challenges of “AI-enhanced Vulnerability identification” and led by FEA aimed to develop an AI-boosted symbolic execution technique, named AI4VULN, designed to efficiently discover vulnerabilities in source code. This approach involves the creation of machine learning models capable of discerning which execution paths in the code can be truncated or omitted, particularly those less likely to contain vulnerabilities. This strategy aims to reduce the overall state space, enabling the symbolic execution to focus more intensively on paths that have a higher likelihood of security issues. By optimizing the symbolic execution process through AI, the task seeks to enhance the effectiveness and depth of security analysis in source code.

The document describes the research directions, architectural design, specification, proof of concept tests and initial versions of the AI4FIX and AI4VULN components that will be fully developed and implemented thorough the project.

The implementation of the models is a work in progress. Proof of concept testing with the collected datasets were conducted using third party online models.

# 1 Introduction

## 1.1 Objective of the document

This document is deliverable D3.1 *AI-driven self-testing and automatic error correction for robustness – Initial version* of AI4CYBER project [1].

The document describes the research directions, plans and initial versions of the prototype implementation of both the AI4FIX component aimed at automatic correction of robustness related weaknesses and the AI boosted vulnerability identification component, AI4VULN, both part of AI4CYBER framework.

## 1.2 Structure of the document

After the introductory Section 1, Section 2 describes the AI4FIX component, the related State of the Art, research results and architecture and plans for the implementation. Section 3 describes the AI4VULN component following the same structure. Finally, Section 4 summarizes the document and provides future directions of research and prototype implementation.

## 1.3 Relationships with other deliverables

The initial tool descriptions presented in the present document relate to the following deliverables:

- *D2.1 AI4CYBER Framework Use cases, Requirements and Architecture – Initial Version*: This deliverable provided the specification of the AI4CYBER framework, including the description of the overall architecture and multiple viewpoints of the framework solution. The document presented the high-level design of the framework components, including both AI4FIX and AI4VULN detailed herein. The present deliverable provides more technical information of the initial prototypes of both components.
- *D3.3 AI-driven self-testing and automatic error correction for robustness – Final version*: this document will present the implemented final versions of the AI4FIX and AI4VULN components. The tools will be demonstrated in the context of WP7, in the Use case 2 banking applications.

## 1.4 Contributors

The following partners have mainly contributed to this deliverable, with the support of other partners in WP3:

- SLAB has coordinated the work in WP3 and has led the edition of the document. Furthermore, SLAB has led the work in AI4VULN component specification, design and prototype development.
- FEA has focused their contribution on the specification, design and development of AI4FIX component.

## 1.5 Acronyms and abbreviations

API      Application Programming Interface      LOC      Lines of Code



APR	Automatic Program Repair	LoC	Lines of Code
AST	Abstract Syntax Tree	LP	Linear Programming
BB	Basic Block	NP	Nondeterministic Polynomial
CFG	Control Flow Graph	npm	Node Package Management
CWE	Common Weakness Enumerations	RAG	Retrieval Augmented Generation
DGRNN	Deep general regression neural network	SMT	Satisfiability Modulo Theories
FDSS	Feedback-Driven Solution Synthesis	TOC	Table of Contents
FP	False Positive	ToE	Target of Evaluation
GPT	Generative Pre-Trained Transformer	TP	True Positive
LLM	Large Language Model	WP	Work Package

## 1.6 Revision history

Version	Date issued	Author	Organisation	Description
0.1	18/01/2023	Ákos Milánkovich	SLAB	TOC revised agreed
0.3	08/09/2023	Ákos Milánkovich	SLAB	Initial version
0.5	15/09/2023	Ákos Milánkovich, Istvan Siket	SLAB, FEA	Intermediate version
0.7	15/12/2023	Ákos Milánkovich, Istvan Siket	SLAB, FEA	Both components integrated draft
0.9	12/01/2024	Ákos Milánkovich	SLAB	Version for internal review-
0.9.1	19/01/2024	Marek Pawlicki	ITTI	Reviewed version
0.9.2	23/01/2024	Ana Rosa Cavalli	MI	Reviewed version
0.9.3	29/01/2024	Ákos Milánkovich	SLAB	Final formatting, improvements based on review
1.0	31/01/2024	Erkuden Rios	TECNALIA	Final submitted

## 1.7 Change log

No change log entries.

## 2 AI4FIX Architectural Design & Implementation – Initial Version

This section describes the initial architecture and implementation of AI4FIX based on the state-of-the-art research results.

### 2.1 AI4FIX – State of the Art Analysis

This section provides the state-of-the-art analysis about AI-based robustness improvements, code, and fix generation techniques, in relation to AI4FIX.

#### 2.1.1 Robustness improvements

In the dynamic landscape of software development, ensuring robustness improvements is paramount to building resilient and trustworthy applications. Recent advancements in the realm of artificial intelligence (AI), particularly Large Language Models (LLMs), have opened new avenues for fortifying the integrity of software systems. Leveraging the capabilities of LLMs offers a promising approach to enhance robustness improvements in terms of security.

Within the context of software system or application development, robustness improvement is a critical consideration for ensuring the security and resilience of the source code. Leveraging Large Language Models (LLMs) to enhance the robustness of source code involves several key aspects.

One of them is the security best practices and guidelines which serve as foundational pillars in the endeavor to fortify the robustness of the source code. By integrating these principles, developers can establish a secure coding framework that mitigates potential vulnerabilities.

Another facet of robustness improvement involves advocating for more widespread and/or specific error handling mechanisms within the source code. This ensures that potential issues are not only identified but are also addressed comprehensively, contributing to the overall reliability and security of the software.

Additionally, employing different static analyzer tools is instrumental in identifying and rectifying bugs within the source code. These tools systematically analyze the codebase, uncovering issues that may range from common programming errors to intricate security vulnerabilities. By proactively addressing these findings, developers strengthen the robustness of the source code against potential exploits.

##### 2.1.1.1 Literature review

A recent study [2] about code generation with LLMs particularly focusing on their capabilities and limitations in handling security issues and generating secure code. The study introduces an approach called Feedback-Driven Solution Synthesis (FDSS), where LLMs receive varied feedback to generate solutions for security problems. The datasets used for evaluation include LLMSecEval, SecurityEval, and PythonSecurityEval. These datasets are designed to test LLMs' abilities in generating secure source code and include a range of Common Weakness Enumerations (CWEs). As the study emphasized, LLMs often struggle with security issues due to limited understanding of security vulnerabilities. Therefore, it explores how LLMs generate code with security issues and their capability to resolve these issues through self-refining or using feedback from external tools like Bandit, a static code analysis tool for Python. Within the FDSS approach the Bandit static analyzer tool was used as a key component to enhance the security capabilities of LLMs. It was utilized as a feedback source as it analyzes the code

generated by the LLMs and identifies potential security vulnerabilities. The feedback includes the type of security issue and the specific line of code where the issue is found. This information is crucial for the models to understand the nature of the security issues and to generate possible solutions for resolving them. In addition to direct feedback, the study also explores the verbalization technique, where Bandit's feedback is transformed into a detailed explanation in natural language. This expanded explanation from Bandit is then provided to the LLMs, which helps them gain deeper insights into the security problems and suggests methods for resolving them. The approach also involves an iterative process where the LLMs use Bandit's feedback to refine and improve the generated code. Each proposed solution, along with feedback from Bandit, is sent back to the model for further refinement, aiming to address and resolve the security issues effectively.

The FDSS approach demonstrates that LLMs can generate solutions to address security issues effectively. The study evaluates the abilities of advanced LLMs, including GPT-4, GPT-3.5, and CodeLlama, using various baseline techniques. The results show that GPT-4 has the highest percentage of refining security issues in all datasets, exceeding 90%. The study also finds that the self-debugging approach is less effective for GPT-3.5 and CodeLlama compared to direct prompting but enhances performance in GPT-4.

In another study [3], which focuses on enhancing program repair using Large Language Models (LLMs) together with static analysis tools. The primary objective was to enhance the capabilities of LLMs in program repair, focusing on fixing critical bugs like Null Pointer Dereference (NPD), Resource Leak (RL), and Thread Safety Violation. The study explores how to leverage static analysis for bug detection, localization, and categorization, paired with LLMs fine-tuned for the program repair task. As a result, a new framework called InferFix was introduced which employs the Infer static code analyzer and transformer encoder model with a Codex Cushman generator model. The Codex model is fine-tuned using the InferredBugs dataset, enriched with relevant program repair patterns. The dataset is a metadata-rich collection of bugs and fixes in Java and C# programming languages extracted using the Infer static analyzer. This dataset is differentiated by the detailed information provided about each bug, including bug type, location, and change history. The InferFix framework can identify critical security and performance issues in the code and with the help of prompt augmentation can generate fixes for the relevant code parts. The prompt augmentation technique is leveraging on the dense retrieval from an external database of historic bugs and fixes, bug type annotations, and syntactic hierarchies. Altogether InferFix achieves impressive top-1 accuracy of patch generation, with 76% in Java and over 65% in C# across the various bug types.

Updating outdated third-party libraries is another crucial aspect of robustness improvement in our approach. As technology evolves, so do potential threats and vulnerabilities. Regularly refreshing dependencies to their latest versions not only introduces performance enhancements but also addresses security patches, safeguarding the source code against potential exploits associated with outdated components.

In addition to the techniques, robustness improvement also implies the possibility to generate patches for code parts that are not vulnerable or not exploitable just like for example in the case of an SQL query in the source code. Even when a vulnerability in a native SQL query may not contain security relevant bug, or cannot be exploitable, adopting a more robust approach, such as utilizing parameterized SQL queries, is recommended. This proactive measure ensures the source code is fortified against potential security threats.

In conclusion, the application of LLMs for robustness improvement in the source code of software systems or applications involves a comprehensive strategy. From adhering to security best practices to advocating for improved error handling, leveraging static analyzers, and updating third-party libraries, developers can fortify the security and resilience of the source code. However, as our research shows, currently in the time of writing this document, there are no available studies that explicitly research the possibilities of robustness improvements of the source code with the help of LLMs. The ones that we found are partially utilizing the different techniques of robustness improvements, but not in accordance with our specific definition of robustness improvements.

### 2.1.2 Software evolution

This section delves into the state-of-the-art solutions for software evolution in source code through the application of AI. The focus spans various dimensions, including refactoring prediction, anti-pattern detection, latent topic analysis, commit-impact analysis, defect density prediction, and change-related prediction. While acknowledging the role of Large Language Models (LLMs), this review extends its scrutiny to encompass a spectrum of AI-driven approaches, elucidating the diverse methodologies that shape contemporary efforts in software evolutions.

**Table 1: Goals of software evolution approaches**

Goal	Reference
Refactoring prediction	[4] [2] [3]
Anti-pattern detection	[4]
Analysis of latent topics	[5]
Commit-impact analysis	[6]
Defect (density) prediction	[7] [8] [9] [10]
Change-related prediction	[11] [12]

#### 2.1.2.1 Refactoring prediction

Refactoring aims at improving the design of a software system without changing its behavior. In [4] the focus was on the prediction of refactoring-prone classes based on several historical code metrics. The number of refactoring is predicted with a classed classifier, first classifying whether a class needs refactoring, then predicting whether it is 1 or more than 1. The refactoring was identified based on information gathered from commit messages, but the different types of refactoring were not differentiated (e. g. create superclass, extract method were treated as a refactoring operation). 80% precision and recall were achieved for predicting non-refactoring prone classes, it was also observed that classes with refactoring are harder to predict. Selection of the timeframe is essential; predicting the refactoring that will happen in the following 2 months based on the previous 3 months yielded better results than the predictions based on the previous 6 months. The most important features were `coChangedNew`, `linesActivityRate`, `coChangedFiles`, `linesChangePerChange`.

Refactoring datasets are often imbalanced, meaning that there are far fewer instances of refactored classes than non-refactored classes. In [5], prediction results on imbalanced and balanced datasets were discussed, where the latter was acquired using Synthetic Minority Over-sampling Technique. The task was to predict whether a class was refactored based on the current version based on the previous version's code metrics. An F-measure of over 92% percent was obtained on every software project with a balanced dataset.

The previous approaches focused on predicting whether a class (or code segment) needs refactoring but did not predict what type of refactoring operation is needed. In [6], several supervised machine learning algorithms predict 7 class levels, 7 method levels and 6 variable levels refactoring based on several code, process and ownership metrics collected from the previous commit. A different model was trained for each refactoring operation, where for every operation, the Random Forest model performed best, with over 84% percent accuracy (but often above 90%). This shows that refactoring prediction/recommendation is possible with high accuracy. However, the exact location of the refactoring operations is unknown. Other observations were that class-level metrics were important for lower level (i.e., method and variable) refactoring and that process and ownership metrics played a crucial role.

### ***2.1.2.2 Anti-pattern detection***

Over the evolution of a software system, if it is not maintained correctly, design problems, such as anti-patterns can occur. Anti-patterns are closely tied to refactoring operations. An individual anti-pattern can be removed by a series of refactoring operations. It works in the other direction as well; an occurrence of an anti-pattern may be detected by suggested refactoring operations. (For example, multiple Extract Method operations may indicate a God Class [7].)

In [4], the aim was to detect the God Class anti-pattern based on the evolution of structural code metrics. There are metrics, such as lack of cohesion in methods, which could indicate a God Class. The F-measure of the proposed convolutional approach was 77% on a highly imbalanced dataset, which was higher than the analyzed previous approaches. An observation is that a higher performance can be achieved by using longer metrics histories.

### ***2.1.2.3 Analysis of latent topics***

Source code can be viewed as a text corpus, where the tokens are analogous to words [8]. LDA (Latent Dirichlet Allocation) was utilized to extract latent topics (in this domain same as concept of concerns) from tokenized Java source code. Each concern is represented with five words. The evolution of each topic can be analyzed over software versions. Over time, based on the topic assignments, the emergence of functionalities and design improvements can be analyzed.

### ***2.1.2.4 Commit-impact analysis***

Commit-impact analysis revolves around how an individual commit influences software quality [9]. An “impactful commit” is defined as a commit, which changed the main module of the system. Several problems could cause an impactful commit not to compile, such as, snapshot dependencies, missing dependencies, committing without compiling in the new environment, committing too early, committing too often, design changes, code cleanups, large refactorings. The following code metric changes were observed: 70% of commits change the lines of code, 35% change the number of functions, and 17% change the number of classes. About 53% of commits change the complexity metric and about 46% introduce new code smells. In most cases, security-relevant impactful commits were below 5% of all impactful commits.

### ***2.1.2.5 Defect density prediction***

Defect density is a software quality metric, which measures the number of defects per software module, often in terms of Lines of Code (LOC), i. e. the number of bugs divided by LOC [10]. Defect density prediction is about predicting the number of defects per module size, which is different from defect prediction, which is a classification problem, which is about determining



whether a program module is defective or not. The data used in this case is often sparse due to the size of modules, where bugs are absent. The proposed DGRNN (Deep general regression neural network) achieved low errors on predicting defect density from code metrics compared to other models.

#### **2.1.2.6 Defect prediction**

In [11], a framework was proposed for identifying error prone modules based on code metrics (Halstead, McCabe [[11]). This task involves classifying modules into two categories (fault-prone and non-fault-prone). A module was deemed fault-prone if it contained one or more errors. 22 classification models were employed and compared. The importance of the chosen model may be overestimated. Most classifiers had over 70% AUC on most of the datasets.

A similar bug prediction was discussed in [12], where 3 supervised models were utilized. In the dataset, already known and fixed bugs were matched with their respective source code elements and the metrics were calculated on those.

A different approach to bug prediction is to classify changes (commits) into two classes, introducing commits and “clean” commits [13]. The approach proposed in [13] uses change deltas (e.g., added code delta, deleted code delta) represented as bag of words vectors, change log, metadata, and other code metrics to classify the commits into the mentioned categories. It was found that using fewer features leads to better results. The best model achieved 0.81 and 0.94 F-measure on the buggy and clean classes, respectively, averaged over all the used projects.

#### **2.1.2.7 Change-related prediction**

Changes in the source code can be categorized into structural and non-structural changes, where structural changes affect the compiled code [13]. Each change can be assigned a change cost. In [14], it was investigated which object-oriented metrics have a high correlation with change-proneness. The object-oriented metrics for each class in a project are further processed into class metric rank lists and a combined rank list. The top 10% of classes in the combined rank list are deemed as change prone. WMC, LOC, RFC, NOM were found to have a high correlation with change-proneness, both individually and combined:

- WMC (Weight Method Class) or McCabe's complexity. It counts the number of branch instructions in a class.
- LOC (Lines of code): It counts the lines of count, ignoring empty lines and comments (i.e., it is Source Lines of Code, or SLOC). The number of lines here might be a bit different from the original file, as we use JDT's internal representation of the source code to calculate it.
- RFC (Response for a Class): Counts the number of unique method invocations in a class. As invocations are resolved via static analysis, this implementation fails when a method has overloads with the same number of parameters, but different types.
- NOM (Number of methods): Counts the number of methods. Specific numbers for total number of methods, static, public, abstract, private, protected, default, final, and synchronized methods. Constructor methods also count here.

The authors of [15] proposed new metrics to measure how classes co-change (co-evolve). Proximity of a pair of classes measures the frequency of their co-change (i.e., the number of commits where both classes changed divided by the number of commits where either of the two are changed.). The Change-Coupling Index of a class captures the change-coupling of a given

class with the rest of the software system. Class Change-Impact-Set is a set of classes for a given class, which are co-changed with other that given class. The latter can be used to predict the ripple effect on the entire system by the onset of changing a class. It was observed that class change-coupling can be predicted by past Change-Coupling Indices and that Change-Coupling Index captures structural dependency among classes but also discovers some hidden dependencies as well.

### 2.1.2.8 Datasets

Predominantly open-source Java projects, whose histories and commit information are mined from CVS, SVN, or GitHub repositories. In some instances, the authors used previously mined datasets, but extracted different features.

**Table 2: Datasets for software evolution**

Name	Comment	Usage
ArgoUML		[4] [4] [5]
Spring		[4]
Eclipse		[5] [13]
Apache projects	38 projects	[6]
	844 Java projects	[3]
	Ant, Lucene, Tomcat, Xerces	[4]
	Apache 1.3	[13]
Android API (Android Opt Telephony, Android Support)		[4]
JEdit		[4] [13]
AEEEM, SOFTLAB, MORPH repositories	in total 28 datasets collected from these repositories	[7]
NASA MDP, JM1, KC1	8 datasets from NASA MDP repository, and two additional datasets	[8]
JFreeChart, Yari, UCDEtector		[14]
ANTLR4, JUnit, Mapdb, McMMO, Mct, Oryx, Titan		[2]
EGit, JGit, Emma, Jpf	collected using SVNSearch	[15]
F-Droid projects	1233 projects	[3]
GitHub projects	9072 out of 10000 most starred Java projects	[3]
Columba, Gaim, GForge, Mozilla, Plone, PostgreSQL, Subversion, JCP		[13]

A new dataset of 46 open-source Java projects was built and discussed in [13].

Some papers may use similar projects, but the mined timeframe or features are often different. For example, in [14], the number of change requests was mined from the issue-tracker.

19580 commits from 38 Apache Java projects were mined and analyzed in [6]. The projects that did not have an update in 2017 and projects having fewer than 3000 commits were left out.



The exact projects were not mentioned in [9], only the fact that 15 Java and Python projects were selected from GitHub.

### 2.1.2.9 Features/metrics

Several features have been used as input for models. Various metrics exist which relate to the size of the source code, OO attributes, developer ownership and habits, complexity, code quality, security violations, or the number of change requests. Even the vectorized source code (e.g., as a BOW vector) was used as a feature for prediction.

The different source code metrics are applicable to different levels (project, method, file, class, method, line). For example, OO metrics, such as Cohesion among methods or Number of children, measure qualities for a given class.

Part of the metrics focus on the commits and measure, for example, the number of added or deleted lines. In some cases, the metrics are applied on the change deltas (e. g. the number of changes lines, the source code BOW vector difference).

Change metrics (e.g., Change-Coupling Index, Class Change-Impact-Set) can capture how an individual class relates to other classes in the whole project. Therefore, all the commits made in each timeframe are used to calculate them.

Metric suites also exist (e.g., McCabe metrics, Halstead metrics), whose parts or entirety are widely used in several applications.

**Table 3: Metrics for software evolution**

Category	Metric	Usage	Description
Size	linesAdded linesModified linesDeleted linesType largeChanges smallChanges	[4]	For each file, within the inspected period. linedAdded, linesModified, linesDeleted relative to total LOC. linesType means whether there are more added or modified files. largeChanges and smallChanges refer to double or half of the average change sizes, respectively.
	Lines of Code variants: with/without whitespace and comments, only code, only comments, only blank, only executable, total	[4] [6] [4] [8] [14] [3]	The widely used metric for measuring the size of a file/class in terms of source code lines.
	Number of functions/ Number of methods (declared) (NMD/NOM)	[6] [4] [9] [14]	Number of functions/methods in a class. Often inner classes are ignored.
	Number of classes (NOC)	[6] [9]	Often inner classes are ignored.
	NAD (Number of attributes/fields declared)	[4] [14]	Often inner classes are ignored.

Category	Metric	Usage	Description
	NBI (Number of Byte-code instructions) IPC (Instructions per method)	[9]	Different approach to measure size instead of measuring only the lines of code.
OO metrics	ATFD (Access to foreign data) NADC (Number of associated data classes)	[4]	ATFD: number of distinct accessed attributes of unrelated classes.
	Lack of cohesion in Methods (LCOM)	[4] [9] [14]	Based on the attributes accessed by each method.
	Coupling Between Objects (CBO)	[9] [14]	
	Number of children (NOCH) Depth of inheritance tree (DIT) Percent Public Instance Variables (PIIV) Access to Public data (APD)	[9]	
	CAM (Cohesion among methods) RFC (Response for a class)	[14]	
Team	authorCount changeCount authorSwitches authorMatch	[4]	Number of authors working on a file, how often they switch. authorMatch refers to the people assigned to an issue and the actual people contributing to the implementation of the issue
Code ownership	quantity of authors minor authors major authors author ownership	[3]	Authors who contributed less than 5% are considered minor, all the others are considered major. Author ownership measures the proportion of commits made by the most active developer.
Work habits	addingChanges modifyingChanges deletingChanges per author per file	[4]	Developer work habits per author per file.
Complexity	changeCount changeActivityRate linesActivityRate bugfix size metrics	[4]	Number of changes/lines relative to the lifetime measured in months. Bugfix metrics are the same as size metrics.
	Number of paths through the code (CX)	[6]	Measured by SonarQube. Complexity is increased by one,

Category	Metric	Usage	Description
			when the control flow splits (e.g., at an if statement).
	WMC (Weighted Method Count)	[4] [9] [14]	McCabe's cyclomatic complexity summed up for all methods of a class.
	Cyclomatic complexity	[9]	Also called McCabe metric.
Code quality	Number of code smells PMD rule violations	[6]	
Problem difficulty	coChangeNew	[4]	How often a file was co-changed with a newly introduced another file.
Relation of classes	coChangedFiles	[4]	Count of co-changed files relative to the change count.
Time constraints	avgDaysBetweenChanges avgDaysPerLine relativePeakMonth peakChangeCount changeActivityRate linesActivityRate	[4]	Number of days between changes or per line of code. Peak month contains the most revisions. Number of changes/lines of code added relative to the months.
Security	SonarQube vulnerabilities PMD Security Code Guidelines violations FindBugs (Malicious Code/Security)	[6]	Number of issues found by the mentioned analyzer tools.
Issue tracking	Number of change requests	[14]	Based on the issue-tracker system.
Change	Change-Coupling Index (CChCI) Class Change-Impact-Set (ChImpactSet)	[15]	CChCI measures how often a class co-changes with other classes. ChImpactSet is a set of classes for a given class that change together.
Process	quantity of commits number of bugfixes number of previous refactoring operations	[3]	
Text	Source code BOW vector Directory/file names New revision source code Change log BOW vector	[13]	"Words" and operators represented as a BOW vector. In some cases, the deltas are considered for a given commit.
	Source code viewed as text	[5]	



Category	Metric	Usage	Description
Halstead metrics	number of distinct operators number of distinct operands total number of operators total number of operands	[7] [8] <sup>1</sup>	Introduced by Maurice Howard Halstead. Can be statically computed
	program vocabulary program length calculated estimated program length volume difficulty effort	[7] [8] <sup>1</sup>	Derivate metrics calculated from each other or the other 4 Halstead metrics.
McCabe metrics		[7] [8] <sup>1</sup>	A metric suite consisting of complexity and data related metrics.
CK metrics		[13] [3] <sup>2</sup>	A set of metrics calculated by the CK tool <sup>3</sup> .
Android metrics	Bad Smell Method Calls WakeLock with no timeout Number of Location Listeners Number of GPS Uses XML Parsers Network Timeouts	[9]	

### 2.1.2.10 Models

Various statistical and machine learning models have been analyzed for robustness improvement tasks. The models are often validated using k-fold cross-validation, with k set to 10 or 5. To overcome the problem of imbalanced datasets, sampling techniques are used, such as SMOTE.

**Table 4: Models for software evolution**

Category	Model	Usage
Decision tree	J48, C4.5	[4] [8]
	LMT (Logistic Model Tree)	[4] [8]
	Decision Tree	[9] [3]
	Random Forest	[8] [9] [3]
SVM	SVM (and/or variants)	[8] [3] [13]
Statistical	Naïve Bayes	[8] [9] [3] [13]

<sup>1</sup> In [8], only some of the Halstead and McCabe metrics were used.

<sup>2</sup> In [3], other source code metrics are collected at class, method, and variable levels.

<sup>3</sup> <https://GitHub.com/mauricioaniche/ck>

Category	Model	Usage
	Bayesian Networks	[8]
	LDA, QDA, LARS, RVM	[8]
	LDA (Latent Dirichlet Allocation)	[5]
	Logistic Regression	[8] [9] [3]
Nearest neighbor	k-NN, K-Star	[8]
	NNge (Nearest Neighbor generalization)	[4]
Neural	CNN (Convolutional Neural Network)	[4]
	GRNN ((extended) Generalized Regression Neural Networks)	[7]
	Neural Network, Multilayer Perceptron	[8] [3]
	Gated Recurrent Unit	[2]
LLMs	ChatGPT, CodeLlama, Copilot, Claude	[16][17][18][19]
Other	ARIMA	[14]
	Combined Rank List	[14]
	Rip (Repeated Incremental Pruning)	[4]

### 2.1.2.11 Granularity

There could be two aspects from which granularity can be considered when looking at software evolution. The first is a structural aspect (e.g., module, file) and the second is a time aspect (i. e. how often the changes are recorded).

### 2.1.2.12 Structural aspect

When discussing how a software evolves, it is important to consider what structural level we are considering. Evolution can be observed on the following levels (ranging from most coarse to the finest): project, module, file, class, method, and line. Some of the different code metrics discussed earlier can be calculated for multiple levels, but some are restricted to a given granularity (e.g., cohesion among methods).

### 2.1.2.13 Time aspect

When using a versioning system (e.g., Git), the changes are recorded as commits, which can provide snapshots of a system. The underlying problem here is that a developer may commit once or twice a day, which means that all the intermediary changes made between commits cannot be found in the versioning system. In [15] an IDE tool was proposed, which records all the high-level changes made to a file. This enables us to reconstruct the path between two commits and have as fine-grained information as possible. Looking at these changes makes it possible to categorize development sessions (i. e. sessions between commits) based on the activity (e.g., bug-fixing session, new development, trial of multiple approaches).

The methods presented above capture the properties of the software in a way that the change of those properties can be analyzed on a timeline to show insights in the evolution of the software. Further research will be required to select the metrics which capture the most of security-relevant changes and would result in the best model for AI4FIX.

### 2.1.3 AI-based program repair

This section processes the possible usage of AI-based Automatic Program Repair (APR), especially using Large Language Models (LLMs). The section details the available models, possibilities, and state-of-the-art results. The results concentrate on the APR of vulnerabilities as this field is the focus of AI4VULN and AI4FIX.

#### 2.1.3.1 *State-of-the-art analysis*

A wide spectrum of tasks related to source code can be handled by LLMs, notably the automatic correction of errors, particularly vulnerabilities as noted in [20]. Since the development of GPT-2, various researchers, including Paik and Wang [21], have explored this domain. The bug-fixing potential of a finetuned GPT-2 model was specifically examined by Lajkó et al. [22] using the BugsJS [23] dataset, featuring elementary JavaScript bugs. Additionally, Lajkó et al. [24] conducted a comparative analysis of fine-tuned and standard versions of GPT-2.

The QuixBugs[25] dataset, a compilation of synthetic bug examples in Python and Java, was utilized by Prenner et al. [26]. OpenAI's Codex was employed to rectify these intentionally inserted flaws in the dataset, showcasing notable results, especially with Python code. They could fix 57.5% of the flaws in the dataset. In a similar vein, Sobania et al. [27] applied QuixBugs for assessing the performance of OpenAI's ChatGPT model, registering a substantial fixing rate (77.5%).

Despite these remarkable findings, it is crucial to recognize that these investigations primarily revolve around minor, straightforward, and artificial programming challenges, potentially not mirroring complex real-world scenarios. Hence, there's a need for further assessment with more intricate programming problems. Studies like those by Charalambous et al. [28] and Hammond et al. [29] explored the capabilities of GPT models in both generating and fixing coding errors. Meanwhile, real-world application-based studies are also being conducted, as seen in the work of Nashid et al. [30] and Xia et al. [31] utilizing actual projects and datasets like Defects4J [32] and ManyBugs [33].

Chunqiu et al. [34] embarked on a comparative study of different language models in the context of automated repair. Their approach involved validating plausible fixes by running test cases. Further, an automated repair tool named AlphaRepair was introduced by Chunqiu and Lingming [35] emphasizing interactive validation of fixes using line-level information and test suites. However, in real-world scenarios, particularly concerning software vulnerabilities, such iterative generation and testing processes may not be feasible due to the typical absence of vulnerability-specific test cases.

To truly understand the efficacy of GPT-based automated vulnerability repair in practical settings, studies need to replicate more authentic scenarios. This involves handling genuine vulnerability cases without preliminary test-based evaluations. Subsequent tests should be confined to verify the accuracy of the devised fixes. A study by Wu et al. [35], using real-life vulnerability cases from the Vul4J [36] benchmark and VJBench, made strides in this direction, though their methodology included providing specific hints and utilizing a test suite for fix validation.

#### 2.1.3.2 *Available models (LLMs)*

As LLMs gained space in software engineering not only engineers adapted to the models, but the models were trained to fit developer needs. There are multiple closed- and open-sourced models. We briefly cover the most likely models to be used with their pros and cons:

**Table 5: Comparison of LLMs**

Model name (Company)	Pros	Cons
Codex (OpenAI)	Trained in source code. Already available via various plugins.	Privacy due to OpenAI. Monthly or per-usage costs.
GPT-4 (OpenAI)	Instructions following. Mixed natural language and source code. Latest transformer model.	Privacy due to OpenAI. Monthly or per-usage costs.
CodeLlama (open-source)	Open-source on Hugging Face. No privacy issues.	Self-hosting costs. Fine-tuning costs. Lower success rate.

### 2.1.3.3 Available benchmarks

As the models require evaluations the necessity of benchmarks with wrong and fixed versions arises. The available benchmarks vary in size, data-source, and available tests. The processed literature contained not exclusively the following benchmarks:

**Table 6: Benchmarks**

Benchmark name	Data source	Pros	Cons
BugsJS	Real-life	High number of examples. Testcases.	Simplicity of bugs.
QuixBugs	Artificial	Multilingual. Commonly used.	Trivial tasks. Simplicity of bugs.
Vul4J	Real-life	Hand-validated. Proof of vulnerability. Complex problems.	Low number.

### 2.1.3.4 Setup of real-life APR

Based on the literature analysis the experiment should reflect scenarios which could be acceptable or feasible on a project. The main restrictions are on the information provided for the model. The fix should not depend on the exact line of the problem as it might not be available in every case. Proof of the problem should not be provided as in most cases it is not available, e.g., test cases. The model should not generate multiple responses as a real-life scenario cannot rely on human prioritization and lastly, the results should be consistent among separate executions, which regarding probabilistic methods is a must-include. In our solution, the VSCode Plugin could provide multiple suggestions for fixing the code, although these multiple suggestions should be good fixes and the developer should not consider the suggestion as invalid fix. Therefore, even with our Plugin providing multiple fixes, we need to examine



the Top-1 results and consistency. As the results might not match the desired outcome it must be examined if the model response could be used as a hint for the developers and therefore used in a semi-automated program repair. The evaluation must be performed on a real-life benchmark such as Vul4J.

In our experiment we executed APR tasks on vulnerabilities using the Vul4J benchmark. We evaluated the generated fixes on Vul4J vulnerability test cases and with manual validations. We executed the fix generations twice to investigate the consistency of fixes. The experiment also included the helpfulness of GPT in semi-automated program repair. The experiment started with prompt engineering which resulted in a prompt that contained description about the project to be fixed, method level vulnerability information, vulnerability category and the vulnerable source code (Java). The prompt requested the model to generate a fix for the provided source code and provide a textual fix (detailed information about how to fix the problem).

Our study showcased the potential of GPT-4 in addressing real-world vulnerabilities, diverging from many studies by focusing on practical application. We used objective tests and human evaluations to assess the feasibility and acceptability of GPT-4's solutions. The findings reveal that GPT-4's proficiency in real-world scenarios is promising but not flawless. It often provides helpful insights for developers, even if it struggles with complex fixes. Most of its successful solutions are simple, best-practice based. Overall, GPT-4 marks a significant step forward in automatic vulnerability fixing, highlighting the need for continued research to refine its capabilities for complex security challenges.

The numerical results from our experiment are the following:

- Using only the vulnerability type, GPT-4 achieved fixing rates of 32.43% and 37.84% across separate executions, averaging 35.14%. These outcomes are relatively lower than those from small, synthetic datasets, yet they align with or slightly surpass those from studies involving real-world vulnerabilities, despite not having precise vulnerability location or details. This suggests potential for future semi-automated or fully automated applications.
- Textual fixes provided by GPT-4 were found to be beneficial in 58.70% and 50.00% of instances for the first and second runs, respectively. Notably, in 19.57% and 10.87% of these cases, the text responses alone were valuable, even when the accompanying source code was not. This indicates a strong potential for using textual fixes in semi-automated program repair and source code review processes.
- To summarize all the results, GPT-4 fixes are reliable across individual runs, reaching a 31.58% pass ratio considering both runs. Considering the two separate runs, we can state that fixes are reliable and consistent, but consistency is not related to vulnerability types but rather to good practices.

This experiment covered a small field of possibilities regarding LLMs in APR although it is evident that LLMs could be used for this task. Further prompt engineering, fine-tuning and methodologies for input should be examined which might be based on documentation generation and program synthesis completely.

#### **2.1.4 AI-generated automated testing**

This section delves into the intricate challenges of automated software testing, particularly in the context of leveraging large language models (LLMs) for code generation and evaluation. One of the core problems addressed is the difficulty in creating effective test suites that can cover a wide range of scenarios and catch potential bugs in code. We discuss the limitations of



existing testing approaches, which often fail to account for the nuances and complexities of real-world software development, such as handling diverse APIs and understanding the intent behind code snippets.

To tackle these challenges, a paper from Tufano et al. [37] presents a comprehensive study in the field of automated unit test generation, focusing on the use of AI models to enhance this process. The research was structured around several key research questions (RQs), each aiming to explore different aspects of unit test generation using AI models, like examining the impact of model pretraining on unit test case generation and investigating how different levels of focal context affect training for unit test generation.

The study addresses the challenge of generating high-quality, syntactically correct unit test cases that conform to standards and invoke a variety of testing APIs. It also focuses on generating test cases that are realistic, accurate, and human-readable.

The datasets used include the Defects4j projects, which provide a reliable infrastructure for generating, compiling, executing, and evaluating test cases for various popular open-source software projects.

The study presented a new approach for automated unit test generation called ATHENATEST. It was based on a BART transformer model, which has undergone both English and source code pretraining. This pretraining is crucial as it helps the model learn meaningful representations from real-world examples, which are then applied to the task of generating unit test cases.

Beyond the initial pre-training, the model was also fine-tuned. It was performed as a translation task, where the source is a focal method (the method to be tested), and the target is the corresponding test case written by a software developer. This process uses a parallel corpus of mapped test cases, aiming to learn the mapping from a focal method to a test case as a conditional probability. The training employs cross-entropy loss and the Adam optimizer, with shared vocabulary embeddings between the Encoder and Decoder. The model was utilized with a specific representation of the input called the focal context method, which is about the inclusion of additional contextual information surrounding the focal method. This context is crucial for the model to generate more accurate and relevant test cases. The method involves experimenting with different levels of focal context to determine which variant provides the best performance. The focal context includes information such as the focal method, class name, constructors, other methods, and fields.

ATHENATEST was evaluated on five Defects4j projects, generating approximately 25K passing test cases (about 16% of the overall generated tests) that covered 43.7% of the focal methods with only 30 attempts. The generated test cases were classified based on syntax correctness, compilability, execution, coverage, and correctness. The results showed that ATHENATEST could generate test cases that were syntactically correct, compilable, and executed correctly, with a significant portion achieving adequate test coverage.

Beside the evaluation on a benchmark, ATHENATEST was also compared with EvoSuite and GPT-3 in terms of code coverage and developers' preferences. The comparison revealed that ATHENATEST outperformed GPT-3 and achieved comparable coverage to EvoSuite. A survey conducted with professional developers showed a strong preference for test cases generated by ATHENATEST over those by EvoSuite, particularly in terms of readability, understandability, and testing effectiveness.

In summary, ATHENATEST represents a significant advancement in automated test case generation, with the advantage of including the ability to generate realistic, accurate, and human-readable test cases that are like those written by developers. It also leverages the power

of transformer models and benefits from both natural language and source code pretraining. However, the approach has some limitations, such as the challenge of incorporating project-level context and supporting multiple testing frameworks. Additionally, deploying large neural models like ATHENATEST in production environments poses significant engineering challenges.

While another study from Schäfer et al. [38] proposes a refined approach to prompt engineering, which is critical for improving the performance of LLMs in generating test cases. The authors introduce a set of prompt refiners—tools that enhance the quality of prompts used to guide the LLMs. These include the DocCommentIncluder, which incorporates documentation comments into prompts, and the SnippetIncluder, which adds usage snippets for functions. The paper evaluates the impact of these refiners on test generation, revealing that while some refiners significantly improve test coverage and pass rates, others show less impact, indicating the nuanced nature of LLM-based test generation.

The paper presents TESTPILOT, an implemented tool that embodies the proposed solutions, and it is used to conduct an empirical study to assess the effectiveness of different prompt configurations. TESTPILOT's performance is measured against several metrics, such as the percentage of passing tests and achieved coverage. The results from TESTPILOT are compared with baseline configurations, demonstrating that the use of prompt refiners can lead to substantial improvements in the quality of generated tests.

The TESTPILOT tool was tested on JavaScript. The document specifies that while the approach is language-agnostic, the current implementation targets JavaScript, and the results cannot be generalized to other languages. The tool was evaluated on 25 Node Package Manager (npm) packages, which indicates that it was indeed focused on the JavaScript ecosystem. The document also compares TESTPILOT's performance to Nessie, another state-of-the-art JavaScript test generator, suggesting that the choice of JavaScript was also to provide a comparative analysis with existing tools in the same programming language domain.

However, the study also uncovers that the inclusion of certain information, like function implementations, can sometimes confuse the model, leading to less effective test cases. This highlights one of the biggest challenges in AI-assisted code generation: the balance between providing sufficient context to the model without overloading it with information that may detract from its performance.

In conclusion, the document suggests that while tools like TESTPILOT and the use of LLMs in software testing represent a significant advancement, there is a delicate balance that must be struck in prompt engineering. The research indicates a promising direction for future work, which includes refining the strategies for prompt construction and exploring the integration of LLMs with other software testing methodologies to enhance the robustness and reliability of automated test generation.

Another study from Siddiq et al. [39] which is an empirical study that is also focusing on the use of LLMs for unit test generation. This research is pivotal in demonstrating the capabilities and limitations of LLMs in software engineering, particularly in automated test case generation. The study utilized several prominent LLMs, including GPT-3.5-Turbo, StarCoder, and two configurations of Codex (Codex 2K and Codex 4K). These models were chosen for their advanced capabilities in code generation and their potential applicability in generating unit tests.

The study primarily investigates the effectiveness of LLMs in generating unit tests for software. It uses two main datasets for evaluation: the HumanEval dataset, containing more than 160

common programming problems with known solutions, and the Evosuite SF110 dataset, comprising 110 mature real world open-source Java projects. These datasets are crucial for assessing the practicality and reliability of the generated unit tests.

One of the significant challenges highlighted in the document is the LLMs' ability to generate high-quality unit tests that are both correct and comprehensive, and free of test smells. The complexity lies in ensuring that these tests cover a wide range of scenarios and effectively catch potential bugs, a task that is often challenging for automated systems. Compounding this difficulty, the complexity of software projects and the limitations of current LLMs in understanding code context and requirements posed significant hurdles.

The study explores different configurations and contexts to optimize the performance of LLMs in test generation. It assesses the quality of unit tests not only based on coverage (line and branch) and correctness but also on the presence of test smells using tools like JaCoCo and TsDetect. The study's results varied across different models and datasets. For the HumanEval dataset, the LLMs achieved line coverage ranging from 67% to 87.7% and branch coverage from 69.3% to 92.8%. Codex (4K) showed the highest line and branch coverage of 87.7% and 92.8%, respectively. However, for the SF110 dataset, the performance was significantly lower, with branch and line coverage less than 2% for all models. Among them, Codex (2K) had the best line coverage at 1.9%, and GPT-3.5-Turbo had the highest branch coverage.

In summary, this document provides an in-depth analysis of the potential and limitations of using LLMs for automated unit test generation. It offers valuable insights into how these models can be optimized and highlights the need for further research to enhance their effectiveness in practical software development scenarios.

## 2.2 AI4FIX System Requirements Analysis

This section will explain how the relevant system requirements identified in D2.1 will be covered by AI4FIX.

**Table 7: System requirements coverage by initial version of AI4FIX component**

Req. ID	Req. name	Req. description	Priority	Coverage	How addressed
SR-003	Vulnerability fixing	The AI4CYBER framework shall be able to fix the vulnerabilities via the AI4FIX component found in the source code by the AI4VULN component.	M	Covered	For the vulnerabilities identified by the AI4VULN component AI4FIX will generate candidate fixes using the generic and custom models built.

Req. ID	Req. name	Req. description	Priority	Coverage	How addressed
SR-004	Robustness improvement	The AI4CYBER framework shall be able to add robustness improvements to the source code verified by tests via the AI4FIX component.	M	Covered	Generic robustness improvements will be added by AI4FIX by the learnt software evolution models. Test will be generated for the proposed fixes using the custom model to ensure their correctness.
SR-005	AI4FIX-AI4VULN interoperability	The AI4FIX component should be able to read the output of the AI4VULN component.	M	Covered	A JSON structure has been defined. See 2.4

The requirements validation will be carried out in WP7.

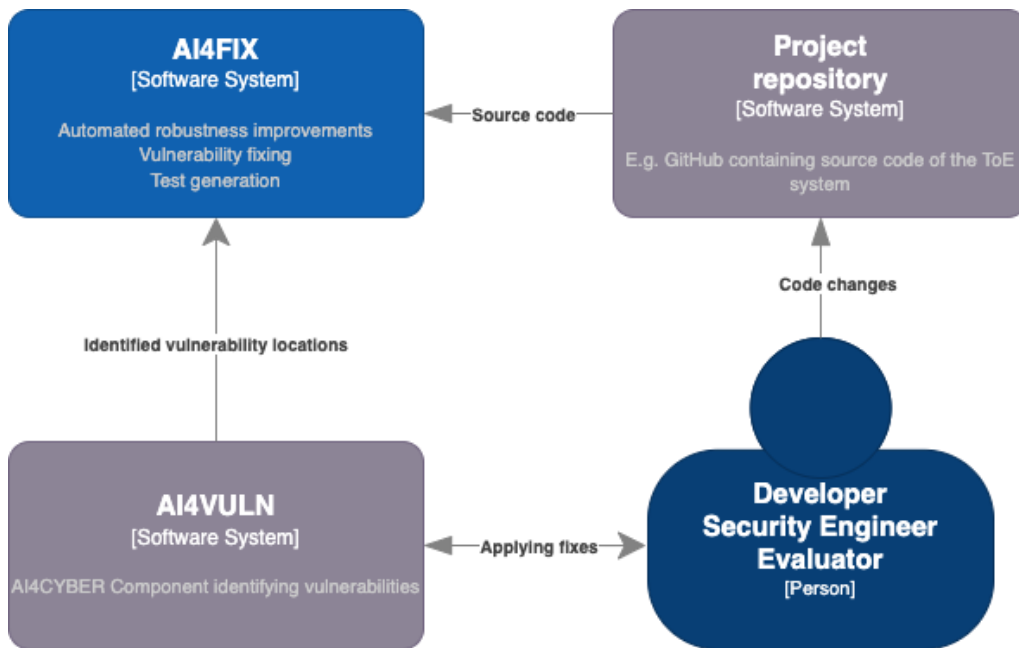
## 2.3 AI4FIX Specification

The AI4FIX tool uses natural language processing techniques harnessing the knowledge from the models (via online API or locally implemented) we build according to *software evolution*. AI4FIX has a VSCode plugin UI for interacting with the developers (for selecting accepting, rejecting, modifying generated fixes, and initiating the execution of the tool).

In the following a detailed specification of the AI4FIX component according to C4 model (<https://c4model.com/>) is provided.

### 2.3.1 System context

At the system level AI4FIX is employed by developers and/or security evaluators to correct the security-relevant bugs and improve robustness.



**Figure 1: System context of AI4FIX**

At the top left of the diagram in Figure 1, we can find AI4FIX depicted as a Software System that serves as the principal component responsible for automated robustness improvements, vulnerability fixing, and test generation. This system directly interacts with the source code to apply these improvements and tests.

AI4VULN is another Software System placed below AI4FIX, which identifies vulnerabilities within the software. It aims to provide the AI4FIX system with identified vulnerability locations, signaling precisely where the AI4FIX system should focus its fixing processes.

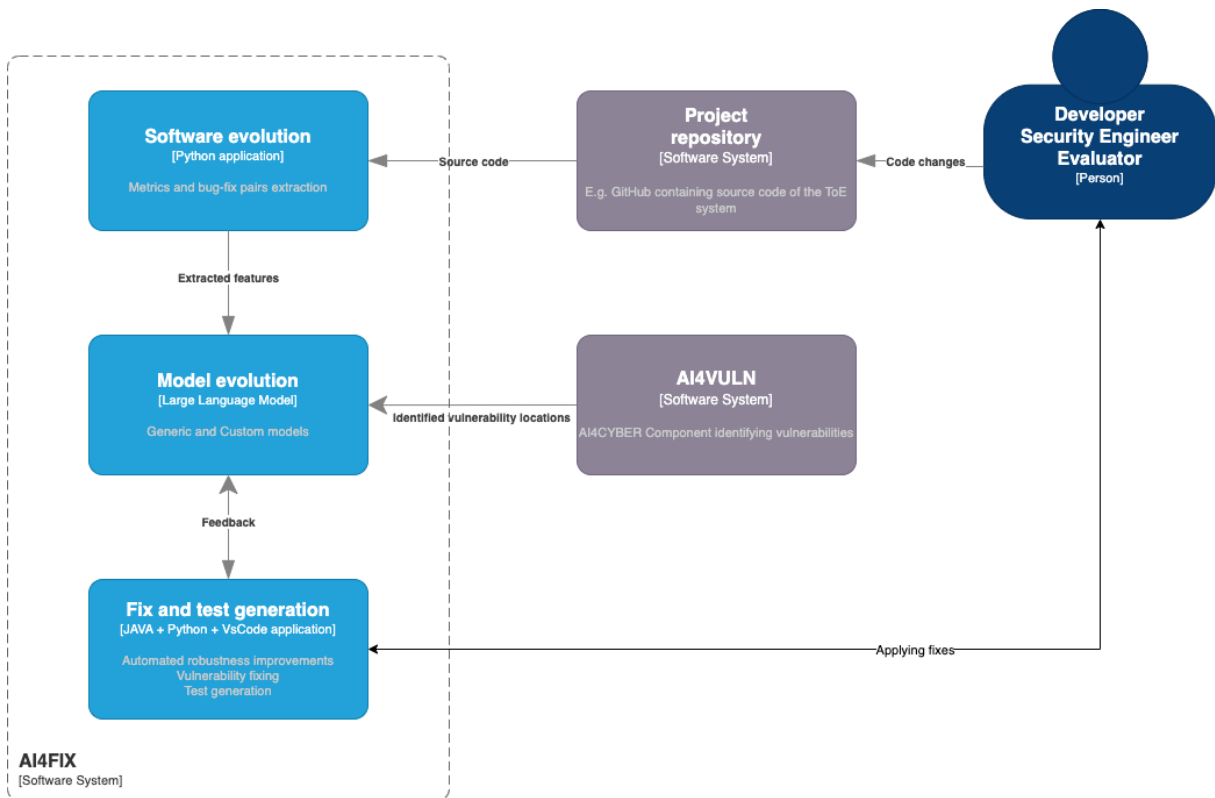
On the right side of the diagram, there is the *Project repository*, which is an external element represented as another Software System and is labelled “E.g., *GitHub containing source code of the ToE system.*” This repository is where the source code of the Target of Evaluation (ToE) resides. Code changes generated by the AI4FIX system are applied here.

At the bottom right, there is a Person icon labelled *Developer, Security Engineer, and Evaluator*, suggesting these roles are involved in the process, providing human feedback and expertise necessary to review and validate the fixes and improvements made by the AI4FIX system.

Arrows in the diagram indicate the flow of information. The arrow labelled *Code changes* shows that modifications made are fed back into the repository. Meanwhile, an arrow labelled *Applying fixes* mutually connects the *Developer/Security Engineer/Evaluator* roles to AI4VULN, which indicates that human-reviewed fixes are incorporated into the vulnerability identification process to refine its accuracy.

### 2.3.2 Container model

At the container level we distinguish the following parts of AI4FIX:



**Figure 2: Container model of AI4FIX**

The container model diagram in Figure 2 provides a more detailed view of the AI4FIX system than the previous context diagram, breaking down the system into its main components or *containers*.

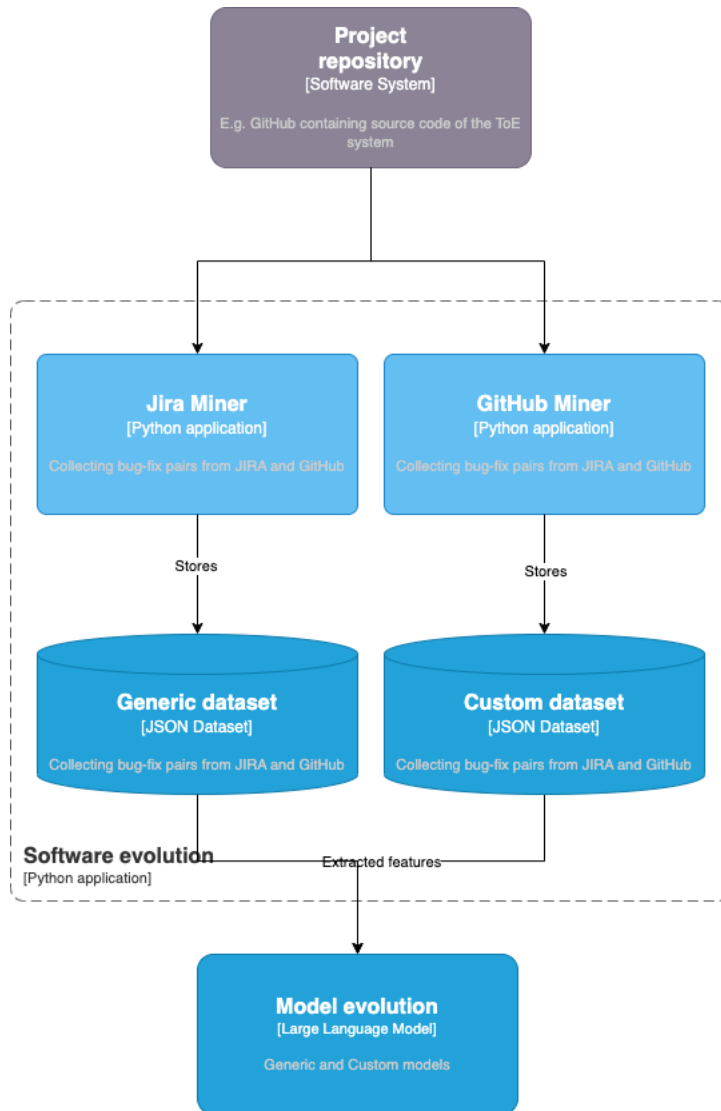
*Software evolution* is a Python application responsible for metrics and bug-fix pairs extraction from the source code.

*Model evolution* is a container characterized as an LLM which uses the extracted features from the Software evolution process. It consists of both generic and custom models, indicating that it is designed to be adaptable to different types of software projects. This container interface with *Identified vulnerability locations*, showing that it uses information about vulnerabilities detected by the AI4VULN system to evolve its models.

*Fix and test generation* is a container that seems to encompass multiple technologies (JAVA + Python + VSCode application) and is responsible for automated robustness improvements, vulnerability fixing, and test generation. This indicates a multifaceted approach to generating solutions for identified issues and verifying that the fixes are correct and effective.

### 2.3.3 Component model

The following three diagrams describe the container level of AI4FIX as an automated system designed to detect and fix software vulnerabilities, with a focus on continuous learning and improvement through machine learning models and human expertise.



**Figure 3: Component model of Software evolution**

The component model of the *Software evolution* component (Figure 3) provides an architectural overview of the parts making up this specific Python application within the AI4FIX system. This model details how the software evolution process involves mining and storing datasets pertinent to software development and maintenance.

*Jira Miner* is a Python application whose role is to collect bug-fix pairs from JIRA and GitHub as described in 3.4.5.1.

*GitHub Miner* is another Python application that parallels the function of the *Jira Miner* but focuses specifically on mining data directly from GitHub repositories. The goal is to collect bug-fix pairs from GitHub, especially those related to commit history and interactions with Static Application Security Testing (SAST) tools, like Spotbugs or Open Static Analyzer.

Both *Jira Miner* and *GitHub Miner* contribute to the construction of two distinct datasets:

*Generic dataset* is a container that stores bug-fix pairs collected from both JIRA and GitHub. JSON format used for storage as it contains a variety of bug and fix examples that can be considered universal or generic across multiple projects.



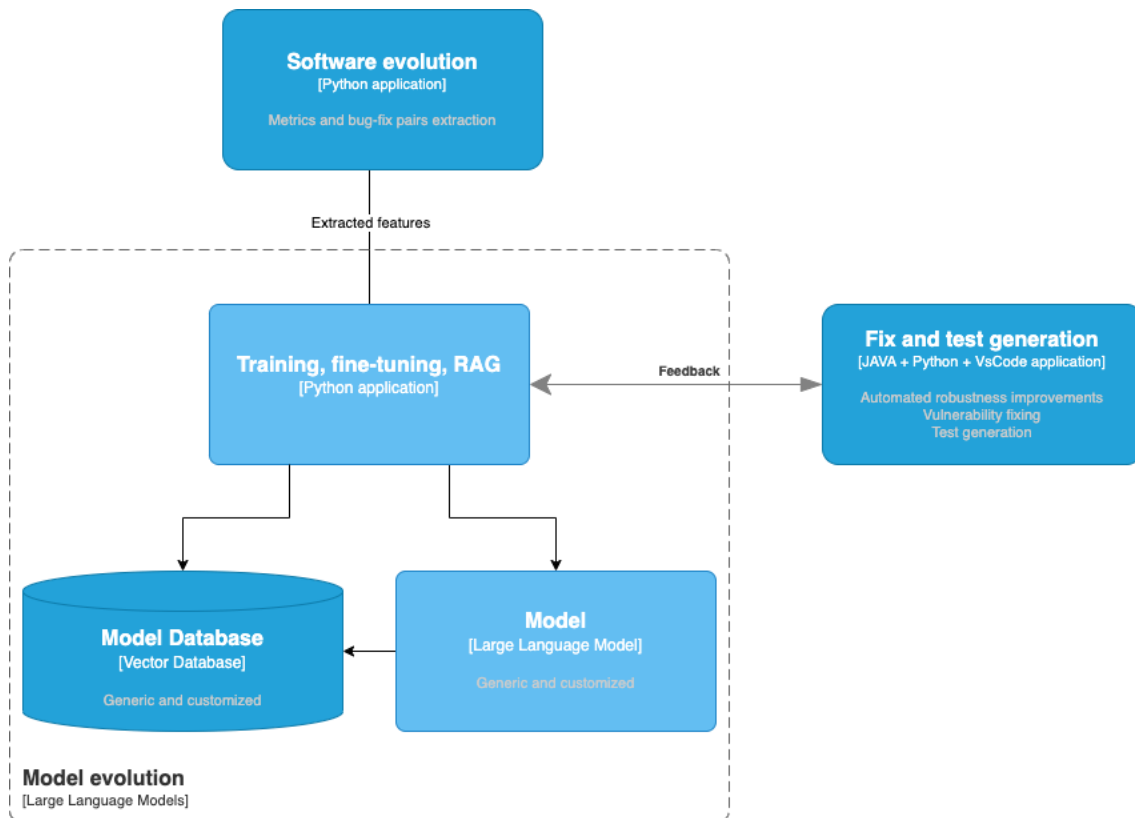
*Custom dataset* is another container like the *Generic dataset* in its purpose but is customized for specific projects. This dataset gathers bug-fix pairs but with a focus on tailoring the collected data to the needs of the ToE's specific software project.

The arrows labelled *Stores* point from the *Jira Miner* and *GitHub Miner* components towards both the *Generic dataset* and the *Custom dataset*, indicating that both miners contribute data to these datasets.

*Extracted features* indicate that the processed and useful information from the datasets is provided to other components or processes, for further analysis or as inputs for model training.

*Model evolution* shows that the extracted features from the datasets feed into a Large Language Model, which is involved in the development of both generic and custom models, as indicated by the component label. This model uses the data to learn and evolve effectively, presumably generating better solutions for fixing bugs and vulnerabilities.

Overall, the components within the *Software evolution* application are responsible for data mining, processing, and storage, with the goal of supporting the continuous improvement of software robustness and security through better-informed machine learning models.



**Figure 4: Component model of Model evolution**

The component model of the *Model evolution* (Figure 4) component details the internal structure and interactions involved in developing and refining the machine learning model within AI4FIX. This is a deeper view into the *Model evolution* container, which was previously described as an LLM, and now we can see its components and their connections.

The model revolves around four main components:

*Software evolution* sits on the periphery as an adjoining container (a Python application), which is not a component of *Model evolution* but is crucial as it supplies *Extracted features* that are inputs for the model evolution process.

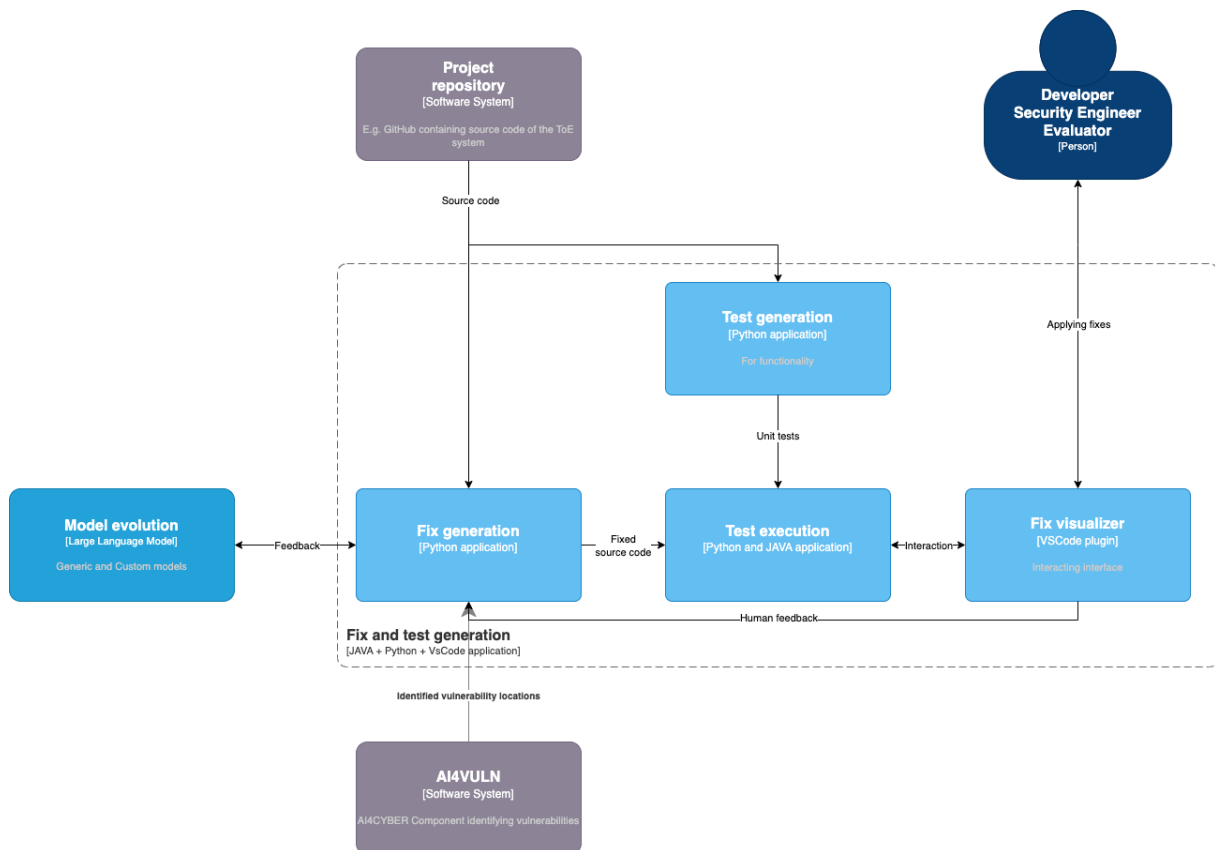


*Training, fine-tuning, RAG* is a Python application that stands as the central component responsible for the development of the machine learning model. RAG (Retrieval-Augmented Generation) is used for enhancing language model predictions by integrating retrieved information. This application processes extracted features and through training and fine-tuning, it continuously improves the model's performance.

*Model Database* is a Vector Database characterized as storing *Generic and customized* data. This database retains the information necessary for the *Training, fine-tuning, RAG* application to operate effectively, it includes encoded representations of generic and project-specific examples that the model can use for learning and reference.

*Model* itself is a Large Language Model being both 'Generic and customized,' indicating that it serves a dual purpose: it is capable of general application but can also be tailored to specific project needs.

On the right-hand side, beyond the dashed boundary of *Model evolution*, is the *Fix and test generation* container. It receives influence from the model's output and provides *Feedback* into *Training, fine-tuning, RAG*. This represents a feedback loop where the results from generated fixes and tests (from the JAVA + Python + VSCode application) help to refine the training process, leading to a machine learning model more adept at solving the types of problems it encounters.



**Figure 5: Component model of Fix and Test generation**

The *Fix and test generation* container (Figure 5), within the *AI4FIX* system, is composed of critical components that interact with both internal processes and external entities to enhance software robustness and security:

*Fix generation* is a Python application that not only generates code fixes based on model outputs but also interacts with the *Project repository*. This interaction involves pushing *Fixed source*

*code* back to the repository, which is an integral part of incorporating enhancements into the actual software project.

*Test generation* creates automated tests designed to verify software functionality. The tests produced by this component are critical as they are used in conjunction with the *Test execution* component and ultimately serve to ensure that the fixes generated do not adversely affect the software's functionality.

*Test execution*, utilizing Python and JAVA technologies, carries out the tests provided by the *Test generation* component. This execution process is crucial as it provides the necessary *Human feedback* loop for the human roles, namely Developers, Security Engineers, and Evaluators. Through their expertise, these individuals contribute to validating the fixes and the functioning of the software. Additionally, this component interacts with the *Fix visualizer* to enable an understanding of the context and impact of the fixes on the software.

*Fix visualizer* is a VSCode plugin that provides a visual representation of the generated fixes. It serves a dual purpose: firstly, it helps developers and other stakeholders to visually comprehend the nature of the fixes within their code environment; secondly, it can interact with the results from the *Test execution* to visualize the outcomes, thereby supporting assessments and potential iterations of fix implementations.

### 2.3.4 Hardware requirements

The estimated hardware requirements for running an offline model on CXB premises for Use case 2 on baking applications.

**Table 8: Hardware requirements for the tools**

Item	Property
CPU cores	16
RAM	32 GB
GPU RAM	64 GB
Storage	1 TB SSD
Operating System	Ubuntu Server Latest
Other	CUDA compatible GPU

In case the estimated hardware requirements cannot be met, the performance may degrade.

### 2.3.5 Software evolution

The following section defines our nomenclature for software evolution and related concepts.

- **Software evolution** – concept: A collection of techniques which aim to extract knowledge for fixing vulnerabilities from the development history of the selected project and other relevant projects.
- **Commit classification:** We analyze historical commits of the repository in order to obtain metrics that we can learn to classify the commits based on their content (e.g., JIRA class): bug fix, new feature, test, revert, etc. and security relevance. The commit classifier can also consider the relevance of the observed code segment to security requirements.
- **SW evolution generic model:** We want to learn – by examining a wide range of commits from various projects – how certain potential vulnerabilities detectable by SAST and

DAST tools (e.g., Spotbugs) are fixed. We collect many bug-fix pairs for security and robustness improvement to train a model which can generate the fixes. We would like to gather all the hidden fixes by running SAST tools on each commit and observing the differences.

- **SW evolution customized model:** We want to learn – by examining the previous commits from the analyzed project – how certain potential vulnerabilities detectable by SAST and DAST tools (e.g., Spotbugs) are fixed in the selected project and customize our generic model with parameters characteristic for the selected project. We also take coding style and naming conventions into account to customize our generated fixes for the selected repository.

## 2.3.6 Code synthesis using AI and LLM

### 2.3.6.1 Evaluating the code synthesis capabilities of LLMs

Code synthesis is currently a hot topic as LLMs are getting better at not only understanding natural language but transforming it to source code. The main scope is creating source code that passes all the functional conditions based on natural language descriptions. Although it is important, factors like software security and source code maintainability are not to be forgotten either.

The related works in this area mostly evaluate one LLM but even ones that compare LLMs usually lack a proper comparison, a proper evaluation. Dominik et al. [40] did a comparative study on Copilot and Genetic program synthesis. Their main goal was to compare two methodologies for program synthesis and did not measure the quality of the synthesized code. In the work of Madi [41] the readability is measured by static code analysis, but no further qualities are checked and is heavily based on human annotators, including 21 people.

A proper evaluation regarding LLMs should focus on 4 elements.

- **Selecting the right prompt** is crucial as the input of LLMs is the prompt. If this prompt must be too detailed, the developer does not gain anything from using an LLM. Creating too abstract descriptions could result in the LLM generating results that fit the abstract description, but the developer needs a more specific solution. Selecting an abstract prompt enough with the best descriptions is called prompt-engineering. Various models might be more dependent on prompting than others.
- **Checking functional validity** is the goal of source code synthesis. Functional validity could be tested by predefined testcases or even human testing, although in an automatic way testcases are preferred.
- **Checking technical quality** is not to be forgotten as this includes the security and quality of source code. It might be checked by source code analyzers or human reviews. When applying a source analyzer during the evaluation, the same settings should be used as in an ongoing project to achieve the best fitting results.
- **Human evaluation** requires tremendous resources although, if possible, it reflects how likely the developers are to use, accept the LLM synthesized source code. Human reviewers should not be aware of the functional and technical evaluation results and should not be aware of which model's code they are reviewing.

Besides these elements additional factors could be investigated. An example is the performance of the source code or Non-functional validity. The complexity of runtime or space could have a great impact on special software.

We evaluated Copilot and ChatGPT 3.5 on the previously mentioned criteria. The descriptions of natural language came from a benchmark created for code synthetization evaluation, PSB2 [42].

We checked functional validity using 10.000 test cases from the benchmark's test suit. The results are in the Table 9 for every task available in the benchmark. The results show what percentage of the 10.000 test cases passed for every task with the LLM generated source code.

**Table 9: Functional validity comparison**

Task name	ChatGPT	Copilot	Diff
basement	100%	100%	0%
bouncing balls	5%	20%	-15%
bowling	5%	0%	5%
camel_case	100%	100%	0%
coin_sums	100%	0%	100%
cut_vector	86%	0%	86%
dice_game	54%	100%	-46%
find_pair	0%	100%	-100%
fizz_buzz	100%	100%	0%
fuel_cost	100%	100%	0%
gcd	100%	100%	0%
indices_of_substring	23%	100%	-77%
leaders	18%	18%	0%
luhn	0%	0%	0%
mastermind	100%	36%	64%
middle_character	100%	100%	0%
paired_digits	100%	81%	19%
shopping_list	100%	0%	100%
snow_day	10%	5%	5%
solve_boolean	75%	100%	-25%
spin_words	100%	100%	0%
square_digits	100%	100%	0%
substitution_cipher	100%	100%	0%
twitter	100%	100%	0%
vector_distance	98%	0%	98%

The technical qualities are checked by SonarQube Scanner. We investigated both code smells and vulnerabilities. We also investigated various code metrics. ChatGPT performed slightly better on these evaluations. Finally, we asked 5 developers to review source code generated by the two models. The developers had to score from  $-2$  to  $+2$  without 0 on the following properties: First Impression, Usability, Readability, Modifiability, Acceptance of the code. The developers also had to decide which code is better without knowing which code is generated by ChatGPT or Copilot. The results are presented in Table 10.

**Table 10: Result of developers' evaluation**

Task name	First Impression		Usability		Readability		Modifiability		Acceptance		G / C
	G	C	G	C	G	C	G	C	G	C	
basement	10	10	10	10	10	10	10	10	10	10	-1
bouncing_balls	9	2	0	3	8	4	9	7	2	3	-3
bowling	2	6	7	5	6	6	4	3	5	4	0
camel_case	6	4	-1	5	8	-2	8	2	-1	4	-1
coin_sums	9	7	5	7	9	6	9	9	10	9	-5
cut_vector	9	-3	8	0	9	4	8	4	9	1	-6
dice_game	8	5	6	9	9	7	9	8	7	9	-3
find_pair	9	0	7	4	8	4	8	5	8	2	-5
fizz_buzz	7	6	10	10	9	8	7	7	9	9	2
fuel_cost	8	6	9	10	10	5	9	9	10	9	-3
gcd	10	-1	10	7	9	4	9	7	10	3	-9
indices_of_substring	9	2	7	7	9	3	9	5	9	6	-2
leaders	9	-6	8	6	10	-1	8	0	9	4	-9
luhn	2	-5	1	-4	8	4	6	2	-1	-7	-5
mastermind	5	3	8	5	8	4	6	3	8	3	-3
middle_character	8	8	9	9	7	10	7	6	8	10	3
paired_digits	9	4	10	6	10	5	9	7	10	4	-6
shopping_list	10	1	10	2	9	-2	9	8	10	3	-8
snow_day	8	4	-8	-1	7	2	3	7	-9	0	3
solve_boolean	3	4	0	4	5	8	2	1	-2	4	2
spin_words	10	5	10	6	10	7	9	7	10	7	-1
square_digits	8	8	9	7	7	6	7	7	9	8	-7
substitution_cipher	4	9	9	4	6	8	2	6	8	5	-1
twitter	10	8	10	8	10	7	10	9	10	8	-4
vector_distance	6	2	6	7	9	7	7	6	7	8	-1

In Table 10, the maximum value is 10, as 5 developers were included and could rate it on max 2. The minimum is  $-10$  similarly. G/C column shows that the developer preferred ChatGPT (G) or Copilot (C). It was measured on  $-2$  to  $+2$  scale without zero too, therefore the close the values to  $-10$  ChatGPT is preferred while values closer to  $+10$  mean Copilot is preferred. Values close to zero show that developer opinions were different.

The results showed that developers did not consider the model synthesized code perfect or out-of-the-box usable, but they considered the code acceptable. Developers slightly preferred ChatGPT generated source code.

### 2.3.6.2 Robustness improvement

Robustness improvements in AI4FIX are designed to be small and automatically applicable fixes which will not alter functionality. Learnt from the datasets of bug-fix pairs in software evolution generic model we can detect potential improvement opportunities and generate candidate fixes using the customized model to be tailored to the current repository. This feature is a work in progress at the time of writing as the models are not yet trained. Based on our initial tests LLMs can improve code snippets but require tight control over functional degradation, therefore test generation needs to be developed first.

### 2.3.6.3 Test generation

We have researched some tools already available for test generation on the market.

This article [43] discusses the significance of AI-driven tools in test automation, highlighting their role in enhancing efficiency, cost-effectiveness, and reliability. It covers the evolution of test automation, the impact of AI on reducing human errors, and the transformation of QA methodologies. The article also emphasizes the growing complexity of applications and the increasing reliance on AI for maintaining quality standards. It presents seven popular AI-powered test automation tools, each with unique features and capabilities, catering to different testing needs in web, mobile, desktop, and API environments. The article concludes by stressing the importance of defining clear goals for test automation to achieve desired outcomes and ROI.

Here are the seven AI-powered test automation tools mentioned in the article, with brief descriptions:

- **Testsigma**: An AI testing tool for web, mobile, desktop applications, and APIs, featuring auto-healing functionalities and natural language processing for test creation.
- **TestCraft**: Built on Selenium, this tool is designed for both manual and automated testing, particularly for web-based software.
- **ACCELQ**: A codeless tool enabling multi-channel test automation (web, mobile, desktop, API), focusing on business process-focused automation.
- **Applitools**: Offers AI-powered visual UI testing and monitoring, integrating with existing tests and providing cross-browser, cross-device testing features.
- **Testim**: An AI-powered platform for custom web applications, enabling rapid creation of UI and end-to-end tests with AI stabilization.
- **Sauce Labs**: Provides comprehensive testing across devices, browsers, and operating systems, with functionalities for functional testing and parallel testing.
- **Functionize**: A cloud-based tool using machine learning and AI for easy test case creation using natural language processing, suitable for desktop and mobile browsers.

We conducted testing of Large Language Model-based test generation. The OWASP Benchmark Project [44] is a Java-based suite designed to assess the effectiveness of software vulnerability detection tools. It is an open-source web application with thousands of test cases, each corresponding to specific CWEs, suitable for testing SAST, DAST, and IAST tools. The benchmark ensures that all vulnerabilities are exploitable, providing a fair and accurate evaluation of these tools' accuracy, coverage, and speed.

The benchmark comprises both positive and negative test cases. Each case, including the negative ones, encompasses functionalities that could potentially be vulnerable, such as command execution or SQL queries. In negative tests, these potentially risky functionalities are designed so that they cannot receive user input, thus rendering them non-exploitable. Conversely, in positive tests, user input can interact with these vulnerable functionalities, making the vulnerabilities exploitable. This design allows for a comprehensive assessment of how security tools handle both secure and vulnerable code scenarios.

The typical test case consists of the following parts:

- GET or POST request handler, which extracts the parameters from the request and passes it to the doSomething function. The result of the doSomething function is passed to the vulnerable code.
- The doSomething function performs some simple or complex operations with the input parameters, resulting that the input parameter may be the part of the output.

So, the vulnerability detection tool should detect the vulnerable functionality correctly and should be able to follow the functionality of the doSomething function.

In the following we show an example test case from the benchmark:

```
/**
 * OWASP Benchmark Project v1.2
 * ...
 */
package org.owasp.benchmark.testcode;
...

@WebServlet(value = "/cmdi-01/BenchmarkTest01609")
public class BenchmarkTest01609 extends HttpServlet {

    private static final long serialVersionUID = 1L;

    ...

    @Override
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");

        String[] values = request.getParameterValues("BenchmarkTest01609");
        String param;
        if (values != null && values.length > 0) param = values[0];
        else param = "";

        String bar = new Test().doSomething(request, param);

        String cmd = "";
        String osName = System.getProperty("os.name");
        if (osName.indexOf("Windows") != -1) {
            cmd = org.owasp.benchmark.helpers.Utils.getOSCommandString("echo");
        }
    }
}
```



```

String[] argsEnv = {"Foo=bar"};
Runtime r = Runtime.getRuntime();

try {
    Process p =
        r.exec(cmd + bar, argsEnv, new
java.io.File(System.getProperty("user.dir")));
    org.owasp.benchmark.helpers.Utills.printOSCommandResults(p, response);
} catch (IOException e) {
    System.out.println("Problem executing cmd - TestCase");
    response.getWriter()

.println(org.owasp.esapi.ESAPI.encoder().encodeForHTML(e.getMessage()));
    return;
}
} // end doPost

private class Test {

    public String doSomething(HttpServletRequest request, String param)
        throws ServletException, IOException {

        String bar = "";
        if (param != null) {
            java.util.List<String> valuesList = new
java.util.ArrayList<String>();
            valuesList.add("safe");
            valuesList.add(param);
            valuesList.add("moresafe");

            valuesList.remove(0); // remove the 1st safe value

            bar = valuesList.get(0); // get the param value
        }

        return bar;
    }
} // end innerclass Test
} // end DataflowThruInnerClass

```

In the above example the `doSomething` function returns the received `param`, which contains the user data. This is because the code inside the `if` block adds `param` to a list, removes the first element of the list (which was "safe"), and then assigns `bar` the value of the new first element in the list, which is `param`. Since the command execution directly adds the return value of the `doSomething` to the executed command name, the command injection is exploitable.

We chose this benchmark for evaluation because all its vulnerable scenarios are susceptible to exploitation. This characteristic made it ideal for our analysis. Specifically, we focused on the command injection test cases to assess the test generation capabilities of the LLMs, as the exploitation in these cases is distinctly observable.



The helper function used by all the test cases (`printOSCommandResults`) printed the result of the command execution to the standard out. It made the verification of the exploitation very simple, since we had to print out with the command injection, which can be checked easily, but will be printed only in case of a successful exploitation. From the lot of possible solutions, we chose to store a simple text file in the working directory of the server, which contained “abcd1234”. The command injection string can be some form of printing out the content of the file, for example ‘; cat hack.txt”.

Throughout the evaluation of test generation, we deliberately avoided specifying the exact means of vulnerability exploitation. Our approach focused on issuing precise instructions on the desired results of the exploitation, thereby facilitating its verification. The test was initiated using the following prompt:

```
Your task is to create Mockito requests to properly call the vulnerable function to test the command injection. You have to execute the "cat hack.txt" function as an injected command. At the end of the test check that the response contains the "abcd1234" string, which indicates that the command is executed.
Do not write an explanation, just give the source code. Use the following example structure to generate the unit test:
<example test structure>
```

We used the following example test case:

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.assertTrue;

import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;
import org.owasp.benchmark.testcode.BenchmarkTest01517;
import javax.servlet.http.*;
import java.io.StringWriter;
import java.io.PrintWriter;

public class BenchmarkTest01517Test {

    @InjectMocks
    BenchmarkTest01517 servlet;

    @Mock
    HttpServletRequest request;

    @Mock
    HttpServletResponse response;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }
}
```



```
@Test
public void testDoPost() throws Exception {
    String param = "; cat hack.txt"; // simulate command injection attempt

    StringWriter stringWriter = new StringWriter();
    PrintWriter writer = new PrintWriter(stringWriter);
    when(response.getWriter()).thenReturn(writer);

    servlet.doPost(request, response);

    writer.flush();

    String responseString = stringWriter.toString();
    assertTrue(responseString.contains("abcd1234")); // Check if the response
contains the expected string
}
}
```

Throughout the evaluation of test generation, we encountered a few manageable issues that can be addressed through postprocessing. Firstly, despite our requirement for code-only submissions, there were instances where a brief explanation preceded the actual code. Secondly, while most code submissions were correctly enclosed within three apostrophes, occasionally, various types of apostrophes were utilized instead.

For the AI to generate a valid test case targeting a genuinely vulnerable piece of code, it needed to overcome two challenges. The first challenge was to create an executable test, and the second was to generate user input that could potentially exploit the vulnerability. Considering that exploiting the vulnerability lay outside our evaluation scope — as it would require extensive knowledge beyond the vulnerable Java file, like the source of utility functions in the benchmark — we aimed to simplify the exploitation process. We provided a default user input ('; cat hack.txt'), which serves as an effective exploit string in most scenarios. In cases where this default input failed to produce a successful test, we automatically attempted the following alternative inputs as well.:

```
"cat hack.txt"
"FOO=cat hack.txt"
```

The OWASP Java benchmark contained 92 vulnerable cases and 103 non vulnerable cases for command injection. We sent all these tests to various LLMs without specifying whether it is vulnerable or not, using the prompt and template we described above. Because the successful test generation (vulnerability exploitation) can be considered as vulnerability detection, we calculated the True Positive Rate (TPR) for each model. The TPR value is defined by the OWASP benchmark True Positive / (True Positive + False Negative), which is also called as recall value<sup>4</sup>.

---

<sup>4</sup> [https://en.wikipedia.org/wiki/Precision\\_and\\_recall](https://en.wikipedia.org/wiki/Precision_and_recall)

Since we accepted only the exploited vulnerability as a successful test, the number of false positive cases (when a non-vulnerable test is marked as vulnerable) was zero, so the False Positive Rate was also 0.

**Table 11: Test generation results**

Model	TP	TN	FP	FN	TPR
gpt 3.5-turbo	47	103	0	45	0.51
<b>gpt 4</b>	<b>65</b>	<b>103</b>	<b>0</b>	<b>27</b>	<b>0.706</b>
gpt-4-1106-preview (gpt4-turbo)	46	103	0	46	0.5

## 2.4 AI4FIX Communication Interfaces

This subsection describes the JSON interoperability interface between AI4FIX and AI4VULN and sharing the information with AI4COLLAB (using Kafka).

Using this JSON structure, AI4VULN can share its output (found potential vulnerabilities) with AI4FIX to generate fixes and relevant tests.

An example of the JSON format is the following:

```
[
  {
    "id": "00001",
    "name": "EI_EXPOSE_REP2",
    "explanation": "...",
    "tags": "CWE-XXXX",
    "items": [
      {
        "patches": [
          {
            "path": "patch_1_EI_EXPOSE_REP2_24_24_34_51.diff",
            "score": 9.962368049953128,
            "explanation": "Repair with clone"
          },
          {
            "path": "patch_2_EI_EXPOSE_REP2_24_24_34_51.diff",
            "score": 9.920913240350998,
            "explanation": "Repair with Arrays.copyOf"
          }
        ],
        "textRange": {
          "endLine": 24,
          "endColumn": 50,
          "startColumn": 33,
          "startLine": 24
        }
      },
      {
        "patches": [
          {
```



```
    "path": "patch_3_EI_EXPOSE_REP2_29_29_36_55.diff",
    "score": 9.962787573897263,
    "explanation": "Repair with clone"
  },
  {
    "path": "patch_4_EI_EXPOSE_REP2_29_29_36_55.diff",
    "score": 9.918276408046792,
    "explanation": "Repair with Arrays.copyOf"
  }
],
"textRange": {
  "endLine": 29,
  "endColumn": 54,
  "startColumn": 35,
  "startLine": 29
}
}
]
}
```

## 2.5 AI4FIX Installation and User Guide

This subsection is dedicated to guiding users through the process of deploying the AI4FIX plugin for Visual Studio Code (VSCode).

The source code for the AI4FIX VSCode plugin is publicly available and can be accessed at the following URL:

<https://github.com/ai4cyber-slab/ai4fix/tree/main/vscode-plugin>

### 2.5.1 Installing dependencies

To set up the required environment for our project, it is essential to install all dependencies. This can be efficiently achieved by using the Node Package Manager (npm). Execute the following command in your terminal or command prompt:

```
npm install
```

This command will automatically download and install all the dependencies listed in the 'package.json' file of the project, ensuring that all necessary modules and libraries are available for the application to run smoothly.

### 2.5.2 Packaging, publishing

For those looking to test an extension on a local installation of Visual Studio Code (VSCode), an essential step is to utilize the Visual Studio Code Extension (vsce) package manager. This tool is pivotal for packaging and publishing VSCode extensions. To install the vsce package manager globally on your system, use the following command in your terminal or command prompt:

```
npm install -g vsce
```

This command will install the vsce package manager globally (-g flag), making it accessible from any directory on your system. This installation allows you to easily package your VSCode

extension, enabling local testing and preparation for eventual publishing to the Visual Studio Code Marketplace.

### 2.5.3 Generate .vsix file:

The .vsix file format plays a crucial role in distributing Visual Studio Code (VSCode) plugins. It is particularly useful for sharing your plugin privately with others who may not have access to the Visual Studio Code Marketplace or for testing purposes.

Steps to Create a .vsix File

- Navigate to the Extension's Root Folder: Start by opening your terminal or command prompt and navigating to the root folder of your VSCode plugin. You can do this using the `cd` (change directory) command. For example:

```
$ cd path/to/vscode-plugin
```

- Package the Plugin: Once in the root folder of the extension, run the `vsce package` command. This command compiles your extension into a .vsix file:

```
$ vsce package  
# aifix4seccode-vscode-x.y.z.vsix generated
```

Upon successful execution, a file named `aifix4seccode-vscode-x.y.z.vsix` will be generated, where `x.y.z` denotes the version of the plugin.

When releasing a new version of your plugin, it is important to update the version number in the `package.json` file at the top of your project. The version numbering follows the semantic versioning pattern:

- Major Version (x): Increment this number for major changes that may not be backward compatible.
- Minor Version (y): Update this for minor changes that introduce new features but are backward compatible.
- Patch Version (z): This is for small, backward-compatible bug fixes.

To install the .vsix file into Visual Studio Code, use the following command in your terminal or command prompt, replacing `x.y.z` with the actual version number of your plugin:

```
code --install-extension aifix4seccode-vscode-x.y.z
```

### 2.5.4 Example of usage on a demo project

Before using the AI4FIX VSCode plugin on a project, it is crucial to configure the extension settings properly. This ensures that the plugin interacts correctly with your development environment and tools. Key configuration parameters include:

- **Executable Parameters:** Specifies the analyzer tool's executable parameters. (e.g.: `java -jar Main.jar`)
- **Executable Path:** Specifies the analyzer tool's folder where the executable (`.jar`, `.exe`, ...) is located.
- **Use Diff Mode:** Change the mode of showing patches in the editor.

Diff mode options:

- **view Diffs** (default): Choosing a fix will show a side-to-side diff view of the original content and the content that the fix would give.
- **view Patch files**: Choosing a fix will show the patch file's content of the fix.

### 2.5.5 Example project

To get a hands-on feel for how AI4FIX works, we recommend using the test-project.zip. This demo project is designed to help you familiarize yourself with the plugin's functionality, such as analyzing code, identifying issues, and applying patches.

### 2.5.6 Open-source repository

The code of AI4FIX will be released into this repository:

<https://github.com/ai4cyber-slab/ai4fix>

## 2.6 AI4FIX Unit Testing

This subsection provides the plans for the unit tests of AI4FIX, which will be executed on the final version of the tool. The results will be presented in D3.3.

**Table 12: System test case AI4FIX\_001**

<b>Test Case ID</b>	AI4FIX_001	<b>Component</b>	AI4FIX
<b>Req ID</b>	SR-003	<b>Priority</b>	Medium
<b>Description</b>	We measure the accuracy of generated patches: TP, FP, TN, FN.		
<b>Tested by</b>	SLAB		
<b>Pre-condition(s)</b>	Dataset for testing: including false positives, negatives, developers		
<b>Test steps</b>			
<b>1</b>	Run AI4FIX with multiple models on the test dataset		
<b>2</b>	Evaluate the results with human assessor		
<b>3</b>	Measure the number of correctly/incorrectly generated patches.		
<b>Input data</b>	Dataset for testing: including false positives, negatives.		
<b>Result</b>	<p><b>Step 1 result</b> The evidence will be the generated patches.</p> <p><b>Step 2 result</b> Human assessment on generated patches (accept/decline + reasoning)</p> <p><b>Step 3 result</b> Statistics from the experiment determine the most successful model.</p>		
<b>Test Case Result</b>	Achieved /Not achieved		

**Table 13: System test case AI4FIX\_002**

<b>Test Case ID</b>	AI4FIX_002	<b>Component</b>	AI4FIX
<b>Req ID</b>	SR-003, SR-004	<b>Priority</b>	High
<b>Description</b>	Human assessment on generated patches (accept/decline + reasoning)		
<b>Tested by</b>	SLAB		
<b>Pre-condition(s)</b>	AI4FIX_001 test		
<b>Test steps</b>			
<b>1</b>	Run AI4FIX_001 test		
<b>2</b>	Evaluate with developers Step 2 of AI4FIX_001 test filtered to true positive results		
<b>3</b>	Evaluate results, create statistics		
<b>Input data</b>	Dataset for testing: including true positives of generated patches. Results expected to be around 80%		
<b>Result</b>	Can be organized to be part of AI4FIX_001 testing activities		
<b>Test Case Result</b>	Achieved /Not achieved		

**Table 14: System test case AI4FIX\_003**

<b>Test Case ID</b>	AI4FIX_003	<b>Component</b>	AI4FIX
<b>Req ID</b>	SR-003, SR-004	<b>Priority</b>	Low
<b>Description</b>	Measuring patch and test generation time		
<b>Tested by</b>	SLAB		
<b>Pre-condition(s)</b>	Dataset for testing		
<b>Test steps</b>			
<b>1</b>	Run AI4FIX on the test dataset		
<b>2</b>	Measure the time required for generating patches and tests for multiple models		
<b>3</b>	Evaluate results		
<b>Input data</b>	Dataset for testing filtered for true positives from AI4FIX_001 tests		
<b>Result</b>	Can be organized to be part of AI4FIX_001 testing activities. The result is expected to be in the realms of a few minutes for 10k LOC.		





<b>Test Case Result</b>	Achieved /Not achieved
-------------------------	------------------------

**Table 15: System test case AI4FIX\_004**

<b>Test Case ID</b>	AI4FIX_004	<b>Component</b>	AI4FIX
<b>Req ID</b>	SR-004	<b>Priority</b>	Low
<b>Description</b>	Compare test coverage before and after running AI4FIX		
<b>Tested by</b>	SLAB		
<b>Pre-condition(s)</b>	Dataset for testing		
<b>Test steps</b>			
<b>1</b>	Run AI4FIX on the test dataset		
<b>2</b>	Run coverage analysis before patching		
<b>3</b>	Run coverage analysis after patching		
<b>4</b>	Evaluate results		
<b>Input data</b>	Dataset for testing: including false positives		
<b>Result</b>	The amount of successfully generated tests is expected to be around 80% of the generated patches (true positives).		
<b>Test Case Result</b>	Achieved /Not achieved		

**Table 16: Integration test case AI4FIX\_005**

<b>Test Case ID</b>	AI4FIX_005	<b>Component</b>	AI4FIX
<b>Req ID</b>	SR-003	<b>Priority</b>	Medium
<b>Description</b>	Integration of AI4FIX with AI4VULN.		
<b>Tested by</b>	FEA, SLAB		
<b>Pre-condition(s)</b>	Completing the integration between AI4FIX and AI4VULN.		
<b>Test steps</b>			
<b>1</b>	We run AI4VULN on its regression tests and on open-source software		
<b>2</b>	We evaluate how AI4FIX uses the output of AI4VULN		
<b>3</b>	We validate the generated path of AI4FIX		
<b>Input data</b>	Regression tests and open-source software		
<b>Result</b>	We can validate that the integration is correct		

---

<b>Test Case Result</b>	Achieved /Not achieved
-------------------------	------------------------

### 3 AI4VULN Architectural Design & Implementation – Initial Version

The AI4VULN system is based on a *symbolic execution engine* for the Java language. The input for the Executor is a Java project in which we detect runtime errors, such as `NullPointerException`, `ClassCastException`, and so on. The output of the symbolic executor includes a list of these exceptions and the complete execution path (stack trace) leading to each exception.

In the following sections, we provide an introduction on symbolic execution, which we continue with the discussion of the work done so far.

#### 3.1 AI4VULN – Symbolic execution-based vulnerability detection

##### 3.1.1 State of the Art Analysis

The idea of symbolic execution was introduced in the 1970s as an elegant and powerful method for software proving, validation, and test generation. In 1976, King introduced the fundamentals of symbolic execution along with the presentation of the EFFIGY system [45]. EFFIGY is one of the first symbolic executor engines. It is written for PL/I programs and only handles integer variables symbolically.

Already at this time, a wide variety of symbolic executor engines were being developed [46], [47], even though the computing power was not yet ready for the challenges, such as the explosion of execution branches or the simulation of the environment of the program.

The evolution of symbolic executors has been defined by the steps taken to overcome these challenges. Various hybrid solutions, machine learning-based methods have been developed. One example is the mixture of concrete and symbolic execution which is called concolic execution or dynamic symbolic execution. The DART (Directed Automate Random Testing) system introduced by Godefroid et al. [48] was the first to suggest this approach. It performs a directed search in the symbolic execution tree through the iterated executions of the investigated software. The process starts with random input values, then during each execution information is collected about the symbolic variables to produce input data that would force the next execution on a new path. CUTE and jCUTE are also concolic testing systems for C and Java [49], [50]. Like the DART system, they collect information about symbolic constraints to produce input data, but they are extended with the capability of handling multithreading. If the constraint solver fails to satisfy a complex expression, the constraints are simplified by replacing some symbolic variables with concrete values. jCUTE can generate JUnit tests for sequential software.

Another technique that is often used is some kind of heuristic for choosing between execution paths. Such techniques may aim to maximize coverage or to find the branches most likely to fail at runtime the fastest. Such techniques can be found in modern, widely used symbolic executors, such as KLEE [51], Java PathFinder and JDART [52].

KLEE is based on the LLVM assembly language and functions as a virtual machine. It powerfully handles environmentally intensive programs, for example by setting up a symbolic filesystem or by simulating faulty system calls. Because it is such an effective tool, KLEE is the basis for countless studies on symbolic implementation. For example, there is a novel regression learning-based search strategy called LEARCH [53] which was implemented on KLEE. LEARCH effectively explores and selects promising states for symbolic execution, thus

addressing the path explosion problem. It uses an iterative learning process using the data extracted from the symbolic execution of the program and generates high-quality, diverse tests. Like KLEE, the Java PathFinder (JPF) [54] tool also behaves like a virtual machine and executes the Java bytecode in a special way. JPF is developed at the NASA Ames Research Center for verifying and checking NASA projects. It has an extension, called Symbolic PathFinder (SPF) [55], for performing symbolic execution. SPF supports concolic execution and can be customized with several constraint solvers. One of them is the CORAL solver [56], which handles complex mathematical functions, making it effective in scientific domains. Two relatively recent surveys on the subject of Symbolic Execution are the work of Cadar and Sen [51] which was presented in 2013, and the detailed paper of Baldoni et al. from 2016 [57]. These provide a more detailed overview of the current state of symbolic execution.

### **3.1.1.1 Exception handling**

Since our goal in symbolic execution is to detect errors at runtime, it is important to properly represent Java exception handling during execution. Representing exception handling in CFG is not a trivial task. Several articles deal with it.

According to Amighi et al., the stack-based nature of the Java virtual machine (JVM) makes it difficult to determine explicitly thrown exceptions, making it hard to decide which handler is being controlled (if any). Implicit exception throwing makes it even harder to determine these [58].

In their work, they have constructed an algorithm that generates a correct control flow graph (CFG) from Java bytecode (JBC), which has been mathematically proven. This means that the CFG they constructed has all the properties of the original source code.

As a starting point, they used the stackless intermediate representation language BIR (Bytecode Intermediate Representation), a transformation of JBC, invented by Demange et al. [59].

The BIR model places assertions in front of all instructions that can throw implicit exceptions. This makes it easier to take implicit exceptions into account.

Unfortunately, their implementation was not available on the internet, so we could not compare it with our flow graphs.

Phan et al. presented an end-to-end model for solving software defect prediction [60]. They generate the CFG of code from the compiled assembly instructions. They apply multi-view, multi-layer directed graph-based convolutional neural networks (DGCNNs) to learn semantic features. They tested their software on four datasets from CodeChef, a programming contest site. They noticed that using the CFG instead of Abstract Syntax Tree (AST) of a program as input for the DGCNNs increases accuracy. They claim their solution can scale well with larger projects. However, they did not test it on more significant projects written in C.

## **3.1.2 Background**

### **3.1.2.1 Symbolic Execution**

During regular execution, the variables of the program have concrete values, meaning that the program follows a specific executional path determined by these values. The basic idea of symbolic execution is that the program is executed on symbolic values instead of concrete values. This execution cannot be considered a real execution, it only means an execution within a synthetic environment, by a static analyzer. When the exact value of a variable cannot be decided (because, for example, it is a user input or method parameter), a symbolic value is

assigned to it. A symbolic variable can contain multiple concrete values that are allowed for its type, for example, a symbolic integer can take arbitrary values within the range of integer type. The possible values of symbolic variables can be bound with constraints. These constraints are usually derived from conditional statements or assignments. If a statement contains symbolic variables, the whole statement will be symbolic, and this applies for logical expressions as well. A symbolic Boolean can either be true or false, therefore, the symbolic engine will continue to execute both the true and false branches of a symbolic conditional statement. This means that theoretically every possible executional path will be explored, and hidden runtime exceptions can be detected.

The tree built up from the executional paths is called the symbolic execution tree. An example is presented in Figure 7. This is a directed, acyclic graph where each node represents a state of the program. Some branches of the symbolic execution tree are unreachable even with symbolic values because the concatenation of the conditional statements leading to them are unsatisfiable. Symbolic engines can cut off these unnecessary branches by maintaining and checking the satisfiability of a path condition (PC). The path condition is a quantifier-free logical formula over the symbolic variables. It contains constraints derived from the conditional statements that must be satisfied to reach the investigated executional state. Symbolic engines use constraint solver algorithms to check the satisfiability of the path condition and to decrease the number of possible branches when a conditional statement is reached. On the contrary, if a path condition is satisfiable, assigning the solutions to the symbolic variables as input values will direct the concrete execution to the state of that PC. Therefore, for example, symbolic execution can be a tool for test input generation.

The following Java code in Figure 6 shows an example function which decides if the three 2-dimensional points given as parameters can form a triangle. There are two erroneous return values: the function returns with -1 if the three points are on a line, and with 1 if the three points are not on one line but the triangle inequality is not satisfied anyway. The return value is 0 if a proper triangle can be composed from the three points.

```
1. public static int isTriangleValid(Point p1,Point p2,Point p3){
2.
3.     double a = euclideanDistance(p1,p2);
4.     double b = euclideanDistance(p2,p3);
5.     double c = euclideanDistance(p1,p3);
6.
7.     if(a == b+c || b == a+c || c == a+b) {
8.         return -1;
9.     }
10.
11.    if(a > b+c || b > a+c || c > a+b) {
12.        return 1;
13.    }
14.    return 0;
15. }
16.
```

**Figure 6: isTriangleValid function**

The implementation of the `euclideanDistance` function is not present, but as its name suggests it calculates the distance between two 2-dimensional points with the classic Euclidean formula. Suppose that the symbolic execution was started with this function, so the actual value of parameter `p1`, `p2`, and `p3` are unknown, therefore, they must be handled symbolically. This



means that the two data members (coordinate  $x$  and  $y$ ) of each Point are symbolic too, so they can hold any value from the domain of their type. Therefore, the value returned by the `euclideanDistance` function can be symbolic, making local variables  $a$ ,  $b$ , and  $c$  also symbolic variables. Theoretically, we have more information about these variables than the parameters of `isTriangleValid` because the symbolic engine executes their initialization. Nonlinear constraints can be derived from the Euclidean formula and then appended to the path condition, however, it is clear that doing it programmatically is not a trivial task. Figure 7 shows the symbolic execution tree built up during the execution of the sample code. Each conditional statement at line 7 and 11 can be split up to three sub-expressions, all of them containing symbolic variables. The short circuit evaluation gives an explanation as to why the two if statements create three-three branching points during the execution, whilst having symbolic variables in the conditions means that both the true and false outcomes of each sub-expression have to be investigated.

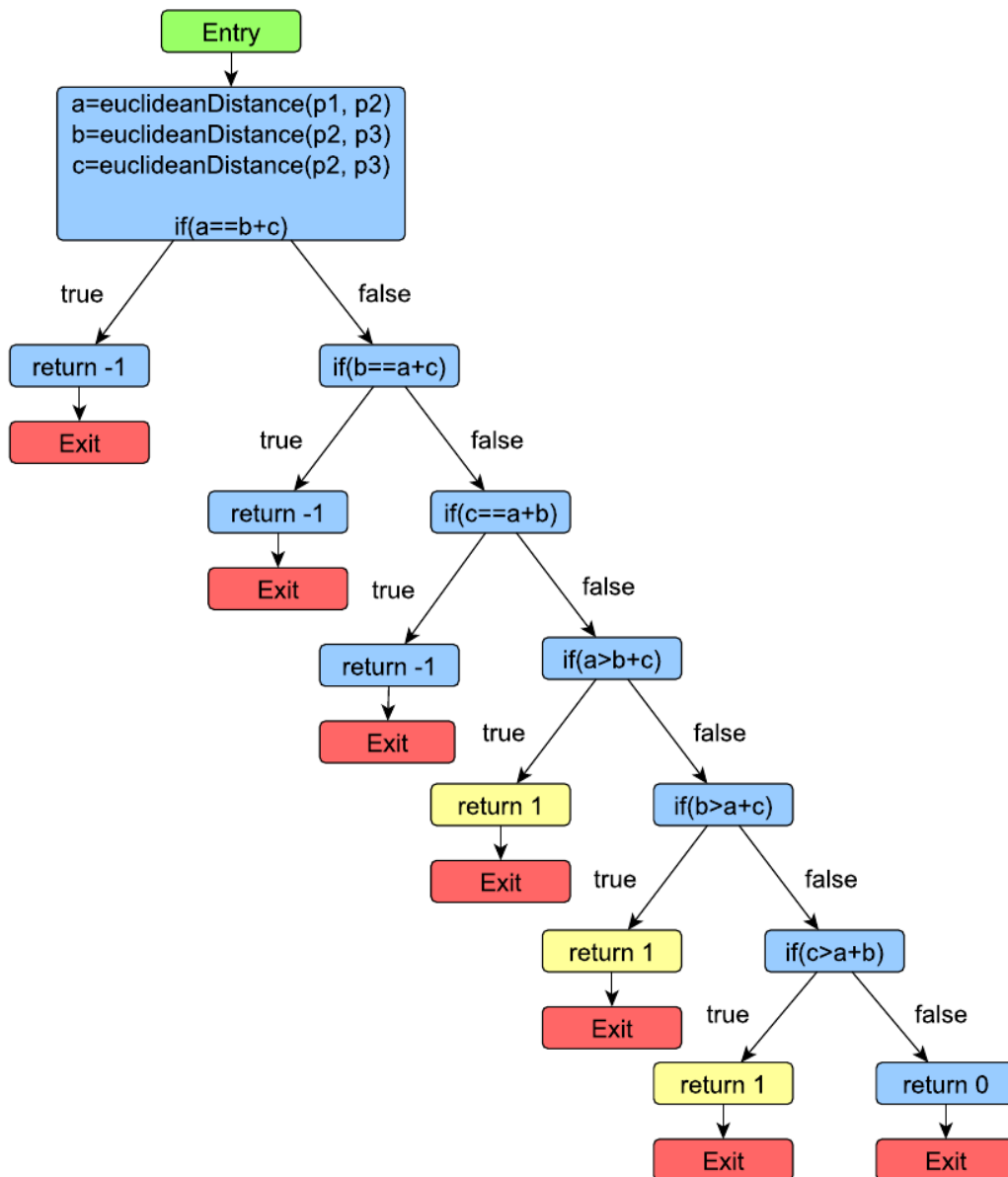


Figure 7: Symbolic execution tree

Path conditions are not presented in Figure 7 because - as previously mentioned - the constraints derived from the initialization are non-linear. We assume that symbolic engines will not build

up and solve constraints like that on their own, mostly because they treat the `sqrt` function grammatically with symbolic variables and an indefinite number of loops instead of using its mathematical meaning. However, a developer is able to detect straightaway the three unreachable states, namely the three `return 1` branches (colored with yellow). If three points given on the two-dimensional plane are not arranged on a line, they definitely form a triangle. Therefore, the `if` condition in line 7 is unnecessary, mathematically infeasible, making the true branches of each sub-expression unreachable. If an unreachable program state is discovered through unsatisfiable path conditions, the state and the sub-tree derived from it should be pruned from the symbolic execution tree. Not only do we avoid needless computations on that path, but possible false-positive warnings will also be eliminated.

This example gives a quick insight into how the symbolic execution is performed in the classical way. Whilst it is a very powerful tool for detecting runtime failures, it has several limitations. One of them that already occurred in the sample code is related to constraint solving. As the example shows, it is not a trivial task to find an efficient solution to reveal unreachable program states. Another major drawback is path explosion. The number of possible executional paths grows with the number of conditional statements almost exponentially. The scenario is even worse with `switch` statements and loops containing symbolic conditions. Since it is not known in advance how many iterations must be done in a loop, symbolic engines must make a guess or unroll the cycle until a limitation is reached, otherwise an infinite number of branches is created. Managing different kinds of features in programming languages symbolically, like static initialization in Java, also gives room for challenges. Simulating the environment of a program (e.g., system calls) in a purely symbolic way can also be a problem. For a more detailed summarization of symbolic execution and its difficulties see the survey of Baldoni et al. [57].

### 3.1.2.2 RTEHunter

In this section, we introduce the symbolic execution tool that we use for vulnerability detection. RTEHunter (abbreviation of RunTimeException Hunter) is a Java symbolic execution engine. It is part of the SourceMeter project, which is developed at FrontEndART Software LTD. SourceMeter [61] analyzes C/C++, Java, C#, Python, and RPG projects. It calculates source code metrics, detects code clones, and finds coding rule violations in the source code. RTEHunter is one of the static analyzers of the SourceMeter Java toolchain. It is designed to detect runtime exceptions in Java source code without executing the application in a real-life environment. Currently it can detect four kinds of common failures:

- *NullPointerException*,
- *ArrayIndexOutOfBoundsException*,
- *NegativeArraySizeException*, and
- *DivideByZeroException*.

RTEHunter performs the analysis by calling the symbolic execution for each method in the program separately. For big systems, this approach is usually a better solution than only starting the execution from the `main()` method [62], [63].

Starting the execution at the entry point of the real-life execution seems like a natural and convenient idea, however the practical limitations mentioned in the previous section would be reached very soon, leaving many parts of the code unexplored.

RTEHunter limits the number of states (nodes) in the symbolic execution tree and the depth of the execution tree as well.



### 3.1.2.3 ASG, CFG

RTEHunter uses the Abstract Semantic Graph (ASG) [64] of the program, which is constructed by the analyzer of SourceMeter. The ASG is a language-dependent representation of the source code that contains every detail of the source code in an internal graph representation. It is similar to an Abstract Syntax Tree but provides additional semantic information. First, RTEHunter builds a language-independent Control Flow Graph (CFG) [65] based on the ASG, and symbolic execution works on this CFG. In a CFG, each node represents a basic block. Basic blocks are the abstraction of straight-line program parts which are guaranteed to be executed sequentially. A jump in the code, like a conditional statement or a return statement, terminates the current basic block, and the outgoing edges connect to the basic blocks that are the targets of that jump. BasicBlock 5 in Figure 8 shows an example for a conditional jump with a true and false outgoing edge. Since RTEHunter performs symbolic execution on each method of the program, a separate control flow graph will be created for all methods. Each CFG for a method contains exactly one entry block and exactly one exit block, even if there are more return statements in a method (see Figure 8). The control can only enter and leave the CFG of a method through these two blocks, making them useful for handling function calls and returns.

The `euclideanDistance` function has a different control flow graph shown on the top of the figure. It is reached through call edges, however, this CFG is only represented with the entry and exit blocks without real content. Inside each basic block, we can find the ASG nodes that are visited in a sequential order. As previously mentioned, each method invocation creates a new basic block, which is why the first three initialization lines of the `isTriangleValid` method appear in four basic blocks (BB 16, 3, 4 and 5). From BasicBlock 5, a new basic block starts at each conditional sub-expression. Note that BasicBlock 6 represents the whole conditional statement at line 5, whilst BasicBlock 10 represents the if statement of line 7. BB 6 and BB 10 are not necessary, they are collector basic blocks introduced to make handling the expressions that contain multiple sub-expressions easier.

There are also true and false out-edges of these basic blocks, but the RTEHunter is aware of the short circuit evaluation so, for example, if the control reaches BB 6 from the true edge of BB 5, there will be no branching in BB 6, only the true edge will be continued. It can be seen from the given control flow graph that RTEHunter performs an interprocedural analysis. The symbolic engine handles method calls by continuing the symbolic execution in the called methods and then returning to the original function as during normal execution.

Built-in or third-party Java functions are also executed, however any warning found in them will be filtered out from the final output.

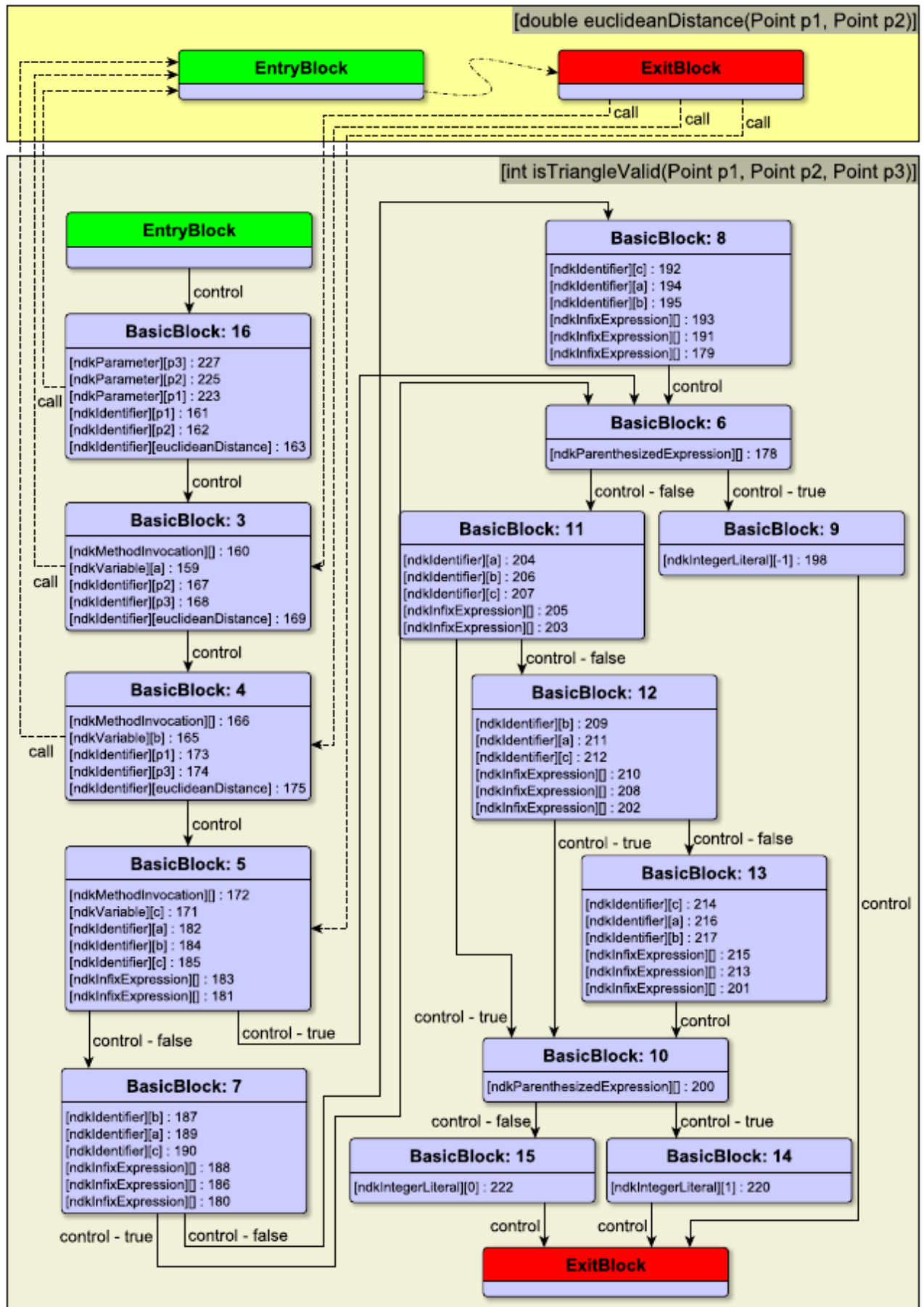


Figure 8: Control Flow Graph (CFG) of the `isTriangleValid` function

Figure 9 shows the symbolic execution tree constructed by RTEHunter. Some basic blocks appear multiple times, but each of them represents a different program state.

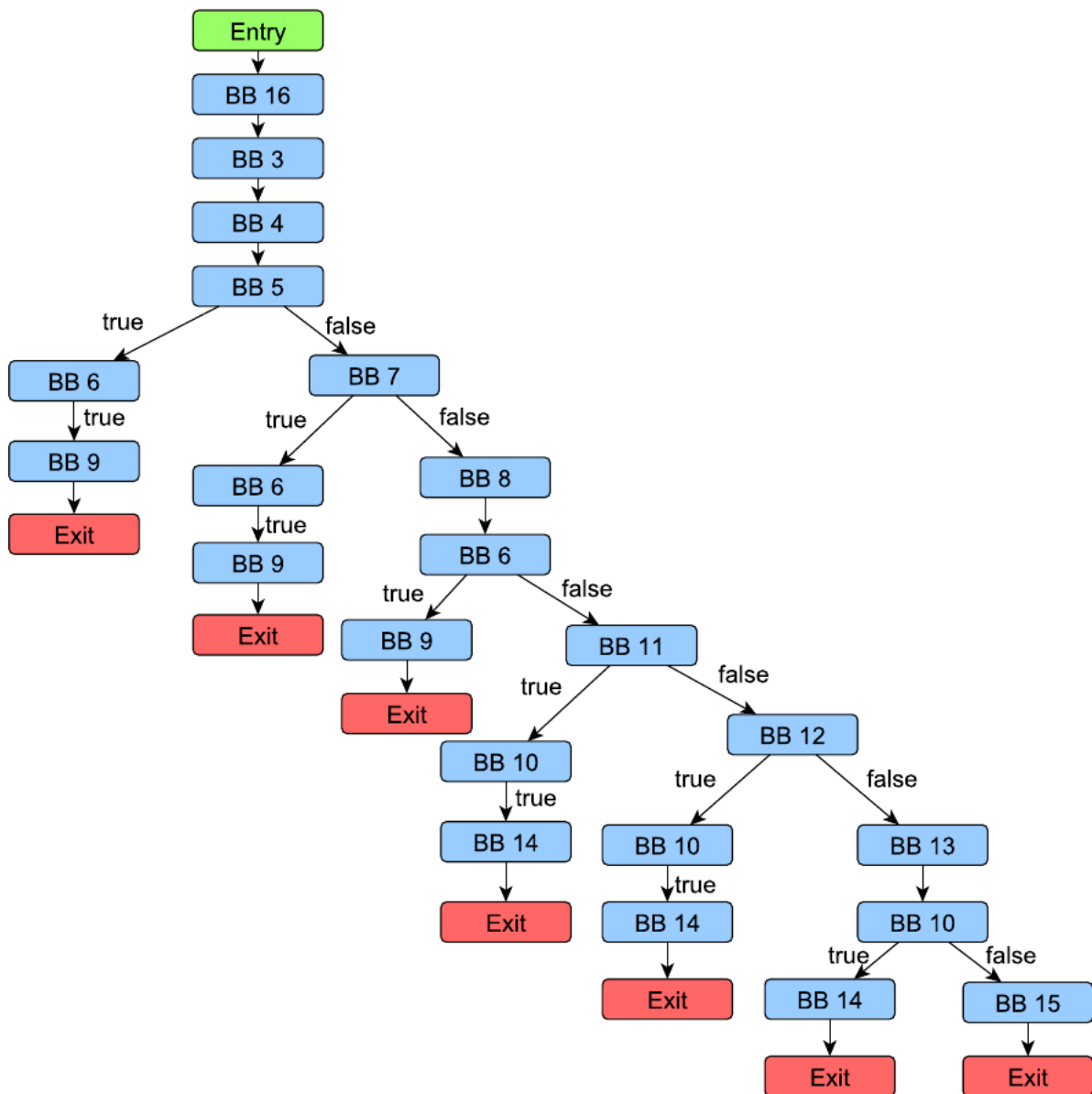


Figure 9: Symbolic execution tree

### 3.1.2.4 Constraint solvers

As mentioned in the previous chapters, in symbolic execution, so-called path conditions are created along each execution path. Checking their satisfiability and cutting off redundant, unavailable execution paths is a crucial task during this simulated execution.

### 3.1.3 Introduction

Many mathematical problem domains can be handled with computer programs, known as solvers. The inputs to these solvers can be formally described, and algorithms are available to calculate the corresponding outputs. Designing a solver involves formalizing the inputs and implementing an algorithm.

It can be observed that multiple solvers can be designed for a single mathematical problem. For popular problems, a simple internet search can yield numerous implementations/solvers.

One of the earliest mathematical problems for which solvers were developed is linear programming [66]. LP solvers are designed to solve linear programming problems, where the input consists of a linear function to be optimized and a finite set of linear constraints. The choice of algorithm and its precise implementation are crucial in this case. The simplex method, developed by Dantzig in the 1950s [67], is one of the most important algorithms and is often listed among the top 10 algorithms [68]. While the simplex method performs well in practice, there are theoretical cases where it can be slow. However, this does not affect applications, as simplex-based LP solvers perform very well. Internal point methods have also been developed based on theoretical considerations, offering alternatives to the simplex method.

During the development of theoretical computer science, the class NP was introduced [69]. This class contains languages, and a language is a mathematical description of a decision problem. In decision problems, a single bit needs to be computed: either accept or reject the input. The language is the set of inputs that need to be accepted. The NP class is not the focus of this thesis, so its definition is not elaborated on. Only the necessary theoretical background is provided.

There is often a significant relationship between two languages (decision problems, i.e., true/false questions). One language may be capable of formulating questions in the other. That is, a question associated with language  $L_1$  can be reformulated for a problem in language  $L_2$ . If the complexity/time required for reformulation is negligible, then the  $L_1$  problem has been reduced to the  $L_2$  language.

A significant result in theoretical computer science is the identification of universal problems in NP. These problems can formulate all problems in NP (every language in NP is reducible to them). The celebrated Cook-Levin theorem [70], [71] states that the SAT language in NP is one such universal problem.

The SAT language involves the decision problem of determining the satisfiability of a propositional logic formula in conjunctive normal form (CNF) [72]. The formalization of a CNF is straightforward, as it consists of a set/sequence of clauses. These clauses must all be satisfied, meaning they are connected by logical 'OR' (or 'AND') operations. A clause is a disjunction of variables and negated variables. Variables/negated variables collectively are called literals. A clause is satisfied if at least one literal evaluates to true. While it is a simple logical problem with easily describable input, a trivial algorithm can be designed. The variables (let their number be  $n$ ) can be evaluated in true/false logical values in a finite number of ways. The total number of evaluations is  $2^n$ . These can be enumerated, and the input can be evaluated on each. The first assignment of values to variables that makes the input CNF true stops the algorithm and leads to an accepting state. If all evaluations are tested without finding one that makes the input CNF true, the algorithm stops with a rejection.

Naturally, the development of SMT solvers did not stop with this naïve algorithm [73]. The  $2^n$  possibilities are too many to check (which we must do if the input is unsatisfiable) therefore it is not suitable for practical use. Typically, the number of logical variables is a few hundred or even thousands. In such cases, the number of possibilities is too large even for the fastest networked computers.

Numerous SAT solver algorithms have been developed. Similar to the simplex method, these algorithms exhibit shortcomings in worst-case analysis, yet frequently demonstrate excellent performance in practical applications. Ongoing advancements in SAT solvers involve the

refinement of heuristics and implementation techniques. Their performance is evaluated, for example, in the form of world championships.

SAT solvers are gaining relevance in many practical scenarios. The Cook-Levin theorem states that every NP problem can be efficiently reformulated as a SAT question. This includes, among other things, the factorization problem necessary for breaking modern encryption. If we want to factorize a specific number, for example, to determine the presence of a small prime divisor, this question can be formulated as a CNF formula. The satisfiability of this formula expresses the existence of the small prime divisor. This example illustrates how useful an SAT solver can be.

Integrating a suitable logical solver into a system can make the system much more efficient. This was a primary motivation for integrating a solver into RTEHunter. The integration process consisted of several steps. First, we had to select the solver to be integrated. However, the number of SAT solvers can be measured in the hundreds. There are numerous solvers for each formula class. One of our selection criteria is that expressive ability of logical formulas should be high. This could it easy for RTEHunter to formulate questions. Other aspects included, for example, the solver's accuracy, speed and whether its code base was still maintained.

Our choice was SMT solvers [74]. Their expressive power surpass those of SAT solvers, and they are widely used and developed.

### **3.1.3.1 SMT solvers**

SMT (Satisfiability Modulo Theories) solvers are programs that decide the satisfiability of formulas modulo theories. Modulo theories generalize the Boolean satisfiability problem. They extend the Boolean satisfiability problem (SAT) to be interpreted within certain formal theories. The name "modulo" arises from the interpretation of these expressions within a specific formal theory. In first-order logic, these expressions are often written with equality, excluding quantifiers, in a manner familiar to Boolean algebra. It is worth noting that SMT formulas can describe SAT formulas, making the design of SMT solvers more challenging than SAT solvers. Since SAT satisfiability is an NP-complete problem, SMT is also NP-hard. For many inputs, satisfiability is undecidable, which means the solver might enter an infinite loop (programs typically handle this with a timeout parameter, terminating after a specified runtime). After a successful run, we obtain information about whether the machine can satisfy the system of conditions we built.

These solvers are frequently used for verification, symbolic execution-based testing, and even for proving program correctness [75], [76], [77].

### **3.1.3.2 SMT-LIBv2 language**

Solvers have developed a language for encoding input and output, known as the SMT-LIBv2 Language. Figure 10 shows an SMT-LIBv2 fragment. "Code lines" written using this language serve as the solver's input. Most solvers can obtain this input either by loading a file or by entering it through the program's interface. The code lines, enclosed in parentheses, typically include settings in the first lines, specifying the logic the solver should use to search for solutions or whether it should generate models, among other options.



```

; Getting values or models
(set-option :print-success false)
(set-option :produce-models true)
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x * 2 y) 20))
(assert (= (- x y) 2))
(check-sat)
; sat
(get-value (x y))
; ((x 8) (y 6))
(get-model)
; ((define-fun x () Int 8)
; (define-fun y () Int 6)
; )
(exit)

```

**Figure 10: SMT-LIBv2 language**

Next come the declarations, where variables and functions can be defined. These declarations specify the inputs and outputs of functions. For example, a declaration could define a function that takes two integer inputs and produces one integer output: (define-fun name (inputTypes) outputType (body)). Following these definitions are the conditions the program attempts to determine if they can be satisfied. If satisfiable, it returns "sat"; otherwise, it returns "unsat." Additionally, in the case of a "sat" output, a model can be requested, providing a concrete solution to the conditions.

It was mentioned earlier that satisfiability is NP-hard, so finding a solution within a reasonable time frame is not guaranteed. Solvers handle this by providing a result if a time limit is set. If the solver cannot find a solution within the specified time, a "timeout" result is returned, and no model can be requested. The purpose of the timeout is to ensure the program terminates.

One interesting aspect compared to other languages is that operators are specified before the variables/values on which the operation is to be performed. This allows for the construction of more complex expressions, as shown in the accompanying image, using parentheses appropriately.

### 3.1.3.3 Improvement of CFG generation

As part of the development of the symbolic executor, the ASG and CFG generator modules were also tested. We wanted to achieve full support for Java 11 language elements.

#### 3.1.3.3.1 Try-with-resources

Try-with-resources was a new feature of Java 11. This is a control structure for automatically handling the resources we specify as the parameter of the try block. If an exception occurs during the execution of the try block, the resources declared in the parentheses of the try-with-resources statement are automatically closed. This is the main advantage of try-with-resources – it ensures that resources are properly closed, even if an exception occurs.

Resources must implement the AutoCloseable interface. The close () method of the interface handles resources to be closed appropriately.



```

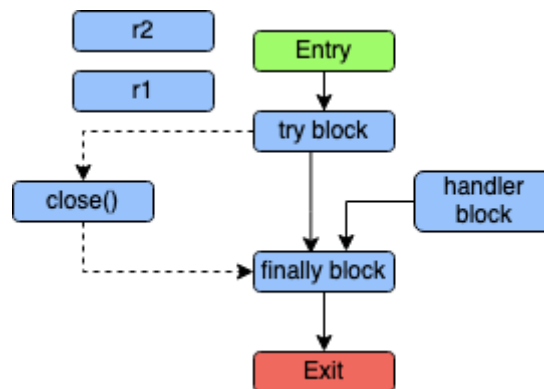
1 try (Resource r1 = new Resource1();
2     Resource r2 = new Resource2()){
3     // try block
4 } catch (Exception e) {
5     // handler block
6 } finally {
7     // finally block
8 }

```

**Figure 11: Try-catch with multiple resources**

Figure 11 shows an example usage of the try-with-resources, containing two AutoClosable resources, r1 and r2. Regardless of whether an exception is thrown in the try block, their close methods will be executed after the try block is executed. Figure 12 shows a simplified CFG that was originally built from the Java code shown in Figure 11. Dashed lines represent method calls (e.g., call to the close () methods at the end of the try block) and solid lines represent control edges. The r1, r2 basic boxes represent the initialization of the two resources. However, these basic boxes are not part of the control flow graph, the control cannot reach them starting from the entry block. This has been corrected in Figure 13. The idea was to implement a new visitor that connects the basic blocks of the new resources into the control flow before the try block. The resources should also be connected to each other respecting the order in which they were declared.

Both Figure 12 and Figure 13 also show a basic block with no edges from the entry block. This is because this basic block represents the error handling, i.e., the catch branch. Every statement in the try block that can throw an exception should be linked to this block as an incoming edge. These edges are not shown in the figures. There will be a further literature review on the exception handling representation.



**Figure 12: Original CFG**



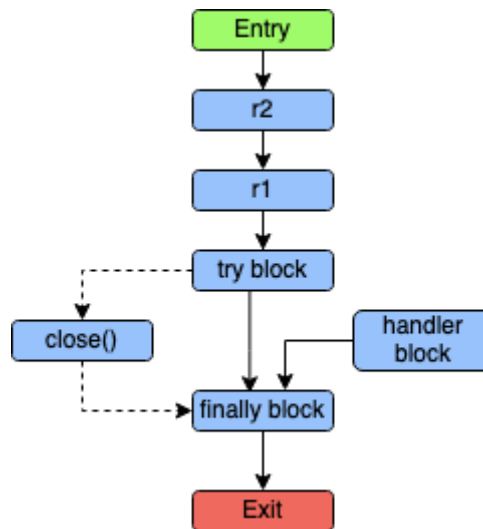


Figure 13: Improved CFG with resource handling

### 3.1.4 Handling finally blocks

In Java, the final block is used in conjunction with a try block to define a set of statements that will be executed regardless of whether an exception is thrown or not. The final block is part of the exception handling mechanism and provides a way to ensure that certain code is always executed, typically for cleanup purposes, regardless of whether an exception occurs.

```

1  try {
2    code1
3    if(condition) {
4      return;
5    }
6    code2
7  } catch (Exception e) {
8    // handler block
9  } finally {
10   // finally block
11 }
12 code3
  
```

Figure 14: Try-catch-finally construct

If a Java code includes a try-catch-finally construction with branching in the try block (like in Figure 14), the CFG needs to appropriately represent the flow of control, including the handling of the final block. Other branching structures are also possible, e.g., a throw statement, continue, break or a return statement. The final block will always be executed, even on an explicit return statement. RTEHunter, integrated with the constraint solver, was no longer able to handle these branching cases well, so we improved it. Figure 15 shows the original representation of the control flow. There is no final block between the block containing the return statement and the exit block, which is a mistake. However, connecting to the existing finally block is no solution either. How does the symbolic executor know whether to go to the code3 block or the exit block after executing the final block? The solution we came up with is cloning the final blocks, making them easier to read but also keeping the correctness of the execution flow. This can be seen in Figure 16.

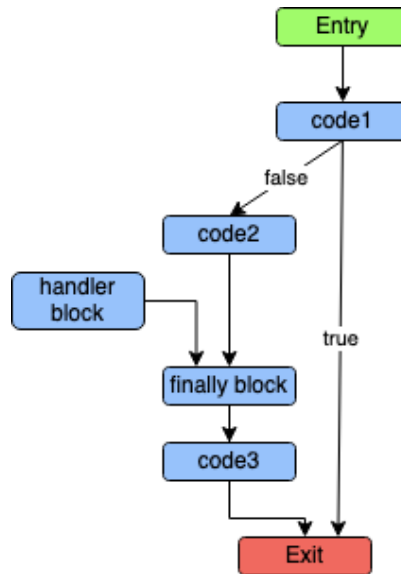


Figure 15: Original CFG for handling finally blocks

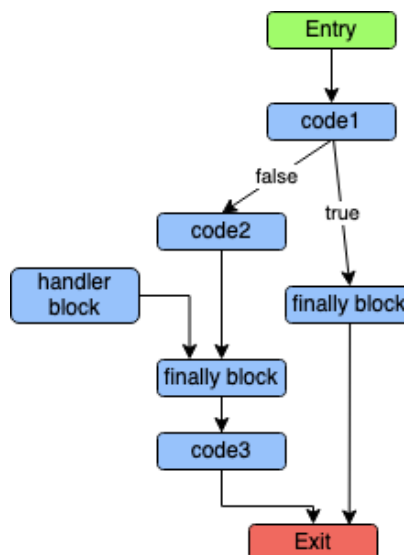


Figure 16: Improved handling of finally blocks

During the implementation of this feature, we had to pay attention to various aspects. For example, duplication is not necessary if the same code block were executed from both input branches after executing the final block. Nested try-catch-finally constructs and labels also needed extra consideration.

### 3.1.5 Constraint solver integration

To make RTEHunter as efficient as possible in detecting run-time errors, we have integrated a new state-of-the-art constraint solver. In this section, we discuss the measurements taken to select the constraint solver and the challenges of integration.

#### 3.1.5.1 Solver Comparison

The project began with research, exploring available solvers, searching for benchmarks on their performance, and comparing them. The results are discussed in the following subsections.

Both paid and open-source solutions exist, but for the project, only freely available solvers with open-source code were tested. During the solution process, even cases where only the solver was used as a runnable program were tested, so the limitation that comes with paid programs, typically not providing access to the source code, did not apply.

Accuracy, speed (for solving a demonstrably solvable task), and support for development were crucial aspects for application purposes. After reviewing several programs and benchmarks, it can be concluded that support is often discontinued, or new versions are infrequent in many cases. The yices2 [78], for instance, performed well in many places a few years ago but has not been updated since 2021. Several documents describe SMT solvers that no longer exist or have fallen far behind in development. Many solution logics can be set for a given solution, and there is no clear "winner."

During our research, we came across the SMT-COMP competition, which aims to compare SMT solvers. The competition evaluates how quickly a solver solves a given problem set (less time results in more points), and the accuracy, determining whether the response is correct (higher accuracy rate earns more points).

Most solutions are either a major solver name or a modified version of it. As shown in Figure 17, in this task, Z3++ and one variant of Z3 were the winners in several "divisions." This is possible because, in the SMT-LIB2 language, the solution logic can be specified, allowing a program to achieve different results. Based on our own tests, we found CVC5 and Z3 to be the most valuable, and their source code is publicly available. Both were tried during integration. Ultimately, the decision was made in favor of Z3 because its build does not require a specific environment, while CVC5 requires a Linux system capable of Linux-Windows cross-build using MINGW.



**Biggest Lead Ranking- Model Validation Track**

Page generated on 2023-07-06 16:06:01 +0000

**Winners**

Sequential Performance	Parallel Performance
Z3++	Z3++

**Sequential Performance**

Solver	Correct Score	Time Score	Division
SMTInterpol	1.49541284	1.90312914	QF_Datatypes <sup>e</sup>
cvc5	1.38431373	0.21522881	QF_Equality+NonLinearArith <sup>e</sup>
SMTInterpol	1.09640523	3.40555517	QF_ADT+LinArith <sup>e</sup>
Bitwuzla	1.07727273	16.51746958	QF_ADT+BitVec <sup>e</sup>
Z3++	1.07597851	1.06674584	QF_NonLinearRealArith <sup>e</sup>
Z3++	1.0369726	2.3003773	QF_LinearIntArith
Bitwuzla	1.02487562	0.81198183	QF_Equality+Bitvec
Z3++	1.01831352	1.15898299	QF_NonLinearIntArith <sup>e</sup>
SMTInterpol	1.00223714	0.47717592	QF_Equality+LinearArith
OpenSMT	1.00164204	0.5399574	QF_LinearRealArith
STP	1.00055432	1.23686617	QF_Bitvec
Bitwuzla	1.00045171	1.55333252	QF_FPArith <sup>e</sup>
Yices2	1.0	3.74833938	QF_Equality

**Figure 17: Participants of an SMT solver competition**

RTEHunter is developed using Microsoft Visual Studio on a Windows system, so it was advantageous not to need to set up a new environment for building CVC5. Conceptually, we tried using the compiled binary to see how usable the solution is, and based on initial tests, its operation is similar to that of Z3. From this, we concluded that any program capable of interpreting the SMT-LIB2 language could be used during integration.

### 3.1.5.2 Measurements and results

To select the best solver, we also carried out our own measurements. To do this, we had to collect SMT-LIBv2 codes and run them to test the accuracy and speed of the solvers. The SMT-LIBv2 codes were obtained from various online sources, benchmarks, and we created our own test codes as well. Unfortunately, some of the online sources were faulty and did not meet the formal language requirements. These faulty files did not pose a problem during measurements because these consistently led to errors in each solver. We tested several SMT solvers, and ultimately, Z3 was chosen. Most solvers were typically either very slow but accurate or very fast but not capable of handling large problem sets. One notable example was exceptionally fast, averaging "solutions" in less than 0.1 seconds. However, we got an answer in less than 10% of cases and in the remaining 90% the program timed out, which is not acceptable for the symbolic executor.

Two of the best-performing programs were CVC and Z3. We conducted various measurements on these programs. It is essential to mention that we verified the output beforehand, but during the measurements, none of the programs output anything to the screen. This helped filter out differences in how each program displays solutions or models. In the case of Z3, we took

advantage of the convenience feature that eliminates the need to specify the logic (other programs also allow skipping this step, but the behavior may change). The measurements are summarized in Table 17.

**Table 17: Results of the measurements**

SMT solver	1 run, 1000 API calls	1000 runs, 1-1 API calls	Solver.exe
Z3	11.053	20.792	14.618
CVC	4.428	62.196	68.627
CVC ALL	4.442	65.876	58.34
CVC QF_UFLIA	4.146	72.195	57.242
z3/cvc Best	2.665943078	0.3342980256	0.2553719297
z3/cvc Worst	2.488293561	0.2879977838	0.2130065426
Best/Worst	1.071394115	1.160765966	1.198892422

At the beginning of each row in Table 17, you can see what was used for the solution (program/logic), and the last 3 rows show the meaning of the numbers listed there. Z3 does not expect logic, so it is never specified there. The choice could be narrowed down there as well, but in online forums, it is often stated that it is "not customary" to use logic, and specifying it is not trivial. For CVC, three types of invocations are visible: without specifying logic, with ALL, and with a specific (randomly chosen) logic. In theory, there should be no difference between ALL and without specifying logic, but CVC, if nothing is specified, signals to at least provide ALL. However, the difference is evident in time. Best and worst represent the best and worst times between Z3 and CVC, regardless of the logic used.

In the second and third columns, the mode of execution is visible, whether we solve one problem 1000 times with the program or solve 1000 problems with it at once (the same problem, only the number of runs differed, either separately or in one batch). In the third column, you can see data for solving the entire set of 1000 problems, where we solved the problem with a precompiled binary. All runtime is in seconds.

At first glance, based on the table, it is apparent that CVC5 significantly outperformed Z3 because it solved 1000 problems faster in one run, and there is a factor of 2.4-2.6 between the runtime values. This is not an error; based on measurements, it was clear that CVC5 is faster with a larger, more complex problem set. It is also visible that there is a difference when choosing logic. Something different happens if nothing is specified, if everything is given, or if we introduce a "limitation." The choice of solution logic was not based on anything specific; we chose a solution logic that could solve the problem(s) because it could also be chosen so that it could not handle the given problem set. Therefore, the choice was only roughly random, with minimal directionality. In the second column, it is evident that if we have many small problems, then Z3 performs significantly better, about 3 times faster than CVC. We found no clear reason why loading CVC multiple times would cause such a significant difference. The only thing we could think of during the research is what we mentioned earlier that CVC5

requires a system capable of Linux-Windows cross-build for its build, while Z3 can be natively built with VS2019 (or later). Binary execution typically falls between the two runtimes. The only difference occurred when we gave the solver a file without specifying the logic.

In Z3, there is a direct API call that solves the problem from SMT-LIBv2 code. Somehow, string API calls are faster than directly written API calls. We did not find clear information about this in an official description, but various forum claims suggest that Z3 converts API calls to strings in the background, which would explain the runtime differences. For the solving of 1000 problems with one run and solving 1000 problems in 1000 runs, it solved them in 10.236 and 15.471 seconds, respectively, based on the table, which are better results than the previous run.

### 3.1.6 Low-level integration challenges

#### 3.1.6.1 Bit operations

In the Java language, it is possible to perform bitwise operations with primitive types, but not in SMTLib2. In Java, bitwise operations are only defined for integers. The reason for this is that, in order to perform the operation, the computer needs to represent the number as a sequence of bits. Floating-point numbers are stored according to the IEEE-754 standard, where the first bit represents the sign (0: positive, 1: negative), followed by 8 bits for the exponent, and the remaining 23 bits for the mantissa. For example, 1.5 is represented as 0 - 01111111 - 10000000000000000000000 (separated by "-" as indicated earlier for the 1-8-23 bit blocks). In this example, it is already evident that a bitwise shift operation would cause issues. Shifting the exponent to the left by 2 would move the 1-bit to the sign's position.

In SMTLib2, bit vectors need to be handled separately. We can only apply bitwise operations to this type, which in Java, we are accustomed to applying easily to integers. The following code snippet allows us to create a bit vector named `x_0` containing 32 bits: (`declare-const x_0 (_BitVec 32)`). This type does not exist in the Java language; however, we need it because bitwise operations cannot be performed differently in SMTLib2. Primarily, we will use it by converting a previously created variable to a bit vector for operations with a variable like (`(_int2bv 8) variable`). With this, we can perform bitwise operations, such as bit shifting or bitwise AND, as needed. Of course, in the example mentioned earlier, we only created an 8-bit bit vector, which does not cover the entire integer range. This needs to be adjusted to Java types.

When performing bitwise operations, it is crucial to ensure that the operations are carried out on bit vectors of the same size. In SMTLib2, you cannot perform a bitwise OR operation on an 8-bit and a 16-bit bit vector. One solution is to create all bit vectors with the largest size possible, ensuring that different sizes won't cause issues. However, this introduces a new problem. If we had a byte type in Java, we would need to handle overflow properly to avoid the larger container producing a result different from what would occur if the code were running in reality.

During conversion, one approach is to use an appropriate all-one vector during the conversion process, eliminating the upper extra bits needed for SMTLib2 to perform the operation. This helps address issues like overflow caused by a bitwise shift that might exceed the maximum value the type can store.

During integration, some modifications were made to the existing solution, and some completely new additions were introduced. The constraint expression class, which previously stored type of information and could negate expressions, etc., was aligned with Java types.

However, with the introduction of bit vectors, a new type had to be created to handle this data. The most straightforward solution was to create a new constructor where the "bitvector" field was set to a default value of `false`, leaving the previous constructor(s) unchanged. This ensures the proper functioning of the previous parts, and the change does not cause issues because the bit vector type does not exist in Java or C++. It was strictly necessary for SMTLib to enable bitwise operations (shift, bitwise AND, bitwise OR, etc.).

We have two types of operations: traditional bitwise operations (e.g., left shift) on bit vectors and traditional operations that involve a bit vector on one side and a primitive number type on the other side (e.g., adding a bit vector to an integer). Handling these cases occurs separately.

In cases where we perform traditional operations on bit vectors, we still need to convert everything to the same type. It can be assumed that if we are using bit vectors for some reason, we are not doing so just once. Therefore, it is preferable to convert all variables involved in the operation to bit vectors.

This approach ensures consistency and avoids potential issues when dealing with mixed types. If, for any reason, bit vectors are used, it is better to convert all variables to bit vectors in every operation.

### 3.1.6.2 *Remainder of division*

RTEHunter was developed for analyzing Java programs, but the program itself is written in C++. This posed challenges during integration. If we can pre-calculate operations, for instance, when both sides of a given operation, such as division, involve numbers (not symbols), we calculate it on the C++ side, and the symbolic solver only receives the computed value. This alone wouldn't be problematic if the toolsets of the two programming languages were entirely identical.

One frequently encountered difference between the two languages was the modulus operation. In Java, the operation `12.3f % 3.2f` is allowed. However, in C++, the `%` operator is not defined for float and double types. It turned out that the modulus operation cannot be handled the same way if a double or float appears on either side of the operation. Fortunately, in the C++ implementation, the `math.h` header provides the `fmod(a, b)` function, which behaves similarly to `%` in Java, but implementation-wise, both cannot be handled identically. For this operation, three different cases are considered: for integers, where it works the same as in Java; for real numbers, where the mentioned difference between Java and C++ exists, and hence a separate function must be used on the C++ side depending on whether a real number is present in the operation; and the third case where a symbol is present on either side of the operation, which is handled as a separate case everywhere.

### 3.1.7 **Integration result**

The integration of the Z3 solver has been completed, but testing and patching are still ongoing. Testing is currently performed on 89 Java test classes, each with 3-4 (or more) false positive (FP) and true positive (TP) errors. The results are reviewed one by one. We carry out minor corrections if necessary. A manual review is essential, because, for example, thanks to the improvements in RTEHunter, we have found an incorrectly categorized true positive runtime error. Table 18 shows partial results of several `ClassCastException` and `ArithmeticException` related tests. The last column shows how many of the errors were found by the original RTEHunter before integration, and the second-to-last column shows the post-integration states. In the case of false positive errors, the lower the number of hits, the better the results.





**Table 18: Improvements in the regression tests**

Test name	Total number of bugs		Improved RTEHunter	Original RTEHunter
ArithmeticExceptions1	FP	2	2	2
	TP	7	3	2
ClassCastException01	FP	2	0	0
	TP	2	1	0
ClassCastException02	FP	1	0	0
	TP	1	0	0
ClassCastException03	FP	2	0	0
	TP	2	1	0
ClassCastException04	FP	2	0	0
	TP	1	1	0
ClassCastException05	FP	2	0	0
	TP	2	2	0
ClassCastException06	FP	2	0	0
	TP	1	1	0
ClassCastException07	FP	2	0	0
	TP	2	2	0
ClassCastException08	FP	2	0	0
	TP	2	2	0

We have highlighted in green the cells where full improvement has been achieved. The original RTEHunter did a very poor job of detecting ClassCastExceptions, but, in most cases, this has been completely fixed. We have highlighted in yellow those cells where there was no or only partial improvement. For example, in the ArithmeticExceptions1 test, we still find the two false positive errors. These are currently being corrected.

### 3.1.7.1 Example

In this section, we show the faulty test case as an example. In Figure 18, you can see that one of the Integer to String casts was labeled a false positive because the else execution branch was considered unavailable. However, the improved RTEHunter detected this false positive, so we took a closer look at what could be wrong. We found that for given initial values  $i, j$ , the if condition can never be true. We have fixed the condition as follows:  $(i - j \% 2) == -1$ .

The investigation on this test case also provided feedback on our % operation implementation.

```
int i = 6 , j = 7;
Object a = new Integer(j);
if(((i-j) % 2) == 0){
    System.out.println((String) a); /* [RTE: TP] */
} else {
    System.out.println((String) i); /* [RTE: FP] Unreachable */
}
```

**Figure 18: Faulty test case**

### 3.2 AI-based vulnerability detection

The application of LLMs for vulnerability detection in code has been an area of significant interest in recent literature. In "Technical Report: Evaluation of ChatGPT Model for Vulnerability Detection," the authors evaluated the performance of ChatGPT and GPT-3 for vulnerability detection tasks but found that the ChatGPT model performed no better than a dummy classifier for both binary and multi-label classification tasks [79]. These findings demonstrate the limitations of LLMs when it comes to vulnerability detection in real-world datasets.

Moreover, "Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT" explored the potential of LLMs for fuzzing deep learning libraries by proposing FuzzGPT, a technique to prime LLMs to synthesize unusual programs for fuzzing [80]. The authors demonstrated that FuzzGPT could substantially outperform TitanFuzz, detecting 76 bugs, with 49 already confirmed as previously unknown bugs, including 11 high-priority bugs or security vulnerabilities [80]. This work highlights the potential of LLMs to be effectively used for fuzzing when primed with the right techniques.

In contrast, "Detecting software vulnerabilities using Language Models" presented a transformer-based vulnerability detection framework called VulDetect, which achieved an accuracy of up to 92.65% in identifying vulnerable software code, outperforming other state-of-the-art vulnerability detection techniques [81]. Additionally, "Predicting Vulnerability in Large Codebases With Deep Code Representation" developed an AI-based system that uses deep representation of Abstract Syntax Trees (ASTs) to identify and alert potential bugs during the development process [82]. These studies showcase the potential of LLMs and other deep learning techniques to detect and predict vulnerabilities in software code.

One significant advancement is the application of LLMs for formal software verification, as described in a paper [83] that presents a method using LLMs trained on both natural language and code. These models are capable of generating and repairing entire proofs, marking a departure from traditional step-by-step or search-based verification methods. This approach allows LLMs to analyze codebases and identify logical errors, potentially leading to the discovery of vulnerabilities, and can be a valuable addition to current vulnerability detection practices in software development due to their learning capacity and compatibility with existing tools.

Another study [84] highlights the capability of OpenAI's GPT-4 in detecting software vulnerabilities, comparing its performance with traditional static code analyzers like Snyk and Fortify. The analysis, covering a range of repositories and programming languages, showed that GPT-4 identified about four times as many vulnerabilities as its counterparts and provided viable fixes with a low rate of false positives. This demonstrates the potential of LLMs not only in detecting but also in repairing software vulnerabilities, indicating a significant advancement in automated software security.

The paper [85] introduces VulBench, a benchmark combining data from CTF challenges and real-world applications to evaluate the effectiveness of Large Language Models (LLMs) in software vulnerability detection. The study finds that LLMs, particularly GPT-4, outperform traditional deep learning models and static analyzers in simpler scenarios like CTF datasets. However, their performance diminishes in more complex real-world datasets, highlighting a limitation in their current ability to analyze intricate software systems. The paper suggests future research should focus on enhancing LLMs' capabilities in processing complex projects and integrating them with other detection techniques, such as fuzzing or static analysis, to maximize their effectiveness in vulnerability detection.

### 3.3 AI4VULN System Requirements Analysis

This section will explain how the relevant system requirements identified in D2.1 will be covered by AI4VULN.

**Table 19: System requirements coverage by initial version of AI4VULN component**

Req. ID	Req. name	Req. description	Priority	Coverage	How addressed
SR-011	Vulnerability detection	The AI4CYBER framework shall be able to identify vulnerabilities present in the source code via the AI4VULN component.	M	Partially covered	AI4VULN will be tested on its regression testcases, open-source software systems containing known vulnerabilities and vulnerability databases. After that, we will apply AI4VULN on the target source code to detect the vulnerabilities. The result will be used by AI4FIX.

The requirements will be validated in WP7. A training validation plan is available in Section 3.7.

### 3.4 AI4VULN Specification

AI4VULN is a command line static analyzer tool that is able to detect vulnerabilities in the source code. Its output is the list of the detected vulnerabilities that is used by AI4FIX, but the users can also use it to correct to validate or to correct the bugs.

In the following a detailed specification of the AI4VULN component according to C4 model (<https://c4model.com/>) is provided.

### 3.4.1 System context

At system level AI4FIX uses the results of AI4VULN to correct security vulnerabilities and developers or security engineers also use it to check, correct or validate the security relevant bugs.

Figure 19 shows the system context diagram of AI4VULN. The input of AI4VULN is the repository of the Java system to be analyzed. The developer who initiates the vulnerability searching process supplies AI4VULN with the file paths. The AI4VULN software system then processes the source code with static analysis. As a result, the developer receives the list of possible vulnerabilities, which can also be forwarded to the AI4FIX system for automatic error correction.

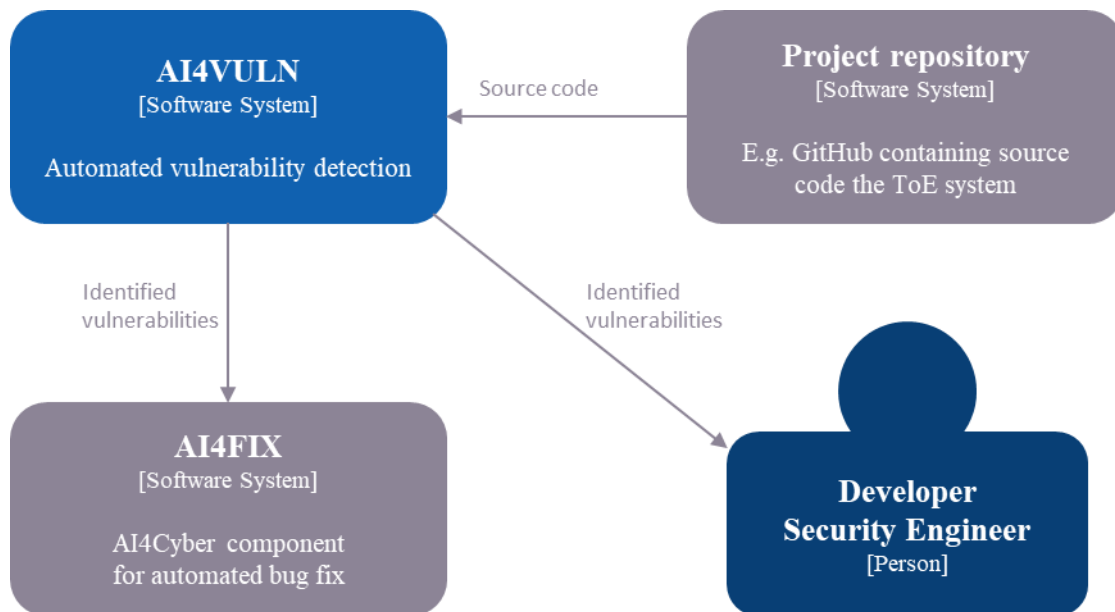
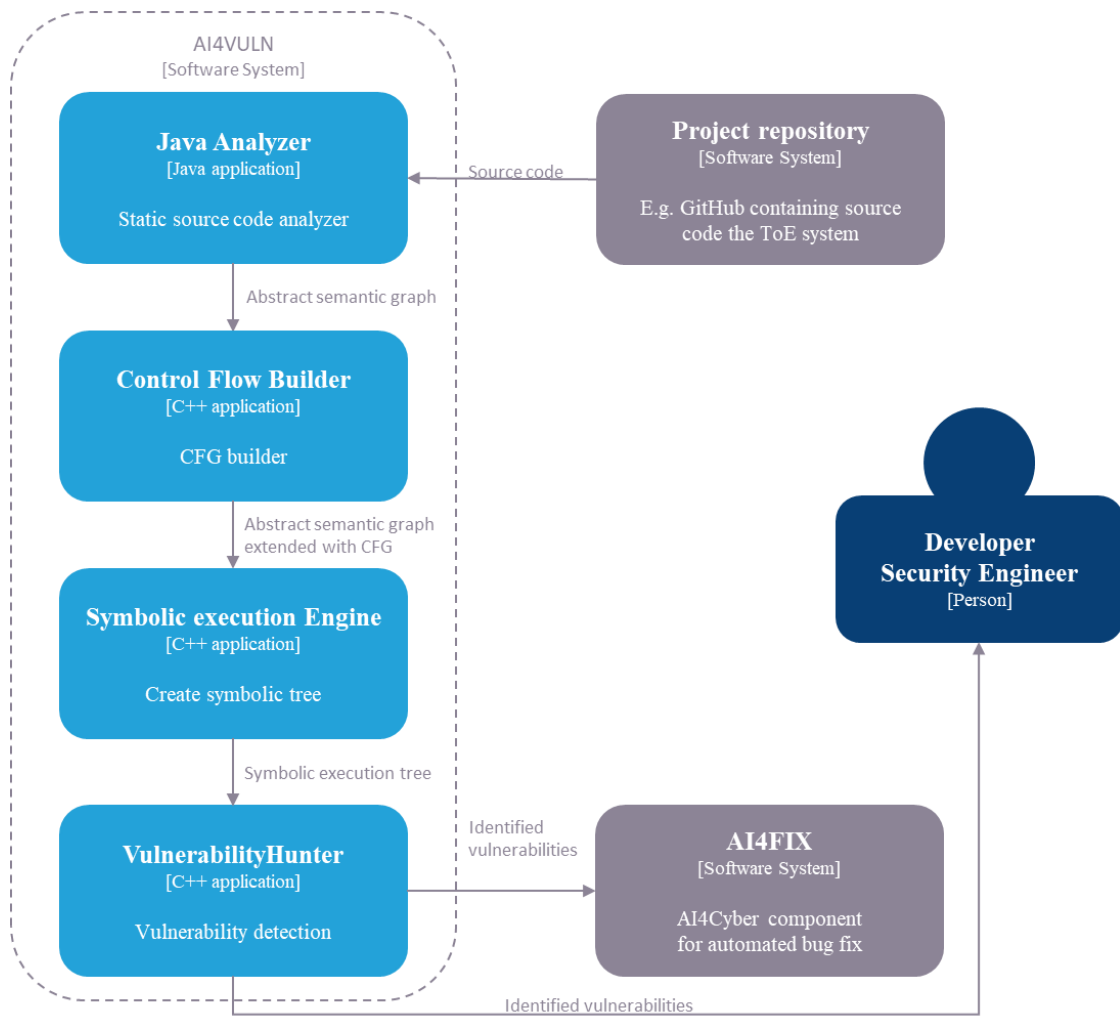


Figure 19: System context of AI4VULN

### 3.4.2 Container model

Figure 20 shows the AI4VULN container diagram. AI4VULN can be divided into four units. First, the source code is analyzed by a language analyzer written in Java. The output of this analysis is the Abstract Semantic Graph (ASG) which means that the input program is translated into a graph representation. Using the ASG, the Control Flow Builder unit generates the Control Flow Graph (CFG). The input of the Symbolic Execution Engine is the ASG supplemented with the CFG. They will be used to simulate the execution of “all” possible execution paths of the program. A symbolic execution tree is created from these execution paths. By traversing the symbolic execution tree, VulnerabilityHunter identifies potential vulnerabilities.



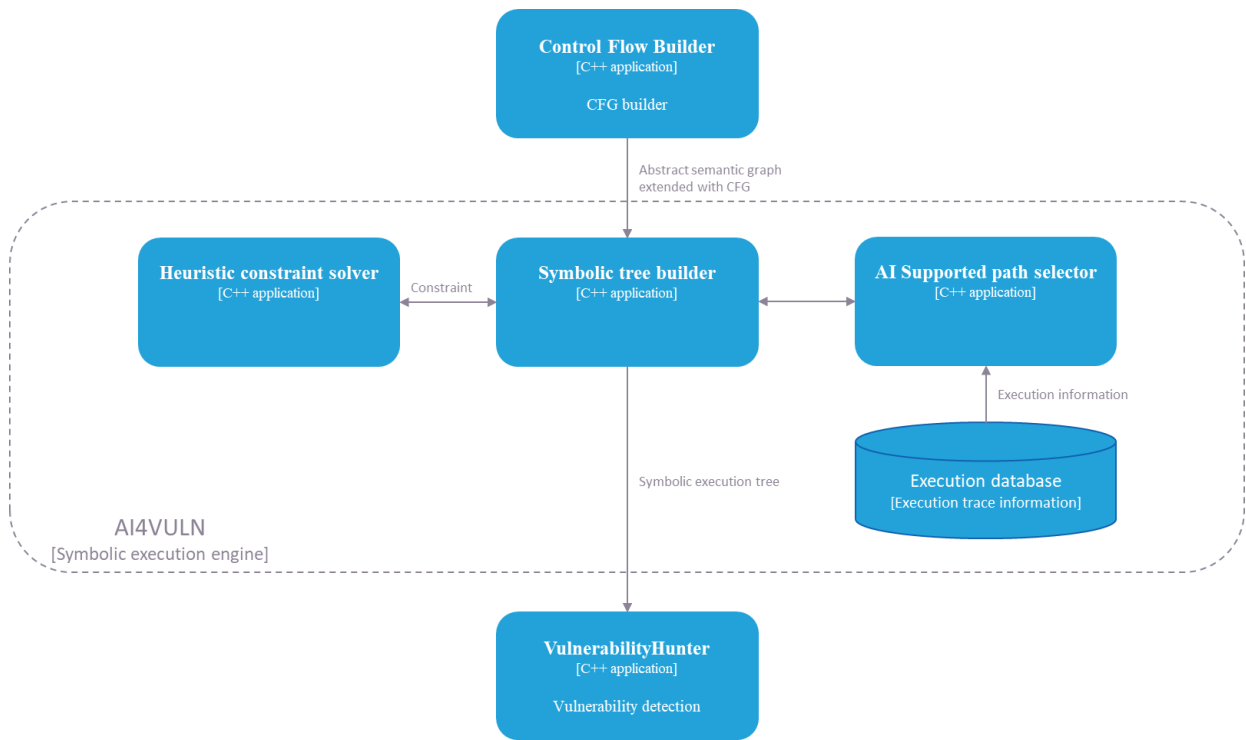
**Figure 20: Container model of AI4VULN**

### 3.4.3 Component model

Figure 21 depicts the component model of the AI4VULN software system. Although AI4VULN consists of four containers, only the Symbolic execution Engine can be further decomposed, the others components are “simple”.

The Symbolic engine uses the CFG and the ASG to generate a tree of possible execution paths. This generation process is supported by two components: the constraint solver and the path selector. The constraint solver can eliminate unsatisfiable execution branches. Since running the constraint solver itself can be very resource-demanding, we introduce heuristics to speed up its operation, hence the name heuristic constraint solver.

The AI Supported path selector is responsible for prioritizing the examination of execution branches on which we are more likely to find vulnerabilities. This is important because the generation of possible execution branches is extremely resource intensive. If we limit the execution time and the usable memory (which we often have to do), important errors may remain undiscovered in the absence of proper prioritization. We train the AI on a database that contains execution branches involving runtime vulnerabilities.



**Figure 21: Component model of AI4VULN**

### 3.4.4 Prospector Dataset

The paper from Sabetta et al. [86] focuses on the challenge of tracing vulnerabilities in open-source software projects and introduces a tool named Prospector, designed, and developed in the AssureMOSS [87] project to assist in this task. The core of the paper revolves around the development and evaluation of Prospector, a tool that utilizes heuristics inspired by human security experts to map vulnerability advisories to their corresponding fixes in source code.

Summary of the Dataset Used in the Paper:

- **Dataset Overview:** The paper discusses the utilization of a dataset comprising over 1300 vulnerabilities across various open-source projects. This dataset is fundamental in testing and validating the efficacy of Prospector.
- **Purpose of the Dataset:** It serves as a test bed for evaluating Prospector's ability to rank candidate commits in such a way that the actual fix for a given vulnerability is among the top-10 candidates in over 90% of the cases. This ability is crucial for reducing the manual effort involved in identifying the correct commits that fix reported vulnerabilities.
- **Scope:** It includes vulnerabilities from 723 open-source projects, developed in seven different programming languages.
- **Content:** Each record in the dataset contains detailed information about a vulnerability, including references to the commits that fix the vulnerability.
- **Usage:** The dataset was used in various configurations during the evaluation of Prospector, helping to determine the tool's effectiveness in different scenarios.
- **Benchmark for Tool Evaluation:** The dataset provides a comprehensive and varied set of real-world cases to assess Prospector's performance.
- **Demonstration of Practical Application:** By applying Prospector to this dataset, the researchers could demonstrate the tool's practical utility in an industrial context.

- **Contribution to Research:** The dataset represents a significant contribution to the field, offering a resource for further research and development of tools and methodologies in vulnerability management.

In essence, the dataset plays a critical role in the paper by providing the means to evaluate the Prospector tool's effectiveness in automating the process of linking known vulnerabilities in open-source software to their respective fixes in the source code. The success of Prospector in handling this dataset highlights its potential to significantly reduce the manual effort required in vulnerability management and improve the efficiency of security practices in open-source software development. In the AI4CYBER project we also leverage on this dataset for training our generic model about finding and fixing security vulnerabilities.

### 3.4.5 JiraMiner dataset

To be able to determine whether a commit is security relevant, we needed labeled data from a trustworthy source instead of manual labeling. Therefore, we built a tool which gathers examples from open-source projects about issue types and code commits.

The JiraMiner dataset is gathered from Apache projects available in JIRA issue tracking platform and their corresponding GitHub repositories for code.

The dataset is available at Zenodo [88]

#### 3.4.5.1 JIRA to GitHub

Each JIRA issue has a description field, which describes the problem and what to do in the issue. On the web frontend of JIRA an 'Issue Links' section can be seen which contains links to other issues, GitHub pull requests or commits. This field however is not available on the REST API (precisely, it is an empty list). In version 3 of the JIRA rest API, accessing remote links could be easier, but the Apache JIRA did not support that version of the API at the time of writing.

Every issue has a large number of custom fields as well, most of them are not filled out.

There are also comments and work log fields. The latter contains time tracking information about users and pull requests. Links to GitHub pull requests or individual commits can be in comments, work log entries and issue descriptions. The following problems arise:

- there aren't any GitHub links for every resolved issue,
- some projects do not use GitHub,
- a JIRA project can relate to multiple GitHub repositories,
- the solution is done in a fork, not in the original repository,
- pull requests are referenced in the work logs or comments, which do not solve the current issue, but relate in some way (e. g. the bug was accepted in that pull request or something was removed which was added in that pull request),
- there could be links to renamed or deleted repositories,
- individual GitHub files are referenced,
- individual files (patch files or complete source files) are uploaded into JIRA and attached to the issue,
- changelog for a larger version upgrade as a comment containing many irrelevant pull requests.



Some projects have a comment like “Issue resolved by pull request x”, which was not investigated. We gathered all the pull requests and commit links from the description, comments and work log fields and filtered out the duplicates. If a commit is referenced in a pull request and by itself as well, then it has to be filtered out. The problems mentioned above could make the dataset fill with irrelevant and misleading commits.

Examples:

- aardvark repository contains spam (<https://issues.apache.org/jira/projects/AAR/issues>)
- 657 (+1 TST=test project) JIRA projects of which 209 does not contain links to any GitHub pull request or commit

### 3.4.5.2 *GitHub to JIRA*

All the referenced GitHub repos from the JIRA projects were collected, which resulted in more than **6000** collected repos. Only those were kept, where the owner of the repo is “apache”, which resulted in **1354** repos. For these repositories all the pull requests were collected, and issue keys from the titles were extracted. If a pull request has a corresponding JIRA issue, then the pull request’s title likely starts with the issue’s key, e.g., “ARROW-18299: [CI][GLib][macOS] Fix dependency install failures”, where “ARROW-18299” is the issue’s key. Not every title starts with the issue key, therefore the first instance of a string matching “[A-Z]+-[0-9]+” was extracted. (Not only the first instance should be collected.) Many pull request titles reference UTF-8, CVEs, RFCs, JDK-s, which are obviously not issue keys.

Problems:

- Some issues are so new that they were not present in the downloaded JIRA issues.
- Some of these issue keys correspond to improvement proposals (e.g., SIP, RocketMQ Improvement Proposals, RIP) or enhancement proposals (Cassandra Enhancement Proposal, CEP), which are used for significant changes (not small fixes).
- Some projects use GitHub Issues to track bugs, improvements, features and reference them in the same format. (e.g., BUG-6331 referring to #6331 GitHub Issue with bug label)
- Often the project’s name in the issue key has a typo. E.g., BOOKKKEEPER instead of BOOKKEEPER.
- Projects tracked with Bugzilla, have a key starting with BZ.
- Apache AIRFLOW has many referenced JIRA issues, but the project has been deleted from Apache’s JIRA or it is restricted.
- APEX has been renamed APEXCORE, meaning that the JIRA issue keys were also renamed, which makes the older pull requests harder to match with the corresponding issue.

The mistyped issue keys are fixed by finding the closest match in the existing project, above a certain similarity threshold. In many cases, this is not enough, the issue keys have to be examined manually.

Number of pull requests	801199
Number of pull requests with a corresponding issue	314533

### 3.4.5.3 JSON format

The JIRAMiner dataset, focusing on JIRA-supported Apache projects (found at Apache JIRA<sup>5</sup>), utilizes a comprehensive JSON structure to encapsulate the collected information. This JSON structure is designed to accurately reflect the various components of a JIRA issue. The structure includes the following elements:

**Project:** (JIRA project<sup>6</sup>):

- self: JIRA REST API resource link
- id: JIRA id
- key: JIRA key
- url: project URL from JIRA if filled out
- issue: list of JIRA issues

**Issue:** (JIRA issue<sup>7</sup>):

- self: JIRA REST API resource link
- id: JIRA id
- key: JIRA key
- issuetype: type of issue (IssueType)
- description: issue description
- status: status (Status)
- prlinks: list of referenced pull request links
- commitlinks: list of referenced commit links
- GitHublinks: list of references GitHub links (currently not in use)
- commits: list of commits

**IssueType** (JIRA issuetype<sup>8</sup>):

- self: JIRA REST API resource link
- name: issuetype name

**Status** (JIRA status<sup>9</sup>):

- self: JIRA REST API resource link
- name: status name

**Commit** (GitHub commit<sup>10</sup>):

- sha: commit hash
- message: commit message:
- files: list of files associated with the commit
- pullrequest: the number of the pull request from which the commit was accessed
- path: path to the downloaded commit on disk

**File** (GitHub file description in a commit):

- sha: file hash
- filename: name of the file
- status: ("added", "removed", "modified", "renamed", "copied", "changed", "unchanged")
- contents\_url: GitHub REST API resource link to the file in the commit's state
- raw\_url: GitHub raw resource link to the file in the commit's state

<sup>5</sup> <https://issues.apache.org/jira>

<sup>6</sup> <https://docs.atlassian.com/software/jira/docs/api/REST/7.6.1/#api/2/project>

<sup>7</sup> <https://docs.atlassian.com/software/jira/docs/api/REST/7.6.1/#api/2/issue>

<sup>8</sup> <https://docs.atlassian.com/software/jira/docs/api/REST/7.6.1/#api/2/issuetype>

<sup>9</sup> <https://docs.atlassian.com/software/jira/docs/api/REST/7.6.1/#api/2/status>

<sup>10</sup> <https://docs.GitHub.com/en/rest/commits/commits>



**Project:**

- issues: **[Issue]**
  - issuetype: **IssueType**
  - status: **Status**
  - commits: **[Commit]**
  - files: **[File]**

**3.4.6 Identifying security relevance**

The commit classification activity which will be used for guiding symbolic execution to relevant code parts was initially done on the JiraMiner dataset. As a first attempt to build a proof-of-concept classifier, we have gathered numerous issues and filtered them based on keywords.

**Table 20: Keyword classes in JIRA**

Issue type	Number of issues
Bug	128152
Improvement	100921
Sub-task	38281
New Feature	23168
Task	20515
Test	4295
Dependency upgrade	1772
Wish	1443
Technical Debt	844
Documentation	685
Story	298
Epic	288
Dependency	66
Other/Misc	135

**Table 21: Number of issues, which have at least one associated java file**

Issue type	Number of issues
Bug	128064
Improvement	100834
New Feature	23181
Test	4298
Dependency upgrade	1772

Security-relevant keywords:

encryption, cve, security, secure, vuln, vulnerability, attack, defense, attacker, hardening

Similar keywords found in the descriptions:

secure, defense, hardening, cve, vulnearbility, security=, secures, defensive, attack, vulnerabilityid, attacks, vulnerability, secure, security, cves, attackers, vulnid, encryptions, encryption, -encryption, vuln, attacker, vulns, security/, harden

Security-relevant pull-request/issue pairs: **8225** (out of which **8210** have associated java files)

Classifier:

- Calculating code metrics for the pull requests and saving them in a separate database/csv for further operations. (This way the whole codebase only has to be processed once.)
- The commits for one pull request have to be merged into one, keeping only the last changes for each file. (The intermediate changes in commits have to be discarded, because the JIRA issues are associated with the pull requests, not the individual commits.)
- A separate one-class classifier (e.g., decision trees, random forest, SVM, Naive Bayes) for each issue type (e.g., for deciding whether it is an "Improvement" or not), or a multi-class classifier for the whole problem.
- A separate model for classifying security-relevant pull requests based on the same data using oversampling techniques to combat the high class-imbalance.

### 3.4.6.1 Metrics

Three types of representations were calculated for each file downloaded by the Apache-JIRA miner:

- class-level code metrics calculated using static analysis,
- a 10-dimensional embedding created using a Word2Vec model trained on Java code,
- a 100-dimensional embedding created using a Doc2Vec model trained on Java code.

The class-level metrics were calculated using the CK tool<sup>11</sup>, which was used in other software evolution related machine learning projects (e.g., change-proneness detection).

The 35 metrics are the following:

- *CBO (Coupling between objects)*: Counts the number of dependencies a class has. The tools check for any type used in the entire class (field declaration, method return types, variable declarations, etc.). It ignores dependencies to Java itself (e.g., java.lang.String).
- *CBO Modified (Coupling between objects)*: Counts the number of dependencies a class has. It is very similar to the CKTool's original CBO. However, this metric considers a dependency from a class as being both the references the type makes to others and the references that it receives from other types.
- *FAN-IN*: Counts the number of input dependencies a class has, i.e., the number of classes that reference a particular class. For instance, given a class X, the fan-in of X would be the number of classes that call X by referencing it as an attribute, accessing some of its attributes, invoking some of its methods, etc.
- *FAN-OUT*: Counts the number of output dependencies a class has, i.e., the number of other classes referenced by a particular class. In other words, given a class X, the fan-out of X is the number of classes called by X via attributes reference, method invocations, object instances, etc.
- *DIT (Depth Inheritance Tree)*: It counts the number of "fathers" a class has. All classes have DIT at least 1 (everyone inherits java.lang.Object). In order to make it happen, classes must exist in the project (i.e., if a class depends upon X which relies in a jar/dependency file, and X depends upon other classes, DIT is counted as 2).
- *NOC (Number of Children)*: It counts the number of immediate subclasses that a particular class has.
- *Number of fields*: Counts the number of fields. Specific numbers for total number of fields, static, public, private, protected, default, final, and synchronized fields.
- *Number of methods*: Counts the number of methods. Specific numbers for total number of methods, static, public, abstract, private, protected, default, final, and synchronized methods. Constructor methods also count here.
- *Number of visible methods*: Counts the number of visible methods. A method is visible if it is not private.
- *NOSI (Number of static invocations)*: Counts the number of invocations to static methods. It can only count the ones that can be resolved by the JDT.
- *RFC (Response for a Class)*: Counts the number of unique method invocations in a class. As invocations are resolved via static analysis, this implementation fails when a method has overloads with the same number of parameters, but different types.
- *WMC (Weight Method Class) or McCabe's complexity*. It counts the number of branch instructions in a class.
- *LOC (Lines of code)*: It counts the lines of count, ignoring empty lines and comments (i.e., it is Source Lines of Code, or SLOC). The number of lines here might be a bit different from the original file, as we use JDT's internal representation of the source code to calculate it.

---

<sup>11</sup> <https://GitHub.com/mauricioaniche/ck>

- *LCOM (Lack of Cohesion of Methods)*: Calculates LCOM metric. This is the very first version of metric, which is not reliable. LCOM-HS can be better (hopefully, you will send us a pull request).
- *LCOM\* (Lack of Cohesion of Methods)*: This metric is a modified version of the current version of LCOM implemented in CK Tool. LCOM\* is a normalized metric that computes the lack of cohesion of class within a range of 0 to 1. Then, the closer to 1 the value of LCOM\* in a class, the less the cohesion degree of this respective class. The closer to 0 the value of LCOM\* in a class, the more the cohesion of this respective class. This implementation follows the third version of LCOM\* defined in [89].
- *TCC (Tight Class Cohesion)*: Measures the cohesion of a class with a value range from 0 to 1. TCC measures the cohesion of a class via direct connections between visible methods, two methods or their invocation trees access the same class variable.
- *LCC (Loose Class Cohesion)*: Like TCC but it further includes the number of indirect connections between visible classes for the cohesion calculation. Thus, the constraint  $LCC \geq TCC$  holds always.
- *Quantity of returns*: The number of return instructions.
- *Quantity of loops*: The number of loops (i.e., for, while do while, enhanced for).
- *Quantity of comparisons*: The number of comparisons (i.e., == and !=). Note: != is only available in 0.4.2+.
- *Quantity of try/catches*: The number of try/catches
- *Quantity of parenthesized expressions*: The number of expressions inside parenthesis.
- *String literals*: The number of string literals (e.g., "John Doe"). Repeated strings count as many times as they appear.
- *Quantity of Number*: The number of numbers (i.e., int, long, double, float) literals.
- *Quantity of Math Operations*: The number of math operations (times, divide, remainder, plus, minus, left shift, right shift).
- *Quantity of Variables*: Number of declared variables.
- *Max nested blocks*: The highest number of blocks nested together.
- *Quantity of Anonymous classes, inner classes, and lambda expressions*: The name says it all. Note that whenever an anonymous class or an inner class is declared, it becomes an "entire new class", e.g., CK generates A.B and A. B\$C, C being an inner class inside A.B. However, lambda expressions are not considered classes, and thus, are part of the class/method they are embedded into. A class or a method only has the number of inner classes that are declared at its level, e.g., an inner class that is declared inside a method M2, that is inside an anonymous class A, that is declared inside a method M, that finally is declared inside a class C, will not count in class C, but only in method M2 (first-level method it is embodied), and anonymous class A (first-level class it is embodied).
- *Number of unique words*: Number of unique words in the source code. At method level, it only uses the method body as input. At class level, it uses the entire body of the class as metrics. The algorithm basically counts the number of words in a method/class, after removing Java keywords. Names are split based on camel case and underline (e.g., longName\_likeThis becomes four words). See WordCounter class for details on the implementation.
- *Number of Log Statements*: Number of log statements in the source code. The counting uses REGEX compatible with SLF4J and Log4J API calls. See NumberOfLogStatements.java and the test examples (NumberOfLogStatementsTest and fixtures/logs) for more info.



- *Has Javadoc*: Boolean indicating whether a method has javadoc. (Only at method-level for now)
- *modifiers*: public/abstract/private/protected/native modifiers of classes/methods. Can be decoded using `org.eclipse.jdt.core.dom.Modifier`.
- *Usage of each variable*: How often each variable was used inside each method.
- *Usage of each field*: How often each local field was used inside each method, local field are fields within a class (subclasses are not included). Also, indirect local field usages are detected, indirect local field usages include all usages of fields within the local invocation tree of a class e.g., A invokes B and B uses field a, then a is indirectly used by A.
- *Method invocations*: All directly invoked methods, variations are local invocations and indirect local invocations.

There were some difficulties with the tool, mainly that often it skipped many files in the same source folder. So, it had to be run multiple times on smaller groups of source files.

The Word2Vec model used for creating an embedding is the same, which was used in the Sorter (Patch prioritizer). A gensim<sup>12</sup> implementation was utilized, which was trained on a large Java corpus<sup>13</sup>. The model creates a 10-dimensional embedding for the individual tokens, which are then averaged to obtain a vector representation for a given code snippet.

The used Doc2Vec model is also the same, which was used in the Sorter. The gensim implementation was also trained in the same large Java corpus. Unlike the previous one, this model creates a document-level embedding (file-level in this case).

### 3.4.6.2 Training results

The metrics and embeddings were further aggregated to get a commit-level feature vector. Two types of simple aggregation were used, one is the averaging of the vectors, and the other is the summation of the vectors.

PyCaret, a low-code machine learning library, was used, which trains many different classifiers (KNN, Logistic regression, SVM, ridge classifier, random forest, ada boost, linear discriminant analysis, extra trees, gradient boosting, light gradient boosting, decision tree, naïve bayes, quadratic discriminant analysis) on the same dataset.

The results were similarly bad on all the datasets (class-level metrics, word2vec embedding, doc2vec embedding, all 3 at once), both commit aggregation techniques (averaging, summation) and classifiers. F1 score was between 0 and 0.2, recall was often below 10%, when recall was high, then accuracy was much worse. The task is hard because the dataset is very imbalanced, only a few thousand positive examples are there compared to the few hundred thousand negative examples. Even with imbalance fixing techniques (SMOTE) used by PyCaret, the results were similarly bad and practically unusable.

Pycaret results using the metrics datasets with averaging as aggregation method and with fixing imbalance.

---

<sup>12</sup> <https://radimrehurek.com/gensim/>

<sup>13</sup> <https://groups.inf.ed.ac.uk/cup/javaGithub/>



**Table 22: Metrics results**

Model	Accuracy	AUC	Recall	Precision	F1	Kappa	MCC
Logistic Regression	0.9953	0.4851	0.0000	0.0000	0.0000	-0.0000	-0.0001
K Neighbors Classifier	0.9953	0.5326	0.0032	0.2833	0.0063	0.0062	0.0291
Ridge Classifier	0.9953	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Ada Boost Classifier	0.9953	0.5759	0.0000	0.0000	0.0000	0.0000	0.0000
Dummy Classifier	0.9953	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000
Quadratic Discriminant Analysis	0.9952	0.4996	0.0000	0.0000	0.0000	-0.0002	-0.0006
Linear Discriminant Analysis	0.9952	0.5746	0.0000	0.0000	0.0000	-0.0002	-0.0005
Random Forest Classifier	0.9951	0.5618	0.0128	0.1921	0.0236	0.0230	0.0471
Light Gradient Boosting Machine	0.9951	0.5913	0.0040	0.0613	0.0075	0.0069	0.0145
Extra Trees Classifier	0.9950	0.5779	0.0287	0.2465	0.0512	0.0503	0.0823
Gradient Boosting Classifier	0.9949	0.5976	0.0024	0.0196	0.0043	0.0034	0.0054
SVM - Linear Kernel	0.9948	0.0000	0.0008	0.0009	0.0009	0.0002	-0.0000
Decision Tree Classifier	0.9916	0.5285	0.0335	0.0393	0.0360	0.0318	0.0320
Naive Bayes	0.1339	0.5281	0.9003	0.0048	0.0096	0.0003	0.0062

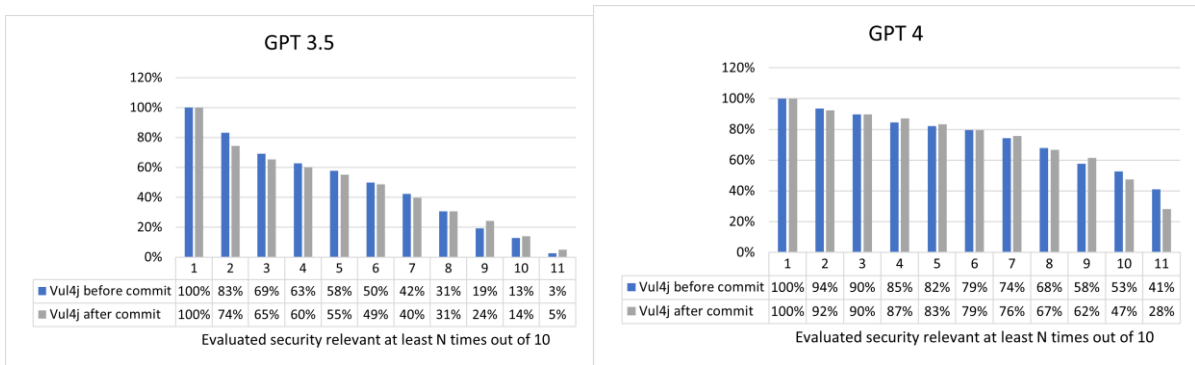
The results show that the approach which uses metrics derived from the source code and commit metadata does not meet the expected level of accuracy. Therefore, as future research we will include the source code (the amount of code required for accurate classification will be subject to research) in the classification data directly using LLM.

### 3.4.6.3 LLM-based identification of security-relevance (proof of concept)

Security relevance evaluation of code snippets with GPT-3.5 turbo and GPT-4 turbo was conducted with a prompt on the Vul4J dataset [90]

GPT was prompted 10 times on each entry of the dataset, once before commit (vulnerable) and once after the commit (corrected) and returned whether it found security relevant or not. It was prompted 10 times to try to minimize the randomness of GPT. Both GPTs evaluated approximately the same amount for the before and after commit code snippets without any

significant difference, so the code snippet does not provide good enough context to GPT to evaluate it from a prompt.



**Figure 22: Security-relevance classification**

We continued with fine-tuning GPT4 with OpenAI playground on MSR-20 dataset [91]:

200 C++ code snippets were provided, 50% insecure, 50% secure to create a fine-tuned GPT for binary classification, where 0 was secure and 1 was insecure. After evaluating it on 250 other code snippets it achieved 52% accuracy with 46.7% precision and 37.7% recall.

### 3.4.7 LLM-based Vulnerability detection (proof of concept)

To test the vulnerability detection capabilities of Large Language Models (LLMs), we forked the AutoGPT<sup>14</sup> project, which aims to develop an autonomous AI capable of performing various tasks without human intervention.

Like our approach in AI-based test generation (Section 2.1.4), we utilized the OWASP Java benchmark for this purpose. This benchmark comprises both vulnerable and non-vulnerable code snippets. Unlike in test generation, the AI's task here is to identify vulnerabilities without the necessity to exploit them. Given the benchmark's characteristics, detecting vulnerabilities posed a greater challenge for the AI compared to generating tests.

We used the following prompt:

You are VulnGPT, EvaluatorAI, whose task is to find vulnerabilities in a given source file. I will provide you with code snippets, and you will analyze them to identify potential security flaws or weaknesses. Your decisions must always be made independently without seeking user assistance. Play to your strengths as an LLM and pursue simple strategies with no legal complications.

**GOALS:**

1. Analyze the provided source code snippet for security vulnerabilities
2. Generate a list of findings and possible fixes
3. Upon request review your previous findings and refine your result according to that

**Constraints:**

1. ~4000 word limit for short term memory. Your short-term memory is short, so immediately save important information to files.
2. If you are unsure how you previously did something or want to recall past events, thinking about similar events will help you remember.

<sup>14</sup> <https://GitHub.com/Significant-Gravitas/AutoGPT>

## 3. No user assistance

## 4. Exclusively use the commands listed in double quotes e.g., "command name"

## Performance Evaluation:

1. Continuously review and analyze your actions to ensure you are performing to the best of your abilities.
2. Constructively self-criticize your big-picture behavior constantly.
3. Reflect on past decisions and strategies to refine your approach.
4. Every command has a cost, so be smart and efficient. Aim to complete tasks in the least number of steps.

## Result types:

VULNERABILITY: Exploitable vulnerability, such as buffer overflow.

POTENTIAL\_VULN: Potential vulnerability without evidence that it can be exploited.

IMPROVEMENT: Robustness or quality improvement, such as error handling and error messages.

You should only respond in JSON format as described below Response Format:

```
{
  "thoughts": {
    "text": "thought",
    "reasoning": "reasoning",
    "plan": "short, bulleted list that conveys long-term plan",
    "criticism": "constructive self-criticism"
  },
  "results": [
    {
      "short description": "name",
      "explanation": "explanation",
      "type": "vulnerability type"
    }
  ]
}
```

Ensure the response can be parsed by Python `json.loads`

Analyze the next source snippet, and respond using the format specified above:

<code snippet>

In the benchmark, each test initially appeared vulnerable, primarily because the exploitability hinged on the 'doSomething' function. Consequently, when processed as a one-shot request, Large Language Models (LLMs) identified every test case as either vulnerable or potentially so. This trend became more pronounced due to the placement of the 'doSomething' function after sections of code that appeared vulnerable. By the time the LLM encountered the 'doSomething' function, it had already assessed the preceding code for vulnerabilities.

To address this behavior, we employed a method of self-reflection. Our aim was to avoid revealing more information about the vulnerabilities than what was present in the raw source code. Therefore, we formulated simple reflection questions for each detected issue, tailoring these questions according to the type of problem reported. Depending on the reported problem type we used the following questions:



```
Are you sure about the vulnerability referenced as <id>?
<id> is really a vulnerability? Is it possible to exploit?
Are you sure about the potential vulnerability referenced as <id>?
Can the improvement referenced as <id> eliminate any potential vulnerability?
```

Using the self-reflection technique, the LLMs have a chance to successfully analyze the doSomething function as part of the whole task and provide a proper decision. However, in some cases, the analysis was proper, but the final decision was incorrect. For example, some of the tests passed the user input to the evaluation using the environment variables, which were executed using the following insecureCmd.sh script:

```
#!/bin/sh

echo Your original string:
eval $FOO
echo Your string reversed value is:
echo $FOO | rev
```

If we passed the content of this script also along with the Java code snippet, the LLM provided the following analysis after the self-reflection question:

```
Upon further review, the 'Command Injection' vulnerability is not exploitable in this case. The user input is passed as an environment variable (FOO) to the insecureCmd.sh script, which then uses 'eval' to execute the value of FOO. However, since the value of FOO is passed as an environment variable and not directly concatenated into the command string, it is not possible for an attacker to inject arbitrary commands.
```

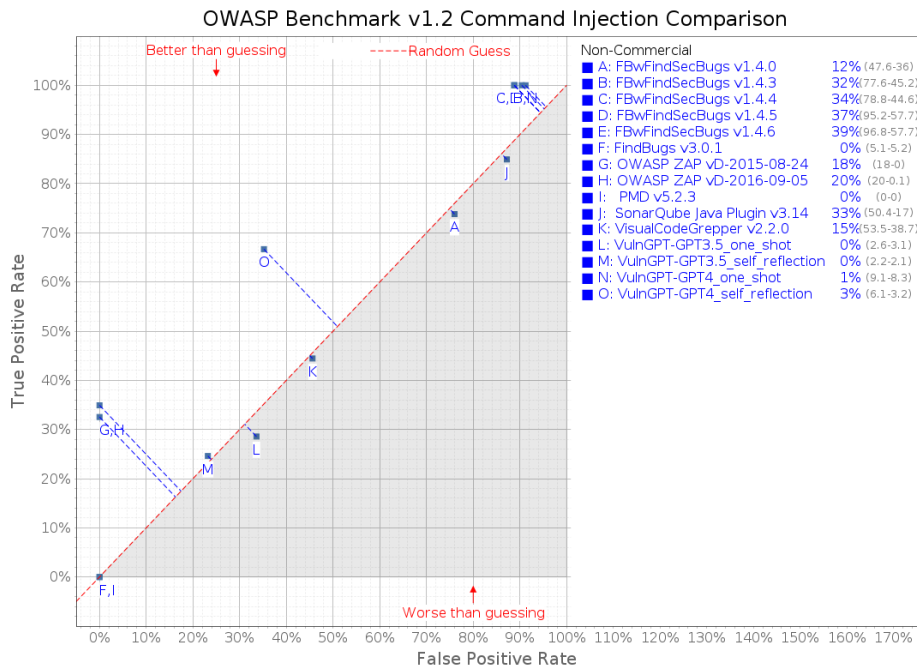
The analysis accurately identified that the user parameter could be passed to the command execution code, yet the assessment of exploitability was flawed.

Due to these identified issues, in the command injection test category the LLMs yielded the following detection statistics:

**Table 23: Command injection test results**

Command injection	Models			
	ChatGPT 3.5		ChatGPT 4	
Prompting	One-shot	Self-reflection	One-shot	Self-reflection
TP	27	23	92	62
TN	68	79	9	66
FP	35	24	94	37
FN	65	69	0	30
TPR	0.29	0.25	1	0.67
FPR	0.34	0.23	0.91	0.35
Precision	0.49	0.52	0.52	0.66
F1	0.36	0.34	0.68	0.66





**Figure 23: Scorecard for command injection results**

As demonstrated in Figure 23, GPT-4, enhanced with self-reflection capabilities, outperformed random guessing by a significant margin. This outcome underscores the promising potential of using Large Language Models (LLMs) in detecting vulnerabilities, especially in comparison to other available tools.

In addition to addressing command injection issues, we also tested the vulnerability detection capabilities of Large Language Models (LLMs) with weak random test cases. Here too, we encountered detection challenges. Although the LLMs identified the issue of insecure randomness from the descriptions, they did not categorize it as a vulnerability. Despite attempts to enhance detection by providing more detailed explanations about the need for secure randomness, there was no improvement in detection accuracy.

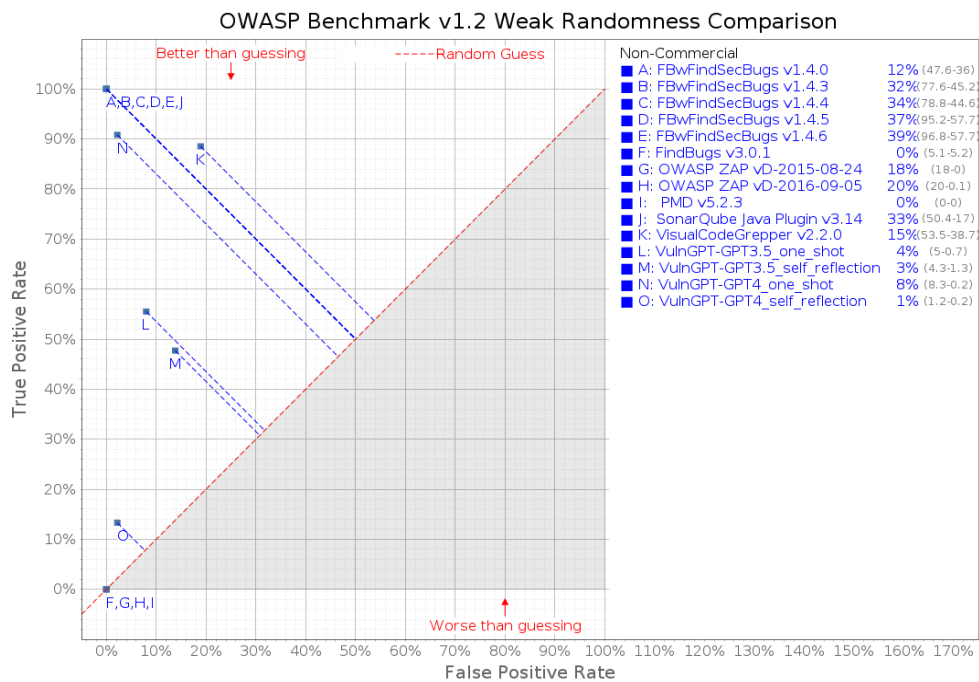
A typical instance where insecure randomness was correctly identified but not deemed significant involved cookie generation. In this scenario, secure randomness is crucial for cookie generation. However, since randomness was part of a test application where the cookies weren't used in any real operations, the issue was overlooked. This highlights a gap in the LLMs' ability to contextualize the importance of vulnerabilities based on their application environment.

Upon further review, the use of Math.random() in this specific context is for generating a 'rememberMeKey' value, which is not used for any security-critical operations such as encryption or authentication. Therefore, the use of Math.random() in this case does not pose a significant security risk, and the previous assessment of this as a vulnerability is not accurate.

Owing to these challenges, one-shot prompting produced superior results for both GPT-3.5 and GPT-4. However, the improvement was notably more significant with GPT-4, as this model demonstrated a markedly better understanding that the evaluated test case did not pose any real threat. As evident in the table below, the one-shot results from GPT-4 were highly accurate.

**Table 24: Testing one-shot and self-reflection techniques**

Weak random	Models			
	ChatGPT 3.5		ChatGPT 4	
Prompting	One-shot	Self-reflection	One-shot	Self-reflection
TP	90	78	201	30
TN	201	188	156	156
FP	18	31	4	4
FN	72	84	20	191
TPR	0.55	0.48	0.91	0.13
FPR	0.08	0.14	0.03	0.03
Precision	0.76	0.70	0.94	0.49
F1	0.64	0.57	0.92	0.21

**Figure 24: Scorecard for weak randomness results**

During our tests we tested the vulnerability detection capabilities for 381 test cases of using weak random. The presented results show that LLMs can perform vulnerability detection for the tested issue types with up to 0,92 F1 score. Individual test results may vary; however, the high amount of test cases imply that the expected performance is close to the observed performance.

### 3.5 AI4VULN Communication Interfaces

This subsection will describe the communication interfaces of AI4VULN. AI4VULN detects the vulnerability in the source code and the output contains position of the vulnerability and the trace information which leads the vulnerability. AI4VULN supports XML and JSON format. An example for XML output is the following:

```
<attribute type = "composite" name = "RH_CCE" context = "warning">
  <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
  <attribute type = "int" name = "Line" context = "" value = "18"/>
```

```

<attribute type = "int" name = "Column" context = "" value = "22"/>
<attribute type = "int" name = "EndLine" context = "" value = "18"/>
<attribute type = "int" name = "EndColumn" context = "" value = "40"/>
<attribute type = "string" name = "WarningText" context = "" value = "Cannot
cast &apos;next&apos; to type &apos;Integer&apos;.&#x0A;&#x0A;"/>
<attribute type = "composite" name = "ExtraInfo" context = "trace">
  <attribute type = "composite" name = "SourceLink" context = "">
    <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
    <attribute type = "int" name = "Line" context = "" value = "12"/>
    <attribute type = "int" name = "Column" context = "" value = "2"/>
    <attribute type = "int" name = "EndLine" context = "" value = "12"/>
    <attribute type = "int" name = "EndColumn" context = "" value = "23"/>
    <attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
  </attribute>
  <attribute type = "composite" name = "SourceLink" context = "">
    <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
    <attribute type = "int" name = "Line" context = "" value = "15"/>
    <attribute type = "int" name = "Column" context = "" value = "9"/>
    <attribute type = "int" name = "EndLine" context = "" value = "15"/>
    <attribute type = "int" name = "EndColumn" context = "" value = "12"/>
    <attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
  </attribute>
  <attribute type = "composite" name = "SourceLink" context = "">
    <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
    <attribute type = "int" name = "Line" context = "" value = "16"/>
    <attribute type = "int" name = "Column" context = "" value = "23"/>
    <attribute type = "int" name = "EndLine" context = "" value = "16"/>
    <attribute type = "int" name = "EndColumn" context = "" value = "31"/>
    <attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
  </attribute>
  <attribute type = "composite" name = "SourceLink" context = "">
    <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
    <attribute type = "int" name = "Line" context = "" value = "16"/>
    <attribute type = "int" name = "Column" context = "" value = "4"/>
    <attribute type = "int" name = "EndLine" context = "" value = "16"/>
    <attribute type = "int" name = "EndColumn" context = "" value = "34"/>
    <attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
  </attribute>
  <attribute type = "composite" name = "SourceLink" context = "">
    <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
    <attribute type = "int" name = "Line" context = "" value = "17"/>
    <attribute type = "int" name = "Column" context = "" value = "14"/>
    <attribute type = "int" name = "EndLine" context = "" value = "17"/>
    <attribute type = "int" name = "EndColumn" context = "" value = "21"/>
    <attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
  </attribute>
  <attribute type = "composite" name = "SourceLink" context = "">
    <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
    <attribute type = "int" name = "Line" context = "" value = "17"/>

```





```

<attribute type = "int" name = "Column" context = "" value = "10"/>
<attribute type = "int" name = "EndLine" context = "" value = "17"/>
<attribute type = "int" name = "EndColumn" context = "" value = "24"/>
<attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
</attribute>
<attribute type = "composite" name = "SourceLink" context = "">
  <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
  <attribute type = "int" name = "Line" context = "" value = "18"/>
  <attribute type = "int" name = "Column" context = "" value = "34"/>
  <attribute type = "int" name = "EndLine" context = "" value = "18"/>
  <attribute type = "int" name = "EndColumn" context = "" value = "38"/>
  <attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
</attribute>
<attribute type = "composite" name = "SourceLink" context = "">
  <attribute type = "string" name = "Path" context = "" value = "test1.java"/>
  <attribute type = "int" name = "Line" context = "" value = "18"/>
  <attribute type = "int" name = "Column" context = "" value = "31"/>
  <attribute type = "int" name = "EndLine" context = "" value = "18"/>
  <attribute type = "int" name = "EndColumn" context = "" value = "40"/>
  <attribute type = "int" name = "CallStackDepth" context = "" value = "0"/>
</attribute>
</attribute>
</attribute>

```

### 3.6 AI4VULN Installation and User Guide

VULHunter is a module of SourceMeter and cannot be executed without it. The detailed user can be found in the SourceMeter package, which can be downloaded from <https://sourcemeter.com/>

### 3.7 AI4VULN Unit Testing

This subsection provides the unit tests of AI4VULN.

**Table 25: System test case AI4VULN\_001**

<b>Test Case ID</b>	AI4VULN_001	<b>Component</b>	AI4VULN
<b>Req ID</b>	SR-011	<b>Priority</b>	High
<b>Description</b>	We measure TP, FP, TN, FN scores of AI4VULN on its regression test		
<b>Tested by</b>	FEA		
<b>Pre-condition(s)</b>	Regression tests for AI4VULN.		
<b>Test steps</b>			
<b>1</b>	We run AI4VULN on its regression tests		
<b>2</b>	We evaluate the results		

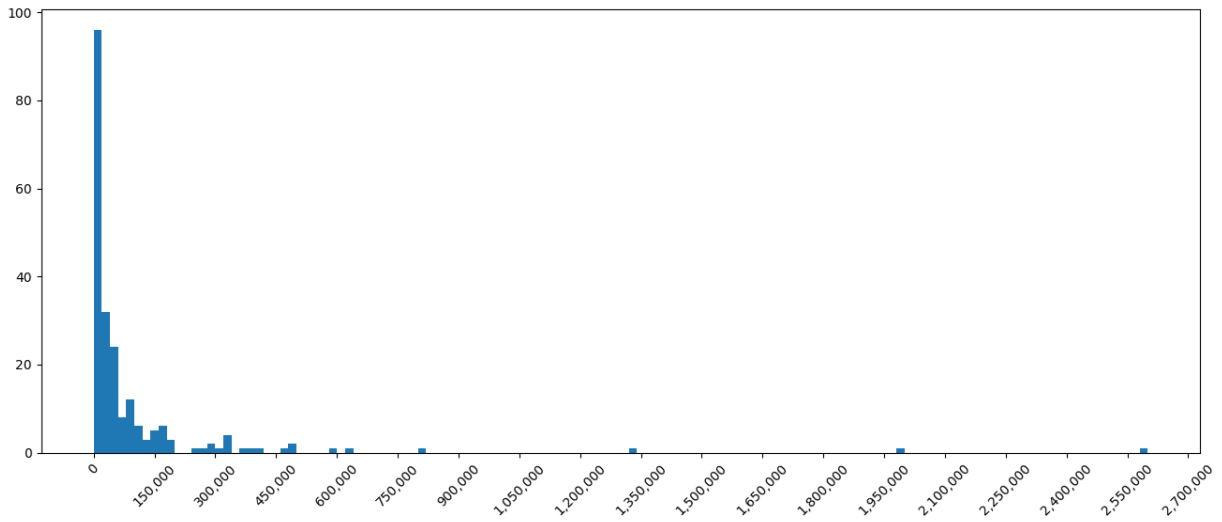
<b>3</b>	We measure the TP, FP, TN, FN scores
<b>Input data</b>	Regression test
<b>Result</b>	The improved version of AI4VULN on its regression test. The improvements can be measured.
<b>Test Case Result</b>	Achieved /Not achieved

**Table 26: System test case AI4VULN\_002**

<b>Test Case ID</b>	AI4VULN_002	<b>Component</b>	AI4VULN
<b>Req ID</b>	SR-011	<b>Priority</b>	High
<b>Description</b>	We measure the performance of AI4VULN on a large set of large open-source software.		
<b>Tested by</b>	FEA		
<b>Pre-condition(s)</b>	Collection of open-source software		
<b>Test steps</b>			
<b>1</b>	We run AI4VULN on a set of large open-source software		
<b>2</b>	We evaluate the results (TP and FP)		
<b>3</b>	We compare the execution time and used memory with the earlier version		
<b>Input data</b>	Collection of large open-source systems and the previous execution results		
<b>Result</b>	We can see the change in the performance		
<b>Test Case Result</b>	Achieved /Not achieved		

### 3.7.1 Validation on open-source projects

The evaluation of RTEHunter on open-source systems will be performed after the small-scale tests are completed. As a starting point, a database of 209 Java systems is already available. These systems are from different domains and their size range from a thousand to 2.5 million lines of code. The histogram of the LOC is presented in Figure 25.



**Figure 25: Lines of Code histogram of the examined Java projects**

Table 27 shows the minimum, maximum, and the average lines of code and the number of classes metric of the systems.

**Table 27: The Lines of Code and Number of Classes metrics of the examined Java projects**

	Lines of Code	Number of Classes
Minimum	993	17
Maximum	2,594,569	16,934
Average	140,653	1,090

### 3.7.1.1 Measuring the improvements

To verify the improvement of our approaches, we will execute both the original and the improved versions of RTEHunter on the Java systems. During the evaluation of the results, we will monitor changes in the detected runtime errors, i.e., the shifting of true positive and false positive alerts. We will also measure performance changes, such as changes in memory consumption and runtime.

## 4 Conclusions

This section concludes this deliverable summarizing the main results and future works.

The deliverable has presented the specification of the initial version of the two components in AI4CYBER Framework dedicated to AI-driven fixing of code and AI-driven vulnerability testing, AI4FIX and AI4VULN, respectively.

AI4FIX is built using a general software evolution model containing robustness improvements and a customized model used for improved test and fix generation capabilities. The initial results show that LLM-based solutions can carry out these tasks.

The initial version of AI4VULN is already able to detect bugs in the code but it is not able to follow the execution paths properly. In this project we improved the symbolic execution engine of the baseline solution so that it can find more bugs and the false positive rate is lower.

The tools are interconnected as the potential vulnerabilities detected by AI4VULN can be corrected by AI4FIX, therefore, the tools are implemented in a common framework of AI-driven code testing.

For the future version of the prototypes, based on the presented research and proof-of-concept results, we will implement locally deployable LLM-based version of AI4FIX and AI4VULN in a common framework. The context size required for improved results will be further studied.

For AI4VULN, after performing the small-scale java tests, we will also it on the large-scale, open-source Java projects. The focus will be not only on the accuracy of the error detection, but on the performance: the amount of memory consumed, and the time required for analysis. Once we get an idea of the current state of the system, we will delve into the development of constraint solver heuristics and AI-supported route selection.

The final version of the tools along with the validation results will be presented in D3.3. The validation of both components will be conducted in the context of selected Use case 2 banking applications, testing and fixing their code.

## References

- [1] “AI4CYBER”, Accessed: Jan. 29, 2024. [Online]. Available: <https://www.ai4cyber.eu>
- [2] K. Alrashedy and A. Aljasser, “Can LLMs Patch Security Issues?,” Nov. 2023, [Online]. Available: <http://arxiv.org/abs/2312.00024>
- [3] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, “Semantic Similarity Metrics for Evaluating Source Code Summarization,” *IEEE International Conference on Program Comprehension*, vol. 2022-March, pp. 36–47, Apr. 2022, [Online]. Available: <http://arxiv.org/abs/2204.01632>
- [4] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, “Mining Software Evolution to Predict Refactoring,” in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, IEEE, Sep. 2007, pp. 354–363. doi: 10.1109/ESEM.2007.9.
- [5] M. Alenezi, M. Akour, and O. Al Qasem, “Harnessing deep learning algorithms to predict software refactoring,” *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 18, no. 6, pp. 2977–2982, Dec. 2020, doi: 10.12928/TELKOMNIKA.v18i6.16743.
- [6] M. Aniche, E. Maziero, R. Durelli, and V. H. S. Durelli, “The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring,” *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1432–1450, Apr. 2022, doi: 10.1109/TSE.2020.3021736.
- [7] A. Barbez, F. Khomh, and Y. G. Gueheneuc, “Deep Learning Anti-Patterns from Code Metrics History,” in *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, Institute of Electrical and Electronics Engineers Inc., Sep. 2019, pp. 114–124. doi: 10.1109/ICSME.2019.00021.
- [8] E. Linstead, C. Lopes, and P. Baldi, “An application of Latent Dirichlet Allocation to analyzing software evolution,” in *Proceedings - 7th International Conference on Machine Learning and Applications, ICMLA 2008*, 2008, pp. 813–818. doi: 10.1109/ICMLA.2008.47.
- [9] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm, “Towards better understanding of software quality evolution through commit-impact analysis,” in *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017*, Institute of Electrical and Electronics Engineers Inc., Aug. 2017, pp. 251–262. doi: 10.1109/QRS.2017.36.
- [10] F. Alghanim, M. Azzeh, A. El-Hassan, and H. Qattous, “Software Defect Density Prediction Using Deep Learning,” *IEEE Access*, vol. 10, pp. 114629–114641, 2022, doi: 10.1109/ACCESS.2022.3217480.
- [11] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,” in *IEEE Transactions on Software Engineering*, 2008, pp. 485–496. doi: 10.1109/TSE.2008.35.
- [12] S. Delphine Immaculate, M. Farida Begam, and M. Floramary, “Software Bug Prediction Using Supervised Machine Learning Algorithms,” in *2019 International Conference on Data Science and Communication (IconDSC)*, IEEE, Mar. 2019, pp. 1–7. doi: 10.1109/IconDSC.2019.8816965.
- [13] S. Shivaji, E. James Whitehead, R. Akella, and S. Kim, “Reducing features to improve code change-based bug prediction,” *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, 2013, doi: 10.1109/TSE.2012.43.
- [14] S. Eski and F. Buzluca, “An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes,” in *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, 2011, pp. 566–571. doi: 10.1109/ICSTW.2011.43.

- [15] A. Parashar and J. K. Chhabra, “Mining software change data stream to predict changeability of classes of object-oriented software system,” *Evolving Systems*, vol. 7, no. 2, pp. 117–128, Jun. 2016, doi: 10.1007/s12530-016-9151-y.
- [16] OpenAI, “ChtGPT.” Accessed: Jan. 22, 2024. [Online]. Available: <https://chat.openai.com/>
- [17] Meta, “Code Llama.” Accessed: Jan. 22, 2024. [Online]. Available: <https://ai.meta.com/blog/code-llama-large-language-model-coding/>
- [18] Microsoft, “Copilot.”
- [19] Anthropic, “Claude.” Accessed: Jan. 22, 2024. [Online]. Available: [claude.ai](https://claude.ai)
- [20] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, “Automatic Program Repair,” *IEEE Softw*, vol. 38, no. 4, pp. 22–27, 2021, doi: 10.1109/MS.2021.3072577.
- [21] I. Paik and J.-W. Wang, “Improving Text-to-Code Generation with Features of Code Graph on GPT-2,” *Electronics (Basel)*, vol. 10, p. 2706, Jan. 2021, doi: 10.3390/electronics10212706.
- [22] M. Lajkó, V. Csuvi, and L. Vidács, “Towards JavaScript Program Repair with Generative Pre-Trained Transformer (GPT-2),” in *Proceedings of the Third International Workshop on Automated Program Repair*, in APR ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 61–68. doi: 10.1145/3524459.3527350.
- [23] P. Gyimesi *et al.*, “BugsJS: a Benchmark of JavaScript Bugs,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 90–101. doi: 10.1109/ICST.2019.00019.
- [24] D. and C. V. and V. L. Lajkó Márk and Horváth, “Fine-Tuning GPT-2 to Patch Programs, Is It Worth It?,” in *Computational Science and Its Applications – ICCSA 2022 Workshops*, B. and M. S. and R. A. M. A. C. and G. C. Gervasi Osvaldo and Murgante, Ed., Cham: Springer International Publishing, 2022, pp. 79–91.
- [25] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, in SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 55–56. doi: 10.1145/3135932.3135941.
- [26] J. A. Prenner, H. Babii, and R. Robbes, “Can OpenAI’s Codex Fix Bugs?: An evaluation on QuixBugs,” in *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2022, pp. 69–75. doi: 10.1145/3524459.3527351.
- [27] D. Sobania, M. Briesch, C. Hanna, and J. Petke, “An Analysis of the Automatic Bug Fixing Performance of ChatGPT,” in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 23–30. doi: 10.1109/APR59189.2023.00012.
- [28] Y. Charalambous, N. Tihanyi, Y. Sun, M. A. Ferrag, and L. Cordeiro, “A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification.” Jan. 2023. doi: 10.48550/arXiv.2305.14752.
- [29] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, “Examining Zero-Shot Vulnerability Repair with Large Language Models,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2339–2356. doi: 10.1109/SP46215.2023.10179324.
- [30] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2450–2462. doi: 10.1109/ICSE48619.2023.00205.



- [31] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery, 2023.
- [32] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *Proceedings of the 2014 international symposium on software testing and analysis*, Jan. 2014, pp. 437–440. doi: 10.1145/2610384.2628055.
- [33] C. Le Goues *et al.*, “The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015, doi: 10.1109/TSE.2015.2454513.
- [34] C. S. Xia and L. Zhang, “Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 959–971. doi: 10.1145/3540250.3549101.
- [35] Y. Wu *et al.*, “How Effective Are Neural Networks for Fixing Security Vulnerabilities,” *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, [Online]. Available: <https://api.semanticscholar.org/CorpusID:258967736>
- [36] 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 *Proceedings of the 19th International Conference on Mining Software Repositories*, in MSR ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 464–468. doi: 10.1145/3524842.3528482.
- [37] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, “Unit Test Case Generation with Transformers and Focal Context,” Sep. 2020, [Online]. Available: <http://arxiv.org/abs/2009.05617>
- [38] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation,” Feb. 2023, [Online]. Available: <http://arxiv.org/abs/2302.06527>
- [39] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. C. Lopes, “An Empirical Study of Using Large Language Models for Unit Test Generation,” Apr. 2023, [Online]. Available: <http://arxiv.org/abs/2305.00418>
- [40] D. Sobania, M. Briesch, and F. Rothlauf, “Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, in GECCO ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1019–1027. doi: 10.1145/3512290.3528700.
- [41] N. Al Madi, “How Readable is Model-Generated Code? Examining Readability and Visual Inspection of GitHub Copilot,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, in ASE ’22. New York, NY, USA: Association for Computing Machinery, 2023. doi: 10.1145/3551349.3560438.
- [42] T. Helmuth and P. Kelly, “PSB2: The Second Program Synthesis Benchmark Suite,” in *2021 Genetic and Evolutionary Computation Conference*, in GECCO ’21. Lille, France: Association for Computing Machinery, Jan. 2021. doi: 10.1145/3449639.3459285.
- [43] “7 AI-Powered Test Automation Tools That You Should Know in 2023.” [Online]. Available: <https://www.linkedin.com/pulse/7-ai-powered-test-automation-tools-you-should/>
- [44] OWASP, “OWASP Benchmark.” Accessed: Jan. 22, 2024. [Online]. Available: <https://owasp.org/www-project-benchmark/>



- [45] J. C. King, “Symbolic Execution and Program Testing,” *Commun ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, doi: 10.1145/360248.360252.
- [46] R. S. Boyer, B. Elspas, and K. N. Levitt, “SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution,” in *Proceedings of the International Conference on Reliable Software*, ACM, 1975, pp. 234–245. doi: 10.1145/800027.808445.
- [47] P. D. Coward, “Symbolic Execution Systems – a Review,” *Software Engineering Journal*, vol. 3, no. 6, pp. 229–239, Nov. 1988, doi: 10.1049/sej.1988.0029.
- [48] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI ’05. ACM, 2005, pp. 213–223. doi: 10.1145/1065010.1065036.
- [49] K. Sen and G. Agha, “CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools,” in *Proceedings of the 18th International Conference on Computer Aided Verification*, in CAV’06. Springer-Verlag, 2006, pp. 419–423. doi: 10.1007/11817963\_38.
- [50] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *Proceedings - International Conference on Software Engineering*, 2007, pp. 416–425. [Online]. Available: [www.scopus.com](http://www.scopus.com)
- [51] C. Cadar, D. Dunbar, D. R. Engler, and others, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.,” in *OSDI*, 2008, pp. 209–224.
- [52] F. Mues Malte and Howar, “JDart: Dynamic Symbolic Execution for Java Bytecode (Competition Contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, 2020, pp. 398–402.
- [53] J. He, G. Sivanrupan, P. Tsankov, and M. T. Vechev, “Learning to Explore Paths for Symbolic Execution,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2021, pp. 2526–2540.
- [54] K. Havelund and T. Pressburger, “Model checking JAVA programs using JAVA PathFinder,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000, doi: 10.1007/s100090050043.
- [55] C. S. Păsăreanu and N. Rungta, “Symbolic PathFinder: Symbolic Execution of Java Bytecode,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, in ASE ’10. ACM, 2010, pp. 179–180. doi: 10.1145/1858996.1859035.
- [56] M. Souza, M. Borges, M. D’Amorim, and C. S. Păsăreanu, “CORAL: Solving complex constraints for symbolic pathfinder,” in *Lecture Notes in Computer Science*, vol. 6617 LNCS, Springer, Berlin, Heidelberg, 2011, pp. 359–374. doi: 10.1007/978-3-642-20398-5\_26.
- [57] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, and I. Finocchi, “A Survey of Symbolic Execution Techniques,” *arXiv preprint arXiv:1610.00502*, 2016.
- [58] A. Amighi, P. de C. Gomes, D. Gurov, and M. Huisman, “Provably correct control flow graphs from Java bytecode programs with exceptions,” *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 6, pp. 653–684, 2016, doi: 10.1007/s10009-015-0375-0.
- [59] T. and P. D. Demange Delphine and Jensen, “A Provably Correct Stackless Intermediate Representation for Java Bytecode,” in *Programming Languages and Systems*, K. Ueda, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 97–113.
- [60] A. Viet Phan, M. Le Nguyen, and L. Thu Bui, “Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction,” in *2017 IEEE 29th International*

- Conference on Tools with Artificial Intelligence (ICTAI)*, 2017, pp. 45–52. doi: 10.1109/ICTAI.2017.00019.
- [61] “The SourceMeter Home Page.” [Online]. Available: <https://www.sourcemeter.com>
- [62] I. Kádár, P. Hegedűs, and R. Ferenc, “Runtime exception detection in Java programs using symbolic execution,” *Acta Cybernetica*, vol. 21, no. 3, pp. 331–352, 2014, [Online]. Available: [www.scopus.com](http://www.scopus.com)
- [63] D. A. Ramos and D. Engler, “Under-Constrained Symbolic Execution: Correctness Checking for Real Code,” in *24th USENIX Security Symposium (USENIX Security 15)*, USENIX Association, 2015, pp. 49–64. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos>
- [64] R. Ferenc, Á. Beszedes, M. Tarkiainen, and T. Gyimóthy, “Columbus Reverse Engineering Tool and Schema for C++,” in *Proceedings of the 18th International Conference on Software Maintenance (ICSM02)*, IEEE Computer Society, Oct. 2002, pp. 172–181.
- [65] F. E. Allen, “Control Flow Analysis,” *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970, doi: 10.1145/390013.808479.
- [66] R. J. Vanderbei, *Linear programming - foundations and extensions*, vol. 4. in Kluwer international series in operations research and management service, vol. 4. Kluwer, 1998.
- [67] J. C. Nash, “The (Dantzig) Simplex Method for Linear Programming,” *Computing in Science and Eng.*, vol. 2, no. 1, pp. 29–31, 2000.
- [68] A.~Townsend, “Top 10 algorithms from the 20th century.” 2016. [Online]. Available: <https://pi.math.cornell.edu/~ajt/presentations/TopTenAlgorithms.pdf>
- [69] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [70] S. A. Cook, “The Complexity of Theorem-Proving Procedures,” *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, vol. 43, pp. 143–152, 2023.
- [71] B. A. Trakhtenbrot, “A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms,” *IEEE Ann. Hist. Comput.*, vol. 6, no. 4, pp. 384–400, 1984.
- [72] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability - Second Edition*, vol. 336. in *Frontiers in Artificial Intelligence and Applications*, vol. 336. IOS Press, 2021.
- [73] M. Heule, “Introduction to Mathematics of Satisfiability, Victor W. Marek, Chapman & Hall/CRC, 2009. Hardback, ISBN-13: 978-143980167-3, 89.95,” *Theory Pract. Log. Program.*, vol. 11, no. 1, pp. 126–130, 2011.
- [74] L. M. de Moura, “Invited talk: Developing Efficient SMT Solvers,” *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories, Bremen, Germany, 17th July 2007*, vol. 257, 2007.
- [75] N. S. Bjørner, “SMT Solvers for Testing, Program Analysis and Verification at Microsoft,” *11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2009, Timisoara, Romania, September 26-29, 2009*, p. 15, 2009.
- [76] J. Vanegue and S. Heelan, “SMT Solvers in Software Security,” *6th USENIX Workshop on Offensive Technologies, WOOT’12, August 6-7, 2012, Bellevue, WA, USA, Proceedings*, pp. 85–96, 2012.
- [77] Z. Chen *et al.*, “Synthesize Solving Strategy for Symbolic Execution,” *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 348–360, 2021.
- [78] “The Yices SMT Solver.” [Online]. Available: <https://yices.csl.sri.com/>

- [79] A. Cheshkov, P. Zadorozhny, and R. Levichev, “Evaluation of ChatGPT Model for Vulnerability Detection.” 2023.
- [80] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, “Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT.” 2023.
- [81] M. Omar, “Detecting software vulnerabilities using Language Models.” 2023.
- [82] A. Tanwar, K. Sundaresan, P. Ashwath, P. Ganesan, S. K. Chandrasekaran, and S. Ravi, “Predicting Vulnerability In Large Codebases With Deep Code Representation.” 2020.
- [83] E. First, M. Rabe, T. Ringer, and Y. Brun, “Baldur: Whole-Proof Generation and Repair with Large Language Models,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1229–1241. doi: 10.1145/3611643.3616243.
- [84] D. Noever, “Can Large Language Models Find and Fix Vulnerable Software?”
- [85] Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang, “How Far Have We Gone in Vulnerability Detection Using Large Language Models,” Nov. 2023, [Online]. Available: <http://arxiv.org/abs/2311.12420>
- [86] A. Sabetta *et al.*, “Known Vulnerabilities of Open-Source Projects: Where Are the Fixes?”
- [87] “AssureMOSS.” Accessed: Jan. 08, 2024. [Online]. Available: [assuremoss.eu](http://assuremoss.eu)
- [88] “JiraMiner dataset.” Accessed: Jan. 29, 2024. [Online]. Available: [10.5281/zenodo.10457999](https://zenodo.org/record/10457999)
- [89] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, “Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design),” *Object Oriented Syst.*, vol. 3, pp. 143–158, 1996, [Online]. Available: <https://api.semanticscholar.org/CorpusID:40937518>
- [90] “Vul4J.” Accessed: Jan. 05, 2024. [Online]. Available: <https://github.com/tuhh-softsec/vul4j>
- [91] “MSR-20.” Accessed: Jan. 05, 2024. [Online]. Available: [https://github.com/ZeoVan/MSR\\_20\\_Code\\_vulnerability\\_CSV\\_Dataset/tree/master](https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset/tree/master)