International Journal of Enhanced Research in Management & Computer Applications
ISSN: 2319-7471, Vol. 12 Issue 8, August-2023, Impact Factor: 7.751

Improving Software Vulnerability and Cybersecurity Threat Detection and Management with the Use of Large Language Models

Geetha Priya D T[1], Dr. Rajesh Koolwal[2]

[1]Research Scholar, Department of Computer Science, JS University, Shikohabad, Uttar Pradesh
[2]Assistant Professor & Supervisor, Department of Computer Science, JS University, Shikohabad, Uttar Pradesh

ABSTRACT

The area of software vulnerability and cybersecurity jobs has found formidable tools in Large Language Models (LLMs) due to its capacity to discover and mitigate security threats. This article discusses many cybersecurity uses of LLMs, including vulnerability discovery, threat prediction, and automated code repair. We begin with a definition of LLMs, go over some of their various applications, and conclude with a literature review that ranks their strengths and weaknesses. We examine the performance of many LLMs on tasks like code summarization and virus detection to show their effectiveness in different areas of cybersecurity. When compared to more traditional approaches, our study demonstrates that LLMs are superior at discovering security issues and providing remedies. Our main emphasis is on the LLM model process and its potential integration with incident response systems and cyber threat detection frameworks. In order to further enhance LLMs' performance, we also cover supplemental approaches and resources, like static and dynamic code analyzers. We also gather data from previous research to demonstrate how LLMs have made software vulnerability and cybersecurity risk detection and mitigation much more efficient. Finally, the paper concludes with suggestions for enhancing LLM implementation, based on prior research.

Keywords: LLMs, Security, Cyber threats, Vulnerabilities, cybersecurity.

INTRODUCTION

Problems with the program's implementation might manifest as a loss of data integrity, slowdowns in performance, or even exploitable security holes. This shows how critical it is to identify and resolve errors quickly. When it comes to efficiently finding software defects, both older techniques, like static analysis, and newer ones, like deep learning-based approaches, have their limitations. While static analysis and DL techniques do a good job of identifying bugs early on, they have a lot of drawbacks, such a high false positive rate and problems with interpretability and feature selection. Developer productivity might take a major hit when these false positives occur.

The latest advances in deep learning (DL) and its potential to address long-standing issues have piqued the curiosity of many in the software engineering community. Many have proposed using Deep Learning (DL) techniques to resolve software issues. To explicitly address these challenges, notable examples are DeepRepair [49] and DLFix [48] that apply DL. In order to comprehend and reproduce the code structure, CURE [50] use a deep learning model that has been trained on a corpus of source code. To enhance code context awareness, DEAR [51] uses a hybrid approach, combining deep learning with spectrum-based defect localization. More research is required to weigh the benefits and drawbacks of each of these approaches in order to determine the optimal strategies for DL-based program defect elimination.

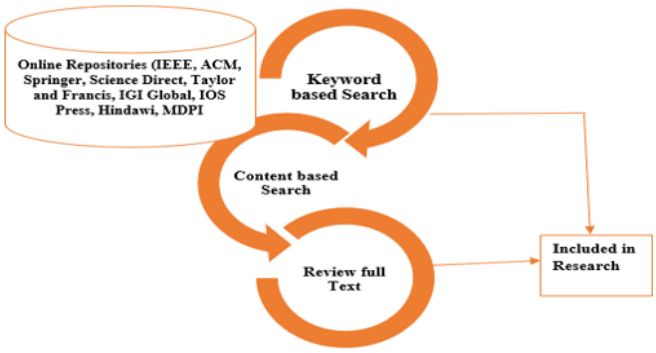
But the arrival of Large Language Models (LLMs) powered by the revolutionary Trans-formers architecture has shook up Natural Language Processing (NLP). A new era of capabilities and applications has dawned, resulting in incredible advancements, all because of this. There has been a lot of recent research showing that LLMs can find security flaws better than deep learning (DL) and conventional methods. [9]

LITERATURE REVIEW

Software bug reports are among the most important components of software. A variety of tracking systems, such as Jira, Bugzilla, Trac, and others, ensure that they are always up to date. Important information on the severity, importance, and remedy is provided by them if a software feature fails to function properly. In addition to that, it includes specifics of the input that was supplied by a number of programmers in order to fix the problem. The purpose of this study is to explore software libraries via the use of text analytics. In this study, a number of different text mining strategies are investigated. These techniques include data extraction, categorization, retrieval, and summarization from issues that have been reported with software. All of the information that is obtained from software bug reports is arranged in a manner that makes it simple to locate the many aspects that are essential. Text mining and machine learning are two methods that are used in the process of developing models for bug prediction. Ontology creation is used to generate models that are able to extract information from the text of software problem reports. The material is then summarized once the models have been constructed. At the end of the chapter, a summary of the findings that were obtained by searching through a huge body of literature on software flaws using a variety of text mining approaches is presented.

METHODOLOGY

Within the Issue Tracking System, a broad range of software assets, such as bug reports, source code, change logs, and many more, are meticulously recorded in great detail. Throughout the years, a great deal of investigation has been conducted on these topics. The process of manually removing these software items from issue tracking repositories is one that is both hard and time-consuming yet ultimately worthwhile. Given the unorganized nature of the data pertaining to software problems, it may be difficult to extract insights that can be put into action. Those who are knowledgeable in the field of automated data extraction have been searching for a way to circumvent this laborious procedure. InfoZilla is a program that was created by N. Bettenburg and his colleagues in order to extract structural aspects from problem reports. These elements include source code, stack traces, and revisions inside the report. When it came to teaching machine learning, it was assumed that using these components would result in an improvement in its performance. Filters such as patch filters, enumeration filters, and stack traces were used in this process. The performance of the tool was evaluated by looking at the bug reports that were submitted to Eclipse [37]. It was hypothesized by M. Nayrolles and colleagues that a mechanism might be used to extract data from bug report and version control systems. They called this approach BUMPER, which stands for Bug Metarepository for Developers and Researchers. It is a web program that is open-source and free to use, and it collects data from queries that are made by users. Inquiries may be classified as either simple or advanced, depending on the level of complexity they need. Gnome, Eclipse, Netbeans, and the Apache Software Foundation are the five distinct platforms that are compatible with this bug reporting application. The fact that a comprehensive understanding of query language is necessary in order to retrieve data from this tool is the most significant disadvantage that it has [38]. According to the findings of Y. Yuk and colleagues, there is a way to get reports of bugs. The proposed method involves first obtaining raw data from various sources, then crawling that data, and then translating it into a data tree layout. XML parsers worked their way through these data trees and retrieved the relevant symbols for each and every problem report [39]. In order to provide an alternative to the traditional technologies that extract textual information from problem reports, R. Malhotra and his colleagues created the Configuration Management System (CMS). Through the use of CVS log files, this tool is able to ascertain the total number of mistakes that have occurred in each category. Additionally, it establishes which versions of the program were altered and the extent of those modifications [40]. The efforts that have been made in the past to find and remove software artifacts are summarized in Table 2.1.



AUTOMATED BUG ANALYSIS

A software bug is defined as a mistake that causes the system to act in a way that is significantly different from what was intended [67]. Those that file bug reports may provide you with information on bugs. When a problem is found with a piece of software while it is being used, a bug report is sent to the appropriate authorities. There are a variety of information that are included in each bug report, including the date, version, problem description, severity, and priority. It is possible for specialists to examine bug reports with the assistance of code traces and a comprehensive explanation of the problem expressed in language that is commonplace. A significant number of programmers provide their recommendations in the form of helpful comments in order to either fix or investigate each problem in further depth. Bug report tracking systems are used by a number of companies in order to monitor and handle a wide variety of software project difficulties. This makes maintaining software much easier. These strategies are helpful because they make it easier for participants in the project, such as developers, testers, and others, to communicate and share a wide range of concerns. Among the many different sorts of challenges that might occur, some examples include bug reports, requests for modifications, additions, additional tasks, and subtasks. These are just a few instances. Jira, Bugzilla, Trac, and Mantis are just few of the many various bug tracking systems that are now available. In spite of this, Git and Bugzilla stand out as the most well-known and extensively used of the available options. Bug reports are a potential source of information that might be useful.

A. Block Diagram for Data Extraction

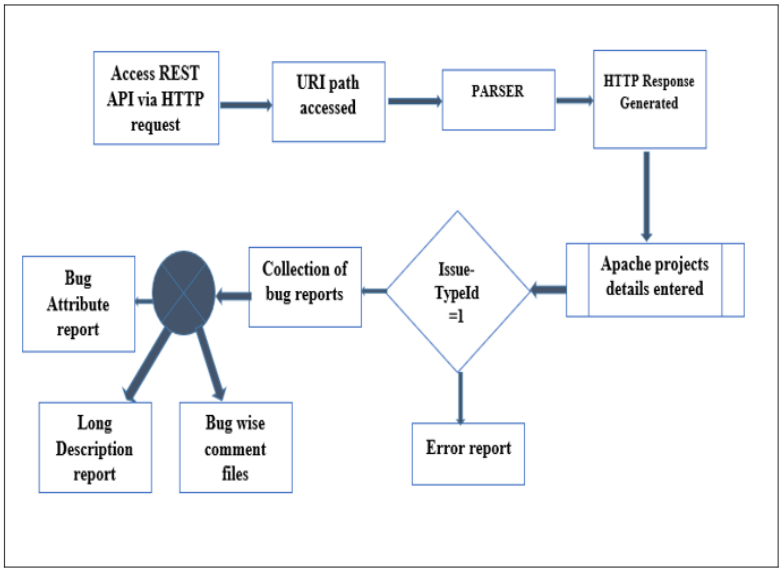


Fig. 1 Procedure for Extracting Data

The block diagram for data extraction represents a step-by-step flow of how raw data is collected, processed, and prepared for analysis. It begins with the data source, such as databases, issue tracking systems, or online repositories, from which relevant data is extracted automatically using extraction tools or APIs. The collected data then moves to a preprocessing block, where it is cleaned, filtered, and formatted to remove duplicates and inconsistencies. After preprocessing, the structured data is stored in a database or data warehouse for efficient access. Finally, the processed data is sent to the analysis or visualization block, where meaningful insights are generated for decision-making.

B. Finding and Fixing Software Issues

Through an examination of the textual descriptions of bug reports and the comments made by a variety of writers, this section investigates software flaws that have been found in a number of Apache Software Foundation (ASF) projects. When significant problems are not addressed and fixed, the usefulness and efficiency of a system are reduced, as stated by resolution. In order to make the system more efficient and effective, it is essential to determine the most serious problems that have not been handled and then allocate them to different members of the staff. Section 3.5.1 enumerates the most critical issues that have not yet been resolved for each and every project that is managed by the Apache Software Foundation. Section 3.5.2 identifies the employees who have made the most contributions to each project in order to resolve the most critical issues. This is done in order to accomplish the aforementioned goal.

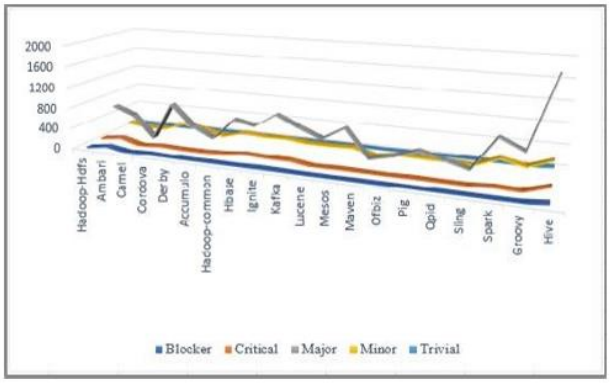


Fig. 2. Quantity of Unresolved Bugs with Varying Levels of Severity

C. Software System Areas Vulnerable to Errors

An analysis of linguistic components, such as a one-line explanation, a comprehensive description, and a large number of notes, is required in order to locate the problem areas that occur most often in bug reports. In the field of text mining, machine learning is used for a variety of purposes, including the classification of material into categories. In order to obtain a feature vector for use in automated document sorting, it is necessary for a document to go through pre-processing steps. In the field of text mining, pre-processing is the process of locating the words, phrases, and characters that will be used as building blocks in following operations. The process consists of three stages: breaking, tokenization, and the elimination of stop words. 1. Words, phrases, keywords, and other string sequences may be "tokenized" by breaking them down into smaller parts. This process applies to all string sequences. Text mining is an essential process because it eliminates commas and marks, converts all capital letters to lowercase, and transforms all capital letters to lowercase. It is via the use of text that tokens, which are units of activity, are created. When you want to employ this phrase, you should say "Have a good day!" while using the terms "have," "a," "good," and "day." (2) Getting rid of terms that aren't necessary: The term "stop words" refers to a small group of words that are used often in any language. Many of the words that are used often in literature have no other function except to act as conjunctions, prepositions, adjectives, or adjectivals. Some examples of these words are "and," "are," "this," "the," and "in." Due to the fact that they are inefficient as document organizers, it is necessary to dispose of them. It is necessary to eliminate stop words in order to enhance the efficiency of the system, concentrate your attention on the most significant phrases, and lower the amount of text data. The objective of the third step, which is referred to as stemming, is to separate the most essential components of a variety of word and grammatical types, including nouns, adjectives, verbs, and other similar types of words. The purpose is to reduce all of the word forms, including their inflated and original variants, to a single standard while maintaining their simplicity. Some examples of suffixes that may be removed are -ly, -ed, -ing, -er, and others. Your memory will increase and your stress level will decrease if you use this strategy while playing word search.

Categorization of Error	Significant Keywords
Logical code error	Throws, exception, divide, unhandled, pointer, throw, uncaught, null pointer, raises, systemoutofmemoryexception, trigger, dividezero, rethrow
Input/output error	Logging, build, classpath, inputformat, api-jarfilemissing, log, loggers, imports, initialized, requestchannels, mapoutputlocationgetfile, displayed, console log
Resource Allocation error	Task_allocation, buffers, memory, synchronized, configuration, memoryfailure, runtime, dynamic, bucketcache
Network Related error	Datanode, localhost, address, port, domain, security, process, https, global, interfaces, binding, virtual, bindexcept, limits

TEXT-BASED BUG SEVERITY PREDICTION

Due to the fact that text mining was first used to the classification process, the bulk of this study is concentrated on that particular aspect. Categoryization is the process of quickly categorizing text into specified categories, and the name "categorization" refers to this activity. For example, Jira and Bugzilla are examples of bug tracking systems that enable users to report software problems. Due to the fact that the situation is so dire, the problem with the software system, as well as its magnitude and relevance, are readily apparent. It is possible that considerably improving the process of discovering

and fixing software mistakes may be accomplished by determining the severity of the defects in advance. In spite of the researchers' best efforts, they have been constrained by a number of challenges that have stopped them from establishing the most effective approach of using text mining to forecast the quality of software.

The severity of a software problem is sometimes evaluated by comparing it to two extremes: important cases and instances that are not considered to be serious. Major faults, critical defects, and stopper flaws are the three categories that classify problems that are considered to be very serious [10]. The fact that the bugs are so little and difficult to see does not appear to bother anybody. It is possible for us to evaluate the quality of the program based on the one-line textual descriptions of software defect reports. [10–12], [34], [42], [47–48], [51], [68–69], [132–134] are some examples of the ways in which researchers have classified problems according to their level of severity using text mining and machine learning. The information was extracted from a substantial body of published material on the subject of using text mining as a method for assessing the quality of software. The application of a number of different machine learning algorithms to a variety of datasets is what is known as practical research. Many different types of learning algorithms are covered in this category, such as decision trees [64], Bayesian learners [135], neural networks [43], ensemble learners [136], support vector machines [60], and a considerable number of others. There is still a lack of knowledge on which machine learning methods are the most effective for estimating the quality of text mining software.

A. Dataset Features and Empirical Evidence

The Bug Report Collection System (BRCS) is a tool that is used for the purpose of gathering information about Apache projects. This information is given in Section 3.2 of Chapter 3. With regard to the selection of datasets, the dataset that has the greatest amount of mistakes is employed. There were thirteen datasets in the Jira folder that had the largest amount of issues. These datasets were Hadoop-HDFS, Hadoop-Common, Apache Cordova, Hive, Ambari, Groovy, Hbase, Lucene, Mesos, Maven, Qpid, Sling, and Spark. The reports that were obtained may be divided into two basic categories: those that address serious bugs and those that do not include severe bugs. It is possible to observe in Table 4.1 that each dataset has been allocated a number of concerns, some of which are significant while others are not as significant. When attempting to estimate the quality of the program at the system level, it is necessary to count the number of errors that are found in each sample. Due to the fact that the found defects are also organized by component, you are able to gain an idea of how serious the software problem is at the component level. In order to determine the extent of the problem, it is possible to choose the components that are the most faulty from among many datasets. The bar graphs that are shown in Table 4.2 and Figure 4.1(a-d) illustrate the projects and the issue regions that are associated with each of them.

Table 1. Software defect statistics broken down by system level severity

Project Name	#of bug reports		Total # of bug reports	Period of collection
	Non-severe	Severe		
Hadoop-hdfs	787	2622	3409	2007-2016
Hadoop-common	1021	4045	5066	2006-2016
Apache cordova	1124	3412	4536	2011-2016
Hive	730	5303	6033	2008-2016
Ambari	261	10156	10418	2011-2016
Groovy	721	3249	3970	2003-2016
Hbase	1464	4618	6082	2007-2016
Lucene	575	1321	1896	2005-2016
Mesos	253	1749	2002	2011-2016
Maven	286	1731	2017	2004-2016
Qpid	581	2665	3246	2006-2016
Sling	431	1727	2158	2007-2016
Spark	921	3323	4244	2012-2016

B. Ontology for Software Bugs Information Extraction using Deductive Reasoning

Over the last several years, ontologies have been more important inside the semantic web because of their ability to ease communication and data sharing. The semantic web, which was established by Gruber, converts disorganized content into structured information that computers can handle with little to no assistance from humans [155]. "A tool that retrieves ontology-based queries" is another way to refer to a semantic search engine. It is possible for us to create a conception of essential knowledge and arrive at a shared understanding of topics by using ontologies. It is possible for an ontology to serve as a means of characterizing an overall viewpoint on the nature of textual meaning analysis. It provides a formal description of knowledge in a particular area by making use of a compilation of ideas and the relationships between them. This idea is described by the term "ontology" [156]. Standard languages, such as Resource Description Framework (RDF) and Ontology Web Language (OWL), make it easier to retrieve data, describe domains, and communicate across different systems when they are paired with ontology. Despite the fact that ontology has the ability to reestablish knowledge of the issue, several sectors still have difficulty identifying concepts and the ways in which they are interrelated [18–20].

Ontology enables the organizing of information and the construction of a uniform language for identifying data when it is applied to the information domain [157]. The single most important aspect of developing and maintaining software in the field of software engineering is the reporting of software issues. They include information such as the project name, the name of the component, the release version, a short description, and other data about software problems, as well as the BugId. Feedback and in-depth details of how to deal with a problem are provided by a variety of authors. The investigation of unorganized bug reports is exhaustive and includes attempts to categorize complaints [42, 160], triage bugs [69, 158], predict flaws [132], evaluate the severity of reports [10, 48, 53, 159], and many other efforts. Acquiring thorough technical information pertaining to the problem is only one of the many uses of bug reports that are not exploited to their full potential. Therefore, it would be a good idea to produce a list of all the people who have helped to resolve the issue that is known as Hdfs-7707. Identities should be supplied for every individual who had a role in the resolution of #Hdfs-7707, not only the person to whom the work is allocated. A lexicon of words related to bugs is necessary if you want to take important information out of a huge collection of reports that describe bugs. Previous research on software bug ontologies did not give precedence to the creation of ontologies that included data from issue reports, comprehensive descriptions, and comments [21], [98], [99], and [161].

C. A New Approach to Building Ontologies

Figure 6.1 shows a diagram of the suggested procedure for creating an ontology. Reports, in-depth explanations, and comments are compiled by the Apache Bug Report Corpus (APBRC) for twenty-one different software bugs. It was developed in order to give the foundation for a theory on defects in software. The attributes are extracted and the text is pre-processed subsequent to the creation of the database. It is possible for you to conceive about BugIds as well as their properties as concepts and attributes, respectively. A formal concept grid is constructed using this method. Ontologies are formed using a variety of links, such as "is-a" and "has-a," and grids serve as the foundation for these ontologies. The study of ontology In order to make an ontology accessible on the internet, one utilizes Web Language, and in order to extract data from inquiries, one needs a reasoner.

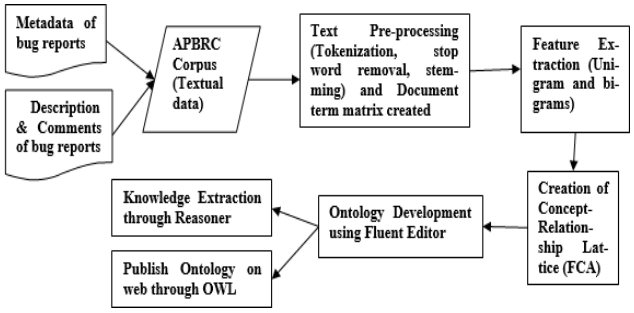


Fig 3 Diagram of the Ontology-Creation Process

D. Auto-Summarization of Software Issue Reports with the Use of Sentence and Keyword Extraction

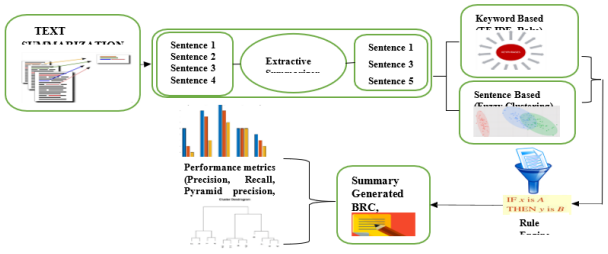


Fig.4 Summary Extraction from Text by Means of a Sentence-and Keyword-Based Approach

This chapter provides a discussion on the second approach that may be used to get a summary of reports related to software issues. The foundation of this strategy to automatically locate errors in software consists of a mix of lines and keywords. In Chapter 7, we spoke about a method that made use of the logical closeness of software bug declarations. The following operators and lines of code are not included: "=", "public static," "^," "sql," "{tmp field}," "{", and "}." The material before the code examples and the code examples themselves may both be educational. This assists developers in understanding the problems and coming up with ways to address them. Code snippets may be used to display data that is in a variety of different document formats. When reporting problems, it is beneficial to have familiarity with code snippets and to make

use of them in order to guarantee that the findings are correct and comprehensible [190]. Compiling an extensive list of software flaws, including relevant code examples, was our main objective in doing this study. Engineers will find bug reports to be readily intelligible and straightforward to understand.

CONCLUSION

English enables software engineers to access a large amount of unstructured data from a variety of applications. Scientists are shown an increasing interest in gathering and examining this kind of data. Unstructured databases are the source of information that is extracted via the technique of text mining. A diverse assortment of strategies for text mining are now being investigated; this includes methods for information retrieval, summarization, sorting, and information extraction, among others. All types of programs are significantly affected by programming errors. As a result, a variety of text mining approaches are used in order to identify mistakes in software.

To automatically find and remove software faults, open-source issue systems need to use structured methodologies. It is the major purpose of this study to uncover such methods. An Apache Project Bug Report Corpus (APBRC) is established by collecting software flaws from twenty projects that are operated by the Apache Software Foundation. Every software problem is formatted in a certain manner that contains a short summary, a reaction, and a degree of severity. Information on the fault, including its identification number, a comprehensive description, the programmer who is accountable, and any comments made by other software developers, as well as the significance of the problem In order to get a variety of attributes, an automated system known as the "Bug Report Collection System" (BRCS) is used.

Machine learning and text analytics are being used to construct models that can anticipate the severity of software faults. Software problems may be categorized into two different types: severe and those that are not significant. The Term Frequency-Inverse Document Frequency (TF-IDF) approach is used to determine the most prominent phrases while you are working with a one-line description of a previously fixed software problem. The following 10 approaches are used in the field of machine learning: boosting, Naïve Bayes, K-nearest neighbor, support vector machine, maximum entropy, decision tree, random forest, bagging, stabilized linear discriminant analysis, and generalized linear models. Both forecasts on the severity of issues at the system level and the component level are possible. The methods are assessed based on a number of different factors, including accuracy, precision, and memory. Boosting provided better results than random forest, which attained an accuracy range from 75% to 97%, in twelve out of the fourteen studies. Boosting lagged behind with a range of 81% to 98% in terms of accuracy. Maximum Entropy, Decision Tree, Naïve Bayes, Support Vector Machine, and K-Nearest Neighbor all fall short of the average performance standard. In conclusion, it need to be noted that SLDA and GLM are the two least accurate algorithms in the field of machine learning. The Friedman Rank Test and the Nemenyi Post-hoc Test are two examples of statistical methods that are used to double-check the results. It will raise a red signal if there is a significant amount of heterogeneity across the various approaches to machine learning.

REFERENCES

[1] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J.: Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620v4 2024

[2] Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M., Amine, Cordeiro, L.: A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. (2023). 10.48550/arXiv.2305.14752

[3] Dipayan Saha, S., Tarek, K., Yahyaei, S.K., Saha, J., Zhou: Mark Tehranipoor, Farimah Farahmandi. LLM for SoC Security A Paradigm Shift. arXiv:2310.06046v1 2023

[4] Ferrag, M.A., Ndhlovu, M., Tihanyi, N., Cordeiro, L.C.: Merouane Debbah, Thierry Lestable, Narinderjit Singh Thandi. Revolutionizing Cyber Threat Detection with Large Language Models. arXiv:2306.14263v2 2024.

[5] Andreas Happe and Jürgen Cito: Getting pwn'd by AI: Penetration Testing with Large Language Models. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 2082–2086. (2023). https://doi.org/10.1145/3611643.3613083

[6] Sakaoglu, S.: 'KARTAL: Web Application Vulnerability Hunting Using Large Language Models: Novel method for detecting logical vulnerabilities in web applications with finetuned Large Language Models', Dissertation, (2023)

[7] Ferrag, M.A., Battah, A., Tihanyi, N., Debbah, M., Lestable, T.: Lucas C. Cordeiro. SecureFalcon The Next Cyber Reasoning System for Cyber Security. arXiv:2307.06616v1 2023

[8] Weng, G., Andrzejak, A.: Automatic Bug Fixing via Deliberate Problem Solving with Large Language Models, in 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Florence, Italy, 34–36. (2023). pp 10.1109/ISSREW60843.2023.00040

[9] David Noever: Can Large Language Models Find and Fix Vulnerable Software. arXiv:2308.10345v1 2023
[10] Hammond Pearce, B., Tan, B., Ahmad, R., Karri: Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models. arXiv:2112.02125v3 2022.
[11] Jingxuan He and Martin Vechev: Large Language Models for Code: Security Hardening and Adversarial Testing. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23). Association for Computing Machinery, New York, NY, USA, 1865–1879. (2023).
[12] https://doi.org/10.1145/3576915.3623175
[13] Xia, C.S., Wei, Y.: Lingming Zhang. Practical Program Repair in the Era of Large Pre-trained Language Models. arXiv:2210.14179v1 2022
[14] Purba, M., Ghosh, A., Radford, B., Chu, B.: Software Vulnerability Detection using Large Language Models, in 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Florence, Italy, 112–119. (2023). pp 10.1109/ISSREW60843.2023.00058
[15] Yuqiang, S., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Yang Liu:. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. arXiv:2308.03314v2 2023.
[16] Huang, K., et al.: An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair, 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, Luxembourg, pp. 1162–1174, (2023). 10.1109/ASE56229.2023.00181
[17] Katsadouros, E., Patrikakis, C.Z., Hurlburt, G.: Can Large Language Models Better Predict Software Vulnerability? in IT Professional. May-June. **25**(3), 4–8 (2023). 10.1109/MITP.2023.3284628
[18] Marwan Omar: Detecting software vulnerabilities using Language Models. arXiv:2302.11773v1 2023
[19] Mamede, C., Pinconschi, E., Abreu, R., Campos, J.: Exploring Transformers for Multi-Label Classification of Java Vulnerabilities, 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), Guangzhou, China, 2022, pp. 43–52, 10.1109/QRS57517.2022.00015 enchmarks. arXiv:2312.12575v2 2024