Machine Learning Based Bug and Vulnerability Prediction

Tamás Aladics

Department of Software Engineering
University of Szeged

Szeged, 2024

Supervisor:

Dr. Rudolf Ferenc

Summary of the Ph.D. Thesis

University of Szeged
Ph.D. School in Computer Science

Introduction

In an increasingly interconnected world, software systems have become the backbone of modern society, profoundly impacting every aspect of our daily lives, from mobile applications to complex systems that manage our transportation and healthcare. This proliferation of software solutions, coupled with an exponential growth in screentime, particularly during the COVID-19 pandemic, has significantly transformed how we live, work, and communicate, bringing convenience and efficiency [20]. However, this rapid integration has also turned these systems into prime targets for cyberattacks, with the surge in vulnerabilities leading to an alarming increase in data breaches and security incidents. The International Business Machines Corporation (IBM) X-Force Threat Intelligence Index in 2020 reported that around 8.5 billion records were compromised in 2019, marking a more than 200 percent increase from the previous year [29]. These incidents often stem from software development oversights and inadequate security protocols, highlighting the urgent need for developers to prioritize security and for organizations to continuously monitor and update their systems to mitigate threats, especially as critical infrastructure becomes increasingly reliant on digital technologies [28].

Recognizing the escalating vulnerabilities in software systems, the cybersecurity community has moved towards a security-first approach in both software development and deployment, moving beyond traditional reliance on manual interventions like code reviews and penetration testing. While these conventional methods have their uses, they demand significant time and resources and may miss finer issues. In response, automated software analysis techniques emerged to complement manual efforts. Static analysis, which inspects code without running it to identify vulnerabilities through predefined patterns, and dynamic analysis, which monitors software behavior in real-time to spot anomalies, together with traditional reviews, represent a effective strategy for enhancing software security.

Even though automated methods like static and dynamic analysis represent a significant advance beyond manual reviews, they encounter scalability and adaptability challenges in the face of software vulnerabilities' evolving nature. The pattern-based solutions these techniques utilize can quickly become outdated with new vulnerabilities' emergence. Furthermore, the vast amount of code in modern software systems renders manual and pattern-based automated analyses increasingly impractical.

The integration of machine learning (ML) into Information Technology, particularly in software security, marks a pivotal advancement by offering solutions that adapt and learn from data, surpassing traditional pattern-based analyses. ML algorithms' capacity to learn from historical vulnerabilities and incidents to predict and detect new threats is a significant leap forward. My contribution includes the creation of the DeepWaterFramework, a tool designed to automate essential ML pipeline processes like execution, hyperparameter optimization, and evaluation across distributed systems [14]. This framework has played a crucial role in our research, facilitating the development and comparison of various ML approaches to software security, demonstrating ML's potential in enhancing automated analysis and the development of more proactive and robust security measures.

Further exploration into machine learning's application in software security reveals its effectiveness in understanding and representing the complex patterns within software systems, such as through source code embedding. This technique, crucial for ML applications, converts code into numerical representations that capture semantic relationships. By employing algorithms like Doc2Vec to analyze these representations, ML models can proactively predict vulnerabilities, allowing for early intervention. Our research has specifically applied Doc2Vec to Java source code, producing numerical vectors that reveal semantic patterns and predict potential bugs, showcasing ML's capacity to improve vulnerability detection and contribute to the development of safer software systems [6].

Acknowledging the growing trend of software vulnerabilities highlights the importance of early detection approach, especially at commit time—when code is ready for sharing but not yet fully integrated. The Just-In-Time (JIT) vulnerability prediction strategy emphasizes the need for timely and proactive security assessments. A key part of JIT prediction is effectively representing code changes through commit representations, analyzing the changes made during commits. Despite challenges due to a lack of datasets for vulnerability-contributing commits, our research introduces a method to generate such datasets from existing fixing commit datasets [4]. By enhancing the SZZ algorithm with a relevance-based filtering process, we've created a dataset focused on Java vulnerabilities, aiding in the development of JIT vulnerability detection methods and facilitating their evaluation.

In our studies on JIT vulnerability prediction, we identified a gap in methods that incorporate the structural information of source code, often encapsulated in forms like Abstract Syntax Trees (ASTs). We introduced the Code Change Tree (CCT) structure, leveraging ASTs to create a change tree that captures the structural differences at the AST level between the pre- and post-commit states. Our approach was evaluated using machine learning models, with Doc2Vec transforming tree structures into vectors [3]. To provide a more precise evaluation, in a comparative study against CC2Vec and DeepJIT, we analyzed their performance in JIT vulnerability prediction, their false positive rates, and the granularity of their predictions [5]. While CCT provides customizable granularity, it demonstrated lower performance compared to CC2Vec and DeepJIT, which showed stronger results in commit-level predictions, offering insights into the trade-offs between granularity and predictive accuracy in different commit representation methods.

The thesis consists of four thesis points. In this booklet, we summarize the results of each thesis point.

№	[6]	[4]	[3]	[5]
I.	♦			
II.		♦		
III.			♦	
IV.				♦

Table 1: Thesis contributions and supporting publications

I. Bug Prediction Using Source Code Embedding Based on Doc2Vec

While crucial for ensuring software quality, detecting bugs remains a challenging task due to the limitations of existing static code analysis tools. These tools often lack robustness due to their reliance on predefined patterns. Machine learning offers a promising alternative by learning patterns from vast amounts of data, potentially improving bug prediction. However, a critical aspect of any machine learning approach is how it represents its input data. In the context of software analysis, these features come directly from the source code itself, taking various forms like code metrics, sequences of functions or classes, and even entire programs. These representations can differ in structure (tokens, statements, functions, etc.) and the form used (text, abstract syntax tree, etc.). Given the importance of code representation in this field, this thesis point explores the effectiveness of using features derived from the Abstract Syntax Tree (AST) compared to solely relying on code metrics for bug prediction.

More precisely, we propose a bug prediction method using code embedding based on Doc2Vec [21], a powerful tool for learning distributed document representations. Doc2Vec extends the well-known Word2Vec algorithm, which learns word embeddings by considering the surrounding words in a text corpus. Doc2Vec builds upon this concept by also learning a document vector, which captures the semantic meaning of an entire document. In the context of this work, we treat classes (including enumerations and interfaces) as the fundamental unit, generating sequences of tokens from them. These class sequences are then treated as documents within a Doc2Vec model, with individual tokens acting as words within the document. Doc2Vec generates a fixed-size vector representation for each class, which can then be used as features for bug prediction tasks.

Our experimentation involves utilizing different Doc2Vec parameterizations. The generated vectors are then used for bug prediction in two ways:

- Standalone Features: The vectors are directly used as input features for machine learning models.
- Combined Features: The vectors are combined with traditional code metrics to form a comprehensive feature set.

To capture the structural information of the source code, we leverage the Abstract Syntax Tree (AST) as an intermediate representation. We traverse the AST in depth-first order, adding the type of each encountered node (e.g., class definition, variable usage) to a sequence. To account for scope changes within the code, we introduce a constant value to the sequence whenever a step back occurs in the tree traversal. This helps differentiate code blocks within distinct scopes (e.g., if statements).

Our decision to employ Doc2Vec stems from two key considerations. Firstly, we can conceptually view classes as "paragraphs" and their code elements as "words." Secondly, Doc2Vec outputs fixed-length vectors, simplifying their integration with various machine learning models. To evaluate this approach we used the Unified Bug Dataset [13], an extensive compilation of Java code bugs, to assess the efficiency of employing Doc2Vec for code embedding in bug prediction. This dataset contains various sources, providing a collection of bugs associated with different levels of source code. It focuses on classes, interfaces, and enumerations, encompassing the analysis of 48,719 entities, with 8,242 identified as faulty.

To update and augment our code metrics, we utilized the OpenStaticAnalyzer toolset [8], incorporating established software metrics such as Lines of Code (LOC), Number of Methods (NM), and more. Also, the Deep Water Framework (DWF) [14] played a crucial role in our experimental endeavors, which included experimenting with different Doc2Vec parameterizations and various machine learning models. Our model array featured traditional algorithms such as Random Forest, Decision Trees, K-Nearest Neighbour (KNN), Support Vector Machines (SVM), Naive Bayes, Linear Classifier, Logistic Regression, alongside two neural network architectures: Standard Deep Neural Network (SDNNC) and Custom Deep Neural Network (CDNNC). We also employed a 10-fold cross-validation and implemented preprocessing strategies like data binarization, standardization, and upsampling.

Table 2: Comparison of different machine learning methods with the same embedding (values are F-scores)

Model name	Embedding	Code metrics	Combined
Bayes	0.301	**0.325**	**0.325**
Linear	0.298	0.401	**0.418**
Logistic	0.311	0.412	**0.430**
Tree	0.374	**0.475**	0.461
Random Forest	0.423	0.515	**0.522**
CDNNC	0.451	0.474	**0.502**
SDNNC	0.467	0.520	**0.533**
KNN	0.463	0.502	**0.524**

Based on the results shown in Table 2, our findings indicate that although Doc2Vec embeddings perform comparably to traditional code metrics, the latter slightly outperform in general. No single Doc2Vec setting emerged as universally optimal across all machine learning models, underscoring the need for customized parameter tuning. When combining Doc2Vec embeddings with code metrics, the results consistently improved, suggesting that the embeddings add valuable semantic information missing from code metrics alone. This synergy between source code embeddings and code metrics not only enhances bug prediction accuracy but also confirms that Doc2Vec provides important insights into the data.

The replication package with the dataset generated for bug prediction is available: `http://doi.org/10.5281/zenodo.4724941`

The Author's Contributions

The author took part in defining the source code embedding methodology. He performed the model training and hyperparameter tuning. He designed the model evaluation scheme. The author performed dimension reduction on some source code examples to showcase the embedding's effectiveness. He ran the static analysis tool on the database entries to generate the source code metrics.

- ♦ **Tamás Aladics**, Judit Jász, Rudolf Ferenc. Bug Prediction Using Source Code Embedding Based on Doc2Vec. Computational Science and Its Applications 21st International Conference (ICCSA 2021), Cagliari, Italy, September 13-16, 2021, Proceedings, Part VII, volume 12955 of Lecture Notes in Computer Science, pages 382–397. Springer, 2021. `https://link.springer.com/chapter/10.1007/978-3-030-87007-2_27`

II. A Vulnerability Introducing Commit Dataset for Java: an Improved SZZ Based Approach

Software vulnerabilities pose a significant and growing threat to software security. Machine learning techniques are increasingly being applied to software engineering tasks, such as quality assurance, to address these challenges. The success of these techniques heavily depends on the availability of suitable datasets for training and evaluation. While datasets containing vulnerability-fixing commits (VFC) can be found, datasets focusing on the commits that introduced the vulnerabilities (VIC) are far less common. This scarcity limits research on critical tasks such as just-in-time vulnerability detection and localization [7, 12, 22].

Existing vulnerability-fixing datasets often build upon publicly disclosed vulnerability databases like the Common Vulnerabilities and Exposures (CVE) [1] and the National Vulnerability Database (NVD) [10]. These resources typically include details about the vulnerability and may also offer links to the relevant fixing patches. Despite the availability of VFC datasets, pinpointing the corresponding commits that originally introduced these vulnerabilities remains a complex challenge. Current attempts to automate this process often involve heuristics and may not be easily scalable.

To address this problem, we propose a two-phase methodology for systematically generating VIC datasets from readily available VFC datasets:

- **Phase 1: Initial Candidate Identification** At the core of our approach lies a refined variant of the SZZ algorithm, known as SZZ Unleashed [11]. SZZ-based algorithms excel at tracing lines in a vulnerability-fixing commit (VFC) back through the repository's history to pinpoint commits that potentially introduced the flawed code. By applying SZZ Unleashed to VFCs in existing datasets, we obtain an initial set of candidate VICs.

- **Phase 2: Filtering and Relevance Scoring** Due to the inherent breadth of SZZ-like algorithms, their raw output often includes false positives and lacks a clear ranking of the candidate VICs' relevance. To mitigate these issues, we implement a filtering phase that assigns a 'relevance score' to each candidate VIC. This score quantifies the degree to which the VIC's changes correlate with the modifications performed in the corresponding VFC, providing a metric for prioritization.

The 'relevance score' calculation is a pivotal component of our method. For each candidate VIC, we iterate over its modified files, seeking corresponding files in the associated VFC. If a match is found, we compute a 'contribution score' that blends two factors:

- File Similarity: We measure the similarity between the corresponding files in the VIC and VFC, such as the proportion of identical lines, excluding whitespace for robustness.

- Base Score: We estimate the relative importance of the file within the VFC's modifications by calculating the ratio of changed lines in that file to the total number of changes made in the VFC. This way, files with more involvement in the fix are assigned higher 'base scores'.

The final 'relevance score' for a VIC is the sum of 'contribution scores' for all its files that have counterparts in the VFC. This score provides a nuanced indicator of the VIC's potential for being a true vulnerability-introducing commit.

To showcase the practicality of our method, we developed the tools *BugIntroducerMiner* and *FilterBugIntroducer*. These tools were used to extract a novel Java vulnerability-introducing commit (VIC) dataset from the project-KB database. This dataset, containing 564 VFC entries with at most two but at least one VIC assigned to each, offers valuable resources for security research. Notably, our method significantly refines the raw SZZ output, reducing the range of VIC entries per VFC from 1-700 to a more manageable number. During the generation process, over 110,000 files from 198 open-source projects were considered.

The dataset and assisting tools are available: `https://doi.org/10.5281/zenodo.5785239`

The Author's Contributions

The author took part in exploring and reviewing the related work. He participated in designing the methodology. He recognized the need for the second, filtering phase. The author defined the relevance score. The author participated in investigating the available SZZ implementations and setting the needed environment up. He was the developer of the tool that generates the vulnerability-introducing dataset. He took part in evaluating and interpreting the results.

- ♦ **Tamás Aladics**, Péter Hegedűs, Rudolf Ferenc. A Vulnerability Introducing Commit Dataset for Java: An Improved SZZ based Approach. In Proceedings of the 17th International Conference on Software Technologies - ICSOFT, pages 68–78. INSTICC, SciTePress, 2022.
  `https://www.scitepress.org/Link.aspx?doi=10.5220/0011275200003266`

III. An AST-based Code Change Representation and its Performance in Just-in-time Vulnerability Prediction

The escalating number of software vulnerabilities presents a critical challenge for software security. The rapid increase in vulnerabilities, highlighted by a 50% rise in open-source vulnerabilities in 2020 alone [19], underscores the urgency of this issue. Just-in-time (JIT) vulnerability prediction aims to address this challenge by detecting vulnerabilities as soon as they are introduced into the codebase. This proactive approach offers the potential to minimize security risks and reduce the costs associated with later remediation.

Traditional vulnerability prediction models (VPMs) often rely on static analysis tools and software metrics. However, these techniques can be limited by scalability issues, high false-positive rates, and difficulty in adapting to new vulnerability patterns [18][9][25]. Machine learning (ML) presents a promising alternative, but its success depends heavily on the quality and representation of the data used for training. A fundamental challenge in JIT vulnerability prediction lies in effectively representing the differences between pre-commit and post-commit code states. Recent research suggests that existing metrics and textual features may not fully capture the nuances of these changes [23]. This motivates the need for novel code change representation techniques that can better inform machine learning models for vulnerability prediction.

In this thesis point, we present a novel approach for representing source code changes called Code Change Tree (CCT). CCTs are designed to effectively represent source code changes for just-in-time (JIT) vulnerability prediction, and their construction involves several steps. First, Abstract Syntax Trees (ASTs) are generated for both the before-change (S_{pre}) and after-change (S_{post}) code states. Each AST is then transformed into a set of unique root-to-leaf paths, outlining the code's hierarchical structure. Next, root-paths from S_{pre} that are identical to those in S_{post} are discarded, retaining the changes made. The remaining, non-identical root-paths form the Code Change Tree, effectively representing the structural alterations.

A crucial aspect of CCTs is the node identification used within ASTs. This scheme considers node type, value, and contextual information (position within the AST) to enable meaningful comparisons even across different ASTs. Additionally, CCTs are flexible in terms of granularity – they can potentially represent changes at various levels (statement, method, class, etc.) since the method works with any AST-possessing code element.

To integrate CCT representations (and similarly, simple AST representations) with machine learning models, we employ Doc2Vec embedding. A Doc2Vec model is trained on a large corpus

Table 3: Results of different source code embedding approaches (F1-score)

Random Guesser	20%							
	Adaboost	**CDNNC**	**Forest**	**KNN**	**Logistic**	**SDNNC**	**Tree**	**Average**
Metrics	37%	32%	38%	38%	19%	29%	37%	33%
Simple	41%	40%	42%	**47%**	30%	**44%**	40%	41%
Change Tree	**41%**	**44%**	**43%**	43%	**38%**	44%	**42%**	**42%**

of Java methods to learn Java-specific semantics. This model then transforms the flattened CCTs (and flattened ASTs in the simpler method) into fixed-length vectors, making them suitable for machine learning input.

In summary, we explore three code change representation methods:

- Metric-Based: Software metrics are calculated for functions involved in the change and concatenated for S_{pre} and S_{post}.
- Simple Code Change: ASTs are generated for S_{pre} and S_{post}, then flattened into token sequences and concatenated.
- Code Change Tree (CCT): Root-to-leaf paths are extracted from each AST. Changes in S_{pre} relative to S_{post} are identified via root-path differences, forming the Code Change Tree. We employ a specific node identification scheme for cross-AST comparisons.

For evaluation, we utilize our previously generated SZZ-based vulnerability-introducing commit (VIC) dataset [4]. This dataset was derived from the project-KB VFC database through SZZ candidate identification followed by relevance score based filtering.

Our experiments demonstrate the strengths of Code Change Trees (CCTs) as a method for representing code changes. The advantage of AST based representation forms over metrics based is evident as there is at least 8% increase in average F1-score. This finding further supports that AST based representations capture (most likely structural) information uncaught by metrics. A not that substantial, but still noticeable difference can be seen between the two AST based approaches, as Code Change Tree performs better by nearly 2% (Table 3). This underscores the importance of capturing structural changes – a key strength of CCTs that extends beyond the information provided by metrics or simple AST flattening.

Furthermore, CCTs significantly reduced the average size of code change representations compared to the Simple Code Change method. On our dataset of 59,340 functions, CCTs had an average node count of 51 compared to 174 nodes for the Simple Change representation – a reduction of over 70%. This highlights CCTs' ability to isolate the most crucial aspects of a change, benefiting downstream machine learning tasks. We believe this efficiency stems from CCTs' focus on changes, discarding unchanged code paths that are irrelevant to the change.

The combination of improved accuracy and compact representation makes CCTs a promising choice for vulnerability detection. Their ability to focus on the core elements of a code change could significantly enhance the speed and reliability of just-in-time vulnerability prediction systems.

The Author's Contributions

The author took part in the related literature's review. The designing of the methodology was also mainly the author's work. He defined and implemented the other code representations. The author set up and run the models' training and hyperparameter tuning environment (DeepWaterFramework). He participated in evaluating and interpreting the results.

- ♦ **Tamás Aladics**, Péter Hegedűs, and Rudolf Ferenc. An AST-based Code Change Representation and its Performance in Just-in-time Vulnerability Prediction. Software Technologies, pages 169–186, Cham, 2023. Springer Nature Switzerland.
`https://link.springer.com/chapter/10.1007/978-3-031-37231-5_8`

IV. Assessment of Commit Representations for Just-in-time Vulnerability Prediction

The increasing complexity and interconnectedness of software systems lead to a rise in software vulnerabilities, posing a significant threat. This trend is evident in Tenable's 2021 report, which highlights a substantial increase in reported CVEs [2]. To mitigate these vulnerabilities, early identification during the development process is crucial. Just-in-time (JIT) vulnerability prediction at the time of code commits offers a timely solution [16].

Commits contain valuable information for vulnerability analysis, including bug fixes, feature additions, code refactoring, and metadata. However, manual analysis of these commits is a daunting and error-prone task, especially in large-scale projects [25]. Machine learning approaches offer a promising alternative for automatic vulnerable commit identification [17].

A critical aspect of these approaches is commit representation – capturing commit information in a way suitable for machine learning algorithms. Representations often rely on commit messages [30], code changes (patches) [24], or a combination of both [15, 16]. Some even incorporate code metrics to supplement commit metadata [26].

Source code representations can take various forms, including raw text, intermediate representations (e.g., ASTs), or derived code metrics. Granularity is also key – affecting representation usability – and can range from the entire commit down to individual lines. This thesis point aims to provide an comparative study of these factors in commit representation by comparing three distinct approaches: CC2Vec, DeepJIT, and a Code Change Tree-based representation:

- DeepJIT [16] analyzes code changes by processing commit messages and corresponding code using convolutional neural networks (CNNs). It embeds raw textual data into arrays and employs two dedicated CNNs to extract relevant features – one for commit messages and another for code changes. The resulting vectors are aggregated to produce a final representation for the commit.

- CC2Vec [15] learns vector representations of code changes in patches. It uses a hierarchical attention network (HAN) to construct vector representations of removed and added code within a given patch. Comparison functions produce features representing the relationship between removed and added code, which are subsequently concatenated to form the final vector representation for the code change in a patch.

- Code Change Tree (CCT) [4] is a novel structure introduced in thesis point III, designed to represent differences between two states of source code at a structural level, utilizing Abstract Syntax Trees (ASTs).

We employed two datasets for this study. The first is a Project-KB derived dataset [27] containing vulnerability entries associated with CVE identifiers, which we showed in thesis point III. The second dataset, Defectors, required only minor refinements. Both datasets share a similar structure – commit SHA, repository identifier, filepath, and vulnerability labels – facilitating

Table 4: The performance measured by various metrics for each model, averaged over a tenfold cross-validation process for ProjectKB and evaluated on the test set for the Defectors dataset.

Dataset	Model	Accuracy	F_1	$F_{0.5}$	F_2	Precision	Recall
ProjectKB	DeepJIT	0.74	0.47	0.41	0.56	0.38	0.64
	CC2Vec	0.59	0.37	0.3	0.51	0.32	0.64
	CCT + RF	0.7	0.33	0.3	0.37	0.29	0.4
	Baseline	0.65	0.22	0.22	0.22	0.22	0.22
Defectors	DeepJIT	0.71	0.35	0.27	0.47	0.24	0.63
	CC2Vec	0.75	0.39	0.31	0.50	0.28	0.63
	CCT + LSTM	0.66	0.3	0.23	0.42	0.20	0.58
	Baseline	0.78	0.13	0.13	0.13	0.13	0.13

vulnerability analysis. To assess model performance on these imbalanced datasets, we used various metrics including accuracy, precision, recall, and F-scores (F_1, F_2, $F_{0.5}$).

Due to the architectural differences of these models, specific preprocessing was necessary to tailor the inputs appropriately. We used the implementations by the authors for DeepJIT [1] and CC2Vec [2] where possible, while for the CCT model, we implemented the following steps:

- Project-KB dataset: We extracted altered functions for method-level analysis. Constructed CCTs were flattened and embedded using Doc2Vec. A random forest model was trained and evaluated on these vectors to determine method-level vulnerability. A commit was labeled as vulnerable if any changed method in it was predicted as vulnerable.
- Defectors dataset: Due to size, the method-mining process was computationally too expensive. For this reason, we opted for a high-level granularity and trained word embeddings for flattened trees. These were fed into a bidirectional LSTM layer followed by a dense layer with a sigmoid activation function to yield vulnerability probability.

Our experimental findings can be seen in Table 4. Our results indicate that commit representations can be helpful for just-in-time (JIT) vulnerability prediction, but their effectiveness varies depending on the specific use case. While all the investigated models (DeepJIT, CC2Vec, and CCT) outperform the simple baseline classifier, the degree of improvement depends on the relative importance of false positives and false negatives, which we measure by using different F-scores: F_2 score is a good indicator in cases where false negatives are costly, and similarly $F_{0.5}$ is a good indicator when false positives are focused. Based on the corresponding $F_{0.5}$ and F_2 scores in Table 4. When the priority is minimizing false negatives (high recall), these representations show significant promise. However, for use cases where minimizing false positives is crucial (high precision), their performance is less impressive.

The CCT-based model, despite having slightly lower overall performance, stands out when a more localized vulnerability prediction is needed. Due to its customizable granularity, it offers the flexibility to pinpoint vulnerable methods within code changes. This makes it a valuable option in scenarios where precision is prioritized and a finer-grained vulnerability analysis is desired.

[1]https://github.com/soarsmu/DeepJIT
[2]https://github.com/CC2Vec/CC2Vec

The Author's Contributions

The author performed the literature review and selected the candidate models. He also collected and filtered the candidate datasets that was used in the evaluation. The methodology was designed incorporating contributions from the author. He implemented and tailored the Code Change Tree model for both Defectors and Project-KB datasets. He set up and ran the independent models (DeepJIT and CC2Vec) based on their official repositories. The author took part in interpreting and presenting the results.

- **Tamás Aladics**, Péter Hegedűs, and Rudolf Ferenc. A Comparative Study of Commit Representations for JIT Vulnerability Prediction. Computers 13, no. 1: 22. `https://doi.org/10.3390/computers13010022`

Summary

In this thesis, we covered four topics regarding improving software quality and security using machine learning approaches. The covered topics include the introduction of a source code embedding method for Java, describing a novel way of generating vulnerability introducing datasets, discussing source code change embeddings and proposing a new approach, and finally, giving insight to the just-in-time scene of vulnerability detection through a comparative study.

First, we designed a *source code embedding* algorithm that is aimed to store the structural information of source code by leveraging it's Abstract Syntax Tree (AST). For a source code entry, we flattened the corresponding AST by traversing it in a depth-first manner, and trained a Doc2Vec model to get a fixed length vector. To validate this approach's effectiveness on the defect prediction task, we compared several machine learning models on a dataset of defective and defect-free source code.

Going forward, as we delved deeper into the subject of software defect prediction we noticed the scarcity of vulnerability introducing datasets - datasets with it's entries being commits that contributed to a vulnerability. To remedy this issue, we proposed a novel way of generating *vulnerability introducing commit datasets* from vulnerability fixing datasets by employing a two-phase method. In the first phase we generate a number of candidate commits, and in the second we perform a relevance-score based filtering to reduce the number of false positives. Using this method we created and shared a new dataset that we used in our other works.

In *commit representation*, we aimed to provide an approach to capture the differences between two states of source code, focusing on structural information. To this end, we designed Code Change Tree, a specific structure that retains only the changes between to ASTs. This approach is applied on the method level and compared to a source code metric based and a simple code change approach. The former is based on metrics calculated by static analysis tools and the latter is based on the entire pre and post commit states' ASTs - as opposed to Code Change Trees, which contain only the changes. For this comparison we used the dataset generated in our previous work and came to the conclusion that Code Change Trees effectively improve the performance over the contending two methods.

Finally, in *JIT vulnerability models*, we gave insight to the current state of commit-time vulnerability prediction. In our work, we studied three distinct approaches: CC2Vec, a hierarchical attention network, DeepJIT, a deep neural network based on convolutional layers and finally Code Change Trees. We compared these methods on two datasets: the modest sized dataset for Java that we introduced in prior work and a larger, recently published dataset for Python called

Defectors. Our results show that while the CC2Vec and DeepJIT work with greater predictive efficiency, Code Change Tree is more flexible and can be applied for more localized predictions (such and file or method level).

Acknowledgements

While this thesis highlights my own contributions, this research would not have been possible without the support and guidance of many individuals.

I'd like to express my gratitude to my supervisor, Dr. Rudolf Ferenc. He introduced me to the world of research and provided invaluable opportunities for learning and growth throughout my studies.

I'm also grateful to Dr. Péter Hegedűs for his close collaboration and constant willingness to help with any questions I had. His insights and support were critical to my progress and publications.

I would like to thank Dr. Jász Judit for her guidance on source code embeddings, especially during my first co-authored publication. Her expertise helped me gain a better understanding of this area.

Finally, thank you to Viszkok Tamás for the productive and enjoyable collaboration on many projects, but especially the Deep Water Framework project.

I recognize that research is a collaborative effort, and I am grateful to everyone who contributed to my journey.

This thesis was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory, and by project no TKP2021-NVA-09 which has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

Tamás Aladics, 2024

References

[1] The mitre corporation - common vulnerabilities and exposures: https://www.cve.org/, Sep 2021. nov. 20.

[2] The 2021 Threat Landscape Retrospective: Targeting the Vulnerabilities that Matter Most, 1 2022.

[3] Tamás Aladics, Péter Hegedűs, and Rudolf Ferenc. An ast-based code change representation and its performance in just-in-time vulnerability prediction. In Hans-Georg Fill, Marten van Sinderen, and Leszek A. Maciaszek, editors, *Software Technologies*, pages 169–186, Cham, 2023. Springer Nature Switzerland.

[4] Tamás Aladics., Péter Hegedűs., and Rudolf Ferenc. A vulnerability introducing commit dataset for java: An improved szz based approach. In *Proceedings of the 17th International Conference on Software Technologies - ICSOFT*, pages 68–78. INSTICC, SciTePress, 2022.

[5] Tamás Aladics, Péter Hegedűs, and Rudolf Ferenc. A comparative study of commit representations for jit vulnerability prediction. *Computers*, 13(1), 2024.

[6] Tamás Aladics, Judit Jász, and Rudolf Ferenc. Bug prediction using source code embedding based on doc2vec. In *Proceedings of the 21th International Conference on Computational Science and Its Applications (ICCSA 2021)*, pages 382–397, Cagliari, Italy, September 2021. Springer International Publishing.

[7] Amr Amin, Amgad Eldessouki, Menna Tullah Magdy, Nouran Abdeen, Hanan Hindy, and Islam Hegazy. Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information*, 10(10), 2019.

[8] OpenStaticAnalyzer Static Code Analyzer. `https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer`, 2021.

[9] Nuno Antunes and Marco Vieira. Benchmarking vulnerability detection tools for web services. In *2010 IEEE International Conference on Web Services*, pages 203–210, 2010.

[10] Harold Booth, Doug Rike, and Gregory Witte. The national vulnerability database (nvd): Overview, 2013-12-18 2013.

[11] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. Szz unleashed: an open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project. *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*, 2019.

[12] Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136:106576, 2021.

[13] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal*, 28:1447–1506, 2020. Open Access.

[14] Rudolf Ferenc, Tamás Viszkok, Tamás Aladics, Judit Jász, and Péter Hegedűs. Deepwater framework: The swiss army knife of humans working with machine learning models. *SoftwareX*, 12:100551, 2020.

[15] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 518–529, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: An end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45, 2019.

[17] Kevin Hogan, Noel Warford, Robert Morrison, David Miller, Sean Malone, and James Purtilo. The challenges of labeling vulnerability-contributing commits. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 270–275, 2019.

[18] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[19] Patricia Johnson. The state of open source vulnerabilities 2021 - whitesource, Jun 2022.

[20] Mina Khan, Kathryn Wantlin, Zeel Patel, Elena Glassman, and Pattie Maess. Changing computer-usage behaviors: What users want, use, and experience. In *Asian CHI Symposium 2021*. ACM, may 2021.

[21] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–1188–II–1196. JMLR.org, 2014.

[22] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *2013 International Conference on Availability, Reliability and Security*, pages 446–454, 2013.

[23] Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, 188:111283, 2022.

[24] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. Commit2vec: Learning distributed representations of code changes. *SN Computer Science*, 2(3), mar 2021.

[25] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 426–437, New York, NY, USA, 2015. Association for Computing Machinery.

[27] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, May 2019.

[28] Hossein Rahimpour, Joe Tusek, Alsharif Abuadbba, Aruna Seneviratne, Toan Phung, Ahmed Musleh, and Boyu Liu. Cybersecurity challenges of power transformers, 2023.

[29] Sarah Sharifi. A novel approach to the behavioral aspects of cybersecurity, 2023.

[30] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 914–919, New York, NY, USA, 2017. Association for Computing Machinery.

Nyilatkozat

Aladics Tamás *"Gépi tanulás alapú hiba és sérülékenység előrejelzés"* című PhD disszertációjában a következő eredményekben Aladics Tamás hozzájárulása volt a meghatározó:

Eredmények	**Tézispontok/ fejezetek**	**Cikk**
- Modelltanítás, paraméter hangolás. - Modellek kiértékelésének megtervezése, végrehajtása. - Dimenziócsökkentés az eredmények reprezentáláshoz. - Statikus kódelemző futtatása, kiértékelése az adatbázison. - Eredmények értékelése, értelmezése.	I. tézispont	Tamás Aladics, Judit Jász, and Rudolf Ferenc. Bug prediction using source code embedding based on doc2vec. In *Proceedings of the 21th International Conference on Computational Science and Its Applications (ICCSA 2021)*, pages 382–397, Cagliari, Italy, September 2021. Springer International Publishing.
- Kapcsolódó szakirodalom feltérképezése, áttekintése. - Módszertan kidolgozása. - Rendelkezésre álló SZZ implmenetációk vizsgálata. - Futtatási környezet felállítása. - Generáló eszközök implementálása, futtatása.	II. tézispont	Tamás Aladics, Péter Hegedűs, and Rudolf Ferenc. A vulnerability introducing commit dataset for java: An improved szz based approach. In *Proceedings of the 17th International Conference on Software Technologies - ICSOFT*, pages 68–78. INSTICC, SciTePress, 2022.
- Kapcsolódó szakirodalom feltérképezése, áttekintése. - Módszertan kialakítása, Code Change Tree definiálása. - Kiértékelésénél használt forráskód reprezentációk implementálása. - Modellek tanítása, paraméter hangolása.	III. tézispont	Tamás Aladics, Péter Hegedűs, and Rudolf Ferenc. An ast-based code change representation and its performance in just-in-time vulnerability prediction. In Hans-Georg Fill, Marten van Sinderen, and Leszek A. Maciaszek, editors, *Software Technologies*, pages 169–186, Cham, 2023. Springer Nature Switzerland.
- Kötődő szakirodalom feltérképezése kandidátus modellek kiválasztása. - Kiértékeléshez használt adathalmazok összegyűjtése, szűrése. - Code Change Tree modell implemetálása a két adathalmazhoz. - Független modellek (CC2Vec	IV. tézispont	Tamás Aladics, Péter Hegedűs, and Rudolf Ferenc. A comparative study of commit representations for jit vulnerability prediction. *Computers*, 13(1), 2024.

Eredmények	**Tézispontok/ fejezetek**	**Cikk**
és DeepJIT) környezetének felállítása - Kiértékelés futtatása. - Eredmények összegzése.		

Ezek az eredmények Aladics Tamás PhD disszertációján kívül más tudományos fokozat megszerzésére nem használhatók fel.

2024. máj. 9.

[illegible]
jelölt aláírása

[illegible]
témavezető aláírása

Az Informatika Doktori Iskola vezetője kijelenti, hogy jelen nyilatkozatot minden társszerzőhöz eljuttatta, és azzal szemben egyetlen társszerző sem emelt kifogást.

2024. május 14.
dátum

SZEGEDI TUDOMÁNYEGYETEM INFORMATIKA DOKTORI ISKOLA * SZEGED *

[illegible]
DI vezető aláírása