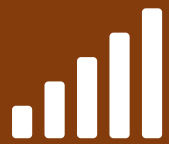




PREPRINT



QuASoQ 2023

11th International Workshop on
Quantitative Approaches to Software Quality

co-located with APSEC 2023
Seoul, South Korea, December 4th , 2023

Editors:

Horst Lichter, RWTH Aachen University, Germany
Selin Aydin, RWTH Aachen University, Germany
Thanwadee Sunetnanta, Mahidol University, Thailand
Toni Anwar, University Petronas, Malaysia

Coyote C++: An Industrial-Strength Fully Automated Unit Testing Tool

Sanghoon Rho¹, Philipp Martens¹, Seungcheol Shin¹, Yeoneo Kim¹, Hoon Heo² and SeungHyun Oh²

¹CODEMIND Corporation, Seoul, South Korea

²Hyundai KEFICO Corporation, Gyeonggi-Do, South Korea

Abstract

Coyote C++ is an automated testing tool that uses a sophisticated concolic-execution-based approach to realize fully automated unit testing for C and C++. While concolic testing has proven effective for languages such as C and Java, tools have struggled to achieve a practical level of automation for C++ due to its many syntactical intricacies and overall complexity. Coyote C++ is the first automated testing tool to breach the barrier and bring automated unit testing for C++ to a practical level suitable for industrial adoption, consistently reaching around 90% code coverage. Notably, this testing process requires no user involvement and performs test harness generation, test case generation and test execution with “one-click” automation. In this paper, we introduce Coyote C++ by outlining its high-level structure and discussing the core design decisions that shaped the implementation of its concolic execution engine. Finally, we demonstrate that Coyote C++ is capable of achieving high coverage results within a reasonable timespan by presenting the results from experiments on both open-source and industrial software.

Keywords

automated unit test, coverage testing, concolic execution, C++, LLVM

1. Introduction

The significance of testing in software engineering is continuously escalating, necessitating thorough validation methods such as white-box testing. However, given the rapid increase in code scale and complexity in the software industry, white-box testing can be time-consuming and resource-intensive[1], often leading to budget constraints. For this reason, there has been a long-standing need for automation in white-box testing.

Lately, efforts to automate white-box unit testing are approaching practical feasibility, with automated testing showing promising results for Java [2, 3], C [4, 5, 6], binary code [7, 8], and a few other programming languages [9, 10, 11]. Conversely, adopting this technology for C++ has proven to be challenging due to the language’s unique features and overall complexity [12]. Implicitly invoked copy or move constructors and templates with all their intricacies are just two examples of C++ language features that are especially difficult to handle in automated white-box unit testing.

In this paper, we introduce Coyote C++, an automated unit testing tool designed for C/C++. With a single click,

Coyote C++ streamlines the entire testing process, from harness generation and test case generation to test execution. The automated test case generation is based on concolic execution, a modern variant of symbolic execution, and features exquisite harness generation capabilities.

The paper outlines the underlying technologies on which Coyote C++ achieves a practical level of high coverage through test case generation. In order to practically utilize automated unit testing tools in the field, we propose that a testing speed of around 10,000 logical LOC of executable statements per hour with statement coverage above 90% and branch coverage above 80% should be desirable. Currently, Coyote C++ is achieving elevated levels of coverage and performance according to these criteria, and is thus being effectively applied and utilized by our customers in the automotive industry.

The rest of this paper is organized as follows. We first look at research on concolic-execution-based unit testing and then examine design decisions made by existing systems to build efficient concolic execution engines in related works. Next, we provide an overview of the implementation of Coyote C++, and present test results obtained from open-source projects and real-world industrial projects. Finally, we conclude the paper by outlining our plans for further improving Coyote C++.

2. Related works

Symbolic execution [13] is a static program analysis technique that interprets programs with symbolic values

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, South Korea


✉ rho@codemind.co.kr (S. Rho); philipp.m@codemind.co.kr

(P. Martens); shin@codemind.co.kr (S. Shin);

yeoneo@codemind.co.kr (Y. Kim); hoon.heo@hyundai-kefico.com

(H. Heo); seunghyun.oh@hyundai-kefico.com (S. Oh)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

rather than concrete values. Due to scalability issues with symbolic execution, this technique has been extended into concolic execution [5, 6]. The main idea of concolic execution is to compute test inputs from path conditions which are obtained by tracking both concrete values and symbolic values. Concolic execution has been anticipated in the automated testing domain due to its known success in test case generation. However, this research has not yet reached a practical level of test generation for whole programs.

Nevertheless, concolic execution is known to be remarkably successful in unit test generation, e.g. for Java [2, 3] and C [4, 5, 6]. For C++ however, automated testing has still been far from viable for industrial purposes despite recent research efforts [14, 12].

When implementing concolic execution there are many options for realizing various aspects of the engine [15]. Especially the engine’s execution mode, analysis target, handling of the path explosion problem, and its memory model can largely affect the performance of the concolic execution engine in terms of coverage and execution time.

2.1. Online/Offline Mode

Concolic execution can be implemented in online or offline mode. In online mode, the concolic execution engine explores multiple paths in a single run by forking on branch points. The advantage of this method is that there is no need to re-execute the common prefixes of multiple paths. However, it requires a substantial amount of memory to store all the states of multiple paths. Offline mode on the other hand explores only one path in a single run. This method requires less memory than online mode, making it better suited for parallelization. However, since offline mode always starts at the beginning of the program for every path, it spends a considerable amount of time on re-examining common path prefixes. Prominent tools using online mode are KLEE [16], MAYHEM [17], and S²E [18], whereas SAGE [7] utilizes offline concolic execution.

2.2. Emulation/Instrumentation

There are two main methods for collecting information about the execution path taken during concrete execution of the program under test. The first method performs symbolic execution at the same time as concrete execution by running the program under test inside of an emulator such as QEMU [19]. The second method instead instruments the program under test with code that handles symbolic execution and the collection of information about the concrete execution of the program. Well-known emulator-based tools are angr [20] and KLEE [16],

while QSYM [21] and CREST [22] are instrumentation-based.

2.3. Mitigating Path Explosion

Another important design decision is how to deal with the path explosion problem commonly encountered when performing concolic execution on programs with complex control flow. In such situations, the search space of concolic execution can grow exponentially due to the many possible combinations of branches. To avoid this issue, concolic execution engines use a variety of heuristic search strategies. Notable search strategies include DFS (depth-first search), BFS (breadth-first search), random path selection, coverage-optimized search, and adaptive heuristics [15, 23].

2.4. Memory Model

When modelling the symbolic memory of a concolic execution engine, one can choose between treating memory addresses as symbolic or concrete values. The symbolic approach can theoretically handle all possible paths, but this approach may cause path constraints to become too complex for current SMT solvers. On the other hand, using concrete addresses might not cover all possible paths due to overly simplified path conditions. In practice, a fully symbolic model is used by tools like KLEE [16], and a concrete address model is used by SAGE [7] among others. Additionally, there are tools like MAYHEM [17] that use a combination of symbolic and concrete addressing schemes.

3. The Design of Coyote C++

3.1. Overview

In this chapter, we present an overview of Coyote C++ and discuss the core decisions that influenced its design. As shown in the diagram in Fig. 1, the Coyote C++ tool is divided into two main parts. The first part builds executable test files based on harness generation, while the second part handles generating test cases through concolic execution.

In the first phase, Coyote C++ uses a harness generator module to automatically generate test stubs and test drivers for test execution and inserts instrumentation code for concolic execution. This instrumentation is performed on LLVM IR level. Next, the binary generation module compiles the created testbed to executable files used in the second part.

While running the executable test file in the second phase, the instrumentation code produces trace files containing information about the concrete program execution on the level of LLVM IR instructions. These trace

QuASoQ 2023 Preprint

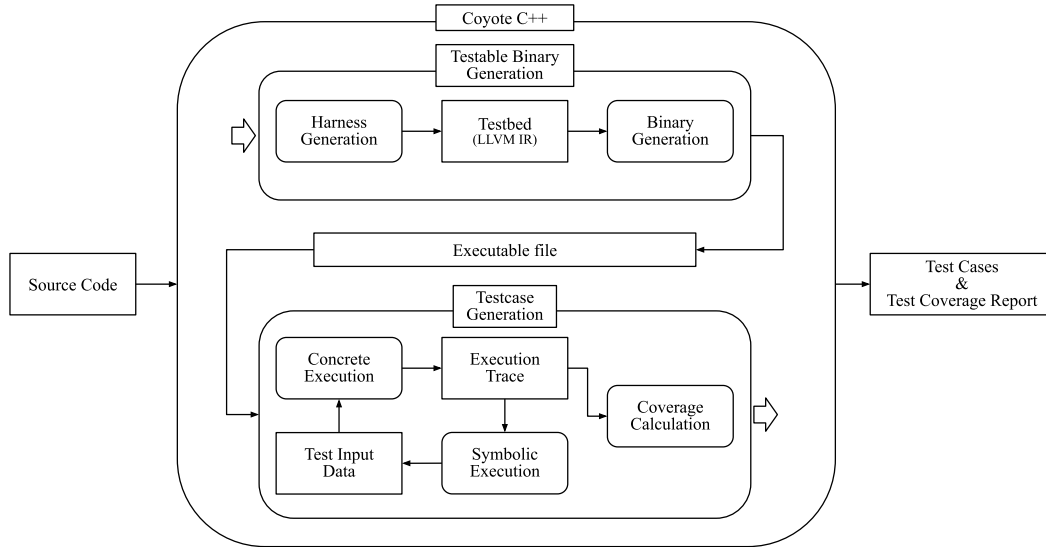


Figure 1: Overview of Coyote C++.

files are then used to reconstruct their respective execution paths, and with this information symbolic execution is performed on the LLVM IR level to generate new test input data. When this concolic execution cycle terminates, the achieved test coverage is calculated based on the generated trace files.

3.2. Design Decisions of Coyote C++

While implementing Coyote C++, many important design decisions had to be made. In the modules responsible for the testable binary generation, these decisions were generally made with the goal of enabling a wide range of transformations on intermediate code models while retaining a sufficiently strong connection between these models and the original source code. Most design decisions affecting the testcase generation phase were strongly influenced by the need to find a suitable tradeoff between the achieved code coverage and performance in terms of test time or resource consumption.

A fundamental design decision made in Coyote C++ is using LLVM IR as its symbolic execution target. This allows for more precision than doing source level symbolic execution while retaining more information about the original source code that would be lost when lowering even further to the assembly level. Also, using LLVM as a foundation for Coyote C++ allows for greater freedom in code transformations during harness generation, bypassing syntactic constraints present on the source code level.

We decided to implement offline testing by inserting instrumentation code into the LLVM IR code of the testbed

during testable binary generation. The main reason for choosing offline testing over online testing is that it is more suitable for parallelization, which is essential for providing good testing performance. Additionally, offline testing is more advantageous from a memory management standpoint.

A key factor for achieving high code coverage is the search strategy that controls in which order the possible execution paths of a program are explored. During testcase generation, the test files are initially executed with all test inputs set to default values. The trace files generated from this are then analyzed using concolic execution techniques to create new test case inputs for visiting new paths. As our search strategy for exploring of candidate paths, we adopted a hybrid approach that combines CCS (Code Coverage Search) and DFS. CCS focuses on exploring code areas that have not been traversed yet, making it advantageous for quickly reaching high coverage. However, because CCS performs rather aggressive pruning on execution paths, it may produce unsatisfiable path conditions in certain situations. To make up for these issues, we also use the DFS strategy in addition to CCS. DFS is a search strategy that has the potential to cover code areas not covered by CCS, but it comes with the drawback of substantial time consumption. Usually, either of these strategies terminates once every branch it has discovered has been explored. Concolic execution may however also be terminated early if a designated amount of test cases has been generated or if a timeout has been reached.

Finally, a significant factor influencing the performance of concolic execution in C++ is the memory model. Sim-

QuASoQ 2023 Preprint

Table 1
Results on Open-Source Projects

Project Info					Coverage		Test Time
Name (C/C++)	Files	Functions	Statements	Branches	Statement	Branch	[m]
nuklear (C)	39	609	9,284	4,309	93.7%	87.1%	55
libsodium (C)	94	887	8,003	1,651	96.5%	89.7%	6
mathc (C)	1	843	4,192	190	99.9%	100.0%	3
aubio (C)	53	520	5,916	1,797	95.7%	92.4%	14
s2n-tls (C)	175	1,621	16,734	15,512	86.7%	81.3%	68
yaml-cpp (C++)	32	367	3,050	1,300	96.9%	95.5%	11
qnite (C++)	48	637	4,294	1,035	95.2%	89.1%	37
json-voorhees (C++)	21	451	2,507	764	92.5%	88.7%	5
QPULib (C++)	24	278	3,561	1,398	87.8%	83.8%	3
jsoncpp (C++)	3	309	2,802	1,148	91.2%	86.3%	11
Total	490	6,522	60,343	29,104	93.6%	89.4%	213

Table 2
Coverage Results from Hyundai KEFICO

Project Info					Coverage		Test Time
Name	Files	Functions	Statements	Branches	Statement	Branch	
Target A	1,855	5,129	129,131	40,718	92.8%	86.8%	N/A
Target B	83	1,774	11,828	3,078	97.4%	90.7%	
Target C	69	375	6,526	2,339	85.5%	79.9%	
Total	2,007	7,278	147,485	46,135	92.9%	86.7%	

ilar to MAYHEM, the approach implemented in Coyote C++ reads values from memory symbolically but writes values to concrete memory addresses. Utilizing symbolic reads in contrast to reading from concrete addresses leads to a more faithful representation of path constraints, thereby enhancing the potential for generating appropriate test cases. For write operations however, we chose to rely on concrete addresses because symbolic writes are prone to making the process of solving the path constraints overly expensive.

4. Experimental Results

To showcase the performance of Coyote C++, we present experimental results for a set of diverse open-source projects as well as several industrial software projects from one of our customers, Hyundai KEFICO. While our tool allows user to add test cases and write driver functions for achieving higher coverage, all experimental results were obtained through one-click automation without any user intervention.

4.1. Experiment on Open-Source Projects

For the first evaluation, we chose to reuse the test set curated by Shin and Yoo for a survey on white-box automated testing tools [24], as it contains open-source projects written in C and C++ from a wide variety of application domains and was composed specifically for the evaluation of automated testing tools such as Coyote C++. This survey also concluded that currently no other commercial tools truly support automated testing for C++ programs. Among open-source tools for C++, CITRUS [12] is no longer publicly available, and we were not able to successfully apply UTBot [14] to the selected test projects due to its rather limited support for the C++ syntax. Thus, unfortunately there were no suitable candidates to compare Coyote C++ against in terms of coverage and test time.

Table 1 shows the statement¹ and branch coverage results achieved by Coyote C++ on the ten open-source projects in the test set as well as the time needed for conducting the automated test generation and execution for each project. Coyote C++ achieves statement coverages between 86.7% (s2n-tls) and 99.9% (mathc) as well

¹As statements we consider only executable lines of code. In contrast to physical lines of code, this excludes e.g. whitespace, comments, and type declarations.

as branch coverages between 81.3% (s2n-tls) and 100% (mathc). Summing up the number of overall covered lines/branches and dividing them by the total number of lines and branches in all ten projects yields a remarkable combined statement coverage of 92.5% and branch coverage of 84.9%.

The test times presented in table 1 were attained from an Intel Core i7-13700 system with 64GB of RAM running Ubuntu 20.04. Overall, the test of all ten projects combined only took about three and a half hours, with individual testing times ranging between three minutes (mathc) and just above one hour (s2n-tls). That makes it more than six times faster than the test times reported in the previously mentioned study [24], which we consider a significant improvement despite possible minor differences between test setups. Furthermore, with the exception of the qnite project, the testing speed on all projects surpasses our definition of practicality, with an overall testing speed of roughly 17,000 statements per hour.

4.2. Results on Industry Projects

Table 2 presents testing results produced by Coyote C++ on automotive control software projects from our customer Hyundai KEFICO, a member of Hyundai Motors Group. As details about these projects such as their actual names are strictly internal information, we will refer to them as target A, B and C.

The coverage results for these industrial projects are quite similar to the open-source projects, with an average statement coverage of 92.9% and an average branch coverage of 86.7%. At our customer, Coyote C++ is employed not in a controlled test environment but rather in a business setting on multiple machines with varying hardware specifications. Due to these circumstances and the fact that a subset of the test results were produced incrementally over a longer period of time, we presently do not have any meaningful test time measurements available to report for these projects.

While project C individually yields a slightly subpar coverage, our notion of practicality in terms of coverage achieved (statement coverage >90%, branch coverage >80%) is upheld both by projects A and B individually as well as all three projects combined. This again reinforces our claim that Coyote C++ is not simply a research prototype which only works on a limited set of specially curated programs but is rather a mature tool that can also handle more challenging industry software. Also, it should be noted that automated testing with such high coverage results for these projects is only possible because Coyote C++ has explicit handling for some common code patterns in embedded software that would usually make automated testing difficult or plainly impossible, such as the usage of fixed memory addresses in code.

5. Conclusion and Future Work

In this paper, we presented Coyote C++, an industry-grade automated testing tool based on concolic execution. After describing the general tool architecture, we discussed the core design decisions for our implementation of its concolic testing engine. Finally, we evaluated the performance of Coyote C++ in terms of achieved coverage and testing time on both a test set of diverse open-source projects and industry code from one of our corporate customers. We were able to demonstrate that Coyote C++ can achieve high statement/branch coverage of around 90% or higher in a reasonable amount of time for software projects from a wide variety of application domains.

While Coyote C++ is already yielding promising results both on open-source projects and in real industry applications, it is our plan to continuously improve the tool both in terms of reliably achieving high coverage results and broadening its capabilities in the field of automated testing.

One goal for the near future is target testing for embedded software. Our tool currently performs host testing, meaning tests are not executed on the hardware that would run the program under test in a production environment, but rather on a separate computer, e.g., a test engineer's computer or a test server. Especially in the embedded domain however, the discrepancy between embedded hardware in the production environment and the consumer or server hardware in the testing environment may lead to inaccurate test results. Thus, we are planning to implement target testing support so that tests may be run directly on production hardware.

Approaching the goal of increasing automated test coverage from a different perspective, we also strive to provide users of our tool with feedback as to how they should change their code so that Coyote C++ will likely yield better coverage results for it. While we would like to give such guidance on the basis of code metrics, our initial investigations have shown that traditional code metrics such as cyclomatic complexity have little to no correlation with automated test coverage. Thus, we see the need for more thorough research involving the development of new code metrics that can serve as a better estimate for the coverage results produced by automated testing and Coyote C++ in particular.

References

- [1] L. Luo, Software testing techniques, Institute for software research international Carnegie mellon university Pittsburgh, PA 15232 (2001) 19.
- [2] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using EvoSuite, ACM

QuASoQ 2023 Preprint

- Trans. Softw. Eng. Methodol. 24 (2014). URL: <https://doi.org/10.1145/2685612>. doi:10.1145/2685612.
- [3] K. Sen, G. Agha, Cute and jcute: Concolic unit testing and explicit path model-checking tools, in: T. Ball, R. B. Jones (Eds.), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 419–423.
- [4] Y. Kim, D. Lee, J. Baek, M. Kim, Concolic testing for high test coverage and reduced human effort in automotive industry, in: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2019, pp. 151–160.
- [5] K. Sen, D. Marinov, G. Agha, CUTE: A concolic unit testing engine for C, *ACM SIGSOFT Software Engineering Notes* 30 (2005) 263–272.
- [6] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random testing, in: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.
- [7] P. Godefroid, M. Y. Levin, D. Molnar, SAGE: white-box fuzzing for security testing, *Communications of the ACM* 55 (2012) 40–44.
- [8] F. Saudel, J. Salwan, Triton: A dynamic symbolic execution framework, in: *Symposium sur la sécurité des technologies de l’information et des communications, SSTIC*, France, Rennes, 2015, pp. 31–54.
- [9] N. Tillmann, J. de Halleux, Pex–white box test generation for .net, in: B. Beckert, R. Hähnle (Eds.), *Tests and Proofs*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 134–153.
- [10] A. Giantsios, N. Papaspyrou, K. Sagonas, Concolic testing for functional languages, *Science of Computer Programming* 147 (2017) 109–134. URL: <https://www.sciencedirect.com/science/article/pii/S0167642317300837>. doi:<https://doi.org/10.1016/j.scico.2017.04.008>.
- [11] K. Sen, S. Kalasapur, T. Brutch, S. Gibbs, Jalangi: A selective record-replay and dynamic analysis framework for javascript, in: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, Association for Computing Machinery, New York, NY, USA, 2013, p. 488–498. URL: <https://doi.org/10.1145/2491411.2491447>. doi:10.1145/2491411.2491447.
- [12] R. S. Herlim, Y. Kim, M. Kim, CITRUS: Automated unit testing tool for real-world C++ programs, in: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 400–410. doi:10.1109/ICST53961.2022.00046.
- [13] J. C. King, A new approach to program testing, *ACM Sigplan Notices* 10 (1975) 228–233.
- [14] D. Ivanov, A. Babushkin, S. Grigoryev, P. Iatchenii, V. Kalugin, E. Kichin, E. Kulikov, A. Misonizhnik, D. Mordvinov, S. Morozov, et al., UnitTest-Bot: Automated unit test generation for C code in integrated development environments, in: *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, 2023, pp. 380–384.
- [15] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, *ACM Comput. Surv.* 51 (2018). URL: <https://doi.org/10.1145/3182657>. doi:10.1145/3182657.
- [16] C. Cadar, D. Dunbar, D. R. Engler, et al., KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs., in: *OSDI*, volume 8, 2008, pp. 209–224.
- [17] S. K. Cha, T. Avgerinos, A. Rebert, D. Brumley, Unleashing Mayhem on binary code, in: *IEEE Symposium on Security and Privacy, SP 2012*, 21–23 May 2012, San Francisco, California, USA, IEEE Computer Society, 2012, pp. 380–394. URL: <http://doi.ieeecomputersociety.org/10.1109/SP.2012.31>. doi:10.1109/SP.2012.31.
- [18] V. Chipounov, V. Kuznetsov, G. Candea, The S2E platform: Design, implementation, and applications, *ACM Transactions on Computer Systems (TOCS)* 30 (2012) 1–49.
- [19] F. Bellard, QEMU, a fast and portable dynamic translator., in: *USENIX annual technical conference, FREENIX Track*, volume 41, California, USA, 2005, p. 46.
- [20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, SoK: (state of) the art of war: offensive techniques in binary analysis, in: *IEEE Symposium on Security and Privacy*, 2016.
- [21] I. Yun, S. Lee, M. Xu, Y. Jang, T. Kim, QSYM: A practical concolic execution engine tailored for hybrid fuzzing, in: *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.
- [22] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446. doi:10.1109/ASE.2008.69.
- [23] S. Cha, S. Hong, J. Bak, J. Kim, J. Lee, H. Oh, Enhancing dynamic symbolic execution by automatically learning search heuristics, *IEEE Transactions on Software Engineering* 48 (2022) 3640–3663. doi:10.1109/TSE.2021.3101870.
- [24] K. Shin, Y. Ryu, Performance and functionality evaluation of white-box software testing tools, part 2, <https://csrc.kaist.ac.kr/blog/2023/01/25/performance-and-functionality-evaluation-of-white-box-software-testing-tools-part-2/>, 2023. Accessed: 2023-10-11.

Software Defect Prediction based on JavaBERT and CNN-BiLSTM

Kun Cheng¹, Shingo Takada²

¹Grad. School of Science and Technology, Keio University Yokohama, Japan

²Grad. School of Science and Technology, Keio University Yokohama, Japan

Abstract

Software defects can lead to severe issues in software systems, such as software errors, security vulnerabilities, and decreased software performance. Early prediction of software defects can prevent these problems, reduce development costs, and enhance system reliability. However, existing methods often focus on manually crafted code features and overlook the rich semantic and contextual information in program code. In this paper, we propose a novel approach that integrates JavaBERT-based embeddings with a CNN-BiLSTM model for software defect prediction. Our model considers code context and captures code patterns and dependencies throughout the code, thereby improving prediction performance. We incorporate Optuna to find optimal hyperparameters. We conducted experiments on the PROMISE dataset, which demonstrated that our approach outperforms baseline models, particularly in leveraging code semantics to enhance defect prediction performance.

Keywords

Software defect prediction, JavaBERT, CNN, BiLSTM, Optuna,

1. Introduction

Software defects present significant challenges to the reliability and performance of software systems, often leading to critical issues such as slow software operation, frequent security vulnerabilities, and software crashes. To address these challenges, researchers have turned their attention to software defect prediction (SDP), a key research area aimed at identifying potentially problematic code early in the development process.

Software Defect Prediction (SDP) is a structured process involving data preprocessing, feature extraction, model building, and evaluation[1]. Feature extraction plays a pivotal role in SDP as it determines the model's data representation. SDP methods have traditionally relied on manual feature engineering, a process involving time-consuming and laborious manual design. However, this approach faces challenges in capturing complex semantics and contextual information embedded in software code as systems become more complex. As a result, there's a growing demand for advanced techniques that can effectively exploit the intrinsic semantic and structural meaning of code, along with its statistical properties.

Recent advances in SDP have shifted towards leveraging structural and semantic features directly from source code or through parsing into an abstract syntax tree (AST)[2]. These modern methods employ these features

in combination with various classification methods, encompassing both traditional algorithms and deep learning techniques.

SDP encompasses two primary domains: Cross-Project Defect Prediction (CPDP) and Within-Project Defect Prediction (WPDP). CPDP involves training a model on one project and applying it to another, addressing the challenge of generalization across different software environments. In contrast, WPDP focuses on building models within the same project, enhancing defect prediction performance by considering unique project characteristics and evolution patterns. For the purpose of this study, our primary focus lies on WPDP, aiming to improve defect prediction performance within a single project.

In this paper, we introduce an innovative approach to SDP that combines Java Bidirectional Encoder Representations of Transformers (JavaBERT) and Convolutional Neural Networks with Bidirectional Long Short-Term Memory (CNN-BiLSTM). By harnessing JavaBERT's contextual understanding of text data and CNN-BiLSTM's capacity to capture structural features, we improve defect prediction performance. Furthermore, we optimize the model's hyperparameters by introducing Optuna, further refining our predictive model.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 presents the design of our proposed approach. Section 4 covers the implementation details based on the design, and Section 5 offers the evaluation results along with a discussion of potential threats to validity. Finally, Section 6 concludes the paper and discusses future work.

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, South Korea

✉ chengkun@keio.jp (K. Cheng); michigan@ics.keio.ac.jp (S. Takada)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

2. Related Work

Researchers have explored various models for feature extraction in software defect prediction, from traditional machine learning to deep learning. Initially, Support Vector Machines (SVM), as employed by Elish et al.[3], gained prominence for identifying defective modules using static code metrics. However, it struggled to uncover deep semantics within the source code. Deep Belief Networks (DBN), introduced by Wang et al.[4], aimed to extract more complex features from code through unsupervised learning. Yet, its limited depth posed challenges in revealing intricate relationships within the source code. Convolutional Neural Networks (CNNs) were used by Li et al.[5] to predict software defects by analyzing structural correlations between code tokens. While proficient in capturing local patterns, CNNs faced challenges in capturing longer-range connections. Wang et al.[6] introduced an RNN (Recurrent Neural Network)-based model for predicting software reliability. Deng et al.[7] and Liang et al.[8] expanded Long Short-Term Memory (LSTM) models in software defect prediction, capturing temporal patterns in code sequences. However, a single LSTM can only capture one direction temporal pattern in the code sequence. Bidirectional LSTM (BiLSTM) models with attention mechanisms emerged. Wang et al.[9] introduced a gated hierarchical BiLSTM model. Uddin et al.[10] combined BiLSTM with attention and BERT-based embeddings.

In short, SVM has difficulty discovering the deep semantics of the source code, DBN has limited depth so it is difficult to understand the complex relationships in the source code, CNN has difficulty capturing long-distance correlations, and RNN and LSTM can only capture a single temporal pattern. BiLSTM may have challenges in capturing local patterns.

To solve these problems, we combine the advantages of CNN in detecting local patterns with the advantages of BiLSTM in processing sequences, allowing for comprehensive code inspection. We further incorporate JavaBERT to dynamically adjust token embeddings based on the entire input sequence, thereby deepening the representation and capturing interdependencies among code tokens.

3. Proposed Methodology

Our software defect prediction method consists of several key steps, all aimed at improving prediction performance. As shown in Figure 1, we first use JavaBERT to convert the code into vector representations. Next, we employ the CNN-BiLSTM model for feature extraction, focusing on local patterns and context. We also incorporate statistical features to fully utilize all available information.

Optuna automatically executes the above combination of JavaBERT and CNN-BiLSTM multiple times, and outputs the best hyperparameter values through these executions. Then we retrain the model in another version of the code based on the obtained hyperparameters and test the model performance.

3.1. Embedding with JavaBERT

BERT (Bidirectional Encoder Representations from Transformers)[11] is a language model widely employed in natural language processing (NLP) tasks. Unlike conventional embeddings, BERT excels at capturing intricate contextual associations. Traditional methods like Word2Vec[12] and GloVe[13] generate static contextual representations, whereas BERT, utilizing multi-layer bidirectional transformers, enables tokens to gather information from both preceding and succeeding tokens.

In our approach, we leverage a pretrained BERT model, JavaBERT[14], fine-tuned for Java code. JavaBERT has been trained on a dataset of 2,998,345 Java files from GitHub open source projects. JavaBERT's transformer architecture dynamically adapts token embeddings based on the entire input sequence, enhancing representation depth and capturing code token interdependencies. The JavaBERT embeddings, denoted as E_{JavaBERT} , are computed by applying the model's encoder to tokenized Java code. For a sequence of code tokens $C = \{c_1, c_2, \dots, c_n\}$, JavaBERT embeddings are computed as:

$$E_{\text{JavaBERT}} = \text{Encoder}_{\text{JavaBERT}}(c_1, c_2, \dots, c_n)$$

Models typically cannot process code text sequences directly. Through JavaBERT, we embed code text into a continuous vector space, using these vectors as inputs to the model, making it easier for the model to compute and understand the code.

3.2. Feature Extraction using CNN-BiLSTM

We combine Convolutional Neural Networks (CNN) and Bidirectional Long Short-Term Memory networks (BiLSTM) to extract features. This is the key part of our approach, where after extracting features with CNN, it is refined with the sequential capabilities of BiLSTM.

3.2.1. Feature Extraction with CNN

Utilizing Convolutional Neural Networks (CNN)[15] for feature extraction involves sliding a small window, known as a filter, over various parts of the code. This filter examines a small segment of the code at a time, calculating a value at each sliding position to create a "feature map."

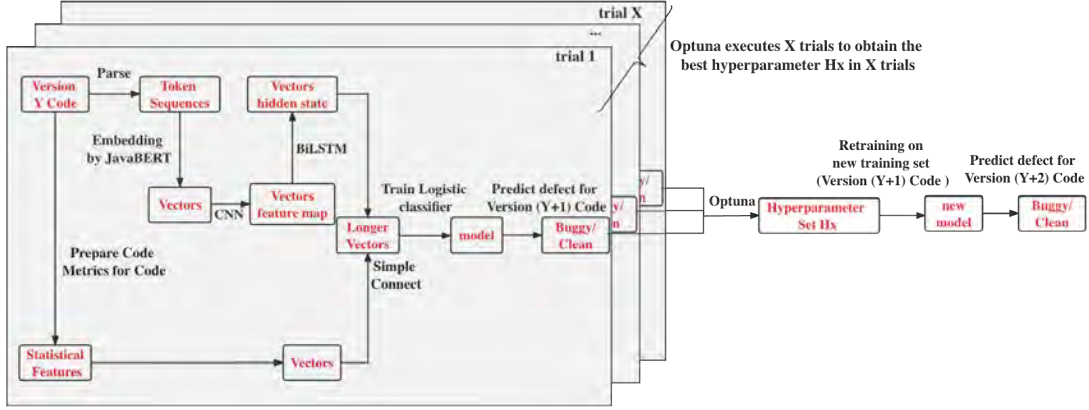


Figure 1: Overview of Methodology

The positions in the code correspond to positions in the feature map. The observed code segment within the filter’s scope is termed the "input sequence slice." As the filter traverses the entire code, it analyzes these input sequence slices, effectively capturing distinct features that characterize the code’s structural and syntactical elements.

The process of feature extraction using CNN is mathematically expressed as:

$$y[i, j] = \sigma \left(\sum_m \sum_n x[i + m, j + n] \cdot w[m, n] + b \right)$$

where $x[i, j]$ is the input at position (i, j) , $w[m, n]$ represents the kernel at position (m, n) , b is the bias, and σ signifies the activation function.

3.2.2. Refinement of Features with BiLSTM

The Bidirectional Long Short-Term Memory (BiLSTM)[16] layer enhances the features extracted by the Convolutional Neural Networks (CNN). What sets BiLSTM apart is its capability to capture both short-term and long-term dependencies within the code, which perfectly complements the local feature extraction carried out by CNN.

The forward and backward computations in BiLSTM can be unified into a single mathematical representation:

$$h_t = \text{BiLSTM}(x_t, h_{t-1}, h_{t+1})$$

In this equation, h_t represents the hidden state at time step t in the Bidirectional Long Short-Term Memory (BiLSTM) model. It is computed based on the input x_t at the current time step, the previous hidden state h_{t-1} , and

the next time step $t + 1$ ’s hidden state h_{t+1} . The BiLSTM model effectively captures sequential patterns and dependencies in data by considering information from both directions. It analyzes the sequence of tokens, capturing dependencies extending both backward and forward within the code. This dynamic construction of code features considers token order, revealing evolving patterns and connections over time, amplifying the feature representation. In summary, we refine the feature maps obtained from CNN using BiLSTM to achieve a comprehensive code representation. This fusion of capturing local patterns and accounting for temporal dependencies improves software defect prediction performance.

3.3. Integration with Statistical Features

Our methodology integrates the refined BiLSTM outputs with statistical features (such as shown in Table 2) extracted from dataset. This step concatenates the vectors obtained from the BiLSTM and the vectors of statistical features obtained from the dataset into longer vectors, making full use of the description information of the code.

3.4. Hyperparameter Optimization by Optuna

Optuna, a powerful hyperparameter optimization framework developed by Akiba et al.[17], plays a vital role in our approach by automating hyperparameter tuning for the CNN-BiLSTM model. There are similar frameworks such as Ray Tune, etc., but Optuna is more lightweight and easier to use. It employs the Tree-structured Parzen Estimator (TPE) algorithm to efficiently explore and exploit the hyperparameter space, enhancing the performance of our Software Defect Prediction task.

In this section, we will discuss a crucial step in our methodology: determining optimal hyperparameters by leveraging shared features among different versions of the same project. Usually, code with similar version numbers exhibits a high degree of similarity. By harnessing these inherent similarities, we attempt to find hyperparameters that can generalize across various versions, ultimately enhancing model performance.

Using the Ant project as an example, our aim is to demonstrate the transferability of hyperparameters obtained from training on one version (e.g., 1.5) to another (e.g., 1.6). This transferability is valid as both versions originate from the same project, sharing similar code structures and functionalities. This enables the hyperparameters obtained from one version to serve as a foundation for other versions within the same project, thereby solidifying our model configuration.

We start by selecting version pairs, using the Ant project as an illustration. Here, we designate version 1.5 for training and version 1.6 for testing. Next, we define the performance metric to optimize, such as the F1 score. Subsequently, Optuna conducts multiple experiments, traversing various hyperparameter combinations and evaluating their performance on the designated testing dataset. Through these iterative experimentation and evaluation stages, Optuna determines the hyperparameter set that maximizes the chosen performance metric.

This process can be represented as:

$$H_x = \text{Optuna } f(\text{Ant 1.5}, \text{Ant 1.6})$$

Here, $f(\text{Ant 1.5}, \text{Ant 1.6})$ embodies the objective function maximized during the hyperparameter optimization process, with Ant 1.5 as the training dataset and Ant 1.6 as the testing dataset. After obtaining optimal hyperparameters H_x through the Optuna process, we seamlessly transfer them across different project versions. H_x is applied to reconfigure the training and testing sets. For instance, in the Ant project, H_x is then used on different version pairs, such as training on Ant 1.6 with H_x and testing on Ant 1.7.

This operation optimizes hyperparameters across version pairs, contributing to enhanced model adaptability and performance in varying project iterations.

4. Experimental Setup

4.1. Research Questions

Our experiment addresses the following research questions (RQ) :

RQ1: How does the performance of our CNN-BiLSTM model compare against baseline models?

Table 1

Selected Projects in the PROMISE Java Dataset

Project	Versions (Buggy Rate)
Ant	1.5, 1.6, 1.7 (0.109, 0.263, 0.224)
Camel	1.2, 1.4, 1.6 (0.36, 0.171, 0.201)
JEdit	3.2, 4.0, 4.1 (0.346, 0.256, 0.263)
Lucene	2.0, 2.2, 2.4 (0.489, 0.611, 0.615)
Poi	2.0, 2.5, 3.0 (0.120, 0.654, 0.641)
Synapse	1.0, 1.1, 1.2 (0.102, 0.270, 0.336)
Xalan	2.4, 2.5, 2.6 (0.163, 0.509, 0.468)

RQ2: How does the performance of the proposed model vary across different software projects and within the different versions of each project in the PROMISE dataset?

RQ3: How do different hyperparameter settings impact the performance of the combined CNN-BiLSTM model in code defect prediction?

4.2. Dataset and Data Preprocessing

Our study uses the PROMISE[18] dataset, exclusively comprised of Java projects. This dataset spans various domains and project scales, providing project details like name, description, version, and bug rate. Table 1 shows an overview of the projects we use that are in the PROMISE Java Dataset. Since Optuna’s process of finding hyperparameters takes a lot of time, we only selected a part of the projects in the PROMISE data set. Statistical features also play a vital role in code analysis, offering insights into code composition and behavior. To enhance our study, we carefully selected a subset of these features, as shown in Table 2.

To prepare the data for analysis, we conducted thorough data preprocessing. Using the "javalang"[19] Python library, we removed redundant code elements such as comments, white spaces, and unnecessary details. This process allowed us to extract essential token sequences, capturing the code’s semantics. To address class imbalance in software defect prediction, we implemented random oversampling exclusively on the "Bug" class files. This deliberate strategy generated synthetic data instances, improving class distribution and mitigating potential bias towards the majority class.

4.3. Experimental Settings

For each project listed in Table 1, we selected the smallest two version numbers to serve as versions Y and Y+1 for Optuna’s hyperparameter optimization process. The search space for the hyperparameters was specified as shown in Table 3. The number of trials for each project was set to 30. After completing these experiments, each project will produce a different set of hyperparameters

Table 2
Selected Statistical Features

Measure of Functional Abstraction (MFA)
Coupling Between Methods (CBM)
Data Access Metric (DAM)
Coupling Between Object classCA (CBO)
Lines Of Code (LOC)
Afferent Couplings (CA)
Number Of Children (NOC)
Lack of COhesion in Methods (LCOM)
Average Method Complexity (AMC)
Inheritance Coupling (IC)
Response For a Class (RFC)
Efferent Couplings (CE)
Measure Of Aggregation (MOA)
Weighted Methods per Class (WMC)
Depth of Inheritance Tree (DIT)
Lack of COhesion in Methods (LCOM3)
Cohesion Among Methods of class (CAM)
Number of Public Methods (NPM)

that allow the model to output the highest F1 score, and a model trained on these parameters using version Y. These hyperparameters were then applied to train new models on version Y+1 for each project. Then the model trained on version Y and the model trained on version Y+1 were evaluated against the code of version Y+2. We conducted each evaluation test three times and calculated the mean to obtain the experimental result.

Table 3
Search Space for Hyperparameters

Hyperparameter	Search Range
Number of Epochs	3 to 10
Batch Size	[16, 32, 64, 128]
Learning Rate	1×10^{-5} to 1×10^{-2} (Log-uniform)
Filter Sizes	[3, 5, 7, 9, 11]
Number of Filters	32 to 512
Hidden Units	[16, 32, 64, 128, 256, 512, 1024]

4.4. Baseline Models

We compare our proposed approach against the following baseline models:

- Support Vector Machine (SVM): SVM, a classic and widely adopted machine learning algorithm, excels in both linear and non-linear classification tasks and is known for its effectiveness in handling high-dimensional data.

- Convolutional Neural Network (CNN): CNNs excel at extracting hierarchical features from structured data, making them suitable for capturing local patterns in software defect prediction.
- Bidirectional Long Short-Term Memory (BiLSTM): BiLSTM enhances LSTM by considering bidirectional information flow, enabling it to capture both past and future contexts.

In assessing the predictive performance, this paper utilizes three widely accepted metrics: precision, recall, and the F1-score.

5. Results and Discussion

In this section, we present the results of our study and discuss their implications, addressing the research questions (RQ) that guide our investigation.

5.1. Impact of JavaBERT-based Embeddings with CNN-BiLSTM Model

To address RQ1, we assessed the performance of our model in comparison to baseline models. Table 4 presents a detailed performance comparison between our CNN-BiLSTM model and the baseline models concerning precision, recall, and F1-score. For instance, "ant_1.5_1.6" represents the experimental results obtained by using version 1.5 of Ant as the training dataset and version 1.6 as the test dataset. The results demonstrate a consistent outperformance of our model across all metrics. Figure 2 complements the table by providing a visual representation of the F1 scores, where the x-axis represents pairs of software versions used for training and testing (e.g., ant_1.5_1.6), and the y-axis represents the corresponding F1 values obtained during testing. This figure shows that the F1 of our model is higher than the base model most of the time.

5.2. Model Performance Variability Across PROMISE Projects and Versions

To address RQ2, Figure 3 presents the F1 scores of our model across different projects and their respective versions in the PROMISE dataset. In this figure, the x-axis represents pairs of software versions used for training and testing (e.g., ant_1.5_1.6), while the y-axis represents the corresponding F1 values obtained during testing. When we examined the model's performance across different projects and its various versions, we observed certain noteworthy patterns. Specifically, within the same project,

QuASoQ 2023 Preprint

Table 4
Comparison of Experimental Results with Baseline Models

project	SVM			CNN			BiLSTM			CNN-BiLSTM		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ant_1.5_1.6	0.5133	0.6304	0.5659	0.5526	0.4565	0.5000	0.5120	0.6957	0.5899	0.6364	0.6848	0.6597
ant_1.6_1.7	0.5849	0.5602	0.5723	0.5230	0.5482	0.5353	0.2486	0.5241	0.3372	0.5868	0.5904	0.5886
ant_1.5_1.7	0.4297	0.6807	0.5268	0.3653	0.7349	0.4880	0.4194	0.7048	0.5258	0.5531	0.5964	0.5739
camel_1.2_1.4	0.3645	0.5103	0.4253	0.5625	0.4966	0.5275	0.3186	0.4966	0.3881	0.4647	0.7724	0.5803
camel_1.4_1.6	0.2687	0.0957	0.1412	0.5392	0.2926	0.3793	0.2908	0.3883	0.3326	0.4957	0.3032	0.3762
camel_1.2_1.6	0.5000	0.1915	0.2769	0.3571	0.1862	0.2448	0.2908	0.3883	0.3326	0.3976	0.5266	0.4531
jedit_3.2_4.0	0.4333	0.1733	0.2476	0.4715	0.7733	0.5859	0.5208	0.6667	0.5848	0.4741	0.8533	0.6095
jedit_4.0_4.1	0.3835	0.7727	0.5126	0.5889	0.6709	0.6272	0.5517	0.7273	0.6275	0.7838	0.3671	0.5000
jedit_3.2_4.1	0.4783	0.1667	0.2472	0.5039	0.8101	0.6214	0.5455	0.7273	0.6234	0.4803	0.7722	0.5922
lucene_2.0_2.2	0.7681	0.4454	0.5638	0.6918	0.7692	0.7285	0.7571	0.4454	0.5608	0.6371	0.9875	0.7745
lucene_2.2_2.4	0.6923	0.7310	0.7111	0.6329	0.6650	0.6485	0.7739	0.4518	0.5705	0.6204	0.9806	0.7600
lucene_2.0_2.4	0.6120	0.9848	0.7549	0.6339	0.7208	0.6746	0.7768	0.4416	0.5631	0.6204	0.9806	0.7600
poi_2.0_2.5	0.6996	0.6573	0.6778	0.6781	0.3992	0.5025	0.8053	0.7339	0.7679	0.7240	0.9839	0.8342
poi_2.5_3.0	0.8560	0.7429	0.7954	0.7038	0.7214	0.7125	0.8547	0.7143	0.7782	0.6943	0.9571	0.8048
poi_2.0_3.0	0.7436	0.7250	0.7342	0.7333	0.5893	0.6535	0.8559	0.7214	0.7829	0.7034	0.9571	0.8109
synapse_1.0_1.1	0.4815	0.2281	0.3095	0.5946	0.3860	0.4681	0.5000	0.3158	0.3871	0.5077	0.5789	0.5410
synapse_1.1_1.2	0.5152	0.3953	0.4474	0.5417	0.4535	0.4937	0.5439	0.3605	0.4336	0.5190	0.4767	0.4970
synapse_1.0_1.2	0.5455	0.2791	0.3692	0.5634	0.4651	0.5096	0.5273	0.3372	0.4113	0.4483	0.4535	0.4509
xalan_2.4_2.5	0.6609	0.4258	0.5179	0.5922	0.4059	0.4817	0.5721	0.3221	0.4122	0.5957	0.7176	0.6510
xalan_2.5_2.6	0.6494	0.5221	0.5788	0.6274	0.6397	0.6335	0.5344	0.3431	0.4179	0.5804	0.8848	0.7010
xalan_2.4_2.6	0.6333	0.5123	0.5664	0.6506	0.2647	0.3763	0.5344	0.3431	0.4179	0.5983	0.8431	0.6999
Average	0.5626	0.4967	0.5020	0.5766	0.5452	0.5425	0.5588	0.5166	0.5164	0.5772	0.7270	0.6295

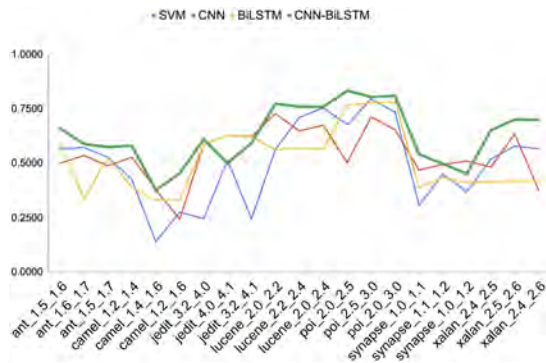


Figure 2: F1 Score Comparison Visualization

such as Lucene, POI, and Xalan, our models show a high degree of performance consistency across different versions. This shows that our model is able to predict results consistently when dealing with different versions of certain projects. This consistency can be partially attributed to the higher code similarity found between versions within the same project, making it easier for models to capture shared features and patterns.

There are some differences between versions of Ant and Synapse, these differences are relatively minor. In contrast, projects such as Camel and JEdit show more performance fluctuations, even within the same project. This

suggests that the predictive performance of our model tends to vary when applied to certain projects. Although we cannot pinpoint the exact reasons behind these changes at this time, we speculate that they may have been influenced by a variety of factors, including project-specific characteristics, code complexity, and domain-related differences.

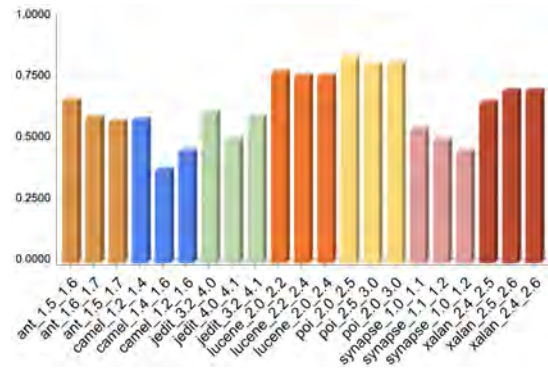


Figure 3: F1 Score Across PROMISE Projects

Table 5
Hyperparameter combinations obtained through Optuna

proj	Optuna Time	num_epochs	batch_size	learning_rate	filter_size	num_filters	rnn_hidden
ant_1.5_1.6	8.18 h	8	64	0.000185	3	186	256
camel_1.2_1.4	30.92 h	7	32	0.000148	7	120	1024
jedit_3.2_4.0	4.32 h	6	32	0.000251	11	157	64
lucene_2.0_2.2	1.21 h	3	128	0.008864	9	178	128
poi_2.0_2.5	3.93 h	6	128	0.000015	3	240	256
synapse_1.0_1.1	3.09 h	7	64	0.000458	5	346	64
xalan_2.4_2.5	35.23 h	5	128	0.000232	5	194	16

5.3. The impact of hyperparameters on the performance of CNN-BiLSTM model

To address RQ3, in this section, we study the impact of hyperparameters on the performance of the CNN-BiLSTM model for code defect prediction. Initially, we set the hyperparameters to the following values: the number of epochs is 10, the batch size is 64, the learning rate is 1e-4, the number of CNN filters is 128, the number of BiLSTM hidden units is 256, and the CNN filter size is 5. After that, we fixed other hyperparameters, and then gradually manually adjust one of the other parameters, the CNN filter or the number of BiLSTM hidden units, to observe changes in model performance.

Figure 4 and Figure 5 show our experimental results, the x-axis is the change in the number of CNN filters and BiLSTM hidden units, and the y-axis shows the F1 score. We can see that the model performance fluctuates greatly when a single parameter changes. For example, the smaller the number of CNN filters, the better the performance of the model. In Figure 5, the F1 score drops after BiLSTM hidden unit is 16, but performs better and tends to be stable after 256. Exploring the impact of each hyperparameter individually would be a time-consuming task, and it is difficult to predict how the model will behave when these hyperparameters are combined. So we used Optuna, which will constantly try to search for hyperparameters that can make the model perform better based on the search algorithm.

Figures 6 and 7 show the F1 score (y-axis) for a certain number of trials (x-axis). Specifically, Figure 6 is a scatter plot, representing the F1 score that was obtained in each trial, e.g., when the trial number is 5, the F1 score is the value for the fifth trial. Figure 7 represents the best model performance that can be achieved based on the search until the current trial model is executed. So, in figure 7, when the trial number is 5, the F1 score is the best F1 score from the first to the fifth trial. We can observe that through continuous repetition and search, Optuna can gradually search for better results. The entire process is automated, which greatly simplifies our hyperparameter tuning process.

Table 5 provides a summary of hyperparameter combinations obtained through Optuna. These combinations have been identified to bring better performance for our code defect prediction model.

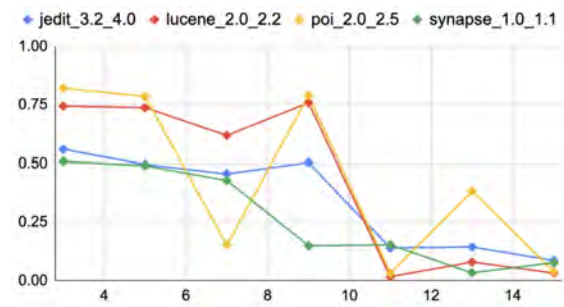


Figure 4: Effect of CNN Filter Length on F1 Score

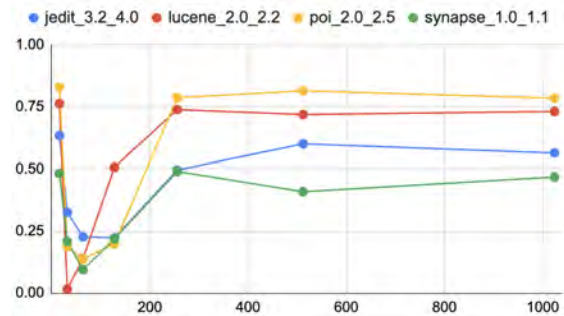


Figure 5: Effect of BiLSTM Hidden Units on F1 Score

5.4. Threats to Validity

In our research, we have identified and addressed several potential threats to the validity of our findings.

The implementation of our Python experimental code for processing source code text and building models poses a potential threat due to the possibility of bugs. To mitigate this, we took measures by leveraging mature third-party

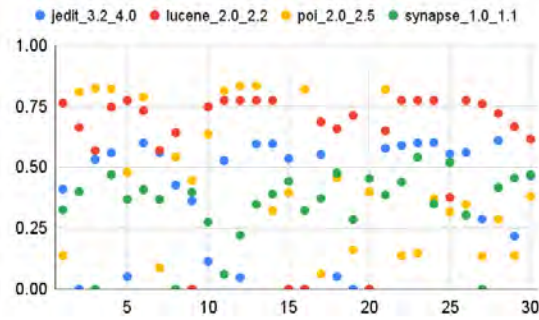


Figure 6: Scatter Plot of F1 Scores Across Optuna Trials

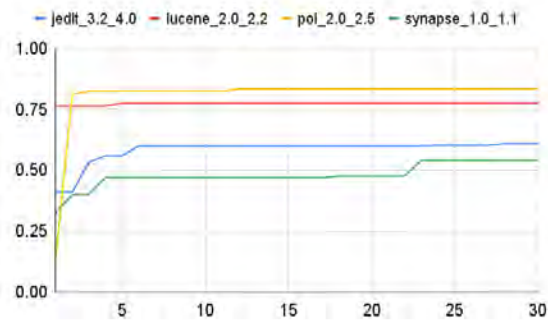


Figure 7: Progressive Improvement of Best Model Performance

libraries (such as javalang and PyTorch) and conducting thorough code inspections. Additionally, we applied random oversampling during data preprocessing, which could introduce bias. Future work will explore alternative methods to handle class imbalance and assess their impact on results. Moreover, the use of Optuna for hyperparameter optimization introduces potential variability in results due to different search spaces and numbers of trials. To reduce these threats, we plan to conduct more extensive searches and explore larger search spaces.

Our choice of a subset of projects from the PROMISE dataset due to time constraints may impact the generalizability of our findings, as the results may not generalize well to other projects. To address this, we intend to include a broader range of projects in future research.

We evaluated our models using a limited set of performance metrics, specifically precision, recall, and F1 measure. To reduce these threats, we will consider incorporating additional metrics such as AUC-ROC and MCC, among others, to provide a more comprehensive assessment of model performance.

6. Conclusion and Future Work

In this research, we have introduced a novel approach that leverages JavaBERT-based embeddings with a CNN-BiLSTM model for software defect prediction. Our approach harnesses semantic and contextual information in program code to enhance prediction accuracy. Through comprehensive experiments on the PROMISE dataset, we have demonstrated the superiority of our model over baseline models based on precision, recall, and F1-score metrics.

Although our study improves the performance of software defect prediction compared to baseline models, we still have many future works to do. In addition to what we discussed in the "threats to validity" session, we can also train the BERT model in different languages to adapt our methods to different programming languages.

References

- [1] S. Omri, C. Sinz, Deep learning for software defect prediction: A survey, in: Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops, 2020, pp. 209–214.
- [2] F. Meng, R. Huang, J. Wang, A survey of software defects research based on deep learning, in: 2023 6th International Conference on Information Systems and Computer Networks (ISCON), IEEE, 2023, pp. 1–5.
- [3] K. O. Elish, M. O. Elish, Predicting defect-prone software modules using support vector machines, Journal of Systems and Software 81 (2008) 649–660.
- [4] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 297–308.
- [5] J. Li, P. He, J. Zhu, M. R. Lyu, Software defect prediction via convolutional neural network, in: 2017 IEEE international conference on software quality, reliability and security (QRS), IEEE, 2017, pp. 318–328.
- [6] J. Wang, C. Zhang, Software reliability prediction using a deep learning model based on the RNN encoder–decoder, Reliability Engineering & System Safety 170 (2018) 73–82.
- [7] J. Deng, L. Lu, S. Qiu, Software defect prediction via LSTM, IET software 14 (2020) 443–450.
- [8] H. Liang, Y. Yu, L. Jiang, Z. Xie, Seml: A semantic LSTM model for software defect prediction, IEEE Access 7 (2019) 83812–83824.
- [9] H. Wang, W. Zhuang, X. Zhang, Software defect prediction based on gated hierarchical LSTMs, IEEE Transactions on Reliability 70 (2021) 711–727.

QuASoQ 2023 Preprint

- [10] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, I. Zada, Software defect prediction employing BiLSTM and BERT-based semantic feature, *Soft Computing* 26 (2022) 7877–7891.
- [11] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, *arXiv preprint arXiv:1810.04805* (2018).
- [12] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, *Advances in neural information processing systems* 26 (2013).
- [13] J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.
- [14] N. T. De Sousa, W. Hasselbring, JavaBERT: Training a transformer-based model for the Java programming language, in: *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, IEEE, 2021, pp. 90–95.
- [15] K. Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, *Biological cybernetics* 36 (1980) 193–202.
- [16] M. Schuster, K. K. Paliwal, Bidirectional recurrent neural networks, *IEEE transactions on Signal Processing* 45 (1997) 2673–2681.
- [17] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework, in: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.
- [18] J. Sayyad Shirabad, T. Menzies, *The PROMISE Repository of Software Engineering Databases.*, School of Information Technology and Engineering, University of Ottawa, Canada, 2005. URL: <http://promise.site.uottawa.ca/SERepository>.
- [19] C. Thunes, javalang: pure Python Java parser and tools, 2020.

QuASoQ 2023 Preprint

Proposals for Improving the Assessment of Medical Device Software in Thailand

Natsuda Kasisopha¹, Songsakdi Rongviriyapanich¹, and Panita Meananeatra²

¹ Thammasat University, 99 Phahonyothin Road, Khlong Nueng, Khlong Luang District, Pathum Thani 12120, Thailand

² National Electronics and Computer Technology Center (NECTEC), 112 Phahonyothin Road, Khlong Nueng, Khlong Luang District, Pathumthani 12120, Thailand

Abstract

The registration of Medical Device Software (MDS) and Software as a Medical Device (SaMD) with the Food and Drug Administration (FDA) is a prerequisite before entering the market. The FDA relies on several international standards as regulatory benchmarks to ensure the quality of MDS. Key components of this regulatory framework include IEC 62340 [1], ISO 14971 [2], and ISO 13485 [3]. Our experience assessing MDS in Thailand highlighted common challenges manufacturers face during software evaluation. Notably, clause 6 (Software Maintenance Process) and clause 8 (Software Configuration Management Process) demonstrate the highest rates of evaluation failure. Clause 7 (Software Risk Management Process) and clause 9 (Software Problem Resolution Process) closely follow, ranking as the second-highest areas of concern regarding evaluation failures. The primary factor contributing to these software evaluation challenges is a deficiency in knowledge and understanding of IEC 62304 [1]. To address this issue, we propose a solution in the form of a chatbot designed to assist manufacturers in comprehending and generating IEC 62304-compliant documents.

Keywords

medical device software, software as a medical device, Thais' medical device software assessor, medical device software obstacle, medical device software quality assessment.

1. Introduction

In Thailand, Medical Device Software (MDS) and Software as a Medical Device (SaMD) are required to register with the Thailand Food and Drug Administration (Thai FDA) [4]. The Thai FDA has established criteria aligned with international standards, including mainly ISO/IEC 62304:2006 - "Medical device software - Software life cycle processes" (IEC 62304) [1], ISO 14971:2019 - "Medical devices - Application of risk management to medical devices" (ISO 14971) [3], ISO 13485:2016 - "Medical devices - Quality management systems - Requirements for regulatory purposes" (ISO 13485) [3], and IEC 60601-1 clause 14 (IEC 60601), which pertains to Programmable Electrical Medical Systems (PEMS) for medical electrical devices [5]. These standards outline the processes, activities, and configuration tasks that form a holistic framework for developing MDS.

IEC 62304 [1] encompasses six processes outlined in clauses 4 to 9. Each clause specifies a breakdown into sub-clauses, activities, and tasks. These sub-clauses are interconnected with other clauses and sub-

clauses within the standard. For instance, sub-clause 4.2 (Risk Management) illustrates its correlation with clause 7 (Software Risk Management Process). Sub-clause 4.2 of IEC 62304 is also interconnected with additional standards, such as ISO 14971 [2]. Furthermore, for manufacturers attaining ISO 13485 [3], adherence to ISO 14971 [2] for risk management is implicitly fulfilled. The visual representation of these interrelations between standards is depicted in Figure 2.

Moreover, IEC 62304 [1] establishes connections with IEC 60601 [5], clause 14, primarily through clauses 4.3 (Software Safety Classification), 5 (Software Development Process), 7 (Software Risk Management Process), 8 (Software Configuration Process), and 9 (Software Problem Resolution Process). The standard comprehensively addresses aspects of Software Life Cycle Processes, encompassing Quality Management Systems (QMS), Software Development Processes (SDP), Software Requirement Specification (SRS), Software Maintenance Process (SMP), Software Risk Management (SRM), Software Configuration Management (SCM), Software System Testing (ST), and related components, as well as the Software Problem Resolution Process (SPR). These elements are essential for the assessment of MDS.

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, South Korea

✉ natsuda.kas@dome.tu.ac.th (N. Kasisopha), rongviriri@tu.ac.th (S. Rongviriyapanich); panita.meananeatra@nectec.or.th (P. Meananeatra)



© 2023 Copyright for this paper by its authors. The use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

© 2023 Copyright for this paper by its authors

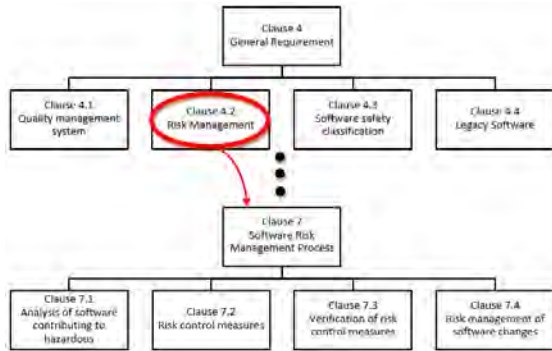


Figure 1: Interrelation between sub-clauses within IEC 62304.

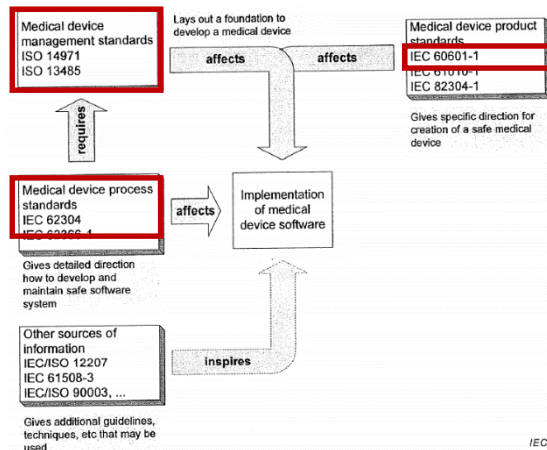


Figure 2: MDS Life Cycle [1].

All MDS must undergo testing in adherence to these standards, following the stipulations of the Thai FDA [6] requirements. The procedure for registering MDS with the Thai FDA is detailed in Section 1.1 of the Regulatory Framework for Medical Device Software in Thailand. Additionally, section 1.2 describes the MDS Software Quality Assurance and Assessment ecosystem, providing a comprehensive overview of the processes and standards involved in ensuring the quality, safety, and regulatory compliance of MDS in the Thai context.

1.1. Regulatory Framework for Medical Device Software in Thailand

The oversight of MDS falls under the scope of the Medical Device Control Division (MDCD) of the Thai FDA [6]. Thai FDA [6] relies on the Health and Science Authority (HSA) of Singapore [7]. The Thai FDA [6] mandates a two-step process for registering MDS and other medical devices. In the first step, known as "Establishment Licensing," the medical device manufacturers must provide business registration documents, complete request forms, and submit other relevant government documents. This step aims to verify the manufacturer credentials, enabling oversight of the quality of medical devices by restricting importation locations for production and storage. The second

step, "Product Registration," necessitates manufacturers to submit comprehensive documentation about the medical device. This includes details such as the device description, intended use, indications, instructions for use, storage conditions, shelf life, contraindications, warnings, precautions, potential adverse effects, alternative therapy options, materials used, product specifications, and the production development flow chart. The submission must align with the essential principles of safety and performance of the medical device as stipulated by the ASEAN Medical Device Directive, EU regulations, Singapore standards, and other applicable guidelines.

Moreover, the submission should summarize verification and validation, incorporating pre-clinical studies, clinical evidence, test reports, clinical evaluation reports, and clinical data. Additionally, the marketing history and safety declaration template documentation must be included. The inclusion of risk management processes that comply with ISO 14971, such as the risk plan, risk control measures, and the risk report, is imperative. A valid certificate of compliance with ISO 13485 or GMP for medical devices and ISO 9001 should be part of the submission. Lastly, the package should also contain a declaration of conformity and a letter of authorization.

Manufacturers must submit documentation to the Thai FDA's E-Submission system to adhere to the product registration process. The submitted documents will be meticulously examined and evaluated in alignment with the risk classification of the medical device to verify compliance with regulatory standards. In the event of uncertainties or the need for additional information, the Thai FDA communicates with the manufacturer. This interaction serves the purpose of seeking clarification and ensuring that all requisite details are accurately furnished. Following a successful review and approval, the Thai FDA issued a certificate for the medical device. The type of certificate, whether "listed", "notified", or "licensed", depends on the risk classification assigned to the device. Subsequently, the issued certification allows the manufacturer to gain authorization to manufacture or import the medical device in Thailand [8].

1.2. Medical Device Software Quality Assurance and Assessment Ecosystem

Medical Device Software Quality refers to the comprehensive set of characteristics, standards, and processes established to ensure that software integrated into medical devices meets predefined quality criteria. This commitment encompasses various elements to guarantee the software's safety, effectiveness, and reliability.

The Medical Device Software Assessment Ecosystem functions as a holistic framework, orchestrating crucial processes, adhering to standards, involving stakeholders, and utilizing tools to evaluate software integrated into medical devices' quality, safety, and regulatory compliance. This complex ecosystem, which is based on established standards such as IEC 62304 [1], ISO 13485 [3], and ISO 14971 [2], provides

a solid foundation for assessment processes. Quality Management Systems (QMS) are pivotal, overseeing the entire software development life cycle and ensuring meticulous documentation and training. The ecosystem integrates robust risk management processes, verification and validation (V&V) activities, and configuration management, as well as change control procedures. Internal and external audit mechanisms gauge adherence to quality standards, while post-market surveillance mechanisms monitor the real-world performance of the software.

Before the release of the MDS to the market, the manufacturer developed the medical device in compliance with established standards. Subsequently, the documentation is forwarded to a testing laboratory for verification and validation according to the IEC 62304 [1] standards. The resulting test report, upon release, is utilized for submission to either the Certification Body (CB) or the Regulatory Body (RB). Once the MDS has successfully undergone registration procedures from the Thai FDA [6], the manufacturer is then authorized to release the MDS to the market to the consumer.



Figure 3: MDS pre-market activities in Thailand.

The MDS assessment approach thoroughly examines the MDS development processes through documentation examination. This process involves a detailed analysis of the system's internal components, ensuring a systematic and comprehensive testing process and other elements of the software life cycle mentioned in Section 1. However, manufacturers who fail to provide the mentioned elements or only partially offer them may be required to request alterations to add information to the document. Manufacturers who do not give any information would fail the testing outcome.

The assessor evaluates three key components of the documents: completeness, accuracy, and consistency of the submitted information. These criteria ensure that the documentation adequately reflects the development processes and meets IEC 62304 [1] in the assessment approach.

The assessment report, also known as the test report, guarantees standard compliance during software development through product release phases. This verifies the safety of the system and confirms the proper functioning of the MDS. The guarantee emphasizes that the development process has been rigorous and thorough, ensuring that the MDS meets user requirements and is fit for use. Additionally, it assures compliance with specified standards of accuracy, safety, and regulatory requirements.

The complete regulatory frameworks. The essential components of quality assurance and assessment ecosystems require delving into the intricate development process, testing, and regulatory compliance.

This work aims to gain insights into robust processes and frameworks that govern the development

and deployment of MDS, contributing to the broader landscape of healthcare technology.

2. Literature Review

The literature encompasses a diverse range of topics related to regulatory compliance and software evaluation of MDS. Literature delves into the regulation's framework compliance to physical medical devices and MDS in the EU. Furthermore, another piece of literature investigates the evaluation of MDS by the Australian Therapeutic Goods Authority (TGA) [9], emphasizing standards including IEC 62304 [1], ISO 14971 [2], and ISO 13485 [3].

The study published by Granlund et al. [10] extensively explores medical devices under the CE mark [11] and the European Commission (EU) [12], focusing on the regulatory frameworks and challenges associated with their evaluation and development. The research highlights the reliance on the Council Directive 93/42/EEC on Medical Devices (MDD) [13] and MEDDEV [14] documents for a standardized application within the CE mark [11] and EU [12]. The challenge organizations face in developing MDS to meet the regulatory requirements of medical devices is that there is no distinction between physical medical devices and standalone software criteria, by classifying both as medical devices.

The paper highlights several regulatory requirement mismatches between physical medical devices and standalone MDS, such as the design change approval process, the use of public cloud computing platforms, the regulation of artificial intelligence and machine learning, and the implementation of a quality management system. The authors emphasize the need for a more streamlined software development and certification process and precise AI/ML-driven systems guidelines. They also suggest that smaller manufacturers could benefit from cooperation or partnerships to navigate the complexities of regulatory compliance in the cloud computing environment.

Ceross and Bergmann's [15] study focuses on recalls and adverse events associated with Software as a Medical Device (SaMD) in Australia. SaMD is distinguished from medical devices with software, and data is collected from three Australian Therapeutic Goods Authority (TGA) [9] databases. The analysis reveals over ninety cases of recall and adverse events for SaMD, with fewer than thirty cases for medical devices with software. The study identifies challenges in risk evaluation associated with SaMD, citing limited regulatory vocabulary for software defects as a key obstacle. The need for regulatory vocabulary support for software developers during early-stage research and development is emphasized, and integration into computer science courses is proposed.

This literature exposes regulatory challenges across various SaMD types stemming from misinterpretation and a lack of guidance. Existing regulatory requirements do not adequately support diverse SaMD development processes and AI MDS. Identifying these challenges underscores the necessity for a tool to assist manufacturers in overcoming significant obstacles in MDS development.

3. Medical Device Software Evaluation

The Software Quality Testing Laboratory (SQUAT) [16] is Thailand's first software testing laboratory certified with TIS 17025 (ISO/IEC 17025 [17]) (Certificate No.: 19T016/0793) by the Thai Industrial Standards Institute (TISI) [18], under the Ministry of Industry. Operating under the Software Engineering and Product Testing Section (SEPT) at the National Electronics and Computer Technology Center (NECTEC) [19], SQUAT is dedicated to verifying system performance by following the criteria outlined in IEC 62304.

Having conducted many MDS evaluations at SQUAT, the challenges encountered while evaluating MDS became evident. Twenty-three MDS evaluation cases from different manufacturers were analyzed, comprising twenty systems classified as Software Safety Class A and three systems classified as Software Safety Class B. The evaluation results, categorized into Pass and Fail for each IEC 62304 [1] clause, revealed that six out of twenty-three manufacturers achieved a fully-passed result. At the same time, the remaining seventeen had a failed outcome.

Further analysis indicated a predominant trend of more failed results than passed in each IEC 62304 [1] clause across all cases. Notably, only clause 4 had a higher pass rate, with twelve cases passing and eleven failing. However, clause 7 has the second highest pass rate, with ten cases passing and thirteen failing. The other four clauses resulted in a majority of failed assessments. Clauses 5 and 9 had eight passed cases and fifteen failed cases. Meanwhile, clauses 6 and 8 showed similar patterns of seven passed and 16 failed results.

In clause 4, the documentation lacks details regarding the decomposition of the software system into software items. Moreover, when a software item is further decomposed into additional software items, these inherit the software safety classification of the original software item (or software system) unless the manufacturer provides a rationale for classifying them differently. Additionally, the rationale should elucidate how the new software items are separated to warrant distinct classification. Suppose the software safety class of a newly created software item differs from the class of the software item from which it was decomposed. In that case, the manufacturer must document the safety class of each software item. Furthermore, there is often an absence of information regarding the identification of legacy software, the rationale for its use, and the risk management associated with legacy software.

In clause 5, specifically under sub-clause 5.1 (Software Development Planning), the deliverables, which encompass documentation of activities and tasks, often fall short of achieving the intended goals. The planning related to software configuration and change management, including software configuration items, system integration, verification and validation, risk management, and the software development process, exhibits a high incidence of failure. In sub-clause 5.2 (Software Requirement Analysis), manufacturers frequently fail to identify all software requirements, such as functional and capability requirements, software

system inputs and outputs, interfaces with other systems, software-driven alarms, warnings, and operator messages, security requirements, user interface requirements implemented by software, data definition and database requirements, installation and acceptance requirements at the operation and maintenance site, requirements related to methods of operation and maintenance, IT-network aspects, user maintenance requirements, and regulatory requirements.

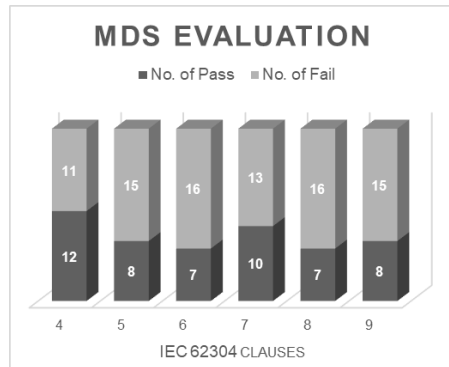


Figure 4: Passed/failed results of MDS evaluations according to IEC 62304.

Moreover, in sub-clause 5.7 (Software System Testing), there is a failure to provide documentation in uniformity with a) reference to test case procedures showing required actions and expected results, b) the test result (pass/fail and a list of anomalies); c) the version of software tested; d) relevant hardware and software test configurations; e) relevant test tools; f) date tested; and g) the identity of the person responsible for executing the test and recording the test results. Lastly, in sub-clause 5.8 (Software Release for Utilization at a System Level), the manufacturer must establish procedures to ensure the released MDS can be reliably delivered without corruption or unauthorized change. These procedures should address the production and handling of MDS media, including replication, media labeling, packaging, protection, storage, and delivery, as appropriate.

In clause 6 (Software Maintenance Process), there is a deficiency in having a software maintenance plan to conduct activities and tasks related to receiving, documenting, evaluating, resolving, and tracking. The usage of the software problem resolution process for analyzing and resolving issues that arise after the release of the MDS is often not adequately addressed. In sub-clause 6.2 (Problem and Modification Analysis), the documentation and evaluation of feedback to ascertain the existence of a problem in a released MDS is either not generated or inconsistently conducted. Additionally, there is a lack of effective implementation of the software problem resolution process to generate problem reports. Consequently, the evaluation and approval of change requests based on the problem reports fail to be addressed appropriately. As a result, there is a failure to identify the approved change requests that impact the released MDS.

In sub-clause 6.3 (Modification Implementation), the manufacturer must modify the instructions outlined in clause 5. Additionally, the release of modifications must align with the provisions specified in 5.8

4. Experience-based Solution

(Software Release for Utilization at a System Level), but these requirements are frequently not fulfilled.

In clause 7 (Software Risk Management Process), there is a failure to maintain the risk management of software changes under sub-clause 7.4. The manufacturer must identify hazardous situations, conduct risk analysis, and implement software risk control measures corresponding to those situations. This ensures an evaluation of potential hazards that may arise following software changes.

In clause 8 (Software Configuration Management Process), most cases fail in sub-clause 8.2 (Change Control). Manufacturers must identify and perform any activities that need to be repeated due to the change, including changes to the software safety classification of software systems and software items. However, manufacturers often fail to verify the change, neglecting to repeat any verification invalidated by the change and failing to account for 5.7 (Software System Testing) and 9.7. Additionally, in sub-clause 8.3, most manufacturers fail to retain retrievable records of the history of controlled configuration items.

For clause 9 (Software Problem Resolution Process), most manufacturers failed to identify and present the process for problem reporting, investigating, and evaluating emerging problems and communicating the problem's existence to relevant parties, as appropriate. The manufacturer approves and implements all change requests, ensuring adherence to the requirements of the change control process. Furthermore, the manufacturer maintains records of problem reports and their resolution, including verification, and updates the risk management file as appropriate. Additionally, the manufacturer analyzes to detect trends in problem reports. Conducting testing, retesting, or regression testing of software items and systems after changes is essential. The manufacturer is required to include the following elements in the test documentation: a) test results, b) anomalies found, c) the version of software tested, d) relevant hardware and software test configurations, e) relevant test tools, f) date of the test, and g) identification of the tester.

The obstacles that resulted in unsuccessful MDS evaluations primarily stemmed from language translation issues and a limited understanding of the interconnected nature of Software Engineering and IEC 62304 [1]. These challenges led to incomplete document submissions, generating uncertainty about the necessary content inclusion. Additionally, manufacturers, mainly with an engineering background, encountered difficulties comprehending the standard's contextual nuances. Lastly, adherence to IEC 62304 [1] guidelines faced constraints due to copyright limitations.

The challenges identified in the MDS evaluation process underscore the critical need for targeted solutions to enhance understanding, compliance, and effective documentation, particularly in adherence to IEC 62304 [1]. The issues identified, such as language translation complexities, limited comprehension of software engineering principles, and constraints related to copyright, highlight the need for a framework that manufacturers navigate during the evaluation process.

Based on experience, various solutions, including short course training (onsite training), information on websites, and other technologies, have been explored to address the challenges highlighted in the preceding section.

Short course training emerges as a promising solution, offering instructors who elucidate the nature and ecosystem of IEC 62304 [1]. The exercises conducted during these courses prove beneficial in helping trainees grasp the concepts and context of IEC 62304 [1]. However, the associated costs of short course training can be prohibitively high, and the inflexible location and schedule may pose challenges for trainees. While hiring a consultant is an effective solution, its affordability remains a concern for manufacturers. Alternatively, numerous websites provide information and explanations on IEC 62304 [1] but lack a structured outline or instructions on applying the standards and producing required documents.

A potential solution lies in the utilization of chatbots. These AI-powered tools offer a simple, quick, and flexible means of assisting manufacturers in creating IEC 62304 [1] documentation. Embedding chatbots into websites or instant messaging software can offer support for IEC 62304 [1] knowledge. The recent release of ChatGPT [20] provides an opportunity, although developing a similar chatbot poses challenges.

This chatbot can be divided into two parts: one for learning user-entered keywords and sentences and another for understanding the regulatory framework, including IEC 62304 [1]. This involves training the bot to fetch essential template links and files for users. The chosen technology for this endeavor is Botpress [21], primarily because of its compatibility with WordPress websites, enabling seamless chatbot integration into an existing platform.

The Botpress [21] architecture for addressing inquiries related to the IEC 62304 [1] standard is structured to provide an intelligent and adaptable chatbot experience. Users interact with the system via a user interface connected to the Botpress Core [21]. The Integration with Generative AI [22], exemplified by models like GPT-3 [23], enhances language understanding and facilitates content generation. User inputs undergo Natural Language Processing (NLP) [24] to identify intent and context, directing queries to the IEC 62304 Query Handler, which interprets and retrieves relevant information from the knowledge base. External resources are accessed through connectors, and an AI training interface ensures ongoing knowledge base updates. The architecture incorporates security measures, logging, analytics tools for user interactions, multi-channel support, and a continuous improvement module that collects feedback for iterative enhancement. The workflow of Botpress [21] is illustrated in Figure 5.

In conclusion, exploring solutions based on a range of experiences emphasizes the potential of chatbots and generative AI to address challenges in comprehending and applying IEC 62304 standards [1].

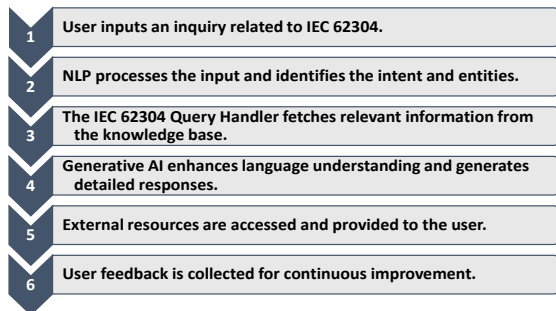


Figure 5: Workflow of Botpress [21].

5. Conclusion

In conclusion, the presented guidelines for improving MDS assessment in Thailand offer a comprehensive framework to enhance MDS's quality, safety, and regulatory compliance. The importance of adhering to international standards, such as IEC 62304 [1], ISO 13485 [3], and ISO 14971 [2], has been underscored throughout the guidelines, emphasizing the need for a robust quality management system.

Incorporating innovative solutions, including integrating chatbots using technologies like Botpress [21], showcases a forward-looking approach to addressing challenges in understanding and implementing complex standards. By leveraging AI-driven tools, manufacturers can benefit from quick, flexible, and accessible support in creating IEC 62304 [1] documentation, ultimately contributing to streamlined processes and improved compliance.

Furthermore, the guidelines advocate for a continuous improvement mindset, focusing on ongoing training, user feedback, and data analysis to adapt to evolving standards and industry best practices. The emphasis on multi-channel support, security measures, and the incorporation of generative AI highlights a commitment to creating a comprehensive and user-friendly ecosystem for MDS assessment.

Overall, these guidelines provide a roadmap for manufacturers, assessors, and regulatory bodies in Thailand to navigate the intricate landscape of MDS assessment, fostering a culture of quality, innovation, and regulatory adherence in the rapidly advancing field of healthcare technology.

References

[1] ISO. "IEC 62304:2006/Amd 1:2015 Medical device software - Software life cycle processes - Amendment 1." <https://www.iso.org/standard/64686.html>.

[2] ISO. "ISO 14971:2019 Medical devices - Application of risk management to medical devices." <https://www.iso.org/standard/72704.html>.

[3] ISO. "ISO 13485:2016." <https://www.iso.org/iso-13485-medical-devices.html>.

[4] "FDA THAI: Food and Drug Administration, Thailand." <https://en.fda.moph.go.th/entrepreneurs-medical-devices/category/how-to-apply-for-permission-on-medical-devices/>.

[5] "IEC 60601-1:2005+AMD1:2012+AMD2:2020 CSV | IEC Webstore." <https://webstore.iec.ch/publication/67497>.

[6] "Thai FDA" <https://en.fda.moph.go.th/home>.

[7] "Health Sciences Authority (HSA)." <https://www.hsa.gov.sg> (accessed 2022-08-23).

[8] "FDA THAI: Food and Drug Administration, Thailand." [Online]. <https://en.fda.moph.go.th/entrepreneurs-medical-devices/category/how-to-apply-for-permission-on-medical-devices/>.

[9] T. G. Administration. "Therapeutic Goods Administration (TGA) | Australian Government Department of Health." <https://www.tga.gov.au/>

[10] T. Granlund, T. Mikkonen, and V. Stirbu, "On medical device software CE compliance and conformity assessment," in 2020 IEEE International Conference on software architecture companion (ICSA-C), 2020: IEEE, pp. 185-191.

[11] "CE marking," https://single-market-economy.ec.europa.eu/single-market/ce-marking_en.

[12] "European Commission, official website," 2023/11/15/ 2023. https://commission.europa.eu/index_en.

[13] CONSIL, (1993, 1993/06/14/). OJ L 169, Council Directive 93/42/EEC of 14 June 1993 concerning medical devices.

[14] "MEDDEV Guidance List - Download," in Medical Device Regulation, ed.

[15] A. Ceros and J. Bergmann, "Evaluating the presence of software-as-a-medical-device in the Australian therapeutic goods register," *Prosthesis*, vol. 3, no. 3, pp. 221-228, 2021.

[16] "Software Quality Testing Laboratory (SQUAT)." <https://www.squat.in.th/>.

[17] "ISO - ISO/IEC 17025 — Testing and calibration laboratories," ISO, 2021/01/26/ 2021. [Online]. Available: <https://www.iso.org/ISO-IEC-17025-testing-and-calibration-laboratories.html>.

[18] "Thai Industrial Standards Institute (TISI)." <https://www.tisi.go.th/home/en>.

[19] "Home - NECTEC: National Electronics and Computer Technology Center," ed, 2007.

[20] "Introducing ChatGPT,". <https://openai.com/blog/chatgpt>.

[21] "Botpress | the Generative AI platform for ChatGPT Chatbots," <https://botpress.com/>.

[22] "Generative AI", Generative AI. <https://generativeai.net>.

[23] "GPT-3," in Wikipedia, ed, 2023.

[24] "Natural language processing," in Wikipedia, ed, 2023.

Identifying Vulnerable Functions from Source Code using Vulnerability Reports

Rabaya Sultana Mim, Toukir Ahammed and Kazi Sakib

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Abstract

Software vulnerability represents a flaw within a software product that can be exploited to cause the system to violate its security. In the context of large and evolving software systems, developers find it challenging to identify vulnerable functions effectively when a new vulnerability is reported. Existing studies have underutilized vulnerability reports which can be a good source of contextual information in identifying vulnerable functions in source code. This study proposes an information retrieval based method called Vulnerable Functions Detector (VFDetector) for identifying vulnerable functions from source code and vulnerability reports. VFDetector ranks vulnerable functions based on the textual similarity between the vulnerability report corpora and the source code corpora. This ranking is achieved modifying conventional Vector Space Model to incorporate the size of a function which is known as the tweaked Vector Space Model (tVSM). As an initial study, the approach has been evaluated by analysing 10 vulnerability reports from six popular open-source projects. The result shows that VFDetector ranks the actual vulnerable function at first position in 40% cases. Moreover, it ranks the actual vulnerable function within rank 5 in 90% cases and within rank 7 for all analysed reports. Therefore, developers can use these results to implement successful patches on vulnerable functions more quickly.

Keywords

vulnerability identification, vulnerable function, vulnerability report, source code, vector space model

1. Introduction

A software vulnerability is a flaw, weakness, or error in a computer program or system that can be exploited by malicious attackers to compromise its integrity, availability, or confidentiality [1]. Software vulnerabilities make software systems increasingly vulnerable to attack and damage, which raises security concerns [2].

Developers need to spend a lot of time in identifying vulnerable function from large codespace when a new vulnerability is reported. Identifying vulnerable functions effectively is a prerequisite of writing a patch for the reported vulnerability. This is essential for enhancing software security by addressing vulnerabilities to mitigate potential risks and threats more effectively at earliest time.

Existing studies have focused on detecting software vulnerabilities employing text-based [3, 4, 5] or graph-based [6, 7] approaches. These approaches either treat source code as plain text or apply graph analysis by representing the source code as graph. In practice, prior text-based studies treat source code as plain text and apply static program analysis or natural language processing. However, the performance of these approaches is not optimal for disregarding the source code semantics. On the other hand, graph based approaches conduct


program analysis which represent the source code semantics as a graph, and then apply graph analysis methods such as Graph Neural Networks (GNN) [8] to identify vulnerabilities. Although these graph-based approaches are more efficient at identifying vulnerabilities taking into account the semantic relationship of various lines of source code but their scalability is substantially less than that of text-based approaches.

However, existing studies have underutilized vulnerability reports which can be a good source of contextual information to detect vulnerability in source code. In this context, the current study aims to verify the role of vulnerability reports in identifying vulnerable function. Vulnerability report can contain contextual information about a vulnerability which may be used to identify vulnerable functions. When a function is vulnerable against a scenario some keywords should be shared between that function and the vulnerability report. These motivate the authors to study whether vulnerable functions can be identified by analysing the source code and vulnerability report.

For this purpose, this study proposes a technique of automatic software vulnerable function identification namely VFDetector. It takes all source code files of a system as input. First, it extracts all source code functions of that system. Then static analysis is performed to extract the contents of those functions. Several text pre-processing analysis such as tokenization, stopwords removal, multiword splitting, semantic meaning extraction and lemmatization are applied on these source code along with the vulnerability report to produce code and

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, Korea

✉ msse1730@iit.du.ac.bd (R. S. Mim); toukir@iit.du.ac.bd (T. Ahammed); sakib@iit.du.ac.bd (K. Sakib)

 © 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

QuASoQ 2023 Preprint

report corpora. In addition, programming language specific keywords is removed for generating code corpora. Finally, to rank the vulnerable functions, similarity scores are measured between the code corpora of the functions and report corpora by a modified version of Vector Space Model (tVSM) where larger methods get more weight while ranking.

In experiments, as an initial study ten Common Vulnerabilities and Exposures (CVE) reports are chosen randomly from six open source GitHub repositories. Based on the commit link available in reports we crawled the corresponding projects before the vulnerability was patched. The result analysis shows that VFDetector ranks the vulnerable functions at the first position in 40% cases, whereas it ranks the actual vulnerable function within top 5 in 90% cases and within top 7 in 100% cases.

It is evident from the results that VFDetector performs promisingly in detecting vulnerable functions against a vulnerability report in a large scale software systems. It is also observed that in Top 5 and Top 7 ranking, the functions which ranks above the actual vulnerable function are the related functions of that vulnerability which acquires higher similarity. It guides a developer to patch those related functions too in order to mitigate that vulnerability from the system.

The remainder of this paper is structured as follows: Section 2 gives an overview of previous studies on vulnerability detection at file level or function level. Section 3 describes our methodology for detecting vulnerable functions in a project. Section 4 reports our experimental findings and the analysis thereof. Section 5 demonstrates the threats to validity of our work. Section 6 motivates future research directions and concludes this paper.

2. Related Work

In recent years, the research community has directed significant attention toward the issue of vulnerability detection, primarily due to the complex challenges it presents. The existing body of literature has introduced numerous methodologies in response to these challenges. These methods can be classified into three distinct categories based on the degree of automation: manual, semi-automatic, and fully automatic techniques.

Manual techniques rely on human experts to create vulnerability patterns. However, all patterns can not be generated manually, which leads to reduced detection efficiency in practical scenarios. In contrast, semi-automatic techniques involve human experts in the extraction of specific features like API symbols [9] and function calls [10], which are then fed into traditional machine learning models for vulnerability detection. Full-automatic techniques utilize Deep Learning (DL) to automatically extract features and construct vulnerability patterns with

out manual expert intervention. Recently, DL based techniques [11, 12, 13] has gained extensive use in detecting source code vulnerabilities due to its ability to automatically extract features from source code. DL based methods can be categorized into text-based and graph-based methods.

Text based methods: The text-based approach in vulnerability detection treats a program's source code as text and employs natural language processing techniques to identify vulnerabilities. Russell et al. [3] introduced the TokenCNN model, which utilizes lexical analysis to acquire source code tokens and employs a Convolutional Neural Network (CNN) to detect vulnerabilities.

Li et al. [4] proposed Vuldeepecker, a method that collects code gadgets by slicing programs and transforms them into vector representations, training a Bidirectional Long Short Term Memory (BLSTM) model for vulnerability recognition.

Zhou et al. [5] introduced μ VulDeePecker, which enhances Vuldeepecker by incorporating code attention with control dependence to detect multi-class vulnerabilities. However, the performance of these text-based approaches is limited because they rely solely on static source code analysis and do not account for the program semantics.

Graph based methods: To address the limitations of text-based methods, researchers have turned to dynamic program analysis to convert a program's source code semantics into a graph representation facilitating vulnerability detection through graph analysis. Zhou et al. [6] introduced Devign which employs a graph neural network for vulnerability identification. This approach includes a convolutional module that efficiently extracts critical features for graph-level classification from the learned node representations. By pooling the nodes, a comprehensive representation for graph-level classification is achieved.

Cheng et al. [7] introduced a different approach named Deepwukong which divides the program dependency graph into various subgraphs after distilling the program semantics based on program points of interest. These subgraphs are then utilized to train a vulnerability detector through a graph neural network. While these graph-based techniques prove more effective in identifying vulnerabilities but it is important to note that their scalability is worse than text based methods due to large number of graph nodes in complex program.

Exploring the existing literature, it is evident that text-based methods lacks in incorporating program semantics while graph-based methods achieve high accuracy considering source code semantics but have scalability issues in complex scenarios. Moreover, due to the underutilization of contextual information like vulnerability reports with source code existing methods fails to detect complicated vulnerabilities in real-world projects. Because whenever

a new vulnerability is reported in a system it is hard to detect in which function the vulnerability exist as the system consist of huge volume of functions. Before using vulnerable reports as a source of contextual information in existing methods, it is important to verify whether vulnerable functions can be identified effectively using these reports. Moreover, identifying vulnerable functions using vulnerability reports can play an effective role to minimize the search space in existing methods.

3. Methodology

This study proposes an approach which detects vulnerable functions from huge volume of files of a large software system using vulnerability reports. The overall process of this approach consist of three distinct steps and those are Source Code Corpora Generation, Vulnerability Report Corpora Generation, Ranking Vulnerable Functions. Each of these steps encompasses a series of tasks as illustrated in Figure 1. At first, all files and their corresponding functions are extracted from a particular version of a software system. Then these source code is processed to create code corpora. Similarly vulnerability report is processed to produce report corpora. Finally, similarity between the report and code corpora is measured using tweaked Vector Space Model (tVSM) to rank the vulnerable source code functions.

3.1. Dataset

We used the benchmark dataset Big-Vul¹ developed by Fan et al. [14]. This dataset comprises reliable and comprehensive code vulnerabilities which are directly linked to the publicly accessible CVE database. Notably, the creation of this dataset involved a significant investment of manual resources to ensure its high quality. Furthermore, this dataset is noteworthy for its substantial scale, being one of the most extensive vulnerability datasets available. It is derived from a collection of 348 open-source Github projects, encompassing a time span from 2002 to 2019, and covers 91 distinct Common Weakness Enumeration (CWE) categories. This comprehensive dataset comprises approximately 188,600 C/C++ functions, with 5.6% of them identified as vulnerable (equivalent to 10,500 vulnerable functions). This dataset provides granular ground-truth information at the function level, specifying which functions within a codebase are susceptible to vulnerabilities.

¹https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset

3.2. Source Code Corpora Generation

Source code corpora consist of source code terms used to assess similarity with vulnerability report corpora. Therefore, the precision of code corpora generation directly impacts the precision of matching, consequently enhancing the accuracy of vulnerability localization. In this step all the folders are extracted from a system with their corresponding C/C++ files. From each of these files all functions are extracted automatically in individual C files which ensures function level analysis. For Example: CVE-2014-2038 of Linux version 3.13.5 consist of 15,675 files which has total 229,682 functions.

This stage generates a vector of lexical tokens by doing lexical analysis on every source code file. There are unnecessary tokens in source code which do not contain any vulnerability related information. These tokens are discarded from source code such as programming language specific keywords (e.g., int, if, float, switch, case, struct), stop words (e.g., all, and, an, the). Many words in the source code may include multiple words. For example, the term "addRequest" consists of the keywords "add" and "Request". Mutiwords are separated using multi word identifier. Furthermore, statements are divided according to certain syntax-specific separators like ' ', '=', '(', ')', '{', '}', '/', and so on. WordNet² is used to derive each word's semantic meanings because a term might have more than one synonym. In specific cases, developers and Quality Assurance (QA) personnel may employ different terminology, even though they are referring to the same scenario with equivalent meanings. For example, the term 'finalize' may have multiple synonyms such as 'conclude' or 'complete.' When describing a situation, if a developer uses 'finalize' but QA opts for 'conclude', it's challenging for the system to identify these variances without considering the semantic meanings of these words. Therefore, the extraction of semantic meaning is crucial in achieving accurate rankings for vulnerable functions.

The final stage of code corpora generation incorporates WordNet lemmatization, a technique that normalizes words to their base or dictionary form. WordNet lemmatization utilizes the comprehensive WordNet lexical database, organizing words into synonymous sets called synsets. This method identifies word lemmas based on the word's part of speech and context within WordNet, offering a more context-aware approach to lemmatization. As a result, it considers a word's meaning and contextual usage, allowing for precise reduction of words. For instance, it transforms "running" to "run" and "better" to "good" based on their meanings and parts of speech, unlike standard lemmatization that typically relies on suffix removal.

²<https://wordnet.princeton.edu/>

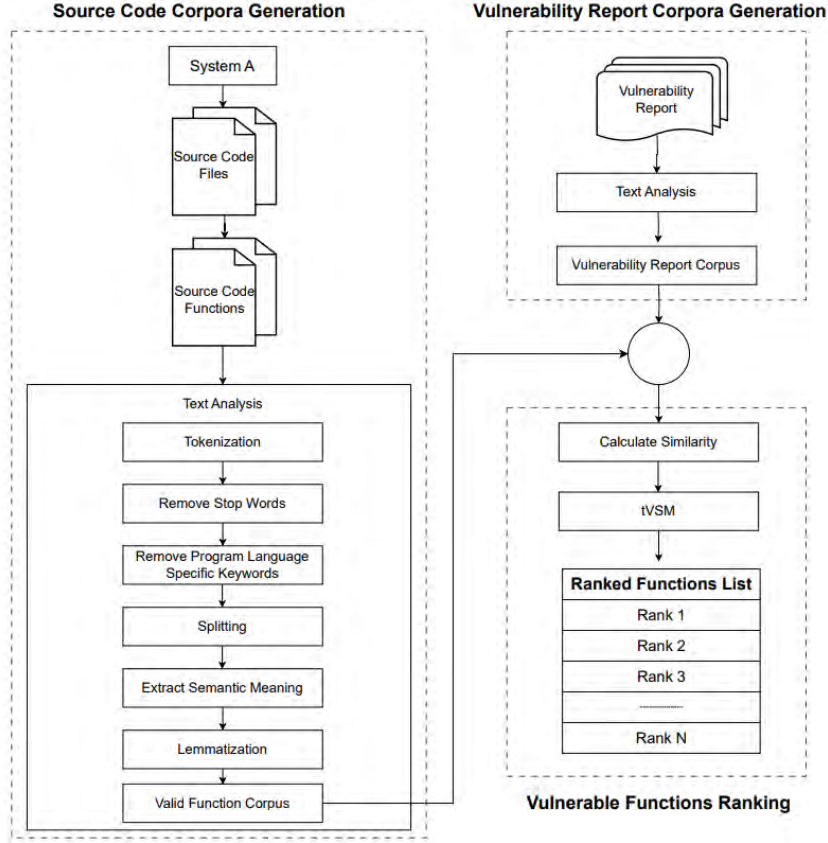


Figure 1: Overview of VFDetector

3.3. Vulnerability Report Corpora

A software vulnerability report contains information like description about the vulnerability, severity rating, vulnerability identifier (CVE-ID), reference to additional sources which gives valuable insights about a software vulnerability issue. However, these reports can also include irrelevant terms such as stop words and words in various tenses (present, past, or future). To refine vulnerability reports, pre-processing is necessary. In the initial stage of vulnerability report corpora creation, stop words are eliminated. We apply WordNet Lemmatizer, similar to what’s used for source code corpora generation, to generate refined report corpora containing only relevant terms.

3.4. Ranking Vulnerable Functions

In this step, relevant vulnerable functions are ranked based on the textual similarity between the query (report corpus) and each of the function in the code corpus. Vulnerable functions are ranked by applying tVSM. We

employ tVSM, which modifies the Vector Space Model (VSM) by emphasising large-scale functions. In traditional VSM, the cosine similarity is used to measure the ranking score between the associated vector representations of a report corpus (r) and function (f), according to Equation 1.

$$Similarity(r, f) = \cos(r, f) = \frac{\vec{V}_r \cdot \vec{V}_f}{|\vec{V}_r| \cdot |\vec{V}_f|} \quad (1)$$

Here, \vec{V}_r and \vec{V}_f are the term vectors for the vulnerability report (r) corpus and function (f) corpus respectively. Throughout the years, numerous adaptations of the $tf(t,d)$ function have been introduced with the aim of enhancing the VSM model’s effectiveness. These encompass logarithmic, augmented, and Boolean modifications of the traditional VSM [15]. It has been noted that the logarithmic version can yield improved performance, as indicated by prior studies [16, 17, 18]. From that point of view, tVSM modified Equation 1 and uses the logarithm of term frequency (tf) and idf (inverse function frequency) to give more importance on rare terms in the functions.

QuASoQ 2023 Preprint

Thus tf and iff are calculated using Equation 2 and 3 respectively.

$$tf(t, f) = 1 + \log f_{tf} \quad (2)$$

$$iff = \log\left(\frac{\#functions}{n_t}\right) \quad (3)$$

Here, f_{tf} represents the frequency of a term t appearing in a function f , $\#functions$ denotes the total count of functions within the search space, n_t signifies the overall number of functions that include the term t . Thus in equation 4 each term weight is calculated as follows:

$$\begin{aligned} weight_{t \in f} &= (tf)_{tf} \times (iff)_t \\ &= (1 + \log f_{tf}) \times \log\left(\frac{\#functions}{n_t}\right) \end{aligned} \quad (4)$$

The VSM score is calculated using equation 5.

$$\begin{aligned} \cos(r, f) &= \frac{\sum_{t \in r \cap f} (1 + \log f_{tr}) \times (1 + \log f_{tf}) \times iff^2}{1} \times \frac{1}{\sqrt{\sum (1 + \log f_{tr}) \times iff^2}} \times \frac{1}{\sqrt{\sum (1 + \log f_{tf}) \times iff^2}} \end{aligned} \quad (5)$$

Traditional VSM tends to give preference to smaller functions when ranking them, which can be problematic for large functions because they may receive lower similarity scores. Past research [19, 20, 21] has indicated that larger source code files are more likely to contain vulnerabilities. Therefore, in the context of vulnerability localization, it's crucial to prioritize larger functions in our ranking. To address this issue, we introduce a function denoted as 'x' (as shown in Equation 6) within the tVSM model, aiming to account for the function's length.

$$x(terms) = 1 - e^{-Normalize(\#terms)} \quad (6)$$

Equation 6 represents a logistic function, specifically an inverse logit function, designed to ensure that larger functions receive higher rankings. We employ Equation 6 to

calculate the length value for each source code function based on the number of terms contained within the function. Here we apply the normalized value of '#terms' as the argument for the exponential function e^{-x} . The normalization process is defined in Equation 7.

Let z denote a set of data, with z_{max} and z_{min} representing the maximum and minimum values of z term, respectively. The normalized value for z term is determined as:

$$Normalize(z) = \frac{z - z_{min}}{z - z_{max}} \quad (7)$$

Considering the above analysis, tVSM score is calculated by multiplying the weight of each function, denoted as $x(terms)$, with the cosine similarity score represented by $\cos(r, f)$, as described in Equation 8:

$$tVSM(r, f) = x(terms) \times \cos(r, f) \quad (8)$$

Once the tVSM score for each function has been computed, a list of vulnerable functions is arranged in descending order of scores. The function with the highest similarity score is positioned at the top of the ranked list.

4. Experiment and Result Analysis

This section provides information on the practical implementation, the criteria used for evaluation and experimental result analysis of this study.

4.1. Implementation

The proposed method is implemented in python (version 3.11.5). The experiment was conducted on an Windows server equipped with an Intel(R) Core(TM) i5-10300H CPU processor @3.0GHz and having 64GB of RAM. The implementation involves various python libraries and NLTK (Natural Language Toolkit) libraries for text processing and feature extraction. It takes function files as input and provides ranking of suspicious vulnerable functions as output.

Table 1
List of Analyzed Open Source Projects

#	Project Name	CVE ID	Source Code
1	Chrome	CVE-2011-3916	github.com/chromium/chromium/tree/f1a59e0513d63758588298e98500cac82ddccb67
2	Radare2	CVE-2017-16359	github.com/radareorg/radare2/tree/1f5050868eedabcbf2eda510a05c93577e1c2cd5
3	Linux	CVE-2013-6763	github.com/torvalds/linux/tree/f9ec2e6f7991e748e75e324ed05ca2a7ec360ebb
4	Linux	CVE-2013-2094	github.com/torvalds/linux/tree/41ef2d5678d83af03012550329b6ae8b74618fa
5	Linux	CVE-2014-2038	github.com/torvalds/linux/tree/a9ab5e840669b19aca2974e2c771a77df2876434
6	ImageMagick	CVE-2017-15033	github.com/ImageMagick/ImageMagick/tree/c29d15c70d0eda9d7ffe26a0ccc181f4f0a07ca5
7	Tcpdump	CVE-2017-13000	github.com/the-tcpdump-group/tcpdump/tree/a7e5f58f402e6919ec444a57946bade7dfd6b184
8	Tcpdump	CVE-2018-14470	github.com/the-tcpdump-group/tcpdump/tree/aa3e54f594385ce7e1e319b0c84999e51192578b
9	FFmpeg	CVE-2016-10190	github.com/FFmpeg/FFmpeg/tree/51020adcecf4004c1586a708d96acc6cbdd050a
10	FFmpeg	CVE-2019-11339	github.com/FFmpeg/FFmpeg/tree/3f086a2f665f99060f6197cddbfacc2f4b093a1

QuASoQ 2023 Preprint

Table 2
Summary of Tested Projects

CVE ID	Total Commits	Total Files	Total Functions	Vulnerable Functions	VFDetector Ranking
CVE-2017-13000	4,466	180	638	<code>extract_header_length()</code>	1
CVE-2018-14470	4,548	185	640	<code>babel_print_v2()</code>	1
CVE-2017-15033	12,558	586	4,694	<code>ReadYUVImage()</code>	1
CVE-2017-16359	16,362	965	9,197	<code>store_versioninfo_gnu_verdef()</code>	1
CVE-2011-3916	93,104	4,929	15,042	<code>WebGLObject()</code>	2
CVE-2019-11339	93,322	2,572	16,724	<code>mpeg4_decode_studio_block()</code>	2
CVE-2016-10190	82,768	2,286	14,713	<code>http_buf_read()</code>	3
CVE-2014-2038	413,259	15,674	229,682	<code>nfs_can_extend_write()</code>	4
CVE-2013-2094	362,534	18,358	257,550	<code>perf_swevent_init()</code>	5
CVE-2013-6763	401,141	19,260	273,898	<code>uio_mmap_physical()</code>	7

4.2. Evaluation

To conduct this research we used the extensive Big-Vul dataset which contains large scale vulnerability reports of C/C++ code from open source GitHub projects. Other C/C++ datasets can also be used. Based on the highest number of vulnerabilities reported, we choosed top six well known projects from this dataset which are Chrome, Linux, Radare2, ImageMagick, Tcpdump and FFmpeg as shown in Table 1. As the selected projects are open-source in nature and are hosted on GitHub, serving as the primary platform for storing code and managing version control. It allows us to extract all essential commits for our analysis. Additional information about the repositories can be found in Table 2. As an initial study, VFDetector was evaluated using ten vulnerability reports from these six open-source projects which are chosen randomly from the dataset. Table 1 lists the analysed project name, CVE ID of report, and the source code link.

To measure the effectiveness of the proposed vulnerability detection method, we use the Top N Rank metric. This metric signifies the count of vulnerable functions ranked in the top N (where N can be 1, 5, or 7) in the obtained results. When assessing a reported vulnerability, if the top N query results include at least one function that corresponds to the location where the vulnerability needs to be addressed, we determine that the vulnerable function is detected successfully. Table 2 includes ten vulnerability reports from six open source projects with their number of commits, total files, total functions, actual vulnerable functions name and finally VFDetector ranking in Top N ranked functions in output. The results of Table 2 shows that among the ten CVE reports VFDetector ranks the actual vulnerable function at the 1st position for four (40%) reports which are CVE ID #13000, #14470, #15033, #16359. For five reports (50%) with CVE ID #3916, #2094, #2038, #10190 and #11339 it ranks the vulnerable function in Top 5 rank. It indicates that nine (90%) reports are ranked in Top 5. For one report CVE-2013-6763 of Linux Kernel version 3.12.1 it ranks the vulnerable function in Top 7 rank i.e., in 7th

position out of total 273,898 functions. Upon manual inspection, we observed that the six functions preceding the vulnerable function exhibit a higher similarity score compared to the actual vulnerable function. The reason behind this can be the inter-connectedness of these six functions with the vulnerable function through function calls. It is also noticeable that projects with less number of functions ranks the vulnerable function in 1st position and with large number of functions the ranking decreases slightly. The reason behind this is larger projects might contain more associated functions which are needed to be fixed in order to address a particular vulnerability.

In summary, the experimental results show that VFDetector can detect vulnerable functions from a huge volume of functions and can also suggest developers with the related functions having highest similarity scores which might need to be patched to address the reported vulnerability. Moreover, to the best of our knowledge we are the first to incorporate vulnerability reports in software vulnerability detection from the concept that vulnerability report’s description contain conceptual information about a reported vulnerability. Based on the promising results in this initial evaluation, the future work can be analyzing more vulnerable reports from diverse projects to make the approach comparable and generalizable.

5. Threats to Validity

In this section, we discussed the potential threats which may affect the validity of this study.

Threats to external validity: The generalizability of the acquired results poses a threat to external validity. The dataset that we used in our research was gathered from open-source. Open-source projects may contain data that differs from those created by software companies with sound management practices. Seven Apache projects are examined in this study. More projects from other systems are needed to be evaluated for the generalisation. However, to overcome this threat large-scale

diversified projects with long change history is to be chosen.

Threats to internal validity: One limitation of our approach is its reliance on sound programming practices when naming variables, methods, and classes. If a developer uses non-meaningful names, it could have an adverse impact on the effectiveness of vulnerability detection. It may not fully represent the characteristics of the whole program. Additionally, our model is evaluated with C/C++ functions and it may encounter challenges in detecting vulnerabilities in other programming languages.

Threats to construct validity: We used the WordNet database and lemmatizer of NLTK library as essential components in text pre-processing to extract word semantics and reduce words to their base forms. Since these resources are well known for their usefulness in NLP, we relied on their accuracy. Moreover, vulnerability reports offer essential information that developers rely on to address and patch vulnerable functions. A bad vulnerability report delays the fixing process. It's worth noting that our approach is dependent on the quality of these reports. If a vulnerability report lacks sufficient information or contains misleading details, it can have a detrimental impact on the performance of VFDetector.

6. Conclusion

Once a new vulnerability is reported, developers need to know which files and particular which function should be modified to fix the issues. This can be especially challenging in large software projects, where examining numerous source code files can be time-consuming and costly.

In this paper, a software vulnerability detection technique has been proposed named as VFDetector for detecting relevant vulnerable functions based on vulnerability reports. Since detecting vulnerabilities from vulnerability report is an information retrieval process, we apply static analysis on both source code and vulnerability reports to create code and report corpora. Finally, VFDetector leverages a tweaked Vector Space Model (tVSM) to rank the source code functions based on the similarity. Our evaluation conducted on six real-world open source projects show that VFDetector ranks vulnerable functions at the 1st position in most cases.

In future, VFDetector can be applied to industrial projects to access the generalization of the results in practice. Besides, dynamic analysis can be incorporated in this approach to improve detection performance. Moreover, minimizing the search space in a function and pinpointing statement-level vulnerabilities is also a potential future scope.

References

- [1] J. Han, D. Gao, R. H. Deng, On the effectiveness of software diversity: A systematic study on real-world vulnerabilities, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2009, pp. 127–146.
- [2] H. Alves, B. Fonseca, N. Antunes, Software metrics and security vulnerabilities: dataset and exploratory study, in: 2016 12th European Dependable Computing Conference (EDCC), IEEE, 2016, pp. 37–44.
- [3] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, M. McConley, Automated vulnerability detection in source code using deep representation learning, in: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE, 2018, pp. 757–762.
- [4] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, in: Proceedings of the 25th Annual Network and Distributed System Security Symposium, 2018.
- [5] D. Zou, S. Wang, S. Xu, Z. Li, H. Jin, μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection, IEEE Transactions on Dependable and Secure Computing 18 (2019) 2224–2236.
- [6] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Advances in neural information processing systems 32 (2019).
- [7] X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui, Deepwukong: Statically detecting software vulnerabilities using deep graph neural network, ACM Transactions on Software Engineering and Methodology (TOSEM) 30 (2021) 1–33.
- [8] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 590–604.
- [9] F. Yamaguchi, M. Lottmann, K. Rieck, Generalized vulnerability extrapolation using abstract syntax trees, in: Proceedings of the 28th annual computer security applications conference, 2012, pp. 359–368.
- [10] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp. 529–540.
- [11] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, Sysevr: A framework for using deep learning to detect software vulnerabilities, IEEE Transactions on Dependable and Secure Computing 19 (2021) 2244–2258.

- [12] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, H. Jin, Vulcnn: An image-inspired scalable vulnerability detection system, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2365–2376.
- [13] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Y. Zhang, Z. Chen, D. Li, Vuldelocator: A deep learning-based system for detecting and locating software vulnerabilities, IEEE Transactions on Dependable and Secure Computing (2021).
- [14] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A c/c++ code vulnerability dataset with code changes and cve summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.
- [15] H. Schütze, C. D. Manning, P. Raghavan, Introduction to information retrieval, volume 39, Cambridge University Press Cambridge, 2008.
- [16] W. B. Croft, D. Metzler, T. Strohman, Search engines: Information retrieval in practice, volume 520, Addison-Wesley Reading, 2010.
- [17] S. Rahman, K. Sakib, An appropriate method ranking approach for localizing bugs using minimized search space., in: ENASE, 2016, pp. 303–309.
- [18] S. Rahman, M. M. Rahman, K. Sakib, A statement level bug localization technique using statement dependency graph., in: ENASE, 2017, pp. 171–178.
- [19] N. E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software engineering 26 (2000) 797–814.
- [20] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (2005) 340–355.
- [21] H. Zhang, An investigation of the relationships between lines of code and defects, in: 2009 IEEE international conference on software maintenance, IEEE, 2009, pp. 274–283.

Formalization and Verification of Go-based New Simple Queue System

Danyang Wang¹, Jiaqi Yin², Sini Chen¹ and Huibiao Zhu¹

¹Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China

²Northwestern Polytechnical University, Xi'an, China

Abstract

NSQ (New Simple Queue) is a real-time distributed messaging platform implemented by Go language. It's designed to operate at scale stably and efficiently handle billions of messages per day. Its decentralized topology guarantees fault tolerance, high availability, and reliable message delivery. Operationally, NSQ is elastic to configure and deploy. With the broad application of the NSQ message system, its security and stability have attracted extensive concentration. Therefore, it is crucial to conduct a rigorous analysis and verification of NSQ's properties. In this paper, we employ process algebra CSP (Communicating Sequential Processes) to model the core functional modules of the NSQ. In addition, we utilize the model checker PAT (Process Analysis Toolkit) to verify five properties of the model, including divergence freedom, reachability, scalability, availability, and flow controllability. The verification results demonstrate that the NSQ system satisfies all the above properties, proving that the system has high flexibility and robustness while providing credible and efficient message delivery.

Keywords

NSQ, Messaging System, Communicating Sequential Processes(CSP), Modeling, Verification

1. Introduction

In the rapidly evolving era of the Internet, the explosion of users and services creates severe challenges for network applications. Conventional monolithic and vertical service architectures can no longer deal with such a volume of data. Distributed services are gradually becoming the mainstream architecture. As a foundational segment in distributed message systems, middleware [1] is important in decoupling, asynchronous communication, traffic clipping, and other issues. It can improve the performance and stability of applications. Therefore, message queue as a critical middleware acquires more attention in the Internet field.

With the evolution of technology, message queues are gradually maturing, resulting in a series of outstanding middleware, including ActiveMQ [2], RabbitMQ [3], Kafka [4], and RocketMQ [5]. These services decouple complex systems and enable asynchronous operations to reduce response times, providing a better user experience. Although the introduction of middleware can significantly improve the performance of a system, we must consider its potential problems and challenges, such as reduced availability due to unstable message queues and data inconsistencies due to concurrent communication. The system needs to introduce additional mechanisms

to ensure high availability and reachability of messages, which increases system complexity. Therefore, excellent message middleware should have high message processing efficiency, robustness, stability, and scalability.

NSQ [6] has emerged from these excellent middlewares in recent years. It is a distributed messaging platform based on Go language [9] with outstanding performance, robustness, and usability. This messaging platform is a user-friendly middleware for real-time messaging services, capable of managing hundreds of millions of messages. In addition, NSQ is fitted to the current concurrent Internet ecosystem due to Go's native strengths in concurrency. Go is a programming language with concurrency features, and its concurrency model was developed based on the process communication concept of CSP (Communicating Sequential Processes) [10, 11]. This feature makes the Go-based NSQ distributed system well-suited to the producer-consumer concurrency problem. Therefore, it is becoming popular within businesses and has also attracted the attention of researchers.

NSQ is suitable for distributed applications and systems that require asynchronous messaging, such as Social Media, Gaming, and other industries that require high concurrency. Until now, existing studies primarily focus on comparing different message queues performance, operability, and other characteristics [8] or delve into practical applications of NSQ [7]. To the best of our knowledge, there has yet to be research about the verification of its properties, which are significant for users. And the fundamental attributes of the system still need to be proven.

Employing a formal verification approach to verify NSQ's fundamental properties offers rigorous proof and

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, South Korea

✉ 51215902076@stu.ecnu.edu.cn (D. Wang); jqyin@nwpu.edu.cn (J. Yin); 52265902002@stu.ecnu.edu.cn (S. Chen); hzbzhu@sei.ecnu.edu.cn (H. Zhu)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

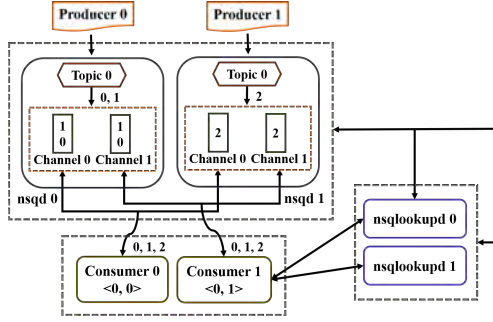


Figure 1: An Instance of the NSQ System

assurance, ensuring the system’s correctness, reliability, and stability. This approach enhances confidence and credibility in the design, implementation, and deployment of the system, which is paramount for developers and users. Consequently, this paper bridges this research gap by adopting formal methods to analyze the NSQ system. We utilize process algebra CSP to formally model the core functional modules and basic workflow of the NSQ such as message publishing, subscription, registration, and querying. Subsequently, leveraging the model checker PAT [12], we verify five properties of the model, including *Divergence Freedom*, *Reachability*, *Scalability*, *Availability*, and *Flow Controllability*. Experimental results demonstrate that the NSQ distributed message platform can guarantee all these properties, proving that the system has outstanding flexibility and robustness.

The remainder of this paper is organized as follows. Section II briefly describes the NSQ system and process algebra CSP. In Section III, we use CSP to model four fundamental components in the NSQ message system. Furthermore, in Section IV, we employ the model checking tool PAT to implement the constructed models and verify five properties we defined. Finally, we summarize this paper and discuss future work in Section V.

2. BACKGROUND

In this section we give a brief description of the NSQ’s architecture and process algebraic language CSP.

2.1. NSQ - New Simple Queue

A typical architecture of the NSQ system is displayed in Fig. 1. Before furthering into the transmission mechanism of the NSQ, we should familiar with the following terms:

- **nsqd**: The NSQ daemon responsible for receiving and delivering messages. nsqd instances manage the actual message storage and distribution.
- **nsqlookupd**: The NSQ lookup daemon that manages topology information. It receives registration information and provides service discovery.

- **Topic**: It is a distinct stream of messages. An NSQ instance can have multiple Topics, each of which can have one or more Channels.
- **Channel**: It is a logical grouping of consumers subscribed to a given Topic. Each Channel receives a copy of all the messages for that Topic.

The Topic and Channel in the NSQ system are implemented by Go’s channel data type. Go-chan builds on the idea of channel in CSP, it allows data transfers and synchronization operations between concurrent processes. A channel with cache space are also the natural way to express queue structure. Therefore, essentially NSQ’s Topic/Channel is a buffered queue for message.

After learning the basic terms, we can introduce the NSQ system further from two core workflows: Message multi-cast and Message consumption.

2.1.1. Message Multi-cast

NSQ designs nsqd to handle multiple data streams concurrently. Each Topic can have one or more Channels. Topics multicast the received messages to Channels, and each Channel receives copies of messages. In practice, Channels map to downstream services that subscribe the Topics. Topics and Channels are not preconfigured but are created upon the first publication or subscription. Within nsqd, Topics and Channels independently buffer data to prevent lagging consumers from affecting other Channels. Messages are delivered to a randomly client when all clients are ready, achieving load balancing.

2.1.2. Message Consumption

Unlike many conventional message queues, NSQ maximizes performance and throughput by pushing data to the client instead of waiting for it to pull. This concept is called the RDY (Ready) state, constituting a form of client-side flow control. This RDY state is a pivotal performance parameter, allowing clients to modulate message by adjusting the RDY value. Once clients establish connections and subscriptions, they assert control over the flow of messages from nsqd by dynamically updating the RDY value.

2.2. CSP

Process Algebra CSP [10, 11] is a formal mathematical method that is widely applied in the design and verification of concurrent systems. This language has been successfully applied in modeling and verifying various concurrent systems and protocols [13, 14]. Parts of the CSP syntax used in this paper is defined as follows:

$$P, Q ::= SKIP \mid c?u \rightarrow P \mid c!v \rightarrow P \mid P \square Q \mid P \parallel Q \mid P \parallel\parallel Q \mid P[[X]]Q \mid P \triangleleft B \triangleright Q \mid P ; Q$$

- SKIP: The process terminates properly.
- $c?u \rightarrow P$: The process receives a value from channel c and assigns it to variable u , then starts P .

- $c!v \rightarrow P$: The process sends value v to channel c and then starts executing process P .
- $P \square Q$: It depicts a general choice between process P and process Q .
- $P ||| Q$: It illustrates interleaving. Processes P and Q run simultaneously and do not share any operations or variables.
- $P \triangleleft B \triangleright Q$: It portrays the execution of process P if the boolean expression b is true; otherwise, process Q will be executed.

3. MODELING

In this section, we construct the model of NSQ distributed architecture as illustrated in Fig. 1.

3.1. Sets, Messages and Channels

For a more detailed understanding of how the components within the NSQ system communicate and interact, we have laid out explanations for the fundamental building blocks used in the model: Sets, Messages, and Communication Channels.

Table 1

The correspondence between sets and constants/variables

Set	Constant / Variable
Module	P(Producer), C(Consumer), D(nsqd), LD(nsqlookupd)
ID	pid(producer ID), cid(consumer ID), did(nsqid), lid(nsqlookupd ID), tid(Topic ID), chid(Channel ID), msgid(message ID)
Command	FIN(Finish), REQ(Queue), SUB(Subscribe), PUB(Publish), REGISTER(Register), MSG(Message), REP(Response), LOOKUPCHA(Lookup channel), LOOKUPD(Lookup nsqd)
Data	chList(registered channel list), dList(nsqd list with specific Topic)
ACK	OK, OUTTIME

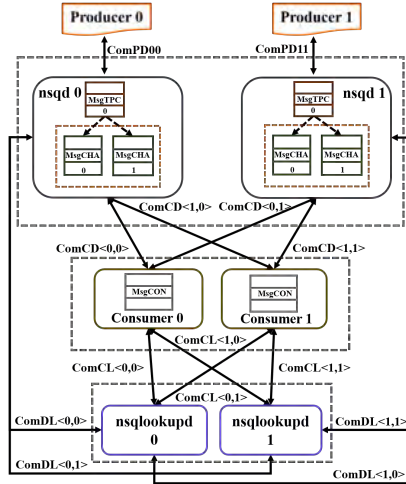


Figure 2: Channels of the NSQ System

Table 1 shows the definitions we defined for the relevant sets employed in the modeling process. The Module set contains all modules of the NSQ messaging system. The ID set consists of unique identifiers for each object within the system. Commands describe the instructions managing interactions within the NSQ, such as message publication (PUB) and subscription (SUB). The Data set indicates the topological information queried by components, and the Ack set is internal feedback.

Based on the above collections, we give the definition of the Message transferred between components:

$$MSG_{req} = \{msg_{req}.A.B.Action.Content \mid A \in Module, B \in Module, Action \in Command, Content \in ID\}$$

$$MSG_{rep} = \{msg_{rep}.A.B.Action.Content \mid A \in Module, B \in Module, Action \in Command, Content \in \{ID, ACK, DATA\}\}$$

$$MSG_{data} = \{msg_{data}.Content \mid Content \in ID\}$$

MSG_{req} denotes the set of request messages, MSG_{rep} means the set of responses, and MSG_{data} represents the set of transmitted data.

Next, we define the Channels responsible for communication between the modules and refer to these Channels with the label COM_PATH .

- **ComCL**: channels between consumer and nsqlookupd.
- **ComPD**: channels between producer and nsqd.
- **ComDL**: channels between nsqd and nsqlookupd.
- **ComCD**: channels between consumer and nsqd.

We also define the channels used internally by components with the label MSG_PATH . These channels have cache space and are responsible for caching messages. Fig. 2 shows all the channels we have defined.

- **MsgTPC**: message cache channels of Topics.
- **MsgCHA**: message cache channels of Channels.
- **MsgCON**: message cache channels of consumers.

3.2. Overall Modeling

The NSQ system embodies an intricate workflow. Due to the page limit, we only present part of the core modeling codes in this section.

The whole $System()$ as below:

$$System() =_{df} \left(\begin{array}{l} \text{|||}_{pid \in PID, did \in DID, lid \in LID, cid \in CID} \\ \left(\begin{array}{l} Producer_{pid} \text{ || } [COM_PATH] \text{ || } nsqd_{did} \\ \text{|| } [COM_PATH] \text{ || } nsqlookupd_{lid} \\ \text{|| } [COM_PATH] \text{ || } Consumer_{cid} \end{array} \right) \end{array} \right)$$

It describes the concurrent model where producers, nsqds, nsqlookupds, and consumers run in parallel and collaborate over the $[[COM_PATH]]$ channels. The pid denotes the producer ID, and PID means the set of pid . Other characters such as did , lid are similar.

3.3. Producer

The producer is responsible for generating and sending messages to corresponding Topics. It communicates with the nsqd directly and publishes messages to the nsqd module through the $ComPD_i$ channel.

$Producer_{pid}() =_{df}$

$$\left(\begin{array}{l} ComPD_i!msg_{req}.pid.did.PUB.tid.MSGID \rightarrow \\ ComPD_i?msg_{rep}.did.pid.REP.msgid.OK \rightarrow \\ updateMsgStates \{PmsgStates[msgid] == 1; \}; \\ (SKIP \triangleleft msgid == 1 \triangleright \\ nextMsg\{MSGID ++\} \rightarrow Producer_{pid}()) \end{array} \right) \\ \triangleleft pid == 0 \triangleright \\ \left(\begin{array}{l} Producer_{pid}() \\ \triangleleft PmsgStates[0] == 1 \&\& PmsgStates[1] == 1 \triangleright \\ \left(\begin{array}{l} ComPD_i!msg_{req}.pid.did.PUB.tid.2 \rightarrow \\ CoPD_i?msg_{rep}.did.pid.REP.2.OK \rightarrow \\ updateMsgStates \{PmsgStates[2] == 1; \} \\ \rightarrow SKIP \end{array} \right) \end{array} \right)$$

We define two type $Producer_{id}$. $Producer_0$ sends messages with message id 0, 1 while $Producer_1$ with id 2. The producers publish three messages to simulate the practical operation of the NSQ. Furthermore, we restrict that $Producer_1$ must wait for $Producer_0$ to finish sending before it sends the message.

3.4. nsqd

The nsqd is daemon that receives, queues, and delivers messages to clients. It handles multiple streams of data at once through the unique design of Topic and Channel. We modeled three core functions of nsqd.

The entire nsqd process execute as flowing:

$Nsqd_{did}() =_{df}$

$ExecLoop_{did}() ||| msgPump_{did}() ||| msgPush_{did}();$

The $ExecLoop_{did}()$ is the main execution loop that drives the core functions of the NSQ daemon. It is responsible for constantly listens requests from other components and processes them according to predefined logic. We model four basic command handling logics, including REQ , SUB , PUB and FIN .

Multicasting and delivery of messages is a core function of nsqd. The relationship between Topics and Channels is established through multicast, ensuring that each Channel receives a copy of all messages associated with a given Topic. This logic is implemented by the $msgPump_{did}()$ process.

$msgPush_{did}()$ is responsible for pushing messages to clients by load balancing strategy. In the NSQ messaging system, this strategy is achieved by employing a random distribution strategy, wherein messages are randomly dispatched to clients subscribed to the same Channel. $Rdy_{cid}[did]$ signifies the number of messages $consumer_{id}$ can process from a specific $nsqd_{id}$. We use the $pushStates_{did}[cid]$ array to mark whether $nsqd_{did}$ is in the state of pushing messages to $consumer_{cid}$. nsqd only sends messages to clients who can process messages. We model this process using the *General Choice* in CSP.

$ExecLoop_{did}() =_{df}$

$$\left(\begin{array}{l} ComPD_i?msg_{req}.pid.did.PUB.tid.msgid \rightarrow \\ \left(\begin{array}{l} creatTopic(did, tid) \\ \triangleleft DTStates_{did}[tid] == 0 \triangleright SKIP \end{array} \right); \\ MsgTPC_j!msg_{data}.tid.msgid \rightarrow \\ ComPD_i!msg_{rep}.did.pid.REP.msgid.OK \rightarrow \\ SKIP \end{array} \right) \\ \square \\ \left(\begin{array}{l} ComCD_k?msg_{req}.cid.did.FIN.tid.chid.msgid \rightarrow \\ updateMsg\{DMStates_{did,tid,chid}[msgid] = -1\} \rightarrow \\ SKIP \end{array} \right) \\ \square \\ \left(\begin{array}{l} ComCD_k?msg_{req}.cid.did.REQ.tid.chid.msgid \rightarrow \\ updateMsg\{DMStates_{did,tid,chid}[msgid] ++\} \rightarrow \\ MsgCHA_i!msg_{data}.tid.chid.msgid \rightarrow \\ SKIP \end{array} \right) \\ \square \\ \left(\begin{array}{l} ComCD_k?msg_{req}.cid.did.SUB.tid.chid \rightarrow \\ \left(\begin{array}{l} createTopic(did, tid); \\ updateChannel\{ \\ DChStates_{did,tid}[chid] = 1; \} \rightarrow \\ Notify(did, tid, cid) \\ \triangleleft DTStates_{did}[tid] == 0 \triangleright \\ \left(\begin{array}{l} updateChannel\{ \\ DChStates_{did,tid}[chid] = 1; \} \rightarrow \\ Notify(did, tid, cid) \\ \triangleleft DChStates_{did,tid}[chid] == 0 \triangleright SKIP \\ addClient\{TCh2C_{did,tid,chid}[cid] = 1; \} \rightarrow \\ pumpMsg\{startMsgPump_{did}[tid] = 1; \} \rightarrow \\ ComCD_k!msg_{rep}.did.cid.REP.SUB.OK \rightarrow \\ SKIP \end{array} \right) \end{array} \right); \\ \square \\ cid \left(\begin{array}{l} \left(\begin{array}{l} MsgCHA_i?msg_{data}.tid.chid.msgid \\ \{pushStates_{did}[cid] = 1; \} \rightarrow \\ ComCD_j!msg_{rep}.did.cid. \\ MSG.tid.chid.msgid \\ \{pushStates_{did}[cid] = 0; \} \rightarrow \\ SKIP \end{array} \right) \\ \triangleleft \left(\begin{array}{l} TCh2C_{did,tid,chid}[cid] == 1 \\ \& Rdy_{cid}[did] > 0 \end{array} \right) \triangleright \\ SKIP \end{array} \right) \\ msgPush_{did}(); \end{array}$$

3.5. nsqlookupd

The nsqlookupd daemon manages the system's topology information. nsqlookupd provides discovery and registration services, which decouple consumers from producers. The formal modeling of nsqlookupd is as follows.

$nsqlookupd_{lid}() =_{df}$
 $Lookup_{lid}() \ ||| \ Register_{lid}() \ ||| \ ErrorHandler_{lid}();$

The $Register_{lid}()$ process handles the registration requests sent by nsqd through the $ComCD_i$ channel, and record nsqd instance by $LDStates_{lid}[did]$. $LTStates_{lid}[tid]$ stores all the registered Topics on the nsqlookupd, and $T2D_{lid}[tid][did]$ holds the corresponding nsqd addresses for each Topic. Similarly, $LChStates_{lid,tid}[chid]$ and $TC2D_{lid}[tid][chid][did]$ serve same functions for Channels.

$Register_{lid}() =_{df}$
 $ComDL_i?msg_{req}.did.lid.REGISTER.tid.chid \rightarrow$
 $addnsqd\{LDStates_{lid}[did] = 1;\} \rightarrow$
 $\left(\left(\begin{array}{l} registerTopic\{ \\ LTStates_{lid}[tid] = 1; \\ T2D_{lid}[tid][did] = 1;\} \rightarrow SKIP \\ \triangleleft chid == -1 \triangleright \end{array} \right) \right);$
 $\left(\left(\begin{array}{l} registerTopicAndChan\{ \\ LTStates_{lid}[did] = 1; \\ LChStates_{lid,tid}[chid] = 1; \\ T2D_{lid}[tid][did] = 1; \\ TC2D_{lid}[tid][chid][did] = 1;\} \rightarrow \\ SKIP \end{array} \right) \right);$
 $Register_{lid}();$

$Lookup_{lid}()$ formalizes nsqlookupd's responses to queries from consumers and nsqd instances using General Choice. $lookupnsqd(lid, tid)$ provides all the stored nsqd address information associated with a specific Topic in nsqlookupd. Similarly, the $lookupChannel(lid, tid)$ returns Channels list under the specified Topic.

$Lookup_{lid}() =_{df}$
 $\left(\begin{array}{l} ComCL_i?msg_{req}.cid.lid.LOOKUPD.tid \rightarrow \\ lookupnsqd(lid, tid); \\ ComCL_i!msg_{rep}.lid.cid.REP.tid.dlist \rightarrow \\ SKIP \end{array} \right)$
 \square
 $\left(\begin{array}{l} ComDL_i?msg_{req}.did.lid.LOOKUPCHA.tid \rightarrow \\ lookupChannel(lid, tid); \\ ComDL_i!msg_{rep}.lid.did.REP.tid.chlist \rightarrow \\ SKIP \end{array} \right);$
 $Lookup_{lid}();$

We also modeled the response of nsqlookupd to connection errors. When nsqlookupd encounters connection timeouts with nsqd, it will receive $OUTTIME$

signal through $ComDL_i$ and then remove all information associated with the corresponding nsqd from its records. This process ensures that the information stored on nsqlookupd remains consistently available.

3.6. Consumer

When a consumer is initiated, it queries nsqlookupd for the addresses of nsqd instances associated with the target Topics. Upon receiving the addresses, it subscribes to all of these instances. Only after these can the consumer activate processes for message retrieval and processing.

Therefore, the modeling of consumer is as follows:

$Consumer_{cid,tid,chid}() =_{df}$
 $ConnToLookups_{cid,tid}();$
 $(Handler_{cid}() \ ||| \ ReadLoop_{cid}());$

$ConnToLookups_{cid,tid}() =_{df}$
 $LOOP(lid : 0..LD) :$
 $SKIP \triangleleft addrLookup[lid] == 0 \triangleright$
 $\left(\begin{array}{l} addLD\{CLStates_{cid}[lid] = 1;\} \rightarrow \\ count\{totalLD = countLD(lid, cid);\} \\ \left(\begin{array}{l} ComCL_i!msg_{req}. \\ cid.lid.LOOKUPD.tid \rightarrow \\ ComCL_i?msg_{rep}. \\ lid.cid.REP.tid.dlist \rightarrow \\ LOOP(did : 0..D) : \\ \left(ConnToNsqd_{cid,did,tid}() \right) \\ \triangleleft dlist[did] == 1 \triangleright SKIP \end{array} \right); \\ \triangleleft totalLD == 1 \triangleright SKIP \end{array} \right);$
 $ConnToNsqd_{cid,did,tid}() =_{df}$
 $SKIP \triangleleft CDStates_{cid}[did] == 1 \triangleright$

$\left(\begin{array}{l} ComCD_i!msg_{req}.cid.did.SUB.tid.c2ch[chid] \rightarrow \\ ComCD_i?msg_{rep}.did.cid.REP.SUB.OK \rightarrow \\ addnsqd\{CDStates_{cid}[did] = 1;\} \rightarrow \\ updateRDY\{Rdy_{cid}[did] = 1\} \rightarrow SKIP \end{array} \right);$
 The above formula models the process of a consumer connecting to nsqlookupds and nsqds. The consumer sends a SUB request to the nsqd through $ComCD_i$ channel. It records connection information in $CDStates_{cid}[did]$ and updates the Rdy_{cid} value of $nsqd_i$. In the formula, the value of Rdy_{cid} is set to 1, indicating the consumer's readiness to process one message from $nsqd_i$.

$readLoop_{cid}() =_{df}$
 $ComCD_i?msg_{rep}.did.cid.$
 $MSG.tid.chid.msgid\{Rdy_{cid}[did] - -;\} \rightarrow$
 $\left(\begin{array}{l} updateRDY\{Rdy_{cid}[did] + +;\} \rightarrow \\ ReadLoop_{cid,did,tid}() \\ \triangleleft Attempts_{cid}[msgid] == -1 \triangleright \\ \left(MsgCON_j!msg_{data}.did.tid.chid.msgid\{ \\ msg2d_{cid}[msgid] = did;\} \rightarrow \\ ReadLoop_{cid,did,tid}() \end{array} \right);$

QuASoQ 2023 Preprint

After completing the subscription, the consumer maintains a TCP connection with nsqd to be ready to receive messages. The diminishing of $Rdy_{cid}[did]$ value implies a decrease in the amount of messages consumers can handle. $Attempts_{cid}[msgid]$ keeps track of the message attempts number. -1 signifies successful processing and will release $Rdy_{cid}[did]$. Otherwise, the message is cached in the $MsgCON_i$ channel for further processing.

$Handler_{cid}()$ is the message-handling module of the consumer process. In our experiment, we use non-deterministic to model the message-processing behavior. We also model aborting re-queuing when the message attempts exceed the maximum value. $MaxAttempts$ defines the maximum number of message attempts allowed by the system.

```

Handlercid() =af
MsgCONi?msgdata.did.tid.chid.msgid →
  ( ComCDj!msgreq.cid.did.FIN.tid.chid.msgid{
    Attemptscid[msgid] = -1; } →
    updateRDY{Rdycid[did] ++; } → SKIP )
  < Attemptscid[msgid] > MaxAttempts >
  ( ( ComCDj!msgreq.
    cid.did.FIN.tid.chid.msgid{
      Attemptscid[msgid] = -1; } →
      updateRDY{Rdycid[did] ++; } → SKIP )
    □
    ( ComCDj!msgreq.
    cid.did.REQ.tid.chid.msgid{
      Attemptscid[msgid] ++; } →
      updateRDY{Rdycid[did] ++; } → SKIP ) )
Handlercid();

```

4. VERIFICATION

In this section, we use the model-checking tool PAT to realize the formal model constructed in section III, and verify its properties. At the same time, the results of properties verification are also shown at the end.

4.1. Implementation

This part presents details of the modeling implementation with the PAT tool, mainly concerning the definition of constants, array variables and channels.

```

#define P 2;      #define C 2;
#define D 2;      #define LD 2;
#define T 1;      #define CHA 2;
#define MsgNum 3;

```

We define constants as above to materialize the architecture of the NSQ system in Fig. 1. P , D , LD , and C represent the number of producer, nsqd, nsqlookupd, and consumer. T and CHA denote that each nsqd has one

Topic and associated two Channels. $MsgNum$ defines the number of messages.

```

#define P 2;      #define C 2;
#define D 2;      #define LD 2;
#define T 1;      #define CHA 2;
#define MsgNum 3;

```

In addition, We define some arrays to store system information, which assists us in confirming the status of processes. $Rdy[C][D]$ is used to record the number of messages the consumer can process. $pushStates[D][C]$ marks whether the nsqd is in the state of pushing messages to the consumer. $LDStates[LD][D]$ logs information about registered instances of nsqd on nsqlookupd. $Attempts[C][MsgNum]$ tracks the status of messages processed on the consumer.

Furthermore, we have implemented the relevant channels in PAT based on the definitions provided earlier. We use multidimensional arrays to store channels between different entities is to avoid resource contention.

```

channel ComPD[P][D] 0;
channel ComPD[P][D] 0;
channel MsgTPC[D][T] MsgNum;
channel MsgCHA[D][T][CH] MsgNum;

```

The channel definitions can be categorized into two types: COM_PATH are used for inter-component communication, where the channel size is set to 0 to achieve process synchronization. Cache channels MSG_PATH are used within components, where the channel size is set to $MsgNum$. These channels are utilized for process synchronization and message buffering.

Given that the NSQ message system operates with multiple producers, nsqds, nsqlookupds, and consumers, we employ a combination of interleaving and loop functions to realize the system's implementation. The comprehensive definition of the NSQ system is presented as follows. $|||i : \{0..N\}@P(i)$; statement means that multiple processes $P(i)$ run interspersed in the PAT.

```

System() =
  |||pid : {0..P-1}; did : {0..D-1}; ldid : {0..LD-1};
  cid : {0..C-1}; tid : {0..T-1}; chid : {0..CHA-1}
  @ ( Producer(pid, tid) || nsqd(did) ||
    nsqlookupd(ldid) || Consumer(cid, tid, chid) )

```

4.2. Properties Verification

In this section, we verify the properties of the constructed model with the model checker PAT. These properties present the flexibility and robustness of NSQ distributed messaging platform.

QuASoQ 2023 Preprint

4.2.1. Divergence Freedom

In NSQ system, if messages can always flow and be handled in the correct way as they should, avoiding invalid or infinite loops, then we think the system is divergence free. It is crucial for message systems because the correctness and stability of the system depends on the correct handling and delivery of messages.

PAT provides the primitive to verify the divergence freedom of the system:

```
#assert System() divergencefree;
```

4.2.2. Reachability

Data reachability is the basic property of message queue. NSQ ensures at least one delivery of a message using the *FIN* and *REQ*, but it does not guarantee data order. In our experiment, we track the attempts of messages with $Attempts[C][MsgNum]$, where the value of -1 indicates the message is finished. Therefore, the definitions of reachability and assertions are as follows:

```
#define Reachability{
  Attempts[0][0] == -1 && Attempts[1][0] == -1
  && Attempts[0][1] == -1 && Attempts[1][1] == -1
  && Attempts[0][2] == -1 && Attempts[1][2] == -1};
#assert System() |=<> Reachability;
```

As the model we constructed has two consumers subscribing to different Channels under the same Topic, each consumer will receive a copy of all messages sent by producers and finish them all eventually. Symbol $\langle\rangle$ means that the system can finally reach *Reachability* state.

4.2.3. Scalability

The NSQ system realizes a distributed decentralized architecture with nsqlookupd, which shows scalability. nsqlookupd manages the topological information of the system and allows nsqd instances to be added for horizontal scaling. In our experiments, the $LDStates[LD][D]$ is initially set to 0, denoting that no nsqd instances are available. When the value changes to 1, it indicates that nsqd instances were dynamically added, demonstrating the system's scalability.

```
#define Scalability{
  LDStates[0][0] == 1 && LDStates[0][1] == 1
  && LDStates[1][0] == 1 && LDStates[1][1] == 1};
#assert System() |=<> Scalability;
```

4.2.4. Availability

nsqlookupd serves as a distributed directory service that supports fault tolerance and redundancy. It maintains

information about the available components of the system forever, and when instances of nsqd are abnormal, it deletes all information about the corresponding instances. We defined a new system to verify the high availability of NSQ. An *OUTTIME* event of $nsqd_0$ is added to the original system, which will be triggered when all messages are finished.

$$System2() = System() ||| \left(\begin{array}{l} |||lid : \{0..LD - 1\} \\ [reachability] \\ @ \left(\begin{array}{l} ComDL_i!msgreq.0.lid. \\ ERROR.OUTTIME \rightarrow SKIP; \end{array} \right) \end{array} \right)$$

The above formula describes the new system, and we verify in the PAT whether nsqlookupd maintains the list of available nsqd. The definition and assertion are as follows:

```
#define Availability{
  LDStates[0][0] == 0 && LDStates[0][1] == 1
  && LDStates[1][0] == 0 && LDStates[1][1] == 1};
#assert System2() |=<> Availability;
```

4.2.5. Flow Controllability

NSQ can dynamically adjust messages' processing rate by changing the consumer's RDY value. To verify this property, we need to demonstrate that nsqd can push messages only if the consumer's *RDY* is greater than 0. Therefore, we introduce the $pushStates[D][C]$ array to store nsqds' status, which indicate whether $nsqd_{did}$ is pushing data to $Consumer_{cid}$. Combined with the $Rdy[C][D]$ array, we give the following definition and assertion.

```
#define Ready00 {Rdy[0][0] > 0};
#define Ready...
#define pushStop00 {pushStates[0][0] = 0};
#define pushStop...
#assert System() |=
  (pushStop00 U Ready00)
  &&(pushStop01 U Ready01)
  &&(pushStop10 U Ready10)
  &&(pushStop11 U Ready11);
```

Our model has four message subscription connections as show in Fig.2. $pushStop00$ defines the state when $nsqd_0$ stops pushing messages to the $consumer_0$, and $Ready00$ defines the state in which the $consumer_0$ is ready to receive messages from $nsqd_0$. The rest of definitions are similar. We use the *Until(U)* syntax from Linear Timing Logic (LTL) to describe the event that *the nsqd stops pushing messages until the Rdy of the corresponding consumer is larger than zero*. This formula verifies if the system can realize flow control.

	Divergence Freedom	Reachability	Scalability	Availability	Flow Controllability
Result	VALID	VALID	VALID	VALID	VALID
Visited States	324159	730077	5143	730225	42775
Total Transitions	1153000	2634052	10565	2634218	120917
Time Used	361.275s	116.7222s	0.5011s	127.0521s	5.3045s
Estimated Memory Used	81164.15KB	59856.97KB	12643.18KB	58307.32KB	18067.96KB

Figure 3: Verification Results of the NSQ System

4.3. Verification and Results

Depending on the definitions and assertions provided above, we use model checker PAT to verify five properties of the NSQ system, including *Divergence Freedom*, *Reachability*, *Scalability*, *Availability*, and *Flow Controllability*. The model checker PAT verifies properties by searching for counterexamples in the system’s state space or reaching the limits of state exploration.

We present a summary of the verification statistics in Fig. 3, including *Visited States*, *Total Transitions*, *Time Used*, and *Estimated Memory Used*.

The verification results of all five properties indicate that the NSQ message queue satisfies all the above properties, proving that the system has high flexibility and robustness while providing credible delivery of messages.

5. CONCLUSION AND FUTURE WORK

In this paper, we focus on the core functionalities of the NSQ message platform, including message publishing, registration, subscription, and querying. With CSP, we formalized critical components of the NSQ architecture, such as producers, consumers, nsqd, and nsqlookupd. Using the model checker PAT, we conducted a rigorous analysis of the constructed NSQ model, verifying five fundamental properties: *Divergence Freedom*, *Reachability*, *Scalability*, *Availability*, and *Flow Controllability*. These properties underscore NSQ’s capacity to handle real-time distributed message delivery at scale, confirming its high flexibility and robustness while ensuring dependable message transmission.

Nonetheless, besides the robustness of message queues, the security of data is extremely important for users. In the future, we will continue to enhance the formalized modeling and verification of NSQ by refining workflows. We will also delving into the system’s security aspects to advance our research outcomes continually.

References

- [1] Bernstein, P. A. (1996). Middleware: a model for distributed system services. *Communications of the ACM*, 39(2), 86-98.
- [2] Snyder, B., Bosnanac, D., & Davies, R. (2011). *ActiveMQ in action* (Vol. 47). Greenwich Conn.: Manning.
- [3] Rostanski, M., Grochla, K., & Seman, A. (2014, September). Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. In *2014 federated conference on computer science and information systems* (pp. 879-884). IEEE.
- [4] Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., ... & Stein, J. (2015). Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12), 1654-1655.
- [5] Yue, M., Ruiyang, Y., Jianwei, S., & Kaifeng, Y. (2017, October). A MQTT protocol message push server based on RocketMQ. In *2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA)* (pp. 295-298). IEEE.
- [6] NSQ: A realtime distributed messaging platform, <https://nsq.io/>
- [7] Lai, X., Wang, H., Zhao, J., Zhang, F., Zhao, C., & Wu, G. (2020, May). HBase Connection Dynamic Keeping Method Based on Reactor Pattern. In *Journal of Physics: Conference Series* (Vol. 1544, No. 1, p. 012122). IOP Publishing.
- [8] Raje, S. N. (2019). *Performance Comparison of Message Queue Methods* (Doctoral dissertation, University of Nevada, Las Vegas).
- [9] Togashi, N., & Klyuev, V. (2014, April). Concurrency in Go and Java: performance analysis. In *2014 4th IEEE international conference on information science and technology* (pp. 213-216). IEEE.
- [10] Brookes, S. D., Hoare, C. A. R., & Roscoe, A. W. (1984). A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3), 560-599.
- [11] Hoare, C. A. R. (1985). *Communicating sequential processes* (Vol. 178). Englewood Cliffs: Prentice-hall.
- [12] PAT: Process Analysis Toolkit. An Model Checker and Refinement Checker for Concurrent and Real-time System. <https://pat.comp.nus.edu.sg/>
- [13] Xiao, L., Zhu, H., Xu, Q., & Vinh, P. C. (2022). Modeling and verifying pso memory model using CSP. *Mobile Networks and Applications*, 27(5), 2068-2083.
- [14] Xu, J., Yin, J., Zhu, H., & Xiao, L. (2023). Formalization and verification of Kafka messaging mechanism using CSP. *Computer Science and Information Systems*, 20(1), 277-306.