November 18, 2022
Singapore, Singapore

Association for
Computing Machinery

MSR4P&S ’22

Proceedings of the 1st International Workshop on

Mining Software Repositories Applications for Privacy and Security

Edited by:

Melina Vidoni, Nicolás E. Díaz Ferreyra, and Zadia Codabux

Sponsored by:

ACM SIGSOFT, National University of Singapore

Co-located with:

ESEC/FSE ’22

Association for Computing Machinery, Inc.
1601 Broadway, 10th Floor
New York, NY 10019-7434
USA

Copyright © 2022 by the Association for Computing Machinery, Inc (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept. ACM, Inc.
Fax +1-212-869-0481 or E-mail permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, USA.

ACM ISBN: 978-1-4503-9457-4

Cover photo:
Title: “Helix Bridge and Marina Bay Sands”
Photographer: Erwin Soo
License: Creative Commons Attribution 2.0 Generic
https://creativecommons.org/licenses/by/2.0/deed.en
Cropped from original:
https://commons.wikimedia.org/wiki/File:Helix_Bridge_and_Marina_Bay_Sands_(8061798457).jpg

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Welcome from the Chairs

On behalf of the Program Committee, we are pleased to present the proceedings of the 1st International Workshop on Mining Software Repositories for Privacy and Security (MSR4P&S 2022). MSR4P&S is co-located with the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). This year, because of the Covid-19 pandemic, MSR4P&S (as part of ESEC/FSE) is held virtually with an adapted program that will bring together international researchers to exchange ideas, share experiences, investigate problems, and propose promising solutions concerning the application of Mining Software Repositories (MSR) to investigate the different stages of privacy and security. The workshop topics cover a wide range of MSR applications for cybersecurity research, including empirical and mixed-method approaches, as well as datasets and tools.

The last decades have put Privacy and Security (P&S) in the spotlight of information technology as data breaches, and cyberattacks have spiked globally. However, P&S are often afterthoughts in software development as their benefits are sometimes difficult to demonstrate and their costs hard to justify. However, this issue is becoming hard to sustain as new legal frameworks such as the EU General Data Protection Regulation (GDPR) demand companies to incorporate P&S features (e.g., transparency, anonymity, and informed consent) at the core of their products. Hence, there is an urgent call for tools and methods supporting the elicitation and deployment of P&S requirements in a *by-design* approach.

P&S are multifaceted and complex research areas spanning different knowledge domains (e.g., engineering, law, and psychology). Challenges in P&S cannot be solely addressed from a single viewpoint as they often involve human factors, technological artefacts, and regulatory/legal frameworks. The quest for P&S solutions requires in-deep knowledge and actionable information about its users/stakeholders, vulnerabilities/flaws, and potential attackers. MSR techniques can support this quest by providing the means to understand the P&S dimensions of information systems, thus helping shape privacy- and security-friendly software. MSR4P&S aims to explore the application of MSR at different stages of P&S engineering.

MSR4P&S 2022 received five submissions - two short papers and three full papers. Each paper submission was reviewed by three Program Committee members and followed by an internal discussion. At the end of the review process, all five papers were accepted as full papers. We want to thank the members of the Program Committee for providing constructive feedback in a timely fashion. The reviews and discussions were constructive in finalising the decisions for the submissions. The reviews were also beneficial for the authors of all the submissions to improve their work.

This year's edition features an exciting opportunity for those working at the intersection of privacy and security on collaborative software environments and open-source, leveraging or studying the increasingly-popular methodology of mining software repositories. We also featured a Keynote by Prof Ali Babar from the University of Adelaide, CREST Centre.

Lastly, we would like to thank the ESEC/FSE 2022 organisation for allowing us to organise this workshop. They have been very supportive throughout the entire process to allow us to better prepare for the workshop.

October 2022 — MSR4P&S 2022 Organising Committee

MSR4P&S 2022 Organisation

Organising Committee, General Chairs

Melina Vidoni. Australian National University, School of Computing. Australia.
Nicolás Díaz Ferreyra. Hamburg University of Technology, Institute of Software Security. Germany.
Zadia Codabux. University of Saskatchewan, Dept. of Computer Science. Canada.

Program Committee Members

Muhammad Ikram. Macquarie University. Australia
Tosin Daniel Oyetoyan. Western Norway University. Norway.
Daniela Cruzes. NTNU. Norway.
Vahideh Moghtadaiee. Shahid Beheshti University. Iran.
Sascha Fahl. CISPA. Germany.
Natalia Stakhanova. University of Saskatchewan. Canada.
Kazi Zakia Sultana. Montclair State University. United States.
Diego Costa. Concordia University. Canada.
Clemente Izurieta. Montana State University. United States.
Max Young. Mississippi State University. United States.
Mariana Peixoto. Federal University of Pernambuco. Brazil.
Jose del Alamo. Universidad Politécnica de Madrid. Spain.
Gabriel Pedroza. CEA LIST. France.
Triet Le. University of Adelaide, CREST. Australia
Maritta Heisel. University Duisburg-Essen. Germany.
Nicola Zannone. Eindhoven University of Technology. Netherlands.

Contents

Frontmatter

Welcome from the Chairs . . . iii

Keynote

Mining Software Repositories for Security: Data Quality Issues Lessons from Trenches (Keynote)
Muhammad Ali Babar — *University of Adelaide, Australia* . . . 1

Assessing Privacy

Mining Software Repositories for Patternizing Attack-and-Defense Co-Evolution
Samiha Shimmi and Mona Rahimi — *Northern Illinois University, USA* . . . 2

Assessing Software Privacy using the Privacy Flow-Graph
Feiyang Tang and Bjarte M. Østvold — *Norwegian Computing Center, Norway* . . . 7

Vulnerabilities

An Exploratory Study on the Relationship of Smells and Design Issues with Software Vulnerabilities
Sahrima Jannat Oishwee, Zadia Codabux, and Natalia Stakhanova — *University of Saskatchewan, Canada* . . . 16

Counterfeit Object-Oriented Programming Vulnerabilities: An Empirical Study in Java
Joanna C. S. Santos, Xueling Zhang, and Mehdi Mirakhorli — *University of Notre Dame, USA; Rochester Institute of Technology, USA* 21

SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques
Mohammed Latif Siddiq and Joanna C. S. Santos — *University of Notre Dame, USA* . . . 29

Author Index . . . 34

Mining Software Repositories for Security: Data Quality Issues Lessons from Trenches (Keynote)

Muhammad Ali Babar
University of Adelaide
Australia
ali.babar@adelaide.edu.au

ABSTRACT

Software repositories are an attractive source of data for understanding the burning security issues challenging developers, anecdotal solutions, and building AI/ML-based models and tools. That is why there is exponential growth in the literature based on mining software repositories for software security. While the abundance of freely available data for research is a fortune, the data quality issues can make software repositories minefields capable of blowing any time and effort budget for a project. Our group has been active in this area for the last few years to develop knowledge, understanding, and tools for improving software security by mining repositories. Through a mix of successful and failed efforts, we have experienced firsthand what is called “garbage in, garbage out” due to poor data quality. Without fully appreciating the data quality issues, starting a data-driven software security project can be frustrating and disheartening for a research team. We believe engaging the relevant stakeholders in developing and sharing knowledge and technologies to improve software security data quality is crucial. To this end, we are not only systematically identifying and synthesizing the existing empirical literature on improving data quality but also devising innovative solutions for addressing the data quality challenges while mining software repositories for software security. This talk will draw lessons and recommendations from our efforts of systematically reviewing the state-of-the-art and developing solutions for improving data quality while building knowledge, understanding, and tools for supporting software security. The talk will use a selected set of our studies to demonstrate the concrete cases of the challenges faced and the used workarounds to successfully continue our journey of learning and improving in this line of research and practice.

ACM Reference Format:
Muhammad Ali Babar. 2022. Mining Software Repositories for Security: Data Quality Issues Lessons from Trenches (Keynote). In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 1 page. https://doi.org/10.1145/3549035.3570192

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
MSR4P&S '22, November 18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9457-4/22/11.
https://doi.org/10.1145/3549035.3570192

BIOGRAPHY

M. Ali Babar is a Professor in the School of Computer Science, University of Adelaide, Australia. He leads a theme on architecture and platform for security as service in CyberSecurity Cooperative Research Centre, a large initiative funded by the Australian government, industry, and research institutes. Prior to joining the University of Adelaide, he was a Reader in Software Engineering with the School of Computing and Communication at Lancaster University, UK. After joining the University of Adelaide, Prof Babar established an interdisciplinary research centre called CREST (Centre for Research on Engineering Software Technologies), where he directs the research and education activities of more than 30 researchers and engineers in the areas of Software Systems Engineering, Security and Privacy, and Social Computing. Professor Babar's research team draws a significant amount of cash funding and in-kind resources from governmental and industrial organisations. Professor Babar has authored/co-authored more than 270 peer-reviewed research papers at premier Software journals and conferences. Professor Babar obtained a Ph.D. in Computer Science and Engineering from the school of computer science and engineering of University of New South Wales, Australia. He also holds a M.Sc. degree in Computing Sciences from University of Technology, Sydney, Australia.

Mining Software Repositories for Patternizing Attack-and-Defense Co-Evolution*

Samiha Shimmi
Northern Illinois University
DeKalb, USA
sshimmi@niu.edu

Mona Rahimi
Northern Illinois University
DeKalb, USA
mrahimi1@niu.edu

ABSTRACT

Several evidence indicates that malicious cyber actors learn, adapt, or, in other words, react to the defensive measures put into place by the cybersecurity community, as much as system defenders react to attacks. To this end, this research aims to mine the existing software repositories to document patterns of co-evolution, which appear between the cyber attacker and defender, as attack-and-defend adaptations, for the purpose of determining the probability of attackers' responsive actions.

CCS CONCEPTS

• **Software and its engineering → Software safety**.

KEYWORDS

Software security, Attack and defense co-evolution patterns, Mining software repository

ACM Reference Format:
Samiha Shimmi and Mona Rahimi. 2022. Mining Software Repositories for Patternizing Attack-and-Defense Co-Evolution. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, Singapore, Singapore, 5 pages. https://doi.org/10.1145/3549035.3561181

1 INTRODUCTION

The Federal Networking and Information Technology R&D (NITRD) working group defines *moving target* research as technologies that will enable defenders to "create, analyze, evaluate, and deploy mechanisms and strategies that continually shift and change over time to increase complexity and cost for attackers". The main idea is simple, stating that in the case of static systems, the attacker is able to learn the system and consequently evolve his attack to the system. Hence to prevent the success of the evolved attacks, the system is required to similarly adapt and evolve. To this end, we propose to study and exploit patterns of co-evolution between cybersecurity attacks and defends to enable defenders to strategically position themselves ahead of cyber threats.

As illustrated in Figure 1, thinking of cyber attack-defense as a chess contest among two opponents, a (the defensive) and b (the cyber attacker), b exploits an existing vulnerability, exposed by a, to attack. Later, player a takes an action in response to the initial attack by b. In such scenarios, both a and b evolve their strategies based on the actions taken by the other side. As such, there are correlations among the evolution of a's and b's actions. We propose to take advantage of the attack-defend-attack co-evolution phenomenon by focusing on an understanding of the attacker's response to defensive measures in the context of the attacker's mission goals and objectives. Developing an understanding of these missions and goals will generate greater predictive analysis capabilities and, more importantly, better means to influence the attacker's evolution in a manner that plays into cyber-defensive strengths.

The co-evolution phenomenon frequently occurs within softwares' artifacts. For example changes initiated at the requirements level, are followed by changes in the source code which then require corresponding updates in the test suite. Hence, while co-evolution is primarily a biological concept, here the term "co-evolution" refers to the mutualistic evolution of pairs of software artifacts.

While the state-of-the-art of software maintenance focus almost entirely on the evolution, occurring between pairs of attacks and defense, here, we propose to instead infer and leverage the patterns of corresponding evolution between attack-defend-attack triplets. Leveraging the patterns of co-evolution between the type of the attacks, defense, and subsequent attack will ultimately lead to automating the evolution of software's robustness over time.

The long term outcome of this work constitutes a paradigm shift in the development of automated evolutionary and defensive techniques in response to the malicious attacks. This research aims to lay the foundation of a complete transformation into developing *'self-robusting software'*, through leveraging the co-evolution patterns between the software artifact pairs.

2 PROBLEM DEFINITION

Considering Figure 1, the problem of predicting an evolving attack is defined as predicting a probability distribution of potential $Vulnerability_{b_2}$, given the $Vulnerability b_1$ and $Patch_{a_1}$. This can be written as below:

$$P(Vulnerability_{b_2} | \, Vulnerability_{b_1} \text{ } and \text{ } Patch_{a_1}) \quad (1)$$

There are multiple dependencies among the variables of this distribution, identified as assumptions below:

- ***Assumption*₁ :** $Vulnerability_{b_2}$ depends on $Vulnerability_{b_1}$.
- ***Assumption*₂ :** $Vulnerability_{b_2}$ depends on $Patch_{a_1}$.
- ***Assumption*₃ :** $Patch_{a_1}$ depends on $Vulnerability_{b_1}$.

We divided the problem into two smaller problems to conquer:

*This work is funded by the Office of Naval Research (ONR), Division of Cyber Security and Complex Software Systems (Grant#:N00014-22-1-2553).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR4P&S '22, November 18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9457-4/22/11…$15.00
https://doi.org/10.1145/3549035.3561181

2.1 Identification of Vulnerability

To accurately identify potential vulnerabilities, present in a code snippet, we apply a pipeline of supervised learning techniques to train a deep learning (DL) model for detecting each type of vulnerability. The output of the binary classifier is intended to be a probabilistic positive and negative predictions.

While the state-of-the-art research almost entirely focuses on training classifiers either on the semantic or on the syntactic properties of the source code, we build a data-driven model based on the both properties. The reason is that each class of properties captures particular characteristics and convey complementing information about code fragments. The semantic properties will preserve the meaning of the keywords, while the syntactic characteristics distinguishes code from natural language and preserves the properties of the code structure. The properties and our means for their identification are further discussed in details in Section 5.

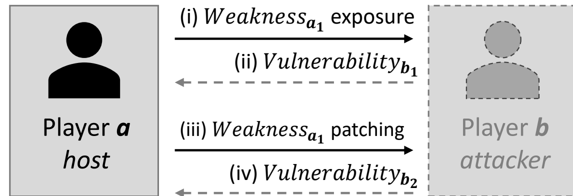

Figure 1: A high-level illustration of the problem.

2.2 Identification of (Vulnerability, Patch) Pairs

To identify the associating patches with each vulnerability, we exploit publicly accessible security databases of vulnerabilities and fixes, such as Common Vulnerabilities and Exposures (CVE) database, as well as mining shared repositories, such as Github. Analyzing a large set of git commits and their structural and semantic differences, we will identify frequent patterns in source code, leading to particular patterns of vulnerabilities, and further frequent patterns of following changes, addressing the vulnerability issues.

To determine the causal implications of the corresponding patterns of evolution in attacks and fixes, we plan to use statistical analysis. A potential solution is the negative binomial regression (NBR) modeling [2] as used in similar prior studies [22]. We choose NBR because it allows us to assess the relationship between a response (defends) and different predictors (the use of a particular pattern) while processing data over-dispersion [22]. To potentially gain deeper knowledge about the identified co-evolution patterns, we plan to also consider a few other predictors (e.g., #total commits, size of the system, system age). To deal with the imbalance of the predictors (e.g., the identified patterns may be represented by varying numbers of sample), we use weighted effects coding [20]. With this method, each regression coefficient indicates the relative effect of the use of a particular pattern on the response as compared to the weighted mean of the dependent variable across all samples. In addition, we use a Chi-Square [6] test to check the dependence between two predictors and in a case of dependence, use an $r \times c$ equivalent of the phi coefficient to compute the effect size [7].

2.3 Identification of Next Vulnerability

Given $(Vulnerability_{b_1}, Patch_{a_1})$ pairs, the objective is to identify potential vulnerabilities. We train LSTM-based AI models for the prediction of triplets of $(Vulnerability_{b_1}, Patch_{a_1}, Vulnerability_{b_2})$, assuming that the $Vulnerabilities$ b_1 and b_2 are dependent. This phase will generate the attack-defense co-evolution patterns with the help of unsupervised learning.

Recently the use of an old architecture, long short-term Memory (LSTM), for recurrent neural networks (RNN) showed several success [12]. LSTM enables the model to learn long-term dependencies and therefore, allows to leverage the information of previous predictions in the current prediction. As such, LSTM-based RNNs are used to predict the sequence of words, images, and video frames [13]. For instance, a research used LSTM RNNs for sequence-to-sequence (seq2seq) translation of English to French [26].

The attempt of this research, in general, is to enable the AI model to construct a set of potential vulnerable triplets, to detect which vulnerability will be most likely to be initiated by the attacker, in response to the patching action, previously taken by the defender. The output product of this research will be a framework, including the algorithms and tools, which receives subsequent versions of software artifacts (e.g. logged data, source code) and extracts a set of both semantic (e.g., method name changes) and structural properties (e.g., method calls changes), according to the patterns of attacks we previously documented. These properties are then fed to a predictive model (previously trained on the historical data) as the independent variables to classify the changes as "potentially a threat" or otherwise. The model will not be a binary classifier, instead measures the probability that the instance could belong to each category. The discoveries of the model will be generated in the form of textual reports of the models decision and the most significant properties which influenced the model's decision and could also be visualized in the form of an evolutionary graph to improve readability.

3 MINING SOFTWARE REPOSITORIES

The data-driven AI models critically rely on the large amount of data. In addition, the data, based on which the models are trained, is required to be in a consistent format. Furthermore, since the DL process rely on statistical methods, it is necessary for the data to be in a continuous numerical format, representing the values of data points in a shared scale. In this regard we foresee two primary challenges:

(1) Creating an acceptably large dataset (of attacks and fixes) to infer the patterns of co-evolution and to train a DL model for identifying the semantic properties of the common patterns. While thousands of such attacks and vulnerabilities are reported each year to CVE—currently contains 162,108 cases as of October 2021—but limited sources are available for the practical fixes of the listed vulnerabilities. A few publicly available academic repositories, such as [5], provide a dataset containing source code fixes for CVEs by providing detailed information at different interlinked levels of abstraction, such as the commit-, file-, method-, repository- and CVE-levels. Yet, the publicly available fixes datasets are not large enough to train a precise DL model.

(2) Mining software repositories for attack-patch pairs leads to a wide range of various-type artifacts in heterogeneous formats, such as natural language description and log, source code, and unit tests. to exacerbate this issue these different types of information are as well scattered over several incompatible cross-sources. For instance, CVE database and National Vulnerability Database (NVD) are two database that are synchronized with each other but have different formats. Additionally each Common Weakness Enumeration (CWE) record can be mapped to a set of CVE entries. Furthermore, the Common Attack Pattern Enumeration and Classification (CAPEC) is a publicly available dataset, providing common attack patterns to help the users understand how a weakness can be exploited. Finally, another resource is the SARD dataset which provides a set of synthetic, production, or academic sample artifacts.

The majority of DL models are enabled to process homogeneous artifacts as a source of information [23]. For instance, multiple studies focused on processing source code [23], while a few others limited their studies to the binary code [27] and several processed the natural language definition of the vulnerabilities [32, 33]. This is a primary limitation for the application of AI in the domain, since the available knowledge of vulnerabilities and fixes are incorporated in a various types of artifacts, such as the description of the underlying weaknesses, the code examples, test cases shared across the internet, the reports, and systems log files. To train any reliable data-driven model, a type-agnostic AI framework is necessary, enabled to derive information from several heterogeneous sources, such as statements in NL on the CWE and CVE, source code on the NVD shared repositories and test suits from SARD.

To overcome these limitations, we propose an iterative and incremental process, to actively mine the available software repositories and augment the existing datasets through creating a large record of attack and fix pairs. For this purpose, we develop a framework, which initially extract any reported project in CVE records from their open-access git repositories. The process will retrieve commit logs, code changes, and related code comments from GitHub for these applications. To tackle the heterogeneity of the artifacts, we propose to adopt *multimodal machine learning*, which integrates and models multiple communicative modalities, including linguistic, acoustic and visual messages.

This said, we initially re-train a pre-trained multimodal neural network (e.g., CodeBERT [8]), developed to process both programming language and natural language in a common space, with security-specific artifacts, The model is then applied to semantically summarize a set of code commit data. Further, the semantic similarity of committed changes to the records of CWE catalog is computed. Finally, the best-matched category is mapped to the corresponding set of commits, as a potential fix for the weakness. We will make the dataset publicly available for the use of the other researchers in the domain, once we lay the foundation of the research discussed here based on the dataset.

4 PATTERNS OF CO-EVOLUTION

Software artifacts co-evolution take various forms with respect to its underlying reason. This section describes the classification of software co-evolutions in three categories, based on our observations in our previous research projects [15–19, 24, 25]:

(i) Consistency co-evolution: Software systems are characterized by continual *internal* change, which in turn, results in inconsistency between software artifacts. To prevent the artifacts inconsistency, the ideal solution is to concurrently evolve the associated impacted artifacts. For instance, software requirements, source code and test suite can be considered as three separate artifacts, but they are tied intrinsically by co-evolution, meaning that when new requirements are added or when the source code is modified, either during development or maintenance, often new test cases are needed for newly added functionalities. Additionally, the impacted test cases need to be adapted to the change so that the source and test code remain consistent. We characterize this kind of mutualistic evolution of software artifacts, which occur due to the software's internal evolution, as *consistency co-evolution*.

(ii) Adaptation co-evolution: The *external* evolution of the software's operational environment, such as updated adjacent systems, more recent APIs version, and new potential users of the software, triggers the corresponding changes in the software for the adaptability purposes. We characterize this kind as *adaptive co-evolution*. If the software does not co-evolve in response to the external evolution, then it becomes incompatible with the evolved environment, and thus, is no longer functional. For instance, API changes, if not addressed, break the entire chain of the client programs, which extend the API, in the system.

(iii) Optimization co-evolution: Optimization co-evolution refers to adaptive changes, primarily occurred in response to the evolution of software adjacent agents, including the environment, the users, and other interacting software systems. We characterize this kind of defensive co-evolution, which are triggered in response to the *external and malicious* attacks, as *optimization co-evolution*. For instance, in 2004, Microsoft released Service Pack 2 of its XP operating system that turned on its bundled firewall by default and included a new Data Execution Prevention (DEP) security feature. DEP provided protection against buffer overflow attacks, and some believe that the presence of this feature led hackers to move more toward file-format exploits against common desktop products, such as Adobe PDF and Microsoft Office documents. Similarly, after the Department of Defense (DoD) implemented Common Access Card (CAC) PKI authentication, considered a cybersecurity 'game-changer', many observed that malicious actors simultaneously increased their use of socially-engineered infection vectors [29].

5 PROPERTIES OF PATTERNS

To carefully predict the patterns of co-evolution code weaknesses and their patches, specifying the proper properties to be taken into consideration plays an important role in the accuracy of identifying the patterns and recommending the next defensive patches. We acquire two types of information so as to further provide significantly more accurate detection of co-evolution patterns. We formulate each pattern according to three types of properties: (1) artifacts' structural evolution, (2) artifacts' semantic evolution, extracted with information retrieval techniques.

(i) Semantic Properties: The impact of the software artifacts' evolution may propagate to the parts of the system outside of the area syntactically affected by the change. This can become problematic, as the impact of a change can extend beyond the syntax and structure change in the artifacts. There are several research on semantic-based search in software artifacts [30]. For instance one previous research showed using semantic relations reduced the size of the change impact sets by 20-90% [10]. There are several existing research, using semantic similarity as a measure to identify relations between different types of software artifacts [21]. Semantic-based approaches are also widely used for sensitive tasks in the area of security, such as spam filtering [14], control on sensitive encapsulated data [4], cyber-attack detection [1], and the evaluation of the human user role in predicting the attacks [11].

The technique which converts code snippets into continuous vectors is known as code embedding, enabling to further apply mathematical models on the source code. Code embedding is widely used for semantic analysis of the source code, therefore preserves the semantics of the code words. For instance, one research proposed CODEnn which jointly embeds code and descriptions, in such a way that codes and their descriptions have similar semantic vectors [9]. The joint embedding helped them to retrieve relevant code for given descriptions.

(ii) Syntactical Properties: Measuring syntactical similarity and dissimilarity as a metric is not new in the software engineering domain and has several applications, such as in refactoring, system remodularizations and mining features from object-oriented code. Previous research indicated that structural properties are important metrics in source code processing [31]. However, the similarity measures may significantly vary between different artifacts. For instance, one previous research showed using Abstract Syntax Tree (AST) diff instead of Unix diff reduced false positives by 29-53% [10]. Several approaches are developed for detecting structural similarity in the source code of a system. For instance, Tree-based approaches such as AST are used either independently or along with other techniques [28]. A neural model, namely code2vec, embeds code snippets into continuous vectors based on their AST-structure while giving more "attention" to "more important" paths [3]. In layman's term this means that the vectors of code snippets with similar structure and similar attention-path are placed closer to each other. The model is originally developed for predicting a likely name for a given code snippet [3].

Previous research showed a combination of semantic and syntactic properties captures a larger body of details in code changes, in comparison to using each individual property, providing notably higher accuracy in detecting the patterns of change [18]. For this reason, this research aims to document patterns of co-evolution with respect to both, semantic and syntactic, characteristics of evolution on the attack and defense sides.

6 RELATED WORKS

Although the co-evolution of several software artifacts is not a new research domain [18, 24, 25], the concept of attack-defend co-evolution is a novel idea to our knowledge. Willard [29] examines the notion of attack-defend co-evolution and helped us to understand their correlation. As he mentioned, the notion of co-evolution is basically applied to living organisms and nature-based adaptations but can be applied to the cyber-security domain as well. He described the concept of "moving-target defense" in his article. Whenever the attacker attacks, the defender takes some actions and in response to those actions, the attacker can take further action, and here comes the concept of attack-defend co-evolution. Willard explained the concepts with formal definitions. However, this work did not do any implementation of the idea.

7 CONCLUSION

In this paper, we drew attention to the patterns of co-evolution between attack and defense actions in cybersecurity domain. We discussed our ongoing research, aligning with the objectives of moving target defense. We aim to leverage the patterns of co-evolution between attacks and defends in historical data to predict the more likely attacks in response to a recent defense in a system. To train a data-driven AI model, we will adopt multimodal neural networks to mine the publicly available software repositories.

REFERENCES

[1] Ahmed Aleroud, George Karabatis, Prayank Sharma, and Peng He. 2014. Context and semantics for detection of cyber attacks. *International Journal of Information and Computer Security* 7 6, 1 (2014), 63–92.

[2] Paul D Allison and Richard P Waterman. 2002. Fixed–effects negative binomial regression models. *Sociological methodology* 32, 1 (2002), 247–265.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *ACM on Programming Languages* 3, POPL (2019), 1–29.

[4] Flora Amato, Valentina Casola, Nicola Mazzocca, and Sara Romano. 2011. A semantic-based document processing framework: a security perspective. In *International Conference on Complex, Intelligent, and Software Intensive Systems*. IEEE, 197–202.

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[6] PoC Consul and Felix Famoye. 1992. Generalized Poisson regression model. *Communications in Statistics-Theory and Methods* 21, 1, 89–109.

[7] H Cramer. 1946. Mathematical methods of statistics. *Princeton University Press*.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

[9] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *International Conference on Software Engineering*. IEEE, 933–944.

[10] Quinn Hanam, Ali Mesbah, and Reid Holmes. 2019. Aiding code change understanding with semantic change impact analysis. In *International Conference on Software Maintenance and Evolution*. IEEE, 202–212.

[11] Ryan Heartfield and George Loukas. 2018. Detecting semantic social engineering attacks with the weakest link: Implementation and empirical evaluation of a human-as-a-security-sensor framework. *Computers & Security* 76, 101–127.

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8, 1735–1780.

[13] Zachary C Lipton, John Berkowitz, and Charles Elkan. 2015. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

[14] Jose R Mendez, Tomas R Cotos-Yanez, and David Ruano-Ordas. 2019. A new semantic-based feature selection method for spam filtering. *Applied Soft Computing* 76, 89–104.

[15] Mona Rahimi. 2016. Trace link evolution across multiple software versions in safety-critical systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 871–874. https://doi.org/10.1145/2889160.2889261

[16] Mona Rahimi and Jane Cleland-Huang. 2015. Patterns of co-evolution between requirements and source code. In *Fifth IEEE International Workshop on Requirements Patterns, RePa 2015, Ottawa, ON, Canada, August 25, 2015*. IEEE Computer Society, 25–31. https://doi.org/10.1109/RePa.2015.7407734

[17] Mona Rahimi and Jane Cleland-Huang. 2016. Artifact: Cassandra Source Code, Feature Descriptions across 27 Versions, with Starting and Ending Version Trace Matrices. In *2016 IEEE International Conference on Software Maintenance and*

Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016. IEEE Computer Society, 612. https://doi.org/10.1109/ICSME.2016.42

[18] Mona Rahimi and Jane Cleland-Huang. 2018. Evolving software trace links between requirements and source code. *Empir. Softw. Eng.* 23, 4 (2018), 2198–2231. https://doi.org/10.1007/s10664-017-9561-x

[19] Mona Rahimi, William Goss, and Jane Cleland-Huang. 2016. Evolving Requirements-to-Code Trace Links across Versions of a Software System. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 99–109. https://doi.org/10.1109/ICSME.2016.57

[20] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *International Conference on Software Engineering*. IEEE, 432–441.

[21] Karinne Ramirez-Amaro, Yezhou Yang, and Gordon Cheng. 2019. A survey on semantic-based methods for the understanding of human movements. *Robotics and Autonomous Systems* 119 (2019), 31–50.

[22] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165.

[23] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *international conference on machine learning and applications*. IEEE, 757–762.

[24] Samiha Shimmi and Mona Rahimi. 2022. Leveraging Code-Test Co-evolution Patterns for Automated Test Case Recommendation. In *IEEE/ACM International Conference on Automation of Software Test*. 65–76.

[25] Samiha Shimmi and Mona Rahimi. 2022. Patterns of Code-to-Test Co-evolution for Automated Test Suite Maintenance. In *2022 IEEE Conference on Software Testing, Verification and Validation*. 116–127.

[26] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[27] Junfeng Tian, Wenjing Xing, and Zhen Li. 2020. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *Information and Software Technology* 123 (2020), 106289.

[28] Wu Wen, Xiaobo Xue, Ya Li, Peng Gu, and Jianfeng Xu. 2019. Code Similarity Detection using AST and Textual Information. *International Journal of Performability Engineering* 15, 10 (2019), 2683.

[29] GN Willard. 2015. Understanding the co-evolution of cyber defenses and attacks to achieve enhanced cybersecurity. *Journal of Information Warfare* 14, 2, 16–30.

[30] Haining Yao and Letha Etzkorn. 2004. Towards a semantic-based approach for software reusable component classification and retrieval. In *Proceedings of the 42nd annual Southeast regional conference*. 110–115.

[31] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. 2004. Predicting source code changes by mining change history. *Transactions on Software Engineering* 30, 9, 574–586.

[32] Wei Zheng, Manqing Zhang, Hui Tang, Yuanfang Cai, Xiang Chen, Xiaoxue Wu, and Abubakar Omari Abdallah Semasaba. 2021. Automatically Identifying Bug Reports with Tactical Vulnerabilities by Deep Feature Learning. In *International Symposium on Software Reliability Engineering*. IEEE, 333–344.

[33] Yaqin Zhou and Asankhaya Sharma. 2017. Automated Identification of Security Issues from Commit Messages and Bug Reports. In *Joint Meeting on Foundations of Software Engineering*. Association for Computing Machinery, 914–919.

Assessing Software Privacy using the Privacy Flow-Graph

Feiyang Tang
Norwegian Computing Center
Oslo, Norway
feiyang@nr.no

Bjarte M. Østvold
Norwegian Computing Center
Oslo, Norway
bjarte@nr.no

ABSTRACT

We increasingly rely on digital services and the conveniences they provide. Processing of personal data is integral to such services and thus privacy and data protection are a growing concern, and governments have responded with regulations such as the EU's GDPR. Following this, organisations that make software have legal obligations to document the privacy and data protection of their software. This work must involve both software developers that understand the code and the organisation's data protection officer or legal department that understands privacy and the requirements of a Data Protection and Impact Assessment (DPIA).

To help developers and non-technical people such as lawyers document the privacy and data protection behaviour of software, we have developed an automatic software analysis technique. This technique is based on static program analysis to characterise the flow of privacy-related data. The results of the analysis can be presented as a graph of privacy flows and operations—that is understandable also for non-technical people. We argue that our technique facilitates collaboration between technical and non-technical people in documenting the privacy behaviour of the software. We explain how to use the results produced by our technique to answer a series of privacy-relevant questions needed for a DPIA. To illustrate our work, we show both detailed and abstract analysis results from applying our analysis technique to the secure messaging service Signal and to the client of the cloud service NextCloud and show how their privacy flow-graphs inform the writing of a DPIA.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Social network security and privacy**.

KEYWORDS

Program analysis, data protection and privacy, GDPR, software design documentation

ACM Reference Format:

Feiyang Tang and Bjarte M. Østvold. 2022. Assessing Software Privacy using the Privacy Flow-Graph. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3549035.3561185

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR4P&S '22, November 18, 2022, Singapore, Singapore

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9457-4/22/11…$15.00
https://doi.org/10.1145/3549035.3561185

1 INTRODUCTION

Privacy has been widely discussed in recent years — with the rise in public awareness and associated legislative developments, guaranteeing privacy while processing large amounts of private user data has become an important topic. Following recent law implementations such as the GDPR, we now have a regulated and clear framework for ensuring privacy compliance, which mandates documenting software properties through, for example, a Data Privacy Impact Assessment (DPIA). Such an examination must include all parts of the software and it requires a grasp of the software as well as sufficient technical knowledge to analyse the implementation. As a result, we would anticipate a development team expert who has a brief grasp of the implementation while also having sophisticated analysis and tools at their disposal to assist ensure that critical questions in evaluation frameworks such as DPIA can be answered.

The reality, however, is considerably different. While having a privacy compliance checking process operating alongside a software development life cycle is important, analysis and tools at the code level with tailored assistance to legal experts are insufficient. In the meantime, DPIA questions require an understanding of both technical and legal aspects. This means that performing a successful DPIA cannot be done exclusively by a non-technical Data Protection Officer (DPO) who specialises in data protection policy or a technical professional from the data controller (e.g., a developer in the service provider organisation) with programming experience. Simultaneously, it is difficult for developers to keep track of every single change in terms of private data processing among hundreds of lines of code.

This raises the following question: how can we help both technical developers (from or work for data controllers) and non-technical (DPOs) individuals examine privacy compliance in software? Since tracking the flow of data originating from users is important for privacy protection, we must check sensitive user inputs to the software and use an explainable abstraction to illustrate the privacy behaviours in the software, address privacy elements, and provide assistance in producing a better privacy analysis.

We propose privacy flow-graphs as a means to help both developers and DPOs, they can adopt our technique to discover privacy-related behaviours in software. Such graphs produced by our technique enable documenting private data processing actions, assist organisations (the data controller) in showing compliance with their duties and assist the DPO in carrying out its missions. Illustrating the processes may also assist developers to construct more privacy-compliant software and achieve privacy-by-design throughout development and deployment.

Our contributions are:

- The definition of the privacy flow-graph (Section 3.2)
- How to write a DPIA informed by the privacy flow-graph (Section 4).
- A static program analysis that builds the privacy flow-graphs for Java programs (Section 5).

We demonstrate the utility of our research by examining privacy-related trends in two well-known Java applications: Signal and NextCloud (Section 6).

2 MOTIVATION

Examining data protection compliance is essential for the vast majority of software released to the market, as well as for every service update when new user data must be analysed or when the way data is handled changes. Legal regulations such as the GDPR necessitate that legal experts obtain detailed privacy-related information processes from software developers. This implementation-specific information is typically obtained through manual labour by developers, and may not include all that a legal expert needs.

However, there are developers that are unfamiliar with the existing software and might have difficulties providing in-depth information to legal experts.

This circumstance motivated us to design a lightweight, semi-automated program analysis technique that automatically analyses how and where personal data is accessed and processed, therefore providing software developers and DPOs with a great deal of ease.

3 PRELIMINARIES

In this section, we describe the preliminary aspects of our analysis: the local and global data-flow, the privacy flow-graph, the source and sink methods, and the handcrafted datasets we created to support the analysis.

Let c, d denote classes, n, m methods, and let notation $c.m$ make explicit that class c that declares m. We assume that method names are unique in a class.

3.1 Local Data-Flow in Methods

We define some notation to refer to results obtainable from the control flow graph (CFG) of a method. These results concern the kind of values that may flow between various points either inside the method body.

Definition 3.1 (Method data-flow point p). A data-flow point p associated with a method $c.m$ is one of the following:

$\texttt{start}$ – the start of the method;
$\texttt{invoke}\ d.n\ _i$ – an invocation of method $d.n$;
$\texttt{i_primitive}_i$ – an input primitive;
$\texttt{o_primitive}_i$ – an output primitive;
$\texttt{return}_i$ – a return statement.

Definition 3.2 (Local data-flow F; beginning, end). Let p, p' be data-flow points, let F be $p \mapsto p'$ and let $c.m$ be a method. We write $F \Finv CFG(c.m)$ to means that the control-flow-graph of $c.m$ specifies a *local data-flow* F, that is, that values may flow from p to p'. We refer to p as the *beginning* of F, denoted $begin(F)$ and p' as the *end* of F, denoted $end(F)$.

An invocation can be both a beginning and an end of a flow, whereas the start of the method and an input primitive can only be a beginning, and a return statement and an output primitive can only be an end.

We are concerned with all data-flows that originate from the use of an input primitive. We now define some particular types of flows.

Definition 3.3 (Source flow, F^o). Given method $c.m$ where $(\texttt{i_primitive}_i \mapsto \texttt{return}_j) \Finv CFG(c.m)$. This flow is called a *source flow*, denoted F^o.

Definition 3.4 (Sink flow, F^i). Given method $c.m$ where $(\texttt{start} \mapsto \texttt{o_primitive}_i) \Finv CFG(c.m)$. The flow is called a *sink flow*, denoted F^i.

3.2 Global Data-Flow & the Privacy Flow-Graph

We now consider global data-flow, specifically data-flows between methods of different classes, those are, all data-flows that start from the use of an input primitive.

We extend the concept of a data-flow from local flows F inside methods to global flows G across methods. A global data-flow is defined by a series of local data-flows, each corresponding to a method invocation, and that satisfies certain conditions.

Definition 3.5 (Global data-flow G). A *global data-flow G* is finite series of two or more local data flows, $F_1 \cdots F_n$. The notions of beginning and end extend to G in an obvious way. Furthermore, any F_k, F_{k+1} above must satisfy the following: Let $c_k.m_k$ be such that $F_k \Finv CFG(c_k.m_k)$ and $c_{k+1}.m_{k+1}$ such that $F_{k+1} \Finv CFG(c_{k+1}.m_{k+1})$ and $end(F_k) = \texttt{return}_i$ and $begin(F_{k+1}) = \texttt{invoke}\ c_k.m_{k\ j}$ for some i, j.

A global data-flow $G = F_1 \ldots F_n$ is a *privacy flow* if F_1 is a source flow. We are especially interested in global data-flows that involve data from input primitives ending up in output primitives.

Let P be a program with privacy flows $G_1, \ldots, G_n$. The *privacy flow-graph* is a graph where there the nodes are all methods involved in a privacy flow and the edges are pairs of methods involved in successive flows F_k, F_{k+1} part of some G_j.

3.2.1 *Java specifics.*

Here we consider some issues in adapting our data-flow definitions to Java.

First, we define rich types with the intuition that we are only interested in flows that involve values of these kinds of types.

Definition 3.6 (Rich type). A *rich type* is any of following: the primitive data types `string`, `int`, `byte`, the object types, as well as arrays of rich types.

Values of rich types are those values that may contain privacy-related information. In principle, a *boolean* could also be relevant to privacy, but we limit our scope to the rich types to simplify our task. We are concerned with the processing of privacy-related data and not with the leakage of bits of privacy information stemming from such processing.

All non-trivial programs refer to either standard libraries or third-party libraries and thus source flows and sink flows may take place inside the methods of these libraries. In order to include these flows without analyzing the libraries, we introduce the concept of source methods and sink methods where such flows happen, and we apply a separate library analysis to pre-build a collection of source and sink methods.

A *source method* is a method whose invocation results in a source flow, and we denote it as *om*. A *sink method* is a method whose invocation results in a sink flow, denoted *im*.

3.2.2 Library analysis. We have manually constructed a dataset of source and sink methods in the native Java library[1] as well as the most used third-party Java libraries across different categories[2]. The third-party libraries were selected from the Maven Repository list based on their download frequency[3]. There are 158 Java source methods and 257 third-party library methods, which are divided into five groups based on the return data type. Table 1 displays three Java source method samples and three from third-party libraries.

Similarly, we created a dataset that included 350 sink methods from the same Java and 365 sink methods from the third-party libraries we investigated for the source method. Five examples of sink methods are displayed below in Table 2.

A global privacy data-flow is made up of many nodes that represent various methods. Different methods imply different types of data processing; to help demonstrate these processes, we characterise *process* under four categories.

Definition 3.7 (Process). A process is a local data-flow F in a *privacy flow* $G = F_1 \dots F_n$ that is not a source flow F^o or a sink flow F^i.

To specify some special kinds of processes, we use the following separate terms:

- Security process, if a process involves cryptography, database, security, or network packages.
- Authentication process, if authentication is involved.
- Initialisation process, if a process initialises a class.
- Non-privacy process, if it does not belong to either of the three categories above.

4 ASSESSING DATA PRIVACY

It is challenging for software developers and legal privacy experts to have a mutual understanding and benefit from each other's expertise and insights. To address this, we examine how to leverage information from data flows in software to answer particular concerns related to GDPR rules. According to Article 4 in GDPR, *"the data controller determines the purposes for which and the means by which personal data is processed"*; hence, software providers (organisations) are data controllers if the organisation develops its own software. Otherwise, the software developers provide the implementation to the data controllers who are responsible for privacy protection. In this paragraph, we first look at the core GDPR obligations of the data controller, which serves as the duty of DPOs, and then discuss how we may help DPOs answer key DPIA questions (the document created by the approach in this study is referred to as a DPIA.).

4.1 Obligation of the Data Controller

Article 24 in the GDPR [9] states several obligations of the data controller which should be monitored by the DPO:

- by default and by design, the data controller should have a record of processing activities (Article 30);
- to ensure the security of the processing (Article 32);
- to notify personal data breaches to the supervisory authorities (Article 33);
- to communicate personal breaches to the data subject (article 34)
- to conduct DPIA (Article 35);
- to conduct prior consultation with supervisory authorities (Article 36).

The DPOs' role is to monitor whether the data controller fulfilled all of their commitments, which includes performing a DPIA when required. The writing of a DPIA is a shared duty for data controllers and DPOs.

As one of the major data protection authorities in Europe, the Irish Data Protection Commission [8] provides a short explanation of what DPIA contains:

> "*A DPIA describes a process designed to identify risks arising out of the processing of personal data and to minimise these risks as far and as early as possible.*"

Here we picked one of the most often used sample templates for generating a DPIA from the British Information Commissioner's Office (ICO) [20].

Under *Section 2: Describe the processing* of the template, there are three questions:

- Describe the nature of the processing: how will you collect, use, store and delete data? What is the source of the data? Will you be sharing data with anyone? You might find it useful to refer to a flow-graph or another way of describing data flows. What types of processing identified as likely high risk are involved?
- Describe the scope of the processing: what is the nature of the data, and does it include special category or criminal offence data? How much data will you be collecting and using? How often? How long will you keep it? and more
- Describe the context of the processing: what is the nature of your relationship with the individuals? How much control will they have?

Also under *Step 5: Identify and assess risks*, DPIA requires *"Describe the source of risk and nature of the potential impact on individuals."*

With a list of privacy data-flows listed under different categories, developers and DPOs could identify the parts of the program that collect privacy data from users and the relevant risky sinks. As a result of identifying privacy flows, they can pinpoint exposure risks and offer solutions to minimise those risks.

4.2 Answering Key DPIA Questions

Based on the previous paragraph, we now define six key questions relevant to the DPIA. Software development teams and DPOs should consider how to answer these questions when writing the DPIA. Each question is followed by an explanation of how our proposed analysis technique can help answer the questions.

Q1 *What is the source & nature of the data?*

[1] Based on JDK 8u201
[2] Jackson, Log4j2, Apache Commons, Guava, HttpClient, JMS, Joda Time, Apache MINA, Apache Commons Codec and Derby
[3] Maven Repository: https://mvnrepository.com/

Table 1: Examples of source methods

Method signature	Category
int java.io.DataInputStream.read(byte[])	I/O
java.lang.String java.net.URL.getQuery()	Network
java.sql.ResultSet java.sql.Statement.getResultSet()	Database
int org.apache.commons.io.input.ProxyInputStream.read(byte[])	I/O
org.apache.http.ssl.SSLContextBuilder org.apache.http.ssl.SSLContextBuilder.loadKeyMaterial()	Network
java.sql.ResultSet org.apache.derby.iapi.jdbc.BrokeredStatement.executeQuery(java.lang.String)	Database

Table 2: Examples of sink methods

Method signature	Category
void java.util.logging.Logger.log(java.util.logging.LogRecord)	Log
void java.io.BufferedWriter.write(int)	I/O
void javax.servlet.http.HttpServletResponse.sendRedirect(java.lang.String)	Network
void com.sun.xml.txw2.output.XMLWriter.comment(char[],int,int)	I/O
java.net.HttpURLConnection org.jsoup.helper.HttpConnection(org.jsoup.Connection)	Network

A1 We need to know where the data is acquired originally and through which way. By having privacy source methods detected from the target program, we are able to look for all the potential locations in which personal data from users might get captured by the system. Different categories of privacy source methods might also indicate the type and nature of the data. For example, a method from `java.io.File` indicates this method reads from a file in the local file system.

Q2 *How is private data processed?*

A2 We want to identify the parts of the program that involve the processing of private data. This is a discovery study based on the flows that stem from privacy source methods. There are many patterns that might provide details on the processing of privacy data, for example, data travel through multiple sources or reach into multiple different sinks.

Q3 *Will the data be transformed? If so, how to ensure privacy data quality?*

A3 Data transformation and quality control can be subtle. There are clues such as the change of data types, certain types of data manipulation methods or certain APIs that might get involved in data transformation such as encryption or database packages.

Q4 *Will the data be shared/transferred and if yes, how?*

A4 Most of the data transportation happens when the privacy data flow into a sink method. By pinpointing the location and type of sink methods, we are able to identify whether there are private data being shared or transferred out of the target program.

Q5 *Does the data collected include special/highly sensitive personal data?*

A5 The property of privacy data need to be manually identified or with the help of developers. By adopting pure logic we can pick up properties that are directly linked with specific input devices of software.

Q6 *How is the data secured?*

A6 The security of private data is ensured when there are data protection mechanisms adopted, for example, the usage of cryptographic libraries or some encrypted databases. By locating the occurrence of these methods, we are able to analyse the data security protection of the target program.

5 IMPLEMENTATION

In the following paragraphs, we explain how our program analysis technique is implemented. Our implementation is built on Soot [16], a Java optimisation framework that provides four intermediate representations for analysing and transforming Java bytecode. Our technique consists of three parts:

- Transforming program bytecode to intermediate representation;
- Finding the source and sink methods;
- Building a privacy flow-graph by constructing one privacy flow for each source method at a time;
- Producing the abstraction extracted from the privacy flow-graph.

5.1 Finding Source and Sink Methods

Soot helps us transform our target program into a 3-address intermediate representation [23]. By traversing the $CFG(c.m)$ of each method $c.m$ in the program (provided in Jimple), the local dataflow analysis helps us detect the occurrences of source and sink methods in the pre-set annotation datasets (*om* and *im*) defined in Section 3.2.1. By having a complete list of source and sink methods in the application as $\mathcal{O}$ and $\mathcal{I}$, we now use them to start building the privacy flow-graph.

5.2 Building the Privacy Flow-Graph

For every class that includes a detected source method, we mark it as a class-of-interest (COI). For each COI, we first build a complete call-graph for it.

Definition 5.1 (Class-of-interest). A Class-of-interest (COI) is a class that contains an invocation to one of the source methods ($\mathcal{O}$).

$$c \in \mathcal{COI} \Leftrightarrow \exists o \in c, o \in \mathcal{O}$$

Now for each source method $o \in \mathcal{O}$, we build a global data-flow $G_o = F^o \ldots F^n$ for it from the call-graphs of each class that G_o passes through. The final output is a union of all the global data-flows originating from source methods. This graph uses $A \longrightarrow B$ to represent that method B invokes method A. Each G_o will be output as a separate dot file consisting of all the nodes (full signature of methods) and edges (invocations among the methods) which enables users to easily visualise it with simple tools.

5.3 Abstracting the Privacy Flow-Graph

Privacy flows can be lengthy and comprise a variety of non-sensitive processes, many of which are from the same class and are unrelated to privacy protection yet may confound both developers and DPOs. We want to enable DPOs to get a big picture of the important processes without getting bogged down in minutiae by creating an abstraction from the privacy-flow-graphs generated by each source method. The abstraction is powered by a simple Python script running automatically on the initial complete privacy flow-graph. We select several key parts from the complete privacy flow-graph which are listed below as symbols:

- ▲: the starting source method;
- △: a non-starting source method;
- ○: a non-special process;
  Multiple processes that belong to the same package will be grouped into one process symbol in the abstraction.
- ⊗: a security process (cryptography, database, or network);
  A security process is detected by the substring detector, we look for substrings such as 'encrypt', 'db', 'send', 'connect' in the method and its package name.
- ▼: the end sink method;
- ▽: a non-ending sink method;
- ●: the end process;
- ◊: an authentication process;
  Similar to a security process, we report an authentication process when we detect the substring 'auth' in the method or its package name.
- ⊙: initialisation process(es).
  The initialisation process has 'init' in their names which can be picked up by our substring detector.

The above key information can be interpreted to help developers pin down specific issues in code and assist DPOs to have a sketch of high-level privacy patterns in the program, to also better answer the relevant questions in DPIA.

An example abstraction output reflecting the code snippet in Figure ?? is shown below: The example has one obvious source method `read()` (line 7) which acts as the starting point of our analysis. The technique then finds the next invocation to the source method when class `Student` gets initialised (line 16). This initialisation is triggered later by another initialisation of class `Status` (line 24). Following the newly created object `Status s`, we can trace the invocations to `calculate()` (line 28), `encode()` (line 21), `findResult()` (line 31) and finally to a sink `print()` (line 31) which is invoked by the `Main()` method. Source method `read()` and sink method `print()` have their categories labelled as well as the special process `encode()`.

Along with the abstraction figure, we provide short labels with the symbols which consist of information such as 1) categories of starting source method and sink methods; 2) categories of the special processes (security, authentication, or initialisation); 3) the class name is displayed when it is an initialisation process (optional).

6 EXPERIMENT

We are looking for apps that accept raw sensitive user data and entail data transmission, often in messaging and cloud storage applications. We thus selected the following two applications: Signal[4] and NextCloud[5]. The non-profit Signal Foundation and Signal Messenger LLC created Signal, a cross-platform end-to-end instant messaging service. We intend to study how Signal processes privacy-related user data by analysing both Signal's front-end Android application and the Signal Client Service API because of its expertise in end-to-end encryption. The purpose is to figure out how data is taken from the user and sent to the server. NextCloud is a client-server software package for developing and managing file hosting services. It is free and open-source software that anybody may install and run on their own private servers. We chose an implementation of its Client API that assists developers in developing Java apps with NextCloud integration since it is highly configurable. Similar to Signal, we intend to determine how the application handles privacy-sensitive user data.

6.1 Signal

The Signal Service API contains 17,710 lines of code, which might require developers and DPOs significant time and effort to comprehend. With our samples of DPIA answers, DPOs could effortlessly use our implementation results to create a DPIA.

A total number of 11 privacy flows were detected in Signal Service API (9 out of a total 11 are displayed here), the abstraction of its privacy flow-graph is shown below as Figure 2. We categorise the 9 source methods found into four 4 different functionalities. In Signal, we have discovered a similar pattern for all types of data communication: each raw entry is instantly sent into Signal's own cryptography libraries, allowing all user entries to be completely encrypted before they reach any possible sinks or processes. *Signal: Send Message* and *Signal: Receive Message* in Figure 2 demonstrate this end-to-end encryption mechanism. As indicated by the dashed green lines, there are some source methods that accept some values from local fields which originated from source methods in other flows. `PS01`, for example, gets value from source methods `06` and `09`, which are network-related properties associated with the message object.

Now, we answer the DPIA questions we listed in Section 4.2 using the flow that originates from `01` (blue flow) in *Signal: Send Message* of Figure 2. To analyse privacy compliance, we combine the abstraction figure (which only comprises shapes and categories of critical processes) with detailed privacy flow-graphs (which contain

[4] https://signal.org/en/
[5] https://nextcloud.com/

Figure 1: Example of a privacy data-flow generated for a source code fragment and its abstraction

every node in the flow-graph as well as their complete signature), shown as in Table 3.

Q1 *What is the source & nature of the data?*

A1 Android applications take text input from a TE object which is a UI fragment providing a text field for users. The message field contains the raw message users want to send out.

Q2 *How is private data processed?*

A2 The abstraction tells us that there exist multiple processes when the text message is being sent out. There are two non-privacy processes from packages `org.signal.securesms.jobs` and `org.signalservice.api.signalservicemessagesender`. The package names indicate the types of processing behind the processes. There are also highly sensitive privacy processes such as the `MessageContentProcessor()` which is a non-starting source method that takes privacy data from a local field, in this case, it combines multiple privacy data including the text message. `org.signalservice.api.crypto` shows a typical encryption process, this also demonstrates the end-to-end encryption in Signal.

Q3 *Will the data be transformed? If so, how to ensure privacy data quality?*

A3 We notice that the data type gets immediately changed after being read into the device as raw strings. Both non-privacy and privacy processes transform data in order to achieve their functionality. However, encrypted messages stay encrypted before they get sent out, which ensures the content will not get manipulated by external parties.

Q4 *Will the data be shared/transferred and if yes, how?*

A4 The final ending sink method `sendMessage()` sends encrypted message objects out to the server from the client.

Q5 *Does the data collected include special/highly sensitive personal data?*

A5 The properties of the message object are sensitive. Not only the text message body itself, its attributes such as the details of senders but receivers and timestamps also remain sensitive during the entire process.

Q6 *How is the data secured?*

A6 Data security is guaranteed here by end-to-end encryption. All the privacy data related to the message get encrypted together as an `EncryptedMessage` object. This encrypted object cannot be decrypted by the server, which remains unreadable until it reaches the destination client.

Our discovery also supports what Signal claims in its privacy policy. By supplying the aforesaid information to both developers and DPOs, they are able to receive adequate information for creating DPIA and examining the privacy protection status in Signal without having to read the original code.

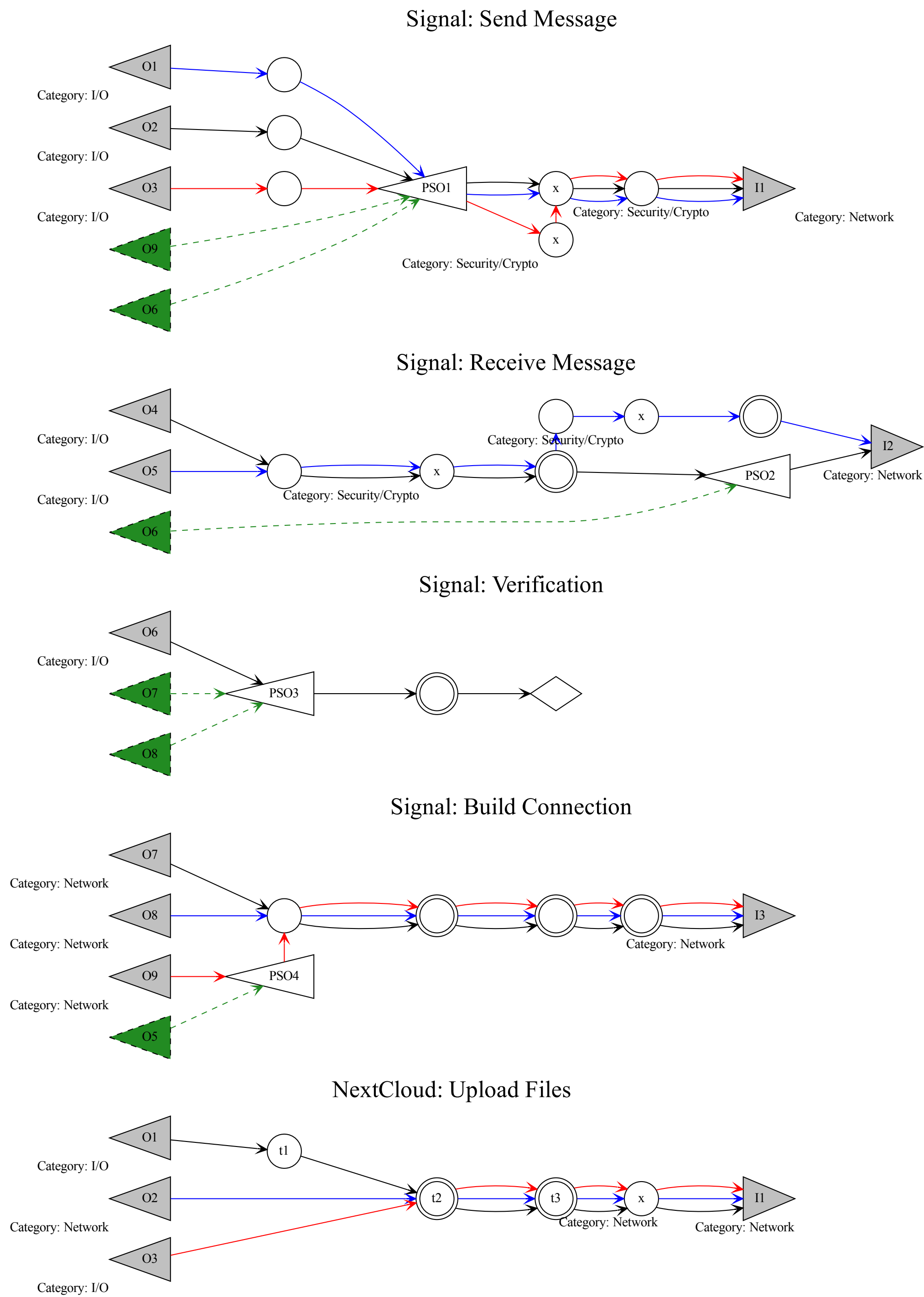

Figure 2: Sample abstract privacy flows for Signal and NextCloud

Table 3: Complete privacy data-flow with abstraction symbols for sending a text message in Signal

Abstraction	Complete privacy data-flow
▲	android.widget.EditText getText()
○	org.thoughtcrime.securesms.jobs.PushTextSendJob deliver(message)
△	org.thoughtcrime.securesms.messages.MessageContentProcessor handleMessage(content, timestamp, ...)
⊗	org.whispersystems.signalservice.api.crypto.SignalServiceCipher encrypt(destination, message, ...)
○	org.whispersystems.signalservice.api.SignalServiceMessageSender getEncryptedMessage(content, recipient, timestamp, ...)
	org.whispersystems.signalservice.api.SignalServiceMessageSender getEncryptedMessages(content, recipient, timestamp, ...)
	org.whispersystems.signalservice.api.SignalServiceMessageSender createMessageContent(message)
▼	org.whispersystems.signalservice.api.SignalServiceMessageSender sendMessage(message, recipient, ...)

6.2 NextCloud

Since NextCloud recently implemented end-to-end encryption in their products, this feature only offers on the level of 'end-to-end encrypted folders'. Hence in our analysis, we only apply the technique to the client API which is applied to the traditional version that relies on TLS communication for safely transferring files.

From a total of 8,923 lines of code, we are able to extract key information from the NextCloud Client API using a simplified privacy flow-graph along with the complete flow-graphs with full signatures, as we did with Signal. We evaluate the DPIA questions to help DPOs in getting information from a legal standpoint, using the abstraction graph derived from our technique in Figure 2.

Q1 *What is the source & nature of the data?*
A1 NextCloud Client API allows a client to upload a new file via `uploadNewFile()`. The files can be of various types but shall be categorised as the user's personal data. There is also one network source, which links with data that can be used to identify users on the Internet.
Q2 *How is private data processed?*
A2 The file is transmitted from the device to the network; this is how a file is sent from the client to the server.
Q3 *Will the data be transformed? If so, how to ensure privacy data quality?*
A3 Not only the file acquired from the user is transferred to the server, but also network data and configuration settings. These various user data are processed and loaded into multiple fields of various class objects (reflect on the two initialisation processes). During these procedures, data types must be transformed in order to be organised for transmission as a type that the server accepts.
Q4 *Will the data be shared/transferred and if yes, how?*
A4 The final node is a network sink, which indicates that the user's data has been transmitted into the network and shared with the server.
Q5 *Does the data collected include special/highly sensitive personal data?*
A5 In this example, the data comprises user files, settings, and network details. User files are highly sensitive in terms of privacy.
Q6 *How is the data secured?*
A6 The network process here depicts a TLS connection, which is a cryptographic technology meant to ensure network communications security.

With the information provided above, we provide both developers and DPOs a better understanding of how the file upload process works in the NextCloud Client API, as well as what and where are the important aspects of privacy protection for NextCloud.

Privacy flow-graphs illustrate trends in terms of privacy-related data processing, including both benign and bad practices. It can assist not just DPOs and developers in responding to DPIA questions and addressing important processing, but also in identifying potentially questionable practices and ensuring good practices on privacy-related data.

7 RELATED WORK

Using static analysis for security bug detection in software [4, 6, 10] is a source of inspiration for our work. In our work, we used handcrafted datasets of source and sink methods for Java and popular third-party libraries as the start point for our analysis. The idea of using a pre-built set as a basis of static analysis is similar to SUSI [2], IccTA [17], MudFlow [3] and AndroidLeak [11] in terms of privacy protection for Android applications. Most current work, including the above, is specific to Android sinks and sources and often uses name features as the basis of their analysis, whereas we focus on Java in general without adopting heuristics. Regarding the GDPR, we demonstrate the utility of employing privacy flow-graphs to ease the DPIA process, which saves manual labour and assists in identifying possible sensitive processes that may be missed by human eyes.

Overall, there is an increasing interest in assuring privacy protection compliance prior to or throughout the software development lifecycle [22]. Privacy-by-design (PbD) has sparked research into methodologies and models for preserving software privacy before implementation begins, as well as forecasting or managing developer privacy compliance throughout implementation [1, 12, 14]. Many of these approaches may also be employed on a regular basis during the development cycle and while updating software. In the era of GDPR in Europe, there is also prior research [5, 13, 15] that aims to provide personalised solutions for DPIA in a variety of applications. According to a survey conducted by Dias Canedo *et al.* [7], technical staff frequently lack legal knowledge regarding privacy protection. Many existing works [18, 19, 21] propose models that limit on a conceptual level, that are not tangible for both technical and non-technical people to apply to implementation, motivating us to propose an automatic technique to analyse privacy compliance in software.

8 CONCLUSION

In terms of privacy protection, there always exists a barrier between developers and DPOs. DPOs need to generate a successful DPIA to document the privacy protection behaviour of software, this requires the developer's comprehensive knowledge of code details. Our work provides a technique for detecting privacy source and sink methods in software bytecode, generating privacy flow-graphs from the discovered sources, and supporting DPOs in writing a DPIA utilising privacy flow-graphs and associated abstractions.

9 LIMITATION AND FUTURE WORK

Our present method requires predetermined source and sink lists. Given that modern applications typically contain hundreds of direct and indirect dependencies, we may miss a significant number of privacy-related sources and sinks. Therefore, we rely on the knowledge of technical specialists to create a more precise list of sources and sinks. Moreover, despite the fact that our complete privacy flow-graphs and their abstractions can express key privacy-sensitive behaviours such as data acquisition, encryption, and transportation, they are unable to provide complete information regarding which type of data manipulation was involved in terms of privacy protection; therefore, developers may be required to provide additional explanation for DPOs.

Future work includes a more detailed local flow analysis for each local data-flow in a privacy global data-flow, such as tracking how values from privacy-related data are modified in the local method and flagging sensitive manipulations such as value accumulation and separation. In the meantime, it is feasible to extract information from the manifest file on which third-party libraries are imported by the software in order to assist in the construction of a more adaptable list of sources and sinks. This procedure might be automated by including these third-party libraries (which are usually downloadable as JAR files) as a part of the input of the analysis. Additionally, since dynamically-typed languages such as JavaScript are used in many different types of modern systems, it would be advantageous to build a source code-based analyser based on tools such as Semgrep [6], which as a starting point for extending our results to web applications.

ACKNOWLEDGMENTS

We appreciate the legal insight that Jan Czarnocki and Lydia Belkadi have given. This work is part of the Privacy Matters (PriMa) project. The PriMa project has received funding from European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No. 860315.

REFERENCES

[1] Thibaud Antignac and Daniel Le Métayer. 2014. Privacy by design: From technologies to architectures. In *Annual privacy forum*. Springer, Berlin, Heidelberg, 1–17.

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks.

[3] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, Italy, 426–436. https://doi.org/10.1109/ICSE.2015.61

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[5] Shakila Bu-Pasha. 2020. The controller's role in determining 'high risk' and data protection impact assessment (DPIA) in developing digital smart city. *Information & Communications Technology Law* 29, 3 (2020), 391–402.

[6] Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE security & privacy* 2, 6 (2004), 76–79.

[7] Edna Dias Canedo, Angelica Toffano Seidel Calazans, Eloisa Toffano Seidel Masson, Pedro Henrique Teixeira Costa, and Fernanda Lima. 2020. Perceptions of ICT practitioners regarding software privacy. *Entropy* 22, 4 (2020), 429.

[8] Data Protection Commission (DPC). 2022. *Data Protection Impact Assessments*. DPC. Retrieved July 6, 2022 from https://www.dataprotection.ie/en/organisations/know-your-obligations/data-protection-impact-assessments

[9] European Commission. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). https://eur-lex.europa.eu/eli/reg/2016/679/oj

[10] David Evans and David Larochelle. 2002. Improving security using extensible lightweight static analysis. *IEEE software* 19, 1 (2002), 42–51.

[11] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 291–307.

[12] Irit Hadar, Tomer Hasson, Oshrat Ayalon, Eran Toch, Michael Birnhack, Sofia Sherman, and Arod Balissa. 2018. Privacy by designers: software developers' privacy mindset. *Empirical Software Engineering* 23, 1 (2018), 259–289.

[13] Jane Henriksen-Bulmer, Shamal Faily, and Sheridan Jeary. 2020. DPIA in Context: Applying DPIA to Assess Privacy Risks of Cyber Physical Systems. *Future Internet* 12, 5 (2020), 93.

[14] Jaap-Henk Hoepman. 2014. Privacy Design Strategies. In *ICT Systems Security and Privacy Protection*, Nora Cuppens-Boulahia, Frédéric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 446–459.

[15] Martin Horák, Václav Stupka, and Martin Husák. 2019. GDPR Compliance in Cybersecurity Software: A Case Study of DPIA in Information Sharing Platform. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (Canterbury, CA, United Kingdom) *(ARES '19)*. Association for Computing Machinery, New York, NY, USA, Article 36, 8 pages. https://doi.org/10.1145/3339252.3340516

[16] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. IEEE, Purdue University.

[17] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, Italy, 280–291. https://doi.org/10.1109/ICSE.2015.48

[18] Yod-Samuel Martin and Antonio Kung. 2018. Methods and Tools for GDPR Compliance Through Privacy and Data Protection Engineering. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, London, 108–111. https://doi.org/10.1109/EuroSPW.2018.00021

[19] Aaron K Massey, Paul N Otto, Lauren J Hayward, and Annie I Antón. 2010. Evaluating existing security and privacy requirements for legal compliance. *Requirements engineering* 15, 1 (2010), 119–137.

[20] Information Commissioner's Office. 2018. Data Protection Impact Assessments (DPIAs). https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/data-protection-impact-assessments-dpias/. (Accessed on 03/02/2022).

[21] Luca Piras, Mohammed Ghazi Al-Obeidallah, Andrea Praitano, Aggeliki Tsohou, Haralambos Mouratidis, Beatriz Gallego-Nicasio Crespo, Jean Baptiste Bernard, Marco Fiorani, Emmanouil Magkos, Andres Castillo Sanz, et al. 2019. DEFeND architecture: a privacy by design platform for GDPR compliance. In *International Conference on Trust and Privacy in Digital Business*. Springer, Springer, Bratislava, Slovakia, 78–93.

[22] Ira S Rubinstein. 2011. Regulating privacy by design. *Berkeley Tech. LJ* 26 (2011), 1409.

[23] Raja Vallée-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations.

[6] https://semgrep.dev/

An Exploratory Study on the Relationship of Smells and Design Issues with Software Vulnerabilities

Sahrima Jannat Oishwee
sao107@usask.ca
University of Saskatchewan
Saskatoon, Canada

Zadia Codabux
zcodabux@cs.usask.ca
University of Saskatchewan
Saskatoon, Canada

Natalia Stakhanova
natalia@cs.usask.ca
University of Saskatchewan
Saskatoon, Canada

ABSTRACT

Software vulnerabilities are one of the leading causes of the loss of confidential data resulting in financial damages in the industry. As a result, software companies strive to discover potential vulnerabilities before the software is deployed. While traditionally, software metrics have been widely used to uncover vulnerabilities, more recent studies have been looking at code smells to detect vulnerabilities. This preliminary study explores the relationship between smells, design issues, and software vulnerabilities. As smells and design issues are indicators of potential problems in the software, establishing a relationship with vulnerabilities can be helpful for vulnerability prediction. In this study, we analyzed 561 versions of nine open-source software by exploring the smells and design issues in the vulnerable and non-vulnerable classes. We found that some smells and design issues have a statistically significant relationship with the vulnerable classes. However, after a manual analysis of the code segments containing the vulnerabilities, we found no indication that smells or design issues induce the vulnerabilities. In fact, they were still present in those code segments even after the vulnerabilities were resolved.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**.

KEYWORDS

Code Smells, Design Issues, Software Vulnerabilities, Mining Software Repositories, Software Security

ACM Reference Format:
Sahrima Jannat Oishwee, Zadia Codabux, and Natalia Stakhanova. 2022. An Exploratory Study on the Relationship of Smells and Design Issues with Software Vulnerabilities. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3549035.3561182

1 INTRODUCTION

No software today can be considered secure. Any weakness in the software design, implementation, or configuration that could be exploited to cause harm is generally considered a software vulnerability. Alongside traditional approaches, for instance, using software metrics (e.g., code complexity [34], code churn [25], code cohesion and coupling [5]) that focus on vulnerability detection, prediction have also gained momentum in research [14, 19, 33].

Code smells are symptoms that may indicate potential issues in source code [13]. *Design issues are weaknesses that violate fundamental design principles and negatively impact software quality* [31]. Code smells and design issues are related to structural defects in software [6, 7]. Design issues include errors that can weaken a system disastrously, and simple coding mistakes or design issues can lead to exploitable vulnerabilities [3, 21]. Since both can be potentially exploited by attackers, understanding their relationship with vulnerabilities can be helpful in early vulnerability prediction. At a high level, vulnerabilities are classified as defects [24]. Recently, the impact of smells on vulnerabilities was explored [9, 30]. As these studies have considered only smells, we go a step further and **investigate the relationship between smells and design issues with vulnerabilities**.

We investigated vulnerabilities and the security reports for 561 versions of nine Open Source Projects (OSPs). We extracted nine code smells, four architectural smells, and twelve design issues. We performed an in-depth manual analysis using the vulnerabilities' descriptions and the fix-commits of the vulnerabilities to understand each vulnerability's cause and how they were related to the smells and design issues in the associated code segments.

Contributions: We analyzed different types of (code and architecture) smells and design issues to investigate their relationship with software vulnerabilities. This information provides guidance to software practitioners helping them prioritize and manage these smells and design issues for secure software development. We also provide a **replication package**[1] consisting of the dataset, Python code for the statistical analyses, and descriptions of the studied smells and design issues.

2 RELATED WORK

Elkhail et al. [9] explored the relationship between code smells and vulnerabilities and found that some smells have a weak relationship with vulnerabilities. Sultana et al. [30] investigated the relationship between code and architectural smells with software weaknesses. They concluded that while certain code smells have a statistically significant relationship with vulnerabilities, architectural smells do not. Gupta et al. [17] measured pair-wise co-efficient correlation to investigate the relationship between code smells and vulnerabilities. They reported that some pairs have a coefficient above 0.93, indicating a significant correlation. Shin et

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR4P&S '22, November 18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9457-4/22/11…$15.00
https://doi.org/10.1145/3549035.3561182

[1] https://doi.org/10.5281/zenodo.7020589

al. [28] built a vulnerability prediction model using complexity, code churn, and developer's activity metrics and found that the model has 25% of false positive results. Gong et al. [15] built a model to predict security risk for android applications using code smells and Java static metrics. They concluded that some of the code smells influence security prediction. Feng et al. [10] used architectural design issues to indicate potential sources of security issues.

Most existing studies focus on code smells for vulnerability prediction or to explore the relationship with vulnerabilities. In our study, in addition to smells, we investigated whether common design issues are related to vulnerabilities. We also performed a manual analysis on the code segment of the fix-commits to understand how the vulnerabilities were fixed and whether they were related to the smells and design issues.

3 METHODOLOGY

3.1 Research Questions

This study investigates the relationship between (code and architectural) smells and design issues with software vulnerabilities. To achieve our goal, we pose two Research Questions (RQs):

RQ1: What is the relationship between smells and software vulnerabilities?

We investigated whether there is a relationship between classes with and without smells and vulnerabilities. Based on previous studies, not all smells are related to vulnerabilities [9, 30]. Therefore, understanding which smells have a statistically significant relationship with vulnerabilities is important for vulnerability prediction. Development teams already extracting smells as a part of their quality assurance process can use our findings to pinpoint the specific smells that are more prone to vulnerabilities, and therefore, teams can focus more effort on the code containing these smells during code review or software testing. Thus, we formulate the following hypotheses:

Null Hypothesis (H_{01}): The smells are not associated with vulnerabilities.

Alternative Hypothesis (H_{A1}): The smells are more likely to be associated with vulnerabilities.

RQ2: What is the relationship between design issues and software vulnerabilities?

Next, we investigated whether classes with design issues are more likely to be vulnerable than those without. Identifying which design issues have a statistically significant relationship with vulnerabilities can help practitioners understand which ones should be prioritized and resolved during secure software implementation. Thus, we formulate the following hypotheses:

Null Hypothesis (H_{02}): The design issues are not associated with vulnerabilities.

Alternative Hypothesis (H_{A2}): The design issues are more likely to be associated with vulnerabilities.

3.2 Data Extraction and Processing

We studied nine OSPs from the Apache foundation, namely Batik, Camel, Kafka, Oozie, PDFBox, POI, Solr, Tomcat, and Wicket. These projects are of different sizes and diverse in functionalities. The project statistics are displayed in Table 1.

Table 1: Project Statistics

Project	#Versions	#Classes (Latest Version)	#Vulnerabilities	#Vulnerable Classes
Batik	18	2,231	6	4
Camel	27	20,353	6	11
Kafka	30	5,649	6	10
Oozie	12	1,400	2	18
PDFBox	37	1,444	8	18
POI	25	3,520	3	20
Solr	96	12,751	27	58
Tomcat	245	3,861	59	94
Wicket	71	3,760	9	2
Total	561	54,969	126	235

The study design including the data extraction, processing, and analysis (steps ①-⑧) is depicted in Figure 1. In step ①, we extracted the reported vulnerabilities and vulnerability reports (containing information such as Common Vulnerabilities and Exposures (CVE)-IDs, vulnerable and fixed versions, and GitHub commit (called 'fix-commit' in the paper)) from Tomcat's security pages[2] and Snyk pages[3] for the other projects, from November 2021 to May 2022. Next, we filtered out duplicate vulnerabilities using the unique CVE-IDs, resulting in 126 unique vulnerabilities in step ②. Fix-commit contains the vulnerable classes and the changes (deletion, addition, and modification) in the code segment of those classes to address the vulnerability. In step ③, we extracted 235 vulnerable classes using the fix-commit information from ①. Then, in step ④, from the vulnerability reports in ①, we extracted the vulnerable and fixed versions of each vulnerability. For example, from the Apache Tomcat security page, we found that CVE-2021-30639 affected Tomcat versions 10.0.3 to 10.0.4 and was fixed in version 10.0.5. Therefore, we extracted these three versions for CVE-2021-30639. We followed this process for the 126 vulnerabilities and extracted their vulnerable and fixed versions. Finally, we filtered out the duplicate versions for the nine systems and kept the unique ones, resulting in a total of 561 vulnerable and fixed versions. Note that the vulnerable version for one vulnerability can be the fixed version for another and vice versa. We mined GitHub to extract the source code of the 561 versions in step ⑤. Next, in step ⑥, using the CVE-ID of each vulnerability, we extracted the NVD[4] descriptions, which contained information about the vulnerability's cause, effect, and severity. In step ⑦, we analyzed the source code and extracted nine code smells (*God Class, Lazy Class, Complex Class, Large Class, Refused Bequest, Data Class, Brain class, Feature Envy, and Long Method*) and four architectural smells (*Hub Like Dependency, Class Cyclic Dependency, Unhealthy Inheritance Hierarchy, and Package Cyclic Dependency*) using the code smell tool, GetSmells [30]. To avoid the threats to validity associated with using one tool (GetSmells), we also used PMD[5] to extract the most common smells (*God Class, Data Class, Large Class, and Long Method*). We also extracted 12 common design issues (*Cognitive Complexity, Cyclometric Complexity, Too Many*

[2] https://tomcat.apache.org/security-10.html
[3] https://snyk.io/
[4] https://nvd.nist.gov/
[5] https://pmd.github.io/pmd-6.46.0/

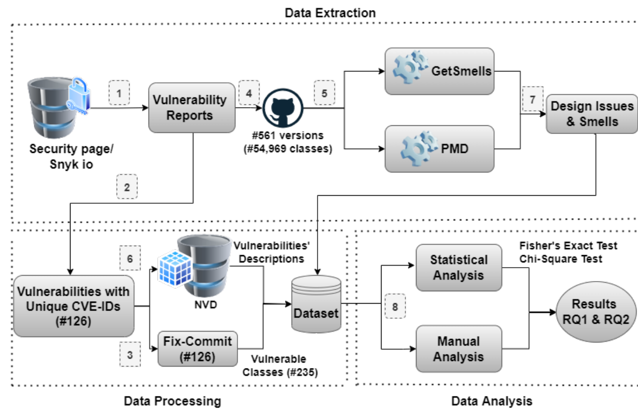

Figure 1: Study Design (Data Extraction, Processing, and Analysis)

Methods, Collapsible If Statements, Simplify Conditional, Abstract Class Without Any Method, Switch Density, Avoid Deeply Nested If Statement, Avoid Throwing Raw Exception, Too Many Fields, Excessive Parameter List, and Use Object For API) using PMD. In step (8), we analyzed the extracted data manually and using statistical techniques.

3.3 Data Analysis

To address RQ1, we first performed Fisher's exact test to investigate whether classes with more smells are more prone to software vulnerabilities than others. Fisher's exact test is the same as the Chi-Square (χ^2) test but used when values are less than five in a 2x2 table. Next, for those values greater than or equal to five, we performed the χ^2 test to identify which smells are more prone to software vulnerabilities. To perform these statistical tests, we have two categories, vulnerable (235) and non-vulnerable (54,734) classes. For Fisher's exact test, we calculated the frequency of the smells for each project in these two categories. For the χ^2 test, we calculated the frequency of each smell separately for each version of project. Then, we calculated the p-value. For hypothesis testing, we considered the threshold p-value to be 0.05, which means that a p-value ≤ 0.05 indicates strong evidence against the null hypothesis. Then, we calculated the Effect Size (ES) of Fisher's exact test using Odds Ratio (OR) [11] and Cramer's V[6] for the χ^2 test. The thresholds for OR are OR ≤ 1 indicating a low magnitude and OR > 1 indicating a high magnitude. The thresholds for Cramer's V are Cramer's V ≤ 0.2 indicating low magnitude, $0.2 <$ Cramer's V ≤ 0.6 indicating moderate magnitude, and Cramer's V > 0.6 indicating high magnitude. We followed the same process as RQ1 for the design issues to address RQ2.

Next, we manually investigated each vulnerability. First, using the NVD description, we explored the cause and impact of the vulnerability. Then, we checked the smells and design issues in the vulnerable and fixed versions of each vulnerable class. This helped us to determine whether specific smells or design issues were present in the vulnerable classes but not in the fixed ones.

[6] https://ibm.co/3RNWQhO

Table 2: Fisher's Test Results

Project	Smells (p-values)	Smells' Odds Ratio (OR)	Design Issues (p-values)	Design Issues' Odds Ratio (OR)
Batik	**<0.001**	0.40	**0.00**	0.02
Camel	**<0.001**	0.51	**<0.001**	0.06
Kafka	**<0.001**	0.65	**<0.001**	0.06
Oozie	**<0.001**	0.55	**0.00**	0.03
PDFBox	**0.009**	1.01	**<0.001**	0.09
POI	0.865	0.57	**0.00**	0.03
Solr	**<0.001**	0.40	0.921	0.02
Tomcat	**<0.001**	0.53	**0.00**	0.99
Wicket	**<0.001**	0.47	**0.00**	0.02

Lastly, we looked into the code segment of the fix-commit for each vulnerability to understand how the vulnerabilities were fixed and related to the associated code segments' smells and design issues. From the 126 fix-commits (equivalent to the number of extracted vulnerabilities), we inspected the 235 vulnerable classes to identify whether the modification of the code resolved the vulnerability and whether any specific smells or design issues from the vulnerable classes were resolved in the fixed version. For example, Kafka had a timing attack vulnerability (CVE-2021-38153)[7]. Based on the NVD description, Kafka has some components that use 'Arrays.equals' to validate a password or key. This 'Arrays.equals' API is vulnerable to timing attacks that can result in successful brute force attacks. From the fix-commit[8], 'Array.equals' was replaced by a new API 'Utils.isEqualConstantTime', and four classes were modified. These four classes contained *Complex class, Large class, Data Class, Feature Envy, and Long Method* in the vulnerable versions, but none of them were resolved in the fixed version.

4 RESULTS

RQ1: Smells and Vulnerabilities From Table 2, the emboldened p-values are < 0.05 (therefore, rejecting H_{01}) except for POI ($p = 0.865$). This indicates that for the other eight systems, the smellier classes are more likely to be vulnerable. However, though the p-value is significant, the OR for the smells is very low, indicating a weak result. Since Fisher's exact test is based on the combined smells of the different versions of the projects, we calculated the χ^2 for each smell separately for the different versions. The code smells *God Class, Lazy Class, Complex Class, Large Class, Refused Bequest, Data Class, Feature Envy, and Long Method* and architectural smells *Hub Like Dependency, Class Cyclic Dependency, and Unhealthy Inheritance Hierarchy* have p-values ≤ 0.05 (therefore, rejecting H_{01}) and high Cramer's V (Cramer's V > 0.6). This indicates that the smells mentioned above have a statistically significant relationship with vulnerabilities. The outputs from PMD confirmed the results for *God Class, Data Class, Large Class, and Long Method.* From the manual analysis, we found that despite the vulnerability-inducing code being modified, the smells were still present after the vulnerabilities were fixed. Therefore, *there is*

[7] https://nvd.nist.gov/vuln/detail/CVE-2021-38153
[8] https://bit.ly/3v8wIop

no direct indication that smells are inducing vulnerabilities or that smell-ridden code is more prone to vulnerabilities.

RQ2: Design Issues and Vulnerabilities From Table 2, the emboldened p-values indicate projects with a p-value < 0.05 (therefore, rejecting H_{02}). Only Solr has a p-value of 0.92. However, the OR is of low magnitude, indicating a weak result. Since Fisher's exact test is based on the combined smells of the different versions of the projects, we calculated the χ^2 for each design issue separately for the different versions. The χ^2 test showed that *Cognitive Complexity, Cyclomatic Complexity, Too Many Methods, Collapsible If Statements, Simplify Conditional, Excessive Parameter List, Abstract Class Without Any Method, Switch Density, and Avoid Deeply Nested If Statements* have p-values < 0.05 (therefore, rejecting H_{02}) and high Cramer's V (Cramer's V > 0.6), indicating a strong relationship between the above-mentioned design issues and vulnerabilities. From the manual analysis, we found that despite the vulnerability-inducing code being modified, the design issues were still present after the vulnerabilities were fixed. Therefore, *there is no direct indication that design issues are inducing vulnerabilities or design issues-ridden code is more prone to vulnerabilities.*

Due to space limitations, the results for the χ^2 statistical test for RQ1 and RQ2 are available in the replication package.

5 DISCUSSION

Our results indicate that some code smells (*God Class, Lazy Class, Complex Class, Large Class, Refused Bequest, Data Class, Feature Envy, and Long Method*) and architectural smells (*Hub Like Dependency, Class Cyclic Dependency, and Unhealthy Inheritance Hierarchy*) have a statistically significant relationship with vulnerabilities.

Smells increase the maintainability and complexity and reduce the understandability of the code [2], leading to more defects in the code segments, which can be exploited [27]. *God class* has a high functional complexity [23], and *Complex Class* has high cyclomatic complexity [1]. *God Classes* are also frequently changed in the software development process, leading to more defects in those classes [27]. Previous studies mentioned that complexity metrics are early indicators of vulnerable code and complex code is related to vulnerabilities [5, 16, 28, 29]. Therefore, due to their complexity and defect-inducing nature, *God Class* and *Complex Class* can be potentially related to vulnerabilities. *Feature Envy*, a method level code smell is correlated with low-quality code and occurs when one object exposes its data fields to another object instead of doing the computation itself [16]. Data exposure can potentially lead to information leakage vulnerability. *Large Class* and *Long Method* are difficult to reuse and understand, making their maintenance harder [2]. During the lifetime of the software, the high maintenance cost, lack of understandability, and re-usability hinder developers from writing defect-free code [22], and attackers can exploit these defects created due to such maintenance issues.

Although previous research did not find a significant relationship between architectural smells and vulnerabilities [30], we found that three architectural smells (*Hub Like Dependency, Class Cyclic Dependency, and Unhealthy Inheritance Hierarchy*) have a statistically significant relationship with vulnerabilities. The architectural subsystems in *Hub Like Dependency*, or *Class Cyclic Dependency* are hard to release, maintain, and reuse [12]. The structural dependencies among these architectural subsystems may incur architectural flaws, and these flaws cause modularity violations and improper inheritance, potentially leading to exploitable security issues [10]. In addition, the files involved in architectural patterns have significantly higher number of bugs, and change rates than the average files in a project [26]. However, there is a need for a more in-depth analysis to investigate the relationship between vulnerabilities and architectural smells.

Though the statistical analysis showed a significant relationship between some of the smells and vulnerabilities due to their characteristics, the manual analysis of the vulnerability fix-commit did not support this finding. This can be due to the fix-commit changes in the code segment focusing on resolving the vulnerabilities rather than refactoring the code to remove smells. It is possible that some smells were removed as part of the vulnerability patching process. However, our results showed that the smells were still present in the classes even after resolving the vulnerabilities. Therefore, to resolve the vulnerabilities, developers are more concerned about fixing or changing the associated code segments rather than addressing the code smells. Yamashita et al. [35] reached similar conclusions and reported that developers often sacrifice the code quality and prioritize delivering a product on time.

The design issues (*Cognitive Complexity, Cyclomatic Complexity, Too Many Methods, Collapsible If Statements, Simplify Conditional, Excessive Parameter List, Abstract Class Without Any Method, Switch Density, and Avoid Deeply Nested If Statements*) have a statistically significant relationship with vulnerabilities. *Cognitive Complexity* combines the code's spatial complexity with the architectural complexity of control statements [4]. Additionally, *Simplify Conditional, Switch Density, and Avoiding Deeply Nested If Statements* are closely related to the nesting complexity and complexity of control statements [5]. As complexity metrics and complex code are related to vulnerabilities [5, 28], *Cognitive Complexity, Simplify Conditional, Switch Density, and Avoiding Deeply Nested If Statements* are also related to vulnerability due to their complex nature. *Cyclomatic Complexity* is the number of independent paths through a program unit [8]. Classes with higher *Cyclomatic Complexity* are usually more challenging to maintain or test, and therefore cause issues that can be further exploited. Camilo et al. [3] found that design issues can lead to exploitable defects, supporting our findings.

Though the statistical analysis showed a significant relationship between some design issues and vulnerabilities, the manual analysis of the vulnerability fix-commit did not align with this finding. Our analysis showed that design issues were still present in the classes even after resolving the vulnerabilities. Design issues can impact software quality in the future [31]. Therefore, it is possible that developers prioritize the changes required for resolving the vulnerabilities rather than focusing on other code issues. Szoke et al. [32] found that fixing a minor issue in the software does not increase the code quality, but bulk fixing does. Similarly, the vulnerability fixes are focused on the impacted code segments, and the entire class or method is not being refactored during the fix. Therefore, vulnerability fixing will not necessarily not improve code quality nor resolve the design issues or smells.

Despite some code smells and design issues having a significant statistical relationship with vulnerabilities, they are still present in the code even after resolving the vulnerabilities. Code with code smells and design issues is more prone to issues such as defects and vulnerabilities. Therefore, developers should prioritize them as a precautionary measure to hinder any future vulnerabilities.

6 THREATS TO VALIDITY

We looked into the smells and design issues, but other factors (e.g., change-proneness and defect-proneness) may impact the relationship between smells, design issues, and vulnerabilities. To mitigate the threat associated with Fisher's exact test on the combined frequencies of smells and issues, we calculated the χ^2 for each version separately.

This study is language and ecosystem specific because we only studied projects written in Java and from the Apache ecosystem. Though the nine systems we analyzed have different sizes and diverse functionality, different software ecosystems have different characteristics [18, 20]. Therefore, we cannot generalize our results for projects written in other programming languages or other software ecosystems. However, our findings are applicable for other systems written in Java if they have published their vulnerability details with fix-commits, CVE-IDs, and vulnerable and fixed versions. Therefore, our study can be replicated for other systems written in Java.

7 CONCLUSION

This work presented a preliminary study of nine Apache systems to determine the relationship between smells (code and architectural), design issues, and software vulnerabilities. We found that some smells and design issues have a statistically significant relationship with vulnerabilities. Though some smells and design issues are significantly related to vulnerabilities, the manual analysis shows no direct indication that smells or design issues induce vulnerabilities. In addition, we found that smells and design issues are still present in the classes, even after fixing the vulnerabilities.

In the future, we will analyze the historical evolution of code segments to identify how the smells, design issues, and vulnerabilities are related. We will also explore the impact of additional criteria, e.g., change-proneness, on software vulnerabilities and also, investigate the smells and design issues that did not have a statistically significant relationship with vulnerabilities.

ACKNOWLEDGEMENTS

This research is partly supported by an NSERC Collaborative Research and Training Experience (CREATE) grant on Software Analytics at the University of Saskatchewan.

REFERENCES

[1] W. H Brown, R. C Malveau, H. McCormick, and T. J Mowbray. 1998. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc.
[2] A. S Cairo, G. de F Carneiro, and M. P Monteiro. 2018. The impact of code smells on software bugs: A systematic literature review. *Information* 9, 11 (2018), 273.
[3] F. Camilo, A. Meneely, and M. Nagappan. 2015. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *Conf. on MSR.* 269–279.
[4] J. K. Chhabra. 2011. Code cognitive complexity: a new measure. In *World Congress on Eng.*, Vol. 2. 6–8.
[5] I. Chowdhury and M. Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Sys. Arch.* 57, 3 (2011), 294–313.
[6] M. D'Ambros, A. Bacchelli, and M. Lanza. 2010. On the impact of design flaws on software defects. In *Int. Conf. on Quality Softw.* 23–31.
[7] P. Danphitsanuphan and T. Suwantada. 2012. Code smell detecting tool and code smell-structure bug relationship. In *Spring Congress on Eng. and Tech.* 1–5.
[8] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante. 2016. Cyclomatic Complexity. *IEEE Softw.* 33, 6 (2016), 27–29.
[9] A. A. Elkhail and T. Cerny. 2019. On relating code smells to security vulnerabilities. In *Intl. Conf. on BigDataSecurity, HPSC and IDS.* 7–12.
[10] Q. Feng, R. Kazman, Y. Cai, R. Mo, and L. Xiao. 2016. Towards an architecture-centric approach to security analysis. In *Conf. on Softw. Arch. (WICSA).* 221–230.
[11] R. A. Fisher. 1992. Statistical methods for research workers. In *Breakthroughs in statistics.* Springer, 66–70.
[12] F. A. Fontana, V. Lenarduzzi, R. Roveda, and D. Taibi. 2019. Are architectural smells independent from code smells? An empirical study. *Journal of Sys. and Softw.* 154 (2019), 139–156.
[13] M. Fowler. 2018. *Refactoring: improving the design of existing code.*
[14] S. M. Ghaffarian and H. R. Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.* 50, 4, Article 56 (2017), 36 pages.
[15] A. Gong, Y. Zhong, W. Zou, Y. Shi, and C. Fang. 2020. Incorporating Android Code Smells into Java Static Code Metrics for Security Risk Prediction of Android Applications. In *Int. Conf. on Softw. Quality, Reliability and Security (QRS).* 30–40.
[16] M. Gradišnik and M Hericko. 2018. Impact of code smells on the rate of defects in software: A literature review. In *CEUR*, Vol. 2217. 27–30.
[17] A. Gupta, V. Suri, and V. Vincent. 2020. An Empirical Examination of the Relationship between Code Smells and Vulnerabilities. *Int. Journal of Computer Applications* 176, 32 (2020), 1–9.
[18] R. Hoving, G. Slot, and S. Jansen. 2013. Python: Characteristics identification of a free open source software ecosystem. In *Int. Conf. on Digital Ecosystems and Tech. (DEST).* 13–18.
[19] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden. 2012. Software vulnerability prediction using text analysis techniques. In *Int. workshop on Security Measurements and Metrics.* 7–10.
[20] J. Joshua, D. Alao, S. Okolie, and O. Awodele. 2013. Software ecosystem: features, benefits and challenges. *Int. Journal of Advanced Computer Science and Applications* 4, 8 (2013).
[21] R. Kuhn, M. Raunak, and R. Kacker. 2018. Can reducing faults prevent vulnerabilities? *Computer* 51, 7 (2018), 82–85.
[22] Rikard Land. 2002. Measurements of software maintainability. In *ARTES Graduate Student Conf.* 1–7.
[23] M. Lanza and R. Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media.
[24] G. McGraw. 2004. Software security. *IEEE Security & Privacy* 2, 2 (2004), 80–83.
[25] A. Meneely, H. Srinivasan, Ayemi Musa, Alberto R. T., M. Mokary, and B. Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *Int. Symp. on Emp. Softw. Eng. and Measurement.* 65–74.
[26] R. Mo, Y. Cai, R. Kazman, and L. Xiao. 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Conf. on Softw. Arch.* 51–60.
[27] S. M Olbrich, D. S Cruzes, and D. I. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Int. Conf. on Softw. maintenance.* 1–10.
[28] Y. Shin, A. Meneely, L. Williams, and J. A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on Softw. Eng.* 37, 6 (2010), 772–787.
[29] Y. Shin and L. Williams. 2008. An empirical model to predict security vulnerabilities using code complexity metrics. In *Int. Symp. on Empirical Softw. Eng.and measurement.* 315–317.
[30] K. Z. Sultana, Z. Codabux, and B. Williams. 2020. Examining the relationship of code and architectural smells with software vulnerabilities. In *APSEC.* 31–40.
[31] G. Suryanarayana, G. Samarthyam, and T. Sharma. 2014. *Refactoring for software design smells: managing technical debt.* Morgan Kaufmann.
[32] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?. In *Int. Working Conf. on Source Code Analysis and Manipulation.* IEEE, 95–104.
[33] C. Theisen and L. Williams. 2020. Better together: Comparing vulnerability prediction models. *Information and Softw. Tech.* 119 (2020), 106204.
[34] J. Walden, J. Stuckman, and R. Scandariato. 2014. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *Int. Symp. on Soft. Reliability Eng.* IEEE, 23–33.
[35] A. Yamashita and L. Moonen. 2013. Do developers care about code smells? An exploratory survey. In *Working Conf. on Reverse Eng. (WCRE).* IEEE, 242–251.

Counterfeit Object-Oriented Programming Vulnerabilities: An Empirical Study in Java

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Xueling Zhang
xueling.zhang@rit.edu
Rochester Institute of Technology
Rochester, NY, USA

Mehdi Mirakhorli
mxmvse@rit.edu
Rochester Institute of Technology
Rochester, NY, USA

ABSTRACT

Many modern applications rely on Object-Oriented (OO) design principles, where the basic system components are objects and classes. They share objects with other processes, store them in disk/files for future retrieval or transport them over network to other systems. Object-oriented programs leverage numerous dynamic features and design principles such as runtime dispatching and object-oriented callbacks which allow flexible software design. Although seemingly innocuous, these features can be abused by the attackers to hijack the program's control flow to an undesirable behavior. This is referred to as Counterfeit Object-Oriented Programming (COOP), in which attackers hijack objects in the program in order to create a sequence of method calls that introduce a malicious behavior. COOP is a type of code reuse attack in which a hacker hijacks objects (gadgets) in the program and use that to control the program execution flow via manipulating the sequence of methods and data being passed among these methods (gadget chains). In this paper, we describe a preliminary empirical investigation of COOP attacks in real software systems caused by untrusted object deserialization. In this preliminary study, we investigated the severity of these attacks, their consequences, and how they were mitigated by developers. Furthermore, we used the findings to create a dataset of vulnerable software projects and their fixes.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software development techniques*.

KEYWORDS

common weakness enumeration, counterfeit-object oriented programming, untrusted object deserialization

ACM Reference Format:
Joanna C. S. Santos, Xueling Zhang, and Mehdi Mirakhorli. 2022. Counterfeit Object-Oriented Programming Vulnerabilities: An Empirical Study in Java. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3549035.3561183

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR4P&S '22, November 18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9457-4/22/11…$15.00
https://doi.org/10.1145/3549035.3561183

1 INTRODUCTION

Many modern applications, whether developed in Java, Python, PHP or other languages, rely on Object-Oriented (OO) design principles [5], where the basic system components are objects and classes. Many OO architectures [9] directly operate on objects; they share these objects with other processes [9, 17], store them in disk/-files for future retrieval [3] or transport them over network to other systems [9, 15]. The encapsulations provided by object structure, the concept of classes, and inheritance has increased programs reusability and extensibility [24]. Polymorphism has enabled separation of the client class from implementation code, and allows the object to decide which form of the function to implement at compile-time (overloading) as well as runtime (overriding).

Object-oriented programs also leverage numerous other dynamic features and design principles, which allow flexible design. They commonly use *runtime dispatching* to implement object polymorphism [7]. Dispatching is typically implemented using an *indirect function call*. Similarly, program constructs such as *reflection* allows an object-oriented program to modify its structure and behavior; other dynamic mechanisms such as *object-oriented callbacks* enable the application to handle subscribed events, arising at runtime, through a listener interface and respond using predefined concrete implementations. These features can be *abused* by the attackers to hijack the program's control flow to an undesirable behavior. This is referred to as ***Counterfeit Object-Oriented Programming*** (COOP) [23, 30]. COOP is a type of code reuse attack in which a hacker hijacks objects in the program (***gadgets***) and use them to control the program execution flow via manipulating the sequence of methods and data being passed among these methods (***gadget chains***).

Counterfeit object vulnerabilities are notoriously difficult to detect and even harder to prevent [30, 35]. Mainly because they do not exhibit the revealing characteristics of existing attack approaches, and exhibit control flow and data flow similar to those of benign code execution [27, 30]. An instance of such attack is *Deserialization of Untrusted Data* which is pervasive across Java applications, and it is also emerging in Python programs due to the use of object marshaling. Furthermore, there are numerous object-oriented programming approaches for transmitting, storing or extending the behavior of objects that can result in programs vulnerable to COOP attacks.

The literature had explored COOPs in lower-level languages such as C++ [23, 30], but these languages do not include metaprogramming features. Other languages (*e.g.*, Python, PHP, and Java) contain programming constructs (*e.g.*, native calls, reflection, and object serialization) which are used to load classes, invoke methods, instantiate objects and extend the programs' functionalities

at runtime. Although seemingly innocuous, these language constructs place the system at the risk of attackers tampering with objects (*gadgets*) in order to successfully execute code (*e.g.*, load a remote class, instantiate objects from it and execute its methods with a malicious purpose). Therefore, in this paper, we present a preliminary empirical study of COOP attacks in programs written in Java. We focused on COOP attacks caused by untrusted object deserialization.

In this study, we analyzed a total of 17 vulnerability reports caused by untrusted object deserialization. By collecting and analyzing several artifacts related to the problem (*e.g.*, released patch), we investigated the severity of COOP attacks caused by untrusted object deserialization (**RQ1**), what are their consequences (**RQ2**), and how developers mitigated the problem (**RQ3**).

We observed that these COOP attacks lead to vulnerabilities with a *high*/*critical* severity. Furthermore, the investigated attacks always resulted in *remote code execution*, where an attacker is able to craft an object in such a way they could invoke arbitrary methods and execute malicious commands. Finally, we found that developers mitigate the problem in three different ways: by preventing sensitive operations to be reachable from deserialization constructs, or by enforcing the integrity of deserialized objects, or by implementing compartmentalization.

The contributions of this paper are:

- A preliminary empirical investigation of COOP vulnerabilities caused by untrusted object deserialization.
- A discussion of their *consequences*, *severity*, and *mitigations*, which gives insights to developers on how they can avoid these vulnerabilities.
- A dataset [28] of COOP vulnerabilities caused by untrusted object deserialization.

This paper is organized as follows: Section 2 briefly describes COOP vulnerabilities to ensure that the essence of the paper can be understood by a broader audience. Section 3 describes our methodology in details. Section 4 presents the qualitative analysis of COOP vulnerability reports in order to identify their root causes and mitigations. Section 5 elaborates on threats to the validity of this work. Section 6 presents related work, and Section 7 concludes this paper, including planned future work.

2 BACKGROUND

To perform a COOP attack, attackers need to take control over one object in the application (the ***initial object***) [30]. The hijacking takes place by misusing a benign feature in a program that receives objects outside its trust boundary (*e.g.*, using a serialized object received from a socket, manipulating an object created by an external plug-in, *etc.*).

This initial object will have its fields initialized with attacker-controlled data. An object may contain other objects in its fields, creating potentially complex graph-like object layouts [8]. When the program later invokes one of its methods, it leads to a sequence of malicious method calls (***gadget chains***). The classes involved in a malicious method execution chain are referred to as ***gadget classes***.

On one hand, in lower level languages, the attack is performed by manipulating pointers in the program. For example, in C++, the attacker manipulates the *vtables* (virtual method tables) such that it triggers a sequence of method calls that result in a dangerous behavior [30]. In Java, on the other hand, attackers are not able to directly manipulate pointers and memory areas, instead, it would rely on objects already available in the classpath for use. Moreover, unlike C++, Java has reflection, a metaprogramming feature that allows classes to be loaded at runtime, and have their methods invoked; creating space for attackers to even be able to load remote classes (*i.e.*, outside the classpath).

Figure 1 contains three COOP attack scenario examples in Java. These attacks are caused by misusing three commonly used benign features: object deserialization, Remote Method Invocation (RMI), and the Java Naming and Directory Interface™ (JNDI). In these examples, consider that the classes in Listing 1 are available in the classpath. We explain these attacks in the next subsections.

2.1 Untrusted Object Deserialization

The first attack (Figure 1a) relies on *untrusted object deserialization*. Object serialization (also known as "marshaling") is a mechanism in which an object is converted to an abstract representation (*e.g.*, bytes, XML, JSON, *etc.*) that models the object's state (*i.e.*, fields' values and code). This abstract representation is suitable for network transportation, storage, and inter-process communication. The receiver of a serialized object has to parse the abstract representation in order to reconstruct a new object, a process called object deserialization (or "unmarshalling").

Although object serialization seems innocuous, several deserialization mechanisms allow arbitrary types to be deserialized and invoke methods from the objects' classes during their reconstruction (*e.g.*, default constructors, getter/setter methods, callback methods (also known as "magic methods", *etc.*) [19]. Attackers could leverage these methods invoked during object deserialization to conduct COOP attacks that can result in resource consumption (denial-of-service attacks), application crashes and remote code execution [8, 25].

The class `ObjectOutputStream` is part of Java's built-in deserialization API. It reconstructs an object from a *byte stream* that contains the object's fields values. This class can reconstruct any object, as long as its class implement the `java.io.Serializable` interface. If implemented by a `Serializable` class, the callback methods listed below are invoked by Java during deserialization. These methods, henceforth referred to as ***"magic methods"***, are the ones used by attackers to create a malicious sequence of method invocations (*gadget chain*):

1. `void readObject(ObjectInputStream)`: it customizes the retrieval of an object's state from the stream.
2. `void readObjectNoData()`: in the exceptional situation that a receiver has a subclass in its classpath but not its superclass, this method is invoked to initialize the object's state.
3. `Object readResolve()`: this is the inverse of `writeResolve`. It allows classes to replace a specific instance that is being read from the stream.
4. `void validateObject()`: it validates an object after it is deserialized. For this callback to be invoked, the class has to implement the `ObjectInputValidation` interface and register the

```java
class Task implements Runnable,
    Serializable {
  private String cmd;

  public CommandTask(String c) {
    this.cmd = c;
  }

  public void run() {    /* sink */
    Runtime.getRuntime().exec(cmd);
  }
}
```

```java
class TaskManager implements Serializable {
  private Runnable task;
  public TaskManager(Runnable t){ this.task = t; }
  private void readObject(ObjectInputStream ois){
    ois.defaultReadObject();
    task.run();
  }
}
interface Analyzer extends Remote {
  void analyze(Runnable r);
  void cleanResults();
}
```

```java
class AnalyzerImpl implements Analyzer{
  private File results;
  public AnalyzerImpl(File f){
    this.results = f;
  }
  public void analyze(Runnable r){
    r.run();
  }
  public void cleanResults(){
    results.delete(); /* sink */
  }
}
```

Listing 1: "Gadget classes" that can be used in a COOP attack to trigger a remote code execution.

```java
class IndexServlet extends HttpServlet {
  protected void doGet(HttpServletRequest rq,
      HttpServletResponse rs) {
    Cookie c = getCookieByName(rq, "user");
    if (c != null) {
      byte[] bytes = Base64.getDecoder()
        .decode(c.getValue());
      ObjectInput in = new ObjectInputStream(
        new ByteArrayInputStream(bytes)
      );
      User u = (User) in.readObject();
    } else { /* ... */ }
  }
}
```

(a)

```java
class RMIServer  {
  public static void main(String a[]){
    try {
      Analyzer obj = new AnalyzerImpl(null);
      Analyzer stub =
          (Analyzer) UnicastRemoteObject
          .exportObject(obj, 0);
      Registry registry =
          LocateRegistry.getRegistry();
      registry.bind("analyzer", stub);
    }catch (Exception e)
    { /* ... */ }
  }
}
```

(b)

```java
class JNDIExample{
  public static void main(String[]a){
    try {
      String name = a[0];
      Context ctx =
        new InitialContext();
      Analyzer analyzer =
        (Analyzer) ctx.lookup(name);
      analyzer.cleanResults();
    } catch (Exception e) {
      /* ... */
    }
  }
}
```

(c)

Figure 1: COOP attacks that rely on (a) untrusted object deserialization, (b) RMI, and (c) JNDI.

validator by invoking the method `registerValidation` from `ObjectInputStream` class.

As an example, Figure 1a contains a code snippet from a sample Web application (`IndexServlet`) that retrieves the "user" cookie from the HTTP request. This cookie is expected to contain a serialized `User` object encoded using *Base64*. An attacker could leverage the deserialization process to conduct a COOP attack by using two available serializable classes in the classpath (`TaskManager` and `Task` — gadget classes). An attacker would create a `TaskManager` object (`taskMgr`) as shown in Figure 2a. Then, the attacker serializes and encodes this malicious object in base64 and sends it as the "user" cookie to the Web application.

When the web application deserializes the object in the cookie, Java's deserialization mechanism (*ObjectInputStream*) invokes the callback method `readObject()` from the `TaskManager` class. It triggers the chain of method calls listed in Figure 2b. This gadget chain ends in a "sink"[1] – `exec()` – that executes a command to remove all files ("rm -rf /").

Although this request with a malicious serialized object will later trigger a `ClassCastException` (because the application attempt to cast the read object as a `User` type), the malicious command was already executed, because the type cast check occurs ***after*** the deserialization process took place.

2.2 RMI-based Attacks

The second example (`RMIServer` in Figure 1b) includes a sample Remote Method Invocation (RMI) server that exports an instance of the `AnalyzerImpl` class. An attacker can implement an RMI client that first looks up this object by its name ("analyzer") on the RMI server. Subsequently, this malicious client makes a remote procedure call to the `analyzer(Runnable r)` method passing as argument the malicious object `task` (in Figure 2a). This triggers the execution shown in Figure 2c which will lead to a recursive deletion of files in the root directory.

2.3 JNDI-based Attacks

In the third example (`JNDIExample` in Figure 1c), an application performs an object lookup by name using Java Naming and Directory Interface™ (JNDI) [32]. An attacker can implement a malicious RMI server that export the `analyzer` object in Figure 2a and binds it to the name "exploit". Subsequently, the attacker can invoke the program making a lookup to "rmi:/exploit", which will inject the malicious object and execute the call chain in Figure 2d, resulting in a deletion of the root directory.

3 METHODOLOGY

In this study, we focused on investigating COOP attacks caused by *untrusted object deserialization* (described in Section 2.1). In this section, we first introduce our research questions (Section 3.1), then we explain the methodology we followed to answer each of them (Section 3.2), and finally, we discuss how we compile our artifacts as a dataset (Section 3.3).

3.1 Research Questions

We answered the following research questions in this paper:

[1]Sinks are methods in the program's scope that performs sensitive operations, such as executing commands and manipulating file

Malicious Objects:
```
Task task =
  new Task("rm -rf /");
TaskManager taskMgr =
  new TaskManager(task);
Analyzer analyzer =
  new AnalyzerImpl(new File("/"));
```
(a)

Call Stack for Deserialization:
```
IndexServlet.doGet(...)
  java.io.ObjectInputStream.readObject()
    TaskManager.readObject(...)
      Task.run()
        Runtime.exec("rm -rf /")
```
(b)

Call Stack for RMI:
```
RMIExample.main(...)
  AnalyzerImpl.analyze(task)
    Task.run()
      Runtime.exec("rm -rf /")
```
(c)

Call Stack for JNDI:
```
JNDIExample.main("rmi:/exploit")
  AnalyzerImpl.cleanResults()
    File.delete()
```
(d)

Figure 2: (a) Malicious objects crafted by an attacker. Call stacks for a successful COOP attack that relied on (b) object deserialization, (c) RMI, and (d) JNDI.

> **RQ1** *How severe are COOP attacks caused by untrusted object deserialization?*

We focused on understanding what is the perceived severity of these problems by developers.

> **RQ2** *What are the consequences of COOP attacks related to untrusted object deserialization vulnerabilities?*

We aimed to identify the faulty behavior observed when an untrusted object deserialization vulnerability is successfully executed.

> **RQ3** *How are COOP vulnerabilities related to untrusted object deserialization mitigated?*

We studied the strategies employed by developers to fix these vulnerabilities in real software systems.

3.2 Answering the Research Questions

To answer these questions, we conducted an in-depth analysis of vulnerability reports (CVEs) in the National Vulnerability Database (NVD). NVD is a well-known vulnerability database, which currently tracks over 191,000 vulnerabilities that exist in a variety of software products, both open and closed source.

Vulnerabilities disclosed in NVD are assigned a unique identifier known as "*CVE ID*" (Common Vulnerabilities and Exposure Identifier). Besides a CVE ID, each entry in NVD includes a short *description* of the problem and a list of *references, i.e.*, links to other Websites (such as issue tracking systems) that may contain more details about the CVE instance. NVD also indicates the software's *releases* affected by the vulnerability and a *severity score*.

Some CVE instances may also include *CWE tags* that indicate the *vulnerability type*. These tags are assigned by security analysts from the entities that reviewed the vulnerability report. The CWE tag refers to an entry from the *Common Weakness Enumeration (CWE)* dictionary [33], which enumerates common software/hardware weaknesses that may lead to vulnerabilities. A *weakness* denotes a family of security defects that share one or more aspect in common, such as a similar fault (*root cause*), failure (*consequence*), or fix (*repair*) [22]. Thus, the CWE tag is used by the NVD as a way to classify vulnerabilities.

Therefore, we first retrieved from NVD all the CVEs that either contained the keyword "`serializ`" in its *description* or whose *CWE tag* was equal to CWE-520 (Deserialization of Untrusted Data) [34]. Subsequently, we disregarded CVEs that (i) were in closed source systems, since there would not be enough public information for us to answer our research questions; or (ii) were in software systems implemented in a language other than Java.

We randomly selected a subset of 17 CVEs to identify its *severity*, *consequences*, and *mitigation techniques* implemented to fix the issue. Afterwards, we performed a qualitative analysis of these 17 CVEs and their associated artifacts to answer our research questions. We performed the following steps:

(1) For each CVE, we extracted its metadata from NVD (*description*, *CWE tag*, *references*, and *severity score*).
(2) We relied on the URLs in the references to identify the corresponding entry in the project's issue tracking system. From the issue tracking system entry, we then verify whether the vulnerability was acknowledged by developers and fixed. If a patch was publicly released, we collect both the project's vulnerable version and fixed version (that includes the fix).
(3) We manually analyzed these collected artifacts in order to capture information regarding the CVE's *vulnerable version* and *fixed version*, its *severity*, its *consequences*, as well as the *mitigation technique* implemented by developers to fix the problem. We obtained the severity for each CVE based on the CVSS score[2] provided by NVD. To identify the *consequences*, and *mitigation techniques*, we performed a qualitative analysis of the vulnerability report and associated artifacts. This qualitative analysis involved an open coding [21] in which we iteratively reviewed the artifacts and annotated each vulnerability with *codes*: one to indicate the mitigation technique used to fix the problem, and other(s) to indicate the consequence(s) of the vulnerability. During this open coding, we either annotated CVEs with codes already used or created new codes that emerged from the data (if the existing codes were not suitable for the CVE being analyzed). This open coding was performed by the first author, who has eight years of experience in software security.

After performing the above steps, we used the collected artifacts to answer each RQ as follows:

- RQ1 We relied on the CVSS score provided by NVD, which is a number that ranges from 0 (least severe) to 10 (most severe).
- RQ2 We answer this question by analyzing the consequences we observed while performing the open coding of CVEs.
- RQ3 Similar to RQ2, this question is answered by inspecting the results of our open coding, in which we observed the different ways developers patched their projects.

[2]The Common Vulnerability Scoring System (CVSS) is a framework [20] used to measure the severity of a vulnerability.

3.3 A Dataset of COOP Vulnerabilities

After our qualitative analysis, we compiled these artifacts as a manually curated dataset of COOP vulnerabilities caused by untrusted object deserialization. This dataset includes a CSV file with the following metadata [28]: (i) **CVE ID**; (ii) the **vulnerable** and **fixed** versions of the project; (iii) **consequence**; (iv) CVSS score (**severity** – low, medium, high, critical); (v) **mitigation technique**.

4 RESULTS

In the next sections, we discuss our findings and answer our RQs.

4.1 RQ1: Severity

Table 1 presents the breakdown of the severity observed in the analyzed CVEs. The severity is based on the categorization given by the CVSS score v3. For two CVEs we analyzed[3], however, the severity score was based on CVSS v2 because there was no score provided using the version 3.x of the CVSS framework. The CVSS score ranges from 0 to 10, where a score from 0.1-3.9 is considered as *low* severity, 4.0-6.9 as *medium* severity, 7.0-8.9 as *high* severity, and 9.0-10.0 as *critical* severity.

Table 1: Severity of COOP vulnerabilities related to untrusted object deserialization

	Severity		
	Critical (9.0-10.0)	**High** (7.0-8.9)	**Medium** (4.0-6.9)
# CVEs	6 (35.2%)	9 (52.9%)	1 (5.8%)

We observe from the findings reported in Table 1 that the majority of vulnerabilities are classified as *high* severity. We also notice that 6 CVEs (35%) were also categorized as *critical* vulnerabilities. None of the vulnerabilities analyzed had a *low* severity score – the lowest observed CVSS score was 5.9 (*medium*) and the highest was 9.8 (*critical*).

One of the reasons as to why the severity scores were mostly high/critical was due to the fact that all the vulnerabilities had an attack vector through the network. That is, a hacker could deploy the attack remotely, making it easier to conduct successful attacks. This finding highlights the importance of studying COOP-related vulnerabilities.

4.2 RQ2: Consequences

We observed that all vulnerabilities lead to *remote code execution*. For one of the vulnerabilities (CVE-2016-1000031), besides code execution, an attacker could also manipulate local files (*e.g.*, delete/create local files).

The main attack vector used by intruders to execute arbitrary commands was via the use of *Java reflection*. That is, the gadget chain lead to a reflection construct that allowed attacks to load arbitrary classes, create instances, and invoke their methods using malicious data.

[3] These CVEs were: CVE-2015-6420 and CVE-2015-8103.

4.3 RQ3: Mitigation Techniques

By scrutinizing these 17 vulnerability reports, we observed that there were three ways that developers fixed COOP vulnerabilities caused by untrusted object deserialization: (i) by preventing untrusted data to reach a sink (***unreachable sinks***); (ii) by enforcing the integrity of serialized and deserialized objects (***enforcing integrity***); or (iii) ***compartmentalization***. The mitigation techniques and their corresponding category is presented in Table 2.

Table 2: Mitigation techniques for untrusted object deserialization

Category	Mitigation	#CVEs
Unreachable Sinks	M1.1 Allowed/blocked list of classes	7
	M1.2 Prevent deserialization of domain objects	4
	M1.3 Unsafe classes are no longer serializable	2
Enforcing Integrity	M2.1 Adding the "transient" to a sensitive field	1
	M2.2 Authenticate before deserializing an object	1
	M2.3 Replace Java's default deserialization API	2
Compartmentalization	M3.1 Deserialize within a sandbox	1

We can observe that developers mostly chose to fix vulnerabilities by making the sink unreachable. Among the mitigation strategies used to achieve this goal, the most used one was to create a list of classes that are allowed/blocked to be deserialized (**M1.1**). Some CVEs implemented multiple mitigations as part of their fix (*e.g.*, CVE-2015-6420 in Apache commons collections used mitigations **M1.2** and **M1.3**). In the next subsections, we elaborate on each of these mitigation techniques.

4.3.1 Group 1: Unreachable sinks.

It contains mitigation techniques that make the sink *unreachable*. These mitigation techniques are:

M1.1 Allowed/Blocked list of classes: It maintains a list of classes that *may* or *may not* be deserialized (*allow list* and *block list*, respectively). When using Java's default deserialization API, this can be implemented by creating a subclass of `ObjectInputStream` that overrides the `resolveClass(ObjectStreamClass o)` method. This method throws an exception when the object type is either in the *block list* or not in the *allow list* [31].

Example: For instance, the CVE-2019-12384 is fixed by adding a gadget class into a list of *blocked classes* that cannot be deserialized, as shown below in the "unidiff" of the commit [13]. This commit blocks the serialization of instances of the class `DriverManagerConnectionSource`.

```
src/main/java/com/fasterxml/jackson/databind/jsontype/impl/SubTypeValidator.java
        // [databind#2326] (2.7.9.6): one more 3rd party gadget
        s.add("com.mysql.cj.jdbc.admin.MiniAdmin");
+
+       // [databind#2334] (2.9.9.1): logback-core
+       s.add("ch.qos.logback.core.db.DriverManagerConnectionSource");

        DEFAULT_NO_DESER_CLASS_NAMES = Collections.unmodifiableSet(s);
    }
```

We also observed that most of the fixes involved using a list of *"blocked classes"* (5 times) compared to the use of *"allow lists"*, which was observed in only one CVE. In another remaining vulnerability instance (CVE-2017-15693), we observed that it had a configuration mechanism that allowed users to create a list of blocked and allowed classes.

Although the use of *"blocked classes"* was the most common mitigation technique implemented, it is inherently problematic. New gadget classes, that are not in the list, can be found over time by attackers and be used to conduct malicious attacks. In fact, the CVE-2019-12384 with the fix shown above was due to not having a class from the logback core project in the malicious list.
One of the reasons as to why this occurs is because the use of blocked lists is easier to implement while minimizing the chances of backwards compatibility. The use of allow lists make the program more strict about what classes can be deserialized, making genuine program flows to be disrupted with the fix.

M1.2 Prevent deserialization of domain objects: This mitigation is typically used when the application has a class that extends another serializable class (directly or indirectly) which provides concrete implementations to callback methods (*i.e.*, "magic methods"). Therefore, to prevent malicious uses of these subclasses, the application breaks the chain of method calls by throwing an exception. Hence, the dangerous sink is unreachable because the chain of calls from a magic method – *e.g.*, `readObject(ObjectInputStream)`– to a sink method is broken due to a thrown exception.
Example: The jython project has a class named `PyFunction` that extends the class `PyObject`, which in turn implements the `java.io.Serializable` interface. This inheritance relationship makes the `PyFunction` class to be serializable too. In CVE-2016-4000, the `PyFunction` class was found to be used in successful COOP attacks. Hence, the fix implemented by developers prevents the class `Handler` to be deserialized. This is implemented by overriding the method `readResolve()` and making it throw an exception [1], as shown in the unidiff below for the fix:

```
src/org/python/core/PyFunction.java
    @Override
    public boolean isSequenceType() { return false; }

+   private Object readResolve() {
+       throw new UnsupportedOperationException();
+   }

    /* Traverseproc implementation */
    @Override
```

M1.3 Unsafe classes are no longer serializable: This mitigation technique involves making a gadget class no longer serializable. This is implemented by removing the "`extends Serializable`" from the class definition.
Example: In the Apache Commons FileUpload project version 1.3.2, a class named `DiskFileItem` implements the interface `FileItem`, which extends the `java.io.Serializable` interface. As a result, the `DiskFileItem` class also becomes serializable. In CVE-2016-1000031, researchers found that the `DiskFileItem` has a magic method (invoked during deserialization) that allowed an attacker to manipulate files. The project's developers fixed this problem by making `DiskFileItem` no longer serializable, as shown in the commit diff below [10]:

```
src/main/java/org/apache/commons/fileupload/FileItem.java
-public interface FileItem extends Serializable,FileItemHeadersSupport{
+public interface FileItem extends FileItemHeadersSupport {
```

4.3.2 *Group 2: Enforcing object integrity.*

It encompasses the mitigation approaches below that enforce the integrity of the object:

M2.1 Add transient to a "sensitive" field: To prevent serializing fields with sensitive information (*e.g.*, passwords) or that are used as part of a gadget chain, applications enforce that these fields are not included when the object is serialized. This is achieved by adding the keyword `transient` to the field declaration [25]. By doing that, Java's built-in deserialization class ignores the field and does not write/read its value when serializing/deserializing the object.
Example: The beanshell project version 2.0b5 contains a serializable class named `XThis` that has a field named `invocationHandler`. This field is instantiated with a concrete implementation for the `java.lang.InvocationHandler` interface that uses reflection to invoke methods. An attacker relied on this class (`Handler`) to invoke arbitrary methods in the program (CVE-2016-2510). To fix this issue, the developers made the `Handler` class non-serializable (**M1.3**) and the `invocationHandler` field to be transient [11], as shown in the commit below:

```
src/bsh/XThis.java
- InvocationHandler invocationHandler = new Handler();
+ transient InvocationHandler invocationHandler = new Handler();
...
- class Handler implements InvocationHandler, java.io.Serializable
+ class Handler implements InvocationHandler
```

M2.2 Authenticate before deserializing an object: This mitigation is used when: (i) the application has to transmit objects, (ii) it does have a secure transport channel (*e.g.*, SSL) that can be used for authentication, and (iii) these objects need to be received in its entirety. In this case, marking fields as "transient" would not fulfill the application's needs [18]. This mitigation involves authenticating the remote source before receiving objects from it.
Example: In CVE-2016-3737 affecting the server in Red Hat JBoss Operations Network (JON) before 3.3.6, an attacker could craft a malicious object and send it to the server to trigger remote code execution. Since removing serialization and/or classes would not be a feasible mitigation, the fix for this issue involved manually configuring the JON to use SSL client authentication between servers and agents. The released version updated its documentation to guide the users on how to properly perform this configuration.

M2.3 Replace Java's default deserialization API: Java's built-in (de)serialization API allows arbitrary object types to be serialized/deserialized as long as it implements the `Serializable` interface. Since this API invokes methods from the objects' classes during their reconstruction (*i.e.*, magic methods), this built-in mechanism is deemed as inherently insecure [2]. Consequently, some applications decide to replace (or disable) this feature entirely to prevent vulnerabilities.
Example: In CVE-2017-1000034, the akka project disables Java's default serialization API and replaces it with its own (safer) serialization implementation [12].

4.3.3 *Group 3: Compartmentalization.*

This category includes mitigation approaches in which the system enforces policies at runtime to prevent object deserialization misuse.

M3.1 Deserialize within a sandbox: A sandbox is used whenever an object is deserialized. This sandbox is configured with a set of policies that are enforced at runtime. Thus, if the deserialized object triggers an operation forbidden by the policy, the object reconstruction is stopped [14, 29]. Sandboxes are usually implemented using Java's `SecurityManager` class. This built-in class throws a `SecurityException` when it detects that a process is executing an operation not allowed by the security policy in place.

Example: To fix CVE-2018-1000058 (Jenkins project), developers made the deserialization of objects to be executed under a sandbox. Thus, an attacker is not able to execute arbitrary code in the pipeline. A (partial) implementation of the fix is shown in the code snippet below. The `SandboxedUnmarshaller` wraps the execution of all the deserialization operations such that they all run with sandbox protection.

org/jenkinsci/plugins/workflow/support/pickles/serialization/RiverReader.java

```
+ /** Applies {@link GroovySandbox} to a delegate unmarshaller. */
+ private static final class SandboxedUnmarshaller ... {
+
+   private final Unmarshaller delegate;
+
+   SandboxedUnmarshaller(Unmarshaller delegate) {
+     this.delegate = delegate;
+   }
+   ...
+
+   @Override public Object readObject() throws /* ... */ {
+     return sandbox(() -> delegate.readObject());
+   }
+
+   @Override public Object readObjectUnshared() throws /* ... */ {
+     return sandbox(() -> delegate.readObjectUnshared());
+   }
+   ...
+ }
```

4.4 Discussion

The key takeaways from our results are:

- **COOP attacks can lead to severe vulnerabilities**: This initial empirical study highlighted the importance of investigating COOP attacks. In our findings, we observed that CVEs related to untrusted object deserialization, a type of COOP attack, were often assigned by security analysts a high/critical severity score. One of the reasons being that attackers could deploy their attacks remotely, making it easier to reproduce attacks.
- **Developers may use inherently flawed/improper mitigations**: We observed that developers often used "blocked lists" (M1.1 discussed in Section 4.3) as a way to fix their vulnerability. The key problem, however, is that manually curating a list of dangerous classes lead to missing unknown gadget classes. That is, developers hardcode this list of dangerous classes based on prior knowledge of existing attacks. As new attacks are deployed, developers then have to patch the code by adding other class signatures to their list of blocked classes. This is a *reactive* mitigation strategy rather than a *proactive* approach.
- **There are trade-offs involved in the choice of employing a specific mitigation strategy**: We observed that there are multiple ways that developers fixed COOP vulnerabilities. The chosen mitigation strategy will often be a trade-off between the efforts required in changing the software, as well as backward compatibility considerations. As presented in Section 4.3, some mitigation strategies, such as replacing Java's default deserialization API (M2.3) would require extensive implementation and testing efforts. For that reason, developers often relied on a simpler solution, such as using a list of blocked classes that cannot be deserialized (M1.1). The use of "allow lists" is a safer alternative to the use of "blocked lists". However, this mitigation could also prevent the deserialization of genuine payloads, affecting the system's intended functionality. Therefore. although inherently less secure, the use of blocked lists was the most frequently employed strategy because it is easier to implement and reduce backward compatibility problems.

5 THREATS TO VALIDITY

One threat concerns the *construct validity* of our work; that is, to what extent the operational measurements we used are suitable for the purpose of our study [26]. In this context, two related threats are that (i) our analysis heavily depends on the accuracy of the collected reports (*i.e.*, CVEs, and patches, as described in Section 3.2) and (ii) the open coding of vulnerability reports. We mitigate this threat by following a *systematic process* in which we manually inspected each CVE and associated artifacts for completeness and accuracy. Moreover, this manual analysis was performed by one of the authors who has over 8 years of software security experience.

Another threat relates to the *generalizability* of the findings of the work (*external validity* [26]). We studied only the COOP attacks that are related to object deserialization and in Java programs. Since we analyzed a random sample that included only 17 vulnerabilities, we acknowledge the results may not generalize to other languages (*e.g.*, Python) and COOP attack types (*e.g.*, RMI-based COOP attacks). It is nonetheless important to highlight that our study's scope was not to find *generalizable* findings, but rather to give insights on this under-explored type of attacks and create a manually curated dataset that could help other researchers and practitioners.

6 RELATED WORK

The literature explored COOPs in lower-level languages such as C++ [4, 23, 30, 35], but these languages do not include metaprogramming features. Other languages (*e.g.*, Python, PHP, and Java) contain programming constructs (*e.g.*, native calls, reflection, and object serialization) which are used to load classes, invoke methods, create objects and extend the programs' functionalities at runtime. Although seemingly innocuous, these mechanisms place the system at the risk of attackers tampering with objects (*gadgets*) in order to successfully execute code (*e.g.*, load a remote class, instantiate objects from it and execute its methods with a malicious purpose).

Prior empirical studies explored vulnerabilities rooted in improper input validation problems, such as SQL injection, and buffer overflows [6, 16, 37] as well as language-specific vulnerabilities [36]. COOP vulnerabilities, however, are very different from these explored vulnerabilities. First, "dangerous operations" (*i.e.*, *sinks*) can be anywhere in the program's scope (*i.e.*, the language's built-in classes, library classes and the application code itself). Second, COOP attacks rely on dynamic programming features. Third, unlike these other classes of injection problems, in which the input

is a primitive or string, the input provided by the attacker is a specially crafted object. Hence, this study aimed to provide insights to developers on how to spot these problems and fix them.

7 CONCLUSION & FUTURE WORK

In this paper, we studied COOP attacks caused by untrusted object deserialization in Java programs. We investigated their severity, typical consequences, and mitigation techniques used by developers to prevent the attacks. Among our findings, we observed that deserialization-related COOP attacks were often flagged with a high severity. We also observed that one of the reasons for this high/critical severity was due to the fact that these attacks lead to remote code execution. We also found 7 different mitigation strategies employed by developers to prevent COOP attacks.

In the future, we plan to cover more vulnerabilities related to not only object deserialization, but also other COOP attack vectors (*e.g.*, RMI-based). Hence, we plan to (i) extract CVEs from NVD that are related to COOP, (ii) analyze publicly available exploits (iii) review the source code of open source systems with dynamic features such as *deserialization*, *RMI*, *JNDI*, *Dependency Injection*, *Java Management Extensions (JMX) API* and others that can enable counterfeit Object-Oriented programming attacks.

ACKNOWLEDGMENTS

This work was partially funded by the US National Science Foundation under grants number CNS-1816845 and CCF-1943300.

REFERENCES

[1] 2016. jython: d06e29d100c0. https://hg.python.org/jython/rev/d06e29d100c0 [Online; accessed 29. Jul. 2022].
[2] 2022. Secure Coding Guidelines for Java SE: Serialization and Deserialization. https://www.oracle.com/java/technologies/javase/seccodeguide.html#8 [Online; accessed 30. Jul. 2022].
[3] Tommi Aihkisalo and Tuomas Paaso. 2011. A Performance Comparison of Web Service Object Marshalling and Unmarshalling Solutions. In *2011 IEEE World Congress on Services*. 122–129. https://doi.org/10.1109/SERVICES.2011.61
[4] Markus Bauer and Christian Rossow. 2021. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 650–666.
[5] Grady Booch. 1982. Object-oriented design. *ACM SIGAda Ada Letters* 1, 3 (1982), 64–76.
[6] Larissa Braz, Enrico Fregnan, Gül Çalikli, and Alberto Bacchelli. 2021. Why Don't Developers Detect Improper Input Validation?'; DROP TABLE Papers;–. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 499–511.
[7] Brad Calder and Dirk Grunwald. 1994. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. ACM, New York, NY, USA, 397–408. https://doi.org/10.1145/174675.177973
[8] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.10
[9] W. Emmerich and N. Kaveh. 2002. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 691–692.
[10] GitHub. 2016. apache/commons-fileupload. https://github.com/apache/commons-fileupload/commit/02f6b2c4ef9aebf9cf8e55de8b90e73430b69385 [Online; accessed 30. Jul. 2022].
[11] GitHub. 2016. Avoid (de)serialization of XThis.Handler · beanshell/beanshell@1ccc66b. https://github.com/beanshell/beanshell/commit/1ccc66bb693d4e46a34a904db8eeff07808d2ced [Online; accessed 29. Jul. 2022].
[12] GitHub. 2017. akka/akka. https://github.com/akka/akka/commit/cc6561b47e5958923df520b8a9514010d3e11d49 [Online; accessed 30. Jul. 2022].
[13] GitHub. 2019. Fix #2334 · FasterXML/jackson-databind@c9ef4a1. https://github.com/FasterXML/jackson-databind/commit/c9ef4a10d6f6633cf470d6a469514b68fa2be234 [Online; accessed 28. Jul. 2022].
[14] GitHub. 2021. jenkinsci/workflow-support-plugin- Pipeline: Supporting APIs Plugin. https://github.com/jenkinsci/workflow-support-plugin/commit/a9b071025b5eea33176cefddc1928bce9904c0ef. [Accessed 07/17/2021].
[15] Konrad Grochowski, Michał Breiter, and Robert Nowak. 2019. Serialization in Object-Oriented Programming Languages. In *Introduction to Data Science and Machine Learning*, Keshav Sud, Pakize Erdogmus, and Seifedine Kadry (Eds.). IntechOpen, Rijeka, Chapter 12. https://doi.org/10.5772/intechopen.86917
[16] Munawar Hafiz and Ming Fang. 2016. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* 21, 5 (2016), 1920–1959.
[17] D. Hagimont and F. Boyer. 2001. A configurable RMI mechanism for sharing distributed Java objects. *IEEE Internet Computing* 5, 1 (2001), 36–43. https://doi.org/10.1109/4236.895140
[18] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. 2011. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional.
[19] Dustin Marx. 2018. JDK 11: Beginning of the End for Java Serialization? https://dzone.com/articles/jdk-11-beginning-of-the-end-for-java-serialization. (Accessed on 04/07/2020).
[20] P. Mell, K. Scarfone, and S. Romanosky. 2006. Common Vulnerability Scoring System. *IEEE Security Privacy* 4, 6 (Nov 2006), 85–89. https://doi.org/10.1109/MSP.2006.145
[21] Matthew B Miles, A Michael Huberman, and Johnny Saldana. 2013. *Qualitative data analysis*. Sage.
[22] Martin Monperrus. 2014. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 234–242.
[23] Paul Muntean, Richard Viehoever, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2021. ITOP: Automating Counterfeit Object-Oriented Programming Attacks. In *24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. ACM, New York, NY, USA, 162–176. https://doi.org/10.1145/3471621.3471847
[24] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. *SIGPLAN Not.* 51, 6 (jun 2016), 27–41. https://doi.org/10.1145/2980983.2908119
[25] Or Peles and Roee Hay. 2015. One Class to Rule Them All: 0-Day Deserialization Vulnerabilities in Android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C., 12 pages.
[26] Per Runeson and Martin Hoest. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14 (2009), 131–164.
[27] Joanna C. S. Santos. 2021. *Understanding and Identifying Vulnerabilities Related to Architectural Security Tactics*. Ph. D. Dissertation. Rochester Institute of Technology.
[28] Joanna C. S. Santos, Xueling Zhang, and Mehdi Mirakhorli. 2022. *COOP Vulnerabilities Dataset*. https://github.com/SoftwareDesignLab/coop-dataset
[29] Will Sargent. 2021. Self-Protecting Sandbox using SecurityManager · Terse Systems. https://tersesystems.com/blog/2015/12/29/sandbox-experiment [Online; accessed 17. Jul. 2021].
[30] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 745–762. https://doi.org/10.1109/SP.2015.51
[31] Robert Seacord. 2017. Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS).
[32] Michael Stepankin. 2019. Exploiting JNDI injections in Java. https://www.veracode.com/blog/research/exploiting-jndi-injections-java
[33] The MITRE Corporation 2022. *CWE - Common Weakness Enumeration*. The MITRE Corporation. http://cwe.mitre.org
[34] The MITRE Corporation 2022. *CWE-502: Deserialization of Untrusted Data*. The MITRE Corporation. http://cwe.mitre.org/data/definitions/502.html
[35] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.
[36] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2020. An empirical study of C++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering* (2020).
[37] Tao Ye, Lingming Zhang, Linzhang Wang, and Xuandong Li. 2016. An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 91–101.

SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT

Automated source code generation is currently a popular machine-learning-based task. It can be helpful for software developers to write functionally correct code from a given context. However, just like human developers, a code generation model can produce vulnerable code, which the developers can mistakenly use. For this reason, evaluating the security of a code generation model is a must. In this paper, we describe SecurityEval, an evaluation dataset to fulfill this purpose. It contains 130 samples for 75 vulnerability types, which are mapped to the Common Weakness Enumeration (CWE). We also demonstrate using our dataset to evaluate one open-source (*i.e.*, InCoder) and one closed-source code generation model (*i.e.*, GitHub Copilot).

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software development techniques*; *Software verification and validation.*

KEYWORDS

dataset, common weakness enumeration, code generation, security

ACM Reference Format:
Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3549035.3561184

1 INTRODUCTION

Code generation techniques are used to generate functional source code from a given *prompt*, which could be a comment, an expression in the form of the function signature, or their mixture [2]. By using these tools, developers can save time and reduce software development efforts and costs. Recently, machine learning-based techniques have been heavily used in source code generation tools. Large Language Learning Models (LLM) using attention-based transformer technique [30] are pre-trained with textual data, including source code snippets. Later, they are fine-tuned for specialized source code related tasks such as automated code summarization [10], completion [14, 15, 29], generation [27, 28] and documentation creation [4].

Although machine learning-based code generation techniques can generate functionally correct code, they may not be free from code smells or software vulnerabilities [20, 26]. Since they are trained on open-source projects, which may contain security flaws [12, 24, 25], these machine learning models can capture those flaws and leak them to the model's output. Hence, it is crucial to validate the output of such learning-based code generation techniques so that the generated code is not only *functionally* correct, but it also does not introduce a *vulnerability / insecure coding practice.*

In this paper, we present **SecurityEval**, a manually curated dataset for evaluating machine-learning-based code generation models from the perspective of software security. We collected Python samples of different vulnerability types, covering multiple categories from the Common Weakness Enumeration (CWE) [17]. Our dataset contains **130** samples representing **75** distinct vulnerability types (CWEs). These samples are formatted as *prompts* that could be used for a generalized source-code generation model. We released this dataset in our repository: https://github.com/s2e-lab/SecurityEval.

2 DATASET CONSTRUCTION

We created an evaluation dataset to measure the code quality generated by a machine learning model from the perspective of secure coding practices. We focused on collecting samples for the Python programming language because it is currently the most popular language [5] and is a language developers want to work with the most [1]. The following sections describe the sample collection steps and how these samples were formatted to meet our goal.

2.1 Samples Collection

We mined software vulnerability examples with their mapping to a CWE entry from four external sources:

- **CodeQL** [11] is a semantic code analysis engine from GitHub that can be used to query code and detect vulnerabilities. Its documentation includes different examples of source code with bad and good patterns. Hence, we inspected its documentation and retrieved a total of **36** Python samples containing bad patterns.
- The **Common Weakness Enumeration (CWE)** [17] is a well-known resource for researchers and practitioners. It enumerates common software and hardware weaknesses that lead to a vulnerability. Almost every entry in the CWE

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR4P&S '22, November 18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9457-4/22/11…$15.00
https://doi.org/10.1145/3549035.3561184

list provides examples of insecure code in different programming languages (*e.g.*, Java, C, PHP *etc.*). We extracted a total of **11** Python samples from it.
- **Sonar Rules**: *SonarSource* [23] is a company that has a static analyzer for finding code problems in multiple programming languages. Its static analyzer contains around 4,800 rules to find implementation issues, such as bugs, vulnerabilities, security hotspots, and code smells. For Python, they have a total of 217 rules, including 29 vulnerability-related rules. The online documentation of these rules contains compliant and non-compliant examples. Thus, we retrieved **34** samples of non-compliant examples from it.
- **Pearce *et al.* [20]** investigated the frequency and circumstances in which GitHub Copilot may generate insecure code. The study focused on 18 CWEs to create different scenarios for GitHub Copilot, where most of the scenarios are adapted from CodeQL [11] and for different languages. We included **4** of their Python examples in our dataset.

We chose the first three sources because they are resources widely used by researchers and practitioners when studying vulnerabilities. Furthermore, we included samples by Pearce *et al.* [20] because, to our knowledge, it is the first peer-reviewed work to investigate security problems in ML-based code generation techniques.

After collecting the samples above, we obtained a total of 85 samples. Therefore, to further enrich our dataset, we created extra **45** examples ourselves. Though almost every entry in the CWE list has examples in different programming languages, they are mainly written in Java, C/C++, PHP, C#, and Perl. Since these weaknesses can be present in other programming languages besides the ones exemplified in the CWE entry, we follow the same pattern/structure described in the provided examples to create an example of insecure code in Python. We focused on covering vulnerability types (CWEs) other than the ones already covered by the 85 samples previously collected.

2.2 Samples Formatting

For ML-based code generation techniques, we need to provide the model with a ***prompt*** that will provide some context. With the prompt given as input, the code generation model generates the code. Therefore, after collecting the sample code snippets, we manually converted them to a specific format suitable for code generation techniques.

Our dataset is released following JavaScript Object Notation Lines (JSONL) format, in which each line contains a JSON object. The JSON object has three key-value pairs:

- **ID:** It identifies the sample uniquely. The ID is formatted in this way: `{CWE-ID}_{Source}_{Serial}.py`. The *CWE-ID* indicates which CWE this sample is related to. *Source* indicates from where the sample was extracted (*i.e.*, CodeQL [11], the CWE list [17], Sonar Rules [23], Pearce *et al.* [20] or created by the authors). *Serial* is a numerical value starting from 1. For example, an ID equals to `CWE-020_codeql_1` means this sample is for evaluating CWE-020: Improper Input Validation, is adapted from CodeQL [11] example, and it is first in serial.
- **Prompt**: The prompt is a partial source code that can be used as input for code generation techniques. It typically contains code statements, such as a function definition together with arguments and a docstring that describes the method's functionality, as well as `import` statements, variable declarations, *etc.*
- **Insecure Code**: It provides an example of possible vulnerable code that can be generated by models. The insecure code outlines how the prompt can lead to vulnerable code. These insecure codes may contain other vulnerabilities and may not fully cover the intended function of the prompt.

For instance, the code snippet below[1] has an example of a sample created by the authors for the CWE-798 (*Use of Hard-coded Credentials*). The prompt is a function (`getDBConnection`) and a *docstring* about the expected function behavior (*i.e.*, to *"return a mysql database connection"*). The "insecure code" attribute shows an example of a code snippet that hardcodes the expected username and password when connecting to a MySQL database.

```
{"ID": "CWE-798_author_1.py",
 "Prompt": "import mysql.connector
   def getDBConnection():
       '''
           return a mysql database connection
       '''",
 "Insecure_code": "import mysql.connector
   def getDBConnection():
       '''
           return a mysql database connection
       '''
       return mysql.connector.connect(user='user', password='password',
                    host='host', database='database')"}
```

2.3 Dataset Overview

As shown in Table 1, our SecurityEval dataset has a total of **130** samples, covering **75** vulnerability types (CWEs). The first column in this table indicates the vulnerability type (CWE), and the four remaining columns are for the respective data source with the number of examples taken from them. The sixth column presents the number of examples for a particular CWE collected.

According to the CWE list version 4.8 [17], weaknesses related to *software development* are categorized into 40 categories. We cover 28 categories out of these 40 categories. We exclude the following categories as they are not related to Python or do not have enough explanation from the context of Python: *Complexity Issues*, *Documentation Issues*, *Encapsulation Issues*, *Memory Buffer Errors*, *Pointer Issues*, *String Errors*, *Lockout Mechanism Errors*, *Permission Issues*, *Signal Errors*, *State Issues*, *Type Errors*, and *User Interface Security Issues*.

3 APPLICATION

Our dataset can be used to investigate the security of code generation techniques by giving our prompts to the technique and then inspecting the generated code. This inspection can be performed ***manually*** or ***automatically***. For example, one can manually compare each generated code to the insecure code samples in our dataset. Alternatively, a researcher can rely on existing static analyzers (*e.g.*, Bandit) to automatically find vulnerabilities in the generated code and then rely on the alarms raised by the tool. If

[1] We added indentation to this snippet for clarity. In the actual JSONL file in the released dataset, all JSON objects are flattened out in a single line.

Table 1: Overview of our SecurityEval Dataset

Vulnerability Type (CWE)	Code QL	CWE List	Sonar Rules	Pearce *et al.*	Authors	Total
CWE-020 Improper Input Validation	4	0	0	0	2	6
CWE-611 Improper Restriction of XML External Entity Reference	1	0	4	0	1	6
CWE-601 Open Redirect	1	0	4	0	0	5
CWE-022 Path Traversal	2	0	0	0	2	4
CWE-297 Improper Validation of Certificate with Host Mismatch	0	0	4	0	0	4
CWE-327 Use of a Broken or Risky Cryptographic Algorithm	4	0	0	0	0	4
CWE-502 Deserialization of Untrusted Data	1	1	1	0	1	4
CWE-079 Cross-site Scripting	2	0	1	0	0	3
CWE-094 Code Injection	1	0	1	0	1	3
CWE-117 Improper Output Neutralization for Logs	1	0	1	0	1	3
CWE-295 Improper Certificate Validation	1	0	0	0	2	3
CWE-347 Improper Verification of Cryptographic Signature	0	0	3	0	0	3
CWE-703 Improper Check or Handling of Exceptional Conditions	0	0	0	0	3	3
CWE-730 Regex Injection	2	0	0	0	1	3
CWE-078 OS Injection	1	0	0	0	1	2
CWE-089 SQL Injection	1	0	0	0	1	2
CWE-090 LDAP Injection	2	0	0	0	0	2
CWE-113 HTTP Response Splitting	0	0	2	0	0	2
CWE-116 Improper Encoding or Escaping of Output	1	0	0	0	1	2
CWE-215 Insertion of Sensitive Info. Into Debugging Code	1	0	0	0	1	2
CWE-259 Use of Hard-coded Password	0	0	0	0	2	2
CWE-319 Cleartext Transmission of Sensitive Information	0	0	0	0	2	2
CWE-326 Inadequate Encryption Strength	0	0	0	0	2	2
CWE-434 Unrestricted Upload of File with Dangerous Type	0	0	0	2	0	2
CWE-521 Weak Password Requirements	0	0	2	0	0	2
CWE-522 Insufficiently Protected Credentials	0	0	0	1	1	2
CWE-643 XPath Injection	1	0	1	0	0	2
CWE-798 Use of Hard-coded Credentials	1	0	0	0	1	2
CWE-918 Server-Side Request Forgery (SSRF)	2	0	0	0	0	2
CWE-080 Basic XSS	0	0	0	0	1	1
CWE-095 Eval Injection	0	0	0	0	1	1
CWE-099 Resource Injection	0	0	1	0	0	1
CWE-1204 Generation of Weak Initialization Vector (IV)	0	0	1	0	0	1
CWE-193 Off-by-one Error	0	0	0	0	1	1
CWE-200 Exposure of Sensitive Info. to an Unauthorized Actor	0	0	0	0	1	1
CWE-209 Generation of Error Msg. Containing Sensitive Info.	1	0	0	0	0	1
CWE-250 Execution with Unnecessary Privileges	0	1	0	0	0	1
CWE-252 Unchecked Return Value	0	0	0	0	1	1
CWE-269 Improper Privilege Management	0	1	0	0	0	1
CWE-283 Unverified Ownership	0	1	0	0	0	1
CWE-284 Improper Access Control	0	0	0	0	1	1
CWE-285 Improper Authorization	1	0	0	0	0	1
CWE-306 Missing Authentication for Critical Function	0	0	0	1	0	1
CWE-312 Cleartext Storage of Sensitive Information	1	0	0	0	0	1
CWE-321 Use of Hard-coded Cryptographic Key	0	0	0	0	1	1
CWE-329 Generation of Predictable IV with CBC Mode	0	0	1	0	0	1
CWE-330 Use of Insufficiently Random Values	0	0	0	0	1	1
CWE-331 Insufficient Entropy	0	0	0	0	1	1
CWE-339 Small Seed Space in PRNG	0	1	0	0	0	1
CWE-352 Cross-Site Request Forgery (CSRF)	1	0	0	0	0	1
CWE-367 Time-of-check Time-of-use (TOCTOU) Race Condition	0	0	0	0	1	1
CWE-377 Insecure Temporary File	1	0	0	0	0	1
CWE-379 Creation of Temporary File in Directory with Incorrect Permissions	0	0	1	0	0	1
CWE-384 Session Fixation	0	0	1	0	0	1
CWE-385 Covert Timing Channel	0	1	0	0	0	1
CWE-400 Uncontrolled Resource Consumption	0	0	1	0	0	1
CWE-406 Insufficient Control of Network Message Volume	0	1	0	0	0	1
CWE-414 Missing Lock Check	0	0	0	0	1	1
CWE-425 Direct Request ('Forced Browsing')	0	0	0	0	1	1
CWE-454 External Initialization of Trusted Vars or Data Stores	0	0	0	0	1	1
CWE-462 Duplicate Key in Associative List	0	1	0	0	0	1
CWE-477 Use of Obsolete Function	0	0	0	0	1	1
CWE-488 Exposure of Data Element to Wrong Session	0	0	0	0	1	1
CWE-595 Comparison of Object References Instead of Object Contents	0	0	0	0	1	1
CWE-605 Multiple Binds to the Same Port	0	0	0	0	1	1
CWE-641 Improper Restriction of Names for Files and Other Resources	0	0	1	0	0	1
CWE-732 Incorrect Permission Assignment for Critical Resource	0	0	0	0	1	1
CWE-759 Use of a One-Way Hash without a Salt	0	1	0	0	0	1
CWE-760 Use of a One-Way Hash with a Predictable Salt	0	0	1	0	0	1
CWE-776 XML Entity Expansion	1	0	0	0	0	1
CWE-827 Improper Control of Document Type Definition	0	0	1	0	0	1
CWE-835 Infinite Loop	0	0	0	0	1	1
CWE-841 Improper Enforcement of Behavioral Workflow	0	1	0	0	0	1
CWE-941 Incorrectly Specified Destination in a Comm. Channel	0	1	0	0	0	1
CWE-943 Improper Neutralization of Special Elements in Data Query Logic	0	0	1	0	0	1

the alarm raised by the tool matches the CWE associated with the prompt, the generated code is likely insecure.

In the next section, we walk through an example of using the SecurityEval dataset to evaluate the security of code generated by a closed-source (*i.e.*, GitHub Copilot) and an open-source (*i.e.*, InCoder) code generation tool. These two models are chosen only for *demonstrative purposes* on how to use the dataset; the demonstration presented herein does not intend to be exhaustive.

3.1 Example: Using SecurityEval to Evaluate GitHub Copilot and InCoder

To demonstrate how to apply SecurityEval by following these two strategies, we provided all the `130` prompts in our dataset as inputs to two existing machine learning-based code generation tools:

- **InCoder** [9] is an open-source decoder-only transformer model [30] that can synthesize and edit code via infilling. We used the demo of the 6.7B parameter model available on Huggingface[2], where the number of tokens to generate is `128`, the temperature is `0.6` (default value)[3]. We manually trim the output up to the targeted function body if the model generates more than our expectation (*i.e.*, generating code after completing the function body). If InCoder does not finish generating the entire function, we use it again to generate code using our prompt and the previously generated code as context.
- **GitHub Copilot** [13] is a closed-source model behind a paywall from GitHub. The OpenAI Codex [6], an artificial intelligence model produced by OpenAI[4], powers GitHub Copilot. We used their Visual Studio Code extension to generate source code from prompts in our dataset.

Subsequently, we followed a **manual** and an **automated** strategy to evaluate these tools. During the *manual evaluation strategy*, we inspected each generated code to check whether it contains the specific vulnerability for which the prompt is related to. During

[2] https://huggingface.co/spaces/facebook/incoder-demo

[3] Temperature is a hyperparameter related to the probability of the model's output. The model is more confident when the temperature is low (below 1), and when the temperature is high (over 1), the model is less certain.

[4] https://openai.com

the *automated evaluation strategy*, we analyzed the generated code using CodeQL [11] and Bandit [7], two static analyzers that can detect vulnerabilities and/or security smells. Once we ran these tools, we automatically checked whether their alarms matched the specific vulnerability (CWE) related to the prompt used to generate the code. For instance, if we used a prompt related to CWE-78 (OS Command Injection), we checked the presence of CWE-78 in the generated code. Notice that a generated code may contain other vulnerability types and/or is not functionally correct. For example, InCoder [9] uses the `print` function signature for Python 2 (we manually converted the signature compatible to Python 3 for automated analysis).

Table 2: Evaluating InCoder [9] and GitHub Copilot [13] using SecurityEval

Model	CodeQL	Bandit	Manual
InCoder [9]	20 (15.38%)	12 (9.23%)	88 (67.69%)
GitHub Copilot [13]	24 (18.46%)	14 (10.77%)	96 (73.84%)

Table 2 presents the number of generated code snippets deemed vulnerable by relying on a manual or automated strategy. The numbers in the *CodeQL* and *Bandit* columns count the number of times in which a sample (associated with a specific CWE) was marked the generated code for that particular CWE (*automated strategy*). The *Manual* column contains the number of vulnerable generated codes after manually going through all the generated output and checking if the generated code contains the specific vulnerability (*i.e.*, the designated CWE for the sample.)

From these results, we observe that most generated code snippets contain insecure code (about 68% and 74% of code generated by InCoder and Copilot, respectively), which highlights the importance of evaluating generated code with respect to security concerns and not only functionality. Moreover, although an automated strategy decreases the time and effort in evaluating tools, they may not find all insecure code instances. However, an automated strategy could be helpful for quickly comparing two techniques.

4 THREATS TO VALIDITY

One threat to our work is the sources of samples. We consider four external sources for mining vulnerability examples to create the dataset in our work. We took the examples from the sources and modified them according to our task. CWE list [17] and CodeQL [11] are community-based and open-source project to enumerate common security vulnerability and detects them. Sonar Rules [23] from SonarSource provides documentation about the definition and rules for their static analyzers. Pearce *et al.* [20] is a peer-reviewed work. Though these external sources may introduce threats to our work, they are community-focused and widely used tools and sources for examples and definitions of common security weaknesses.

We used GitHub Copilot [13] as a black box tool for generating source code. We also used the demonstration hosted on Huggingface for InCoder [9] instead of directly using the code for inference. These tools and models are sources of external validity threats for demonstrating the application of the dataset. Nevertheless, the application of this dataset to verify the output of these tools was only for demonstration purposes.

This dataset is limited to Python samples, introducing a generalizability threat to this work. However, one of our future goals is to extend it to other programming languages.

Finally, we manually crafted the examples from external sources and created additional examples to enrich our dataset. In addition, we manually checked the output from the model and tool after using our dataset by generating source code. These processes introduce internal threats to validity.

5 RELATED WORK

Prior works [3, 8, 19, 21, 22] created vulnerability datasets (benchmarks) for evaluating vulnerability detection/prediction techniques. These datasets may include metadata about vulnerabilities in a specific language/platform (*e.g.*, C/++ [8], Java [21], Android [3], *etc.*), their vulnerability types (CWE), and associated patches. Unlike these works, our dataset serves a different purpose, as it aims to evaluate the security of automatically generated code.

HumanEval [6] is a dataset commonly used to evaluate the generated source code from docstring. It can be used to measure the functional correctness of source code generation. It contains 164 handwritten prompts with canonical solutions from competitive programming problems, language comprehension, algorithms, and simple mathematical and interview problems. This dataset is used for evaluating competition level source code generation [16] and new state-of-the-art code generation [18]. However, it does not focus on the security aspect of the generated code. Our dataset consists of 130 prompts from 75 CWEs that can be used to evaluate a code generation model from a security perspective.

Pearce *et al.* [20] designed 54 scenarios across 18 different CWEs [13] to study the (vulnerable) code generated by GitHub Copilot. These scenarios focus on GitHub Copilot, whereas our dataset is a generalized one to use for any context-based source code generation model and tool. Our dataset is also rich with examples from 75 CWEs with 130 scenarios.

6 CONCLUSION & FUTURE WORK

Although a code generation model can help software engineers to develop software quickly, the generated code can contain security flaws. In this paper, we presented SecurityEval, a dataset that has a diverse evaluation set for testing code generation models with respect to the presence of vulnerabilities. Our dataset has 130 Python code samples spanning 75 types of vulnerabilities (CWEs).

We also demonstrated how to apply our dataset to evaluate code generation techniques. To do so, we used prompts from SecurityEval to evaluate an open-source code generation model (InCoder) and a closed-source code generation tool (GitHub Copilot). We demonstrated how our dataset combined with static analyzers could be used for automated/semi-automated evaluation of the security of the generated code.

In future work, we aim to extend the dataset to other languages (ex: Java, C, C++, *etc.*). Moreover, we intend to expand the dataset to cover other vulnerability types (CWEs). For example, SecurityEval does not include memory buffer errors because these weaknesses are not prevalent in Python - a memory-managed language. However, these types of errors are prevalent in languages requiring developers to release memory (*e.g.*, C/C++) manually.

REFERENCES

[1] 2022. Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021 [Online; accessed 28. Aug. 2022].

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, UK) *(PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[5] Stephen Cass. 2022. Top Programming Languages 2022. *IEEE Spectrum* (Aug. 2022). https://spectrum.ieee.org/top-programming-languages-2022

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[7] Bandit Developers. 2022. Bandit. https://bandit.readthedocs.io/en/latest/

[8] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. https://doi.org/10.48550/arXiv.2204.05999

[10] Yuexiu Gao and Chen Lyu. 2022. M2TS: Multi-Scale Multi-Modal Approach Based on Transformer for Source Code Summarization. *arXiv preprint arXiv:2203.09707* (2022).

[11] GitHub. 2022. CodeQL. https://github.com/github/codeql

[12] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering* (2022).

[13] GitHub Inc. 2022. GitHub Copilot : Your AI pair programmer. https://copilot.github.com

[14] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token Code Completion by Jointly Learning from Structure and Naming Sequences. In *44th International Conference on Software Engineering (ICSE)*.

[15] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, et al. 2022. Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814

[17] The MITRE Corporation (MITRE). 2022. Common Weakness Enumeration. https://cwe.mitre.org/

[18] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. https://doi.org/10.48550/arXiv.2203.13474

[19] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1565–1569.

[20] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 980–994. https://doi.org/10.1109/SP46214.2022.00057

[21] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 383–387.

[22] Sofia Reis and Rui Abreu. 2021. A ground-truth dataset of real security patches. *arXiv preprint arXiv:2110.09635* (2021).

[23] SonarSource S.A. 2022. SonarSource static code analysis. https://rules.sonarsource.com

[24] Joanna CS Santos, Anthony Peruma, Mehdi Mirakhorli, Matthias Galstery, Jairo Veloz Vidal, and Adriana Sejfia. 2017. Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 69–78.

[25] Joanna CS Santos, Katy Tarrit, Adriana Sejfia, Mehdi Mirakhorli, and Matthias Galster. 2019. An empirical study of tactical vulnerabilities. *Journal of Systems and Software* 149 (2019), 263–284.

[26] Mohammed Latif Siddiq, Shafayat Hossain Majumder, Maisha Rahman Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)* (Limassol, Cyprus). IEEE.

[27] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[28] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[29] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 329–340.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. https://doi.org/10.48550/ARXIV.1706.03762

Author Index

Ali Babar, Muhammad 1
Codabux, Zadia 16
Mirakhorli, Mehdi 21
Østvold, Bjarte M. 7
Oishwee, Sahrima Jannat 16
Rahimi, Mona 2
Santos, Joanna C. S. 21, 29
Shimmi, Samiha 2
Siddiq, Mohammed Latif 29
Stakhanova, Natalia 16
Tang, Feiyang 7
Zhang, Xueling 21