Exploring the Correlation between Technical Debt Factors and Software Security in Open Source Projects

Thesis · February 2025

DOI: 10.13140/RG.2.2.30561.42087

CITATION
1

READS
14

1 author:

George David Apostolidis
Centre for Research and Technology Hellas
5 PUBLICATIONS **1** CITATION

SEE PROFILE

Exploring the Correlation between Technical Debt Factors and Software Security in Open Source Projects

By

George david apostolidis

February 2025

Supervisor
Alexander Chatzigeorgiou

ABSTRACT

Technical debt, caused by inadequate design or coding decisions made for expediency, poses significant risks to software quality and security. To investigate the relationship between Technical Debt and Security, the study examined open-source projects with specialized tools. Statistical analyses were performed at both project and file levels, time-series data were reviewed, and individual code commits were assessed. The findings show that higher levels of technical debt are associated with weak security performance. Projects with more unsolved security issues and larger class-level codebases are more likely to face quality and security challenges. Over time, improvements in code quality were discovered to improve security outcomes, emphasizing the significance of managing technical debt in order to enhance software security and reliability.

ΠΕΡΙΛΗΨΗ

Το τεχνικό χρέος, που προκαλείται από ανεπαρκείς αποφάσεις σχεδιασμού ή προγραμματισμού που λαμβάνονται για λόγους σκοπιμότητας, ενέχει σημαντικούς κινδύνους για την ποιότητα και την ασφάλεια του λογισμικού. Για να διερευνηθεί η σχέση μεταξύ του τεχνικού χρέους και της ασφάλειας, η μελέτη εξέτασε έργα ανοικτού κώδικα με εξειδικευμένα εργαλεία. Πραγματοποιήθηκαν στατιστικές αναλύσεις τόσο σε επίπεδο έργου όσο και σε επίπεδο αρχείων, εξετάστηκαν δεδομένα χρονοσειρών και αξιολογήθηκαν μεμονωμένες δεσμεύσεις κώδικα. Τα ευρήματα δείχνουν ότι τα υψηλότερα επίπεδα τεχνικού χρέους συνδέονται με αδύναμες επιδόσεις ασφάλειας. Έργα με περισσότερα άλυτα ζητήματα ασφάλειας και μεγαλύτερες βάσεις κώδικα σε επίπεδο κλάσης είναι πιο πιθανό να αντιμετωπίσουν προκλήσεις ποιότητας και ασφάλειας. Με την πάροδο του χρόνου, ανακαλύφθηκε ότι οι βελτιώσεις στην ποιότητα του κώδικα βελτιώνουν τα αποτελέσματα της ασφάλειας, τονίζοντας τη σημασία της διαχείρισης του τεχνικού χρέους προκειμένου να ενισχυθεί η ασφάλεια και η αξιοπιστία του λογισμικού.

ACKNOWLEDGMENTS

I'm thrilled to present this thesis, a journey made possible by the support and encouragement of many individuals. My deepest gratitude goes to Professor Alexander Chatzigeorgiou, who believed in me and granted me the creative research freedom to pursue this project. His guidance and motivation were truly extraordinary, pushing me to achieve more than I thought possible.

I also want to express my heartfelt thanks to Miltiadis Siavvas and Dimitris Tsoukalas, whose unwavering support and expertise were the foundation of this thesis. Their supervision and technical insights were indispensable, and I couldn't have accomplished this without them.

I am deeply grateful to Ilias Kalouptsoglou for his role as a motivator and for his creative contributions, which added a unique dimension to this project.

A heartfelt thank you to my family for their unwavering emotional support, which kept me grounded throughout this journey.

I am also grateful to fellow students George Fakidis, Archontis Kostis, and Apostolos Chalis for their camaraderie and shared insights, which enriched this experience. Finally, I want to thank my friends for their understanding and encouragement, which helped me stay motivated and focused.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Rational

Open source projects are vital in today's fast-paced software development environment because of their collaborative nature and widespread adoption. However, these initiatives frequently generate technical debt—a trade-off chosen to hasten development—which can lead to major maintenance issues in the future. The presence of technical debt has far-reaching consequences for software security, as unresolved debt can introduce vulnerabilities and raise the likelihood of breaches. Poor security measures, on the other hand, can compound technological debt, putting system integrity at risk. Previous studies [1] and [2] established a significant link between security indices and technical debt, highlighting the significant impact technical debt may have on software security.

Building on that framework, the purpose of this study is to investigate in greater detail how technical debt factors and software security in open-source projects are related. The study looks at how technical debt affects security as well as how these two aspects interact in order to offer useful insights into how they work together. For developers, project managers, and the open-source community, this knowledge is essential because it allows them to prioritize initiatives that enhance software security and resilience in addition to streamlining development procedures.

1.2 Objectives

The purpose of this study is to look into how certain technical debt variables affect software security. Although technical debt is frequently linked to decreased scalability and maintainability, it also exposes undiscovered or growing security flaws. This study aims to determine the relationship between specific features of technical debt and the development of security threats by examining empirical data and case studies from various open-source projects. The results are intended to draw attention to crucial areas for intervention in order to improve the security resilience of open-source ecosystems. Additionally, the study will investigate the complex connection between software security and technical debt and will attempt to identify trends and connections relating technical debt measurements to security outcomes by utilizing statistical models and data-driven methodologies.

1.3 Structure of the Study

This study aims to provide a thorough investigation into the relationship between technical debt issues and software security in open source projects. It starts with an introduction that defines technical debt, describes its different aspects, and discusses the consequences for software security in the context of open-source development. The following sections, "Theoretical Background" and "Related Work" critically evaluate previous research and frameworks on technical debt, software security, and its interaction in open-source ecosystems. This review provides the theoretical underpinning for the analysis that follows.

The following section, "Research Methodology," describes the research technique used, including data gathering methodologies, open-source project selection criteria, and metrics for measuring technical debt and software security. This section also describes the statistical

approaches and models used to analyze the data and determine correlations between technical debt indicators and security threats. The "Results" section next offers the empirical investigation's findings, highlighting major insights gained from the research of technical debt dimensions and their impact on software security.

Finally, the "Conclusion" summarizes the study's key findings, reinforces its contributions to the field, and offers possibilities for future research. By conducting the study in this manner, the research aims to carefully and comprehensively investigate the complicated relationship between technical debt and software security, giving useful insights to improve decision-making and strengthen the resilience of open-source software development techniques.

Chapter 2

Theoretical Background

2.1 Software Quality

Software quality is a critical component of software engineering, determining the success and dependability of software products. Several factors influence software quality, including development procedures, product performance, maintainability, and customer satisfaction. A thorough understanding of these dimensions is required for developing reliable and maintainable software systems.

It is challenging to accurately determine software quality. Kitchenham [3] states that quality is "hard to define, impossible to measure, and easy to recognize." Similarly, Gillies [4] states that "quality is generally transparent when present, but easily recognized in its absence". When software quality is visible, it is demonstrated by the software's performance and ability to meet requirements. In contrast, the absence of quality becomes obvious when a software fails to fulfill expectations, contains errors, or becomes difficult to maintain.

Maintainability is an important aspect of software quality, described as "the extent to which software is capable of being changed after deployment" [5]. This adaptability is critical for correcting issues that were not detected during testing, enhancing performance, and adapting to changing needs. The concept of Clean Code, established by software engineer Robert C. Martin [6], is an important method for improving maintainability. Clean Code

promotes a set of ideas and practices for producing code that is clear, manageable, and efficient, resulting in smoother post-deployment changes.

As software systems expand and evolve, they frequently accumulate Technical Debt, which can impede further progress. To mitigate this, writing "clean" new code is critical. This method reduces technical debt and prevents software degradation over time [7]. Developers may ensure that their software is durable and adaptive to change by prioritizing maintainability and using clean coding principles.

Product Quality Model (ISO/IEC 25010)

The ISO/IEC 25010 standard [8], often known as the Software Quality Model, is an important framework for evaluating and improving software quality. This standard outlines critical quality attributes that should be addressed during software evaluation, providing a structured approach for developers and stakeholders to make informed decisions across the development and deployment phases.

The method emphasizes a variety of non-functional requirements, often known as software quality attributes, that have a direct impact on the overall quality of a product. These characteristics, such as compatibility, usability, and dependability, are usually considered and thoroughly examined throughout the architectural design phase. The ISO/IEC 25010 model is intended to assess "the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value." This makes the model essential for ensuring that software not only meets functional needs, but also meets broader quality expectations that influence user experience and satisfaction.

The product quality model identifies nine important quality aspects needed to evaluate and improve software solutions. Functional suitability assesses whether software fits

SOFTWARE PRODUCT QUALITY								
FUNCTIONAL SUITABILITY	PERFORMANCE EFFICIENCY	COMPATIBILITY	INTERACTION CAPABILITY	RELIABILITY	SECURITY	MAINTAINABILITY	FLEXIBILITY	SAFETY
FUNCTIONAL COMPLETENESS FUNCTIONAL CORRECTNESS FUNCTIONAL APPROPRIATENESS iso25000.com	TIME BEHAVIOUR RESOURCE UTILIZATION CAPACITY	CO-EXISTENCE INTEROPERABILITY	APPROPRIATENESS RECOGNIZABILITY LEARNABILITY OPERABILITY USER ERROR PROTECTION USER ENGAGEMENT INCLUSIVITY USER ASSISTANCE SELF-DESCRIPTIVENESS	FAULTLESSNESS AVAILABILITY FAULT TOLERANCE RECOVERABILITY	CONFIDENTIALITY INTEGRITY NON-REPUDIATION ACCOUNTABILITY AUTHENTICITY RESISTANCE	MODULARITY REUSABILITY ANALYSABILITY MODIFIABILITY TESTABILITY	ADAPTABILITY SCALABILITY INSTALLABILITY REPLACEABILITY	OPERATIONAL CONSTRAINT RISK IDENTIFICATION FAIL SAFE HAZARD WARNING SAFE INTEGRATION

Figure 2.1: ISO/IEC 25010

both stated and implicit criteria, with emphasis on sub-attributes such as functional completeness, accuracy, and appropriateness. Performance efficiency measures how effectively the software uses resources, with a focus on temporal behavior, resource use, and capacity. Compatibility assesses the software's capacity to perform in various hardware or software settings, including sub-attributes such as coexistence and interoperability. Usability investigates how well users may attain their goals, taking into account elements such as learnability, operability, and error protection. Reliability refers to the software's consistency in fulfilling its intended functions over time, which includes fault tolerance and availability. Security provides the security of sensitive information by addressing issues such as confidentiality, integrity, and non-repudiation. Maintainability measures the simplicity of modifying, upgrading, or adapting software, with a focus on sub-attributes such as modularity, reusability, and analyzability. Portability assesses a system's capacity to be transferred between multiple contexts. Interaction capability analyzes a system's ability to exchange information with other systems.

Understanding these quality criteria is essential for creating and sustaining high-quality software systems. However, as software systems expand and change, they frequently gather Technical Debt—the implied cost of additional effort resulting from selecting a quick,

easy solution over a longer-term, more successful strategy. Managing technical debt is essential for ensuring that software is maintainable, dependable, and efficient over time. By proactively addressing technical debt, developers can prevent the progressive decline of software quality, retaining its value to stakeholders and assuring the system's long-term success.

2.1.1 Technical Debt

Ward Cunningham's 1992 paper [9] developed the notion of Technical Debt (TD), which is analogous to financial debt in economics. It is a metaphor for the long-term problems that occur when existing quality concerns in software products are not addressed early enough in the software development lifecycle (SDLC). TD was originally linked with software implementation at the code level [10], but the metaphor has now grown to embrace other phases of the SDLC, including architecture, design, documentation, requirements, and testing. TD, like financial debt, incurs "interest" in the form of higher future expenditures, which are frequently the result of poor design and code quality. Various theories, methodologies, and instruments have been created to enable researchers and practitioners identify, quantify, and payback TD throughout the development process [11].

Managing TD is more challenging than managing financial debt due to uncertainties in software development [12]. Effective management of TD necessitates a thorough understanding of the current situation of Technical Debt Management. TD is famous for its interdisciplinary character, integrating concepts from both software engineering and finance theory [13].

Despite the expanding number of strategies, methods, and tools aimed at controlling TD, these tools have yet to attain the necessary maturity to successfully handle all elements of TD management [11]. Furthermore, there is no globally agreed standard for estimating

and managing TD, making it impossible to determine how well present solutions correspond with TD management activities including identification, measurement, and repayment. As a result, researchers, developers, and managers frequently perceive TD differently, making it difficult to discern between software quality compromises caused by TD and those that are not.

Nonetheless, the evolution of a software system is frequently accompanied by an evolution of its TD. A tool or method capable of assisting software project managers in making decisions under uncertainty by forecasting future TD would be extremely useful. Many researchers have explored areas indirectly related to TD, such as code smells [14], fault-proneness [15], and evolutionary trends [16], but the integration of these information into TD management is still an evolving field.

2.1.2 Technical Debt Tools

Building on the discussion of the issues and current state of TD management, the investigation of TD tools becomes an important priority. These technologies are specifically built to handle the difficulties of detecting, measuring, and managing TD across the software development lifecycle (SDLC). They use a variety of approaches and functions, including static code analysis and sophisticated dashboard solutions that interact with project management systems. Using these tools, software teams can acquire a better understanding of code quality, prioritize debt repayment schemes, and ultimately improve the maintainability and reliability of their software systems. This section examines at popular TD tools and approaches, emphasizing their critical roles in enabling efective Technical Debt Management practices.

The table below provides an overview of numerous TD tools, each using a differ-

ent methodology to measure and quantify TD inside software projects. These technologies provide a variety of approaches to TD management, assisting organizations in identifying, prioritizing, and addressing software quality issues across the whole SDLC.

Name (Release year)	**Type**	**Principal**	**Index**
SonarQube (2007)	Code	Time to remove issues	Cost to develop one line of code × number of lines of code
SQuORE (2010)	Design and code	Time to remove issues	No
CodeMRI (2013)	Design	Not estimated	Interest—not mentioned
Code Inspector (2019)	Architectural, design, and code	Effort needed to avoid high TD	A function of violations, duplications, and readability/maintainability issues
DV8 (2019)	Architectural	Number of affected files and lines of code	Penalties: additional bugs and/or changes in lines of code
SymfonyInsight (2019)	Code	Time to remove issues	Number of issues × time needed to remove the issue

Table 2.1: Characteristics of TD Index (TDI) tools [17]

SonarQube [18] improves code quality by identifying issues that impact maintainability and dependability. It estimates TD by calculating the time required to resolve bugs, allowing teams to prioritize refactoring efforts to reduce maintenance costs and enhance sustainability.

SQuORE [19] addresses design and code concerns to enhance software quality. It provides an overview of TD based on remediation time, which helps teams prioritize efforts to improve maintainability and dependability.

CodeMRI [20] assesses design-related issues, tracking TD, and focusing on architectural flaws and areas for improvement. It aids in the identification of possible difficulties, reducing future maintenance challenges.

Code Inspector [21] quantifies TD by combining metrics like as violations and main-

tainability issues into one measurement. It assists teams in proactively controlling and lowering debt throughout the development process.

DV8 [22] examines architectural concerns by evaluating the impact of TD on affected files and code lines. It assists in the prioritization of upgrades in order to reduce maintenance requirements and assure system stability.

SymfonyInsight [23] assesses code issues to enhance software quality and calculates TD by multiplying found issues by remediation time. This analysis helps to prioritize refactoring in order to improve code maintainability and overall system efficiency.

2.1.3 TD Classifier

While prior tools provide quantitative insights into certain aspects of TD, such as code quality and architectural integrity, as well as overall maintainability and scalability, the TD Classifier improves on these capabilities by categorizing discovered faults.

The TD Classifier [24] uses an accurate approach to create a classification model for high-TD software classes. This methodology uses extensive data gathering and an empirical benchmark [25] to build a "ground truth" for classification. The dataset contains 18,857 Java classes that have been categorized as high-TD or not high-TD, and it includes a variety of measures such as code-related metrics (such as structural features and size) as well as development process metrics. To improve accuracy, the dataset is supplemented with 18 independent variables that represent diverse factors impacting TD. The following phases include painstaking data preparation, which addresses issues like as missing values and class imbalance using techniques such as SMOTE oversampling.

The technique concludes in model selection and evaluation, during which seven classifiers are carefully tested, with Random Forest emerging as the best performer due to its

high F2-measure score and recall rates. The TD Classifier is a web application with a scalable backend administered by RESTful APIs and a Python-based microservice architecture, ensuring accessibility and smooth integration. The React-based frontend of the SDK4ED platform [26] allows for user interaction and visualization of results, facilitating practical adoption and continuing assessment in real-world software development settings.

The TD Classifier is used to calculate high-TD classifications and associated metrics across the organization's projects. These measures are: CBO, WMC, DIT, RFC, LCOM, NPM, Fanin, Fanout, NOC, NCLOC, duplicated lines, duplicated lines density, total lines, high TD classes, and high TD classes density. This tool will play an important role in this study, allowing for the identification and analysis of software classes with significant levels of TD.

2.2 Software Security

Software security is a vital discipline in software engineering that focuses on ensuring that software systems function properly even in the face of hostile attacks. Software security, which first appeared in the early 2000s, addresses widespread vulnerabilities caused by software defects, which can range from implementation errors, such as buffer overflows, to design flaws, such as insufficient error handling. As the complexity and interconnection of software applications increase, the need to incorporate comprehensive security measures into every level of the SDLC becomes more pressing. Proactive strategies, such as secure design principles, comprehensive risk analysis, and rigorous testing methodologies, aim to create software that not only meets functional requirements but also effectively withstands evolving cyber threats [27].

According to Gary McGraw and Sean Barnum's book "Knowledge for software se-

curity" [28], knowledge management plays a crucial part in improving software security methods. The work underlines the scarcity of competent software security practitioners and suggests knowledge management as a strategy for effectively spreading essential insights and best practices. McGraw and Barnum divide security knowledge into principles, guidelines, rules, attack patterns, historical risks, vulnerabilities, and exploits, with each offering distinct viewpoints and approaches for improving software security. This systematic approach makes it easier to apply security expertise throughout the SDLC—from initial requirements and architectural design to testing, deployment, and maintenance. Organizations can strengthen their software against potential vulnerabilities and exploits by incorporating these knowledge frameworks, hence increasing overall cybersecurity resilience and successfully managing risks.

In 2018, McAfee [29] reported that 80 billion malicious scans attack vulnerable systems everyday, compromising 780,000 records through hacking incidents. Despite worrying data, security professionals still struggle to persuade and incentivize developers to prioritize vulnerability detection and mitigation [30]. This demonstrates the cybersecurity community's continued difficulty to adequately communicate the crucial relevance of proactive security measures in software development techniques. Bridging this gap necessitates not just technical solutions, but also cultural and organizational changes that prioritize security from the start of software design and development.

2.2.1 Security Tools

In the context of security tool analysis, a thorough and holistic approach is required to properly address the wide range of difficulties connected with software security. Rule-based tools are an essential component of security analysis, relying on specified rules to detect known vulnerabilities and enforce code standards. These techniques are especially successful

at spotting simple problems; nevertheless, they frequently have shortcomings when dealing with complicated or dynamic threats that do not follow static or rigid patterns.

In contrast, machine learning (ML)-based solutions have the capacity to combat dynamic and sophisticated threats by autonomously learning from massive datasets to discover abnormalities and patterns associated with prospective assaults. The adaptive characteristics of ML-based solutions enable them to change in response to emerging threats, resulting in a more resilient and proactive protection system.

Furthermore, quantitative tools play an important role in the security analysis landscape by providing metrics and measurements for assessing risks and vulnerabilities. These quantitative insights enable security professionals to make informed decisions and prioritize security operations using objective, data-driven judgments. The combination of rule-based, ML-based, and quantitative techniques is a strong and comprehensive method for addressing the numerous challenges of current software security.

Nessus [31] is a popular cybersecurity tool that automates the discovery and remediation of known security flaws. It excels at detecting security flaws that have been discovered either accidentally or deliberately by hacker groups, security corporations, or independent researchers. By quickly discovering these vulnerabilities, Nessus reduces the danger of malicious actors exploiting them. However, while Nessus is praised for its wide capabilities, it creates complexity issues, particularly for new users, due to a lack of detailed installation and usage documentation.

OpenVAS [32] is an open-source vulnerability assessment tool designed to detect and manage security vulnerabilities in IT networks. Its distinctive feature is its capacity to do in-depth scans that identify active services, open ports, and running applications across a variety of computers. OpenVAS is driven by a scan engine that is continuously updated and includes Network Vulnerability Tests (NVTs), which are specialized plugins designed to find

Tool	Type	Description
Nessus	Vulnerability Scanner	Widely used with a broad spectrum of scan options and plugins.
OpenVAS	Vulnerability Scanner	Full-featured, capable of unauthenticated and authenticated testing.
GFI LanGuard	Network Auditing Tool	Provides visibility into network elements, vulnerability assessment, and patch management.
Arachni Scanner	Web Application Security Tool	Open-source Ruby tool for web app security.

Table 2.2: Rule-Based Vulnerability Scanners

vulnerabilities across a variety of operating systems and apps.

GFI LanGuard [33] is a comprehensive network auditing tool known for its extensive patch management capabilities across several OS systems and apps. It is especially useful for conducting comprehensive vulnerability assessments on networks that involve not just traditional hardware but also virtual environments and mobile devices. GFI LanGuard uses well-known security databases, such as OVAL and SANS Top 20, to run in-depth scans of operating systems, virtual settings, and installed programs to find vulnerabilities.

Arachni [34] is an open-source Ruby framework designed for penetration testers to perform extensive scans and evaluations of online applications. What distinguishes Arachni is its adaptive learning capabilities, which allows it to study HTTP replies in real time during scans. This capability enables the tool to navigate and adapt to the dynamic nature

of modern web applications, efficiently handling the challenges provided by non-static web pathways, including their cyclomatic complexities. As a result, Arachni can detect a wider range of potential attack routes than typical web scanners, which may struggle with such complexities.

Devign [35] is a cutting-edge vulnerability identification methodology that encapsulates source code functions into a uniform graph structure using several syntactic and semantic representations. Devign uses this composite graph to successfully learn and discover vulnerable code patterns. This methodology establishes a new benchmark for machine-learning-based vulnerability identification, especially in real-world open-source projects. Building on the success of manually crafted vulnerability patterns using code property graphs, which combine syntactic and dependency semantics, Devign automates the process by utilizing advanced graph neural network (GNN) techniques to identify vulnerable code patterns without the need for manual crafting.

The REVEAL tool [36] aims to improve vulnerability prediction by overcoming limitations in existing systems. It follows a two-step process: feature extraction and training. In the feature extraction phase, REVEAL converts real-world source code into graph embeddings, capturing the code's structural and semantic features. During the training step, a representation learner is used to distinguish between susceptible and non-vulnerable code using the extracted characteristics.

IneVul [37] is a Transformer-based tool for vulnerability prediction, with a particular focus on detecting vulnerable functions and specific lines inside the code. Its performance has been extensively assessed on a huge dataset containing over 188,000 C/C++ functions, and it outperforms current techniques. LineVul improves F1-measure by 160-379 percent for function-level vulnerability predictions and Top-10 Accuracy by 12-25 percent for line-level predictions. Additionally, it reduces the effort required by 29-53% at 20% recall, highlighting

its efficiency. These findings highlight LineVul's excellent precision and granularity, making it an invaluable tool for security analysts who can rapidly and reliably identify susceptible lines of code. This improved accuracy enables more concentrated repair efforts, ultimately contributing to a more secure software development process.

Tool	**Type**	**Description**
Devign	Vulnerability Identification Model	Uses graph neural networks to learn and identify vulnerable code patterns by encoding source-code functions into a joint graph structure.
REVEAL	Vulnerability Prediction Tool	Utilizes graph embeddings and a representation learner to predict vulnerabilities, based on data from Linux Debian Kernel and Chromium.
LineVul	Transformer-based Vulnerability Prediction Tool	Predicts vulnerable functions and lines, demonstrating superior performance over existing methods on a dataset of over 188,000 C/C++ functions.

Table 2.3: Machine Learning (ML)-based Tools

Quantitative methods in software security evaluation include a variety of methodologies for objectively measuring and evaluating the security of software systems. These methods use many metrics and data sources to offer a numerical representation of software security posture [38]. Static analysis, for example, is crucial for finding potential vulnerabilities without executing the code, resulting in a fundamental dataset of security alarms. Furthermore, software metrics, such as complexity metrics, have been useful in anticipat-

ing vulnerabilities by indicating areas prone to security problems [27]. The problem is to cohesively integrate these various measures to generate a complete security score that appropriately reflects the overall resilience of software products. The following section describes an innovative quantitative analysis approach that seeks to refine the aggregation and interpretation of security metrics to boost software security assessment capabilities [39].

2.2.2 Security Assessment Model

Despite the widely acknowledged necessity of quantitative security evaluation in secure software development, current research lacks a feasible paradigm for assessing internal software security risk. A hierarchical security assessment model (SAM) [40] was created to evaluate the internal security level of software products utilizing low-level indicators such as static analysis alarms and software metrics. Following ISO/IEC 25010 criteria, the model systematically integrates these low-level indicators using a series of thresholds and weights to generate a high-level security score that represents the internal security level of the software under consideration.

The model was calibrated with a comprehensive code repository that included 100 popular Java apps from the Maven Repository, as well as knowledge from the Common Weakness Enumeration (CWE). To generate a reliable set of weights that reflect the knowledge expressed by CWE rather than relying solely on expert judgments, a novel weight elicitation approach has been developed, grounded in the Analytic Hierarchy Process (AHP) [41] and the SMARTS/SMARTER [42] decision-making techniques.

The suggested model's effectiveness was evaluated through a series of experiments using 150 prominent open-source Java applications from GitHub and 1200 test cases from the OWASP Benchmark. The results of these studies confirmed the model's capacity to correctly

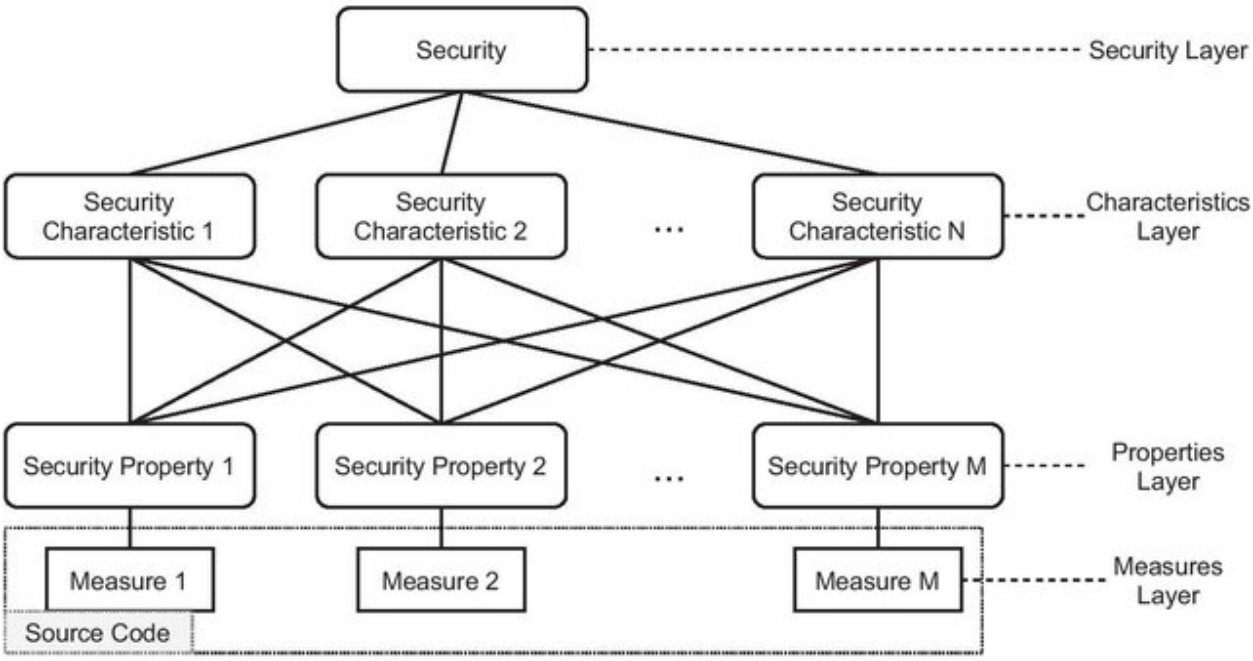

Figure 2.2: The overall structure of the proposed security assessment model (SAM) [40]

reflect the internal security level of software applications at both the project and class levels of granularity, while retaining appropriate discretion power. Preliminary evidence supporting the model's capacity to distinguish between susceptible and secure software components was also shown, indicating that it has the potential to serve as a foundation for vulnerability prediction. To the best of our knowledge, this is the first fully automated and operationalized security assessment model documented in the relevant literature, built and tested on a large volume of empirical data encompassing 250 real-world software applications, totaling approximately 20 million lines of code.

Currently, the proposed SAM use the CKJM Extended tool to compute software metrics that define its metric-based characteristics. Because the CKJM Extended tool operates at the bytecode level, this requirement limits the model's usability by requiring the source code to be compiled before analysis, which can be a time-consuming procedure. Future work aims to make the suggested approach more feasible by replacing the CKJM Extended tool

with alternative open-source static analysis tools that operate directly on software product source code, such as JHawk and OpenStaticAnalyzer.

The current version of the model focuses on three important security characteristics, namely Confidentiality, Integrity, and Availability, which are considered the primary security criteria of information systems [43], also known as the CIA triad [44]. However, new security requirements, such as authenticity and non-repudiation, could be added to improve the model's completeness, allowing more fine-grained evaluations and addressing specific security demands of distinct software.

Low-level security measures, such as static analysis alerts and software metrics, are first derived from the source code. These metrics, along with a set of thresholds, assign ratings to higher-level attributes such as Complexity. These evaluations are then combined using a weighted average to produce ratings for security features such as confidentiality. Finally, these scores are averaged to establish the software project's overall security score, which is expressed as a number between 0 and 1. This combined number, known as the Security Index, provides a comprehensive knowledge of the project's security level, much exceeding the insight provided by individual low-level measurements.

This emphasis on well-supported attributes assures the model's integrity and usefulness to developers and project managers. The aggregated Security Index, built from ratings of basic security features, is critical in giving a full security assessment. The next section looks into how it is calculated and what it means for evaluating software security.

2.2.3 Security Index

The security assessment model's goal is to meaningfully aggregate the code-level measures in order provide a high-level security score that reflects the software product's internal security

level. The Security Index (SI) is a continuous value in the range of 0 to 1. Values close to zero indicate low security, whereas values close to one suggest a high level of security.

The SI is calculated in two phases, similar to previous quality models [45] [46] [47]. In the first stage, the model evaluates the system-level values of the code-level measures used to evaluate specific security features. Each property is awarded a rating or score in the interval $[0, 1]$, indicating how efficiently the relevant security property is met. The second stage uses security property ratings to assess overall security qualities. This stage also results in the final computation of the SI based on the ratings of the provided security attributes.

Initially, the model performs static analysis on the software product's source code to identify potential security issues, which are divided into seven vulnerability areas (NPE, Assignment, Exception Handling, Resource Handling, Logging, Adjustability, Misused Functionality, and Synchronization). It also computes four software metrics (cohesion, complexity, coupling, and encapsulation), which are lower-level measures that must be aggregated and standardized before being used to system-level evaluation.

The Static Analysis Vulnerability Density (SAVD) [48] [49] evaluates the security of software products at the system level. SAVD is derived by dividing the total number of security-related static analysis warnings provided by a static code analyzer for a specific software product by the total lines of code (LOC) and normalizing it per thousand lines of code. For each vulnerability category, the SAVD is calculated using the following formula:

$$\text{SAVD}_i = \frac{1000 \times N_i}{\text{LOC}}$$

where:

- SAVD_i is the static analysis vulnerability density for the i-th vulnerability category,

- N_i is the total number of static analysis alerts for the i-th category,

- LOC represents the total lines of code of the software product.

Using SAVD, the security assessment process can investigate the software product's vulnerability categories, allowing for a more fine-grained security evaluation by determining its susceptibility to certain types of vulnerabilities. This method, used in prior studies [47], provides system-level insights by standardizing SAVD data throughout the entire product.

Proper aggregation and normalization processes are utilized for system-level evaluations. The sum of class-level metric values is weighted by each class's lines of code and divided by the system's total amount of lines of code. This strategy, developed by Wagner [50] and adopted in other quality models [51] & [45], guarantees that system-level metrics accurately represent the overall software product.

Each security property in the model is linked to a code-level measure, and the system-level value is used to assign a score between 0 and 1 using a utility function. The utility function structure converts measure values into property ratings [50]. The utility function is structured as follows:

$$f_p(s) = \begin{cases} 1, & , s \leq t_l \\ \dfrac{0.5}{t_m - t_l}(s + t_l - 2t_m) & , t_l \leq s \leq t_m \\ \dfrac{0.5}{t_u - t_m}(s - 2t_u) & , t_m \leq s \leq t_u \\ 0 & , s \geq t_u \end{cases}$$

where: - s is the system-level value of the corresponding code-level measure,

- t_l, t_m, and t_u are the lower, middle, and upper thresholds, respectively.

The final SI of the software product is computed by averaging the ratings for the three security measures. Given the CIA's importance in information security, their weights in the final assessment are equal. However, the model is very adaptable, allowing users to modify the importance of security attributes simply adjusting the weights in the XML configuration file without having to recalibrate.

It's important to understand that the SI is a relative score. This means that it compares the evaluated software product to a list of well-known software items. A Security Index of 0.8 (five stars) ranks the software among the top 10% of products in terms of security. This allows project managers to analyze the product's security preparation and make informed decisions about its release to market.

Chapter 3

Related Work

This chapter examines at similar studies that investigate the relationship between TD and software security. It focuses on two key approaches: machine learning-based algorithms for forecasting security concerns using TD metrics, and statistical evaluations of the relationship between TD and software vulnerabilities. This chapter will cover different strategies to illustrate the current state of research on how TD might be used to identify security vulnerabilities, as well as the benefits and limitations of various approaches.

3.1 Empirical Evaluation Approach

The study by Siavvas et al. [1] investigates the relationship between TD and software security through a large dataset of real-world software projects. The authors hope to determine whether TD can be used as a reliable indicator of software vulnerabilities, allowing quality managers and engineers to better assess security risks during development. The study studies 50 open-source Java apps to evaluate if TD and security metrics are related, and preliminary findings suggest a significant relationship between these two characteristics.

The authors employed two criteria to investigate the relationship. To start, they used SQALE Density, a statistic from SonarQube [18] that measures the quantity of TD in software applications. The higher the SQALE Density value, the more noticeable the TD, indicating code that will need more maintenance or may cause performance issues. In

essence, a high SQALE Density score suggests that a codebase may need to be addressed due to potential concerns. FindBugs developed the Static Analysis Vulnerability Density (SAVD) measure [52] to assess the number of security vulnerabilities in code. A higher SAVD value suggests a greater number of security flaws or potential dangers. This data is crucial for identifying weaknesses that hostile actors could use to compromise the application's security.

The study's primary purpose was to conduct a thorough comparison of SQALE Density and SAVD, two measurements that evaluate TD and security vulnerabilities, respectively. To achieve this goal, the study thoroughly generated two independent rankings of software packages based on each metric. These rankings were then submitted to a detailed Spearman's Rank Correlation Coefficient analysis, a statistical technique for determining the strength and direction of a monotonic relationship between two variables. By closely analyzing the relationship between these two variables, the study hoped to give valuable insights on the interaction of technical debt and security risks within software systems.

The empirical analysis found a high positive correlation between TD and software security issues. Comparing the SQALE Density rankings (for TD) to the SAVD rankings (for security vulnerabilities) found that apps with higher levels of TD had higher security vulnerability densities. This relationship was statistically significant, indicating that an increase in TD could predict an increase in software vulnerabilities, and vice versa.

The study's weaknesses include a small sample size and insufficient comprehensive analysis of TD and security issues. To address these limits, future research should use a larger code repository and investigate more clear connections between TD elements and security problems.

3.2 Machine Learning Approach

Another study by Siavvas et al. [2] investigates how common TD indications (such as bugs, code smells) can predict security concerns in software, with a focus on project and class-level granularity. The study looked at 210 open-source Java projects, using SonarQube to produce TD indicators and the Static Analysis Vulnerability Density (SAVD) as a security metric. Machine learning models were employed to detect security threats. At the project level, Logistic Regression outperformed Random Forest (F1 score = 70.8%).

At the project level, TD indicators effectively predicted Security Risk Levels (SRLs), and at the class level, the indicators differentiated between vulnerable and clean classes. This demonstrates that TD indicators can be used to detect security hotspots in software, potentially leading software engineers and project managers to prioritize high-TD regions that may contain vulnerabilities. The data support the idea that TD could be used as an indirect indicator of software security.

The authors emphasize the importance of TD in security management and argue for its early introduction into the software development lifecycle. They believe that knowing the relationship between TD and security issues will help companies optimize resource allocation and improve security testing procedures. While the current study focused on Java applications, future research plans to look into other programming languages, commercial software, and new security metrics such as the Attack Surface. The investigation relied on SonarQube for static analysis, but future studies will use a variety of open-source and commercial static analysis tools to see how results change between platforms.

3.3 Other Approaches

Despite substantial research into the various aspects that may signal security vulnerabilities in software systems, very few studies have focused solely on TD indicators such as code smells or code duplication. Prior research has primarily focused on software metrics, which are only tangentially related to TD and have little relationships with software security. However, relying solely on these indirect signs is insufficient to reach broad generalizations about the relationship between TD and security.

In recent years, researchers have begun to look into whether TD can be used to assess software security. Rindell et al. proposed principles for expanding the TD concept to include security [53] [54], while others have investigated prioritizing security problems as TD items while treating them as quality issues [55] [56]. However, these efforts have been primarily theoretical, with little empirical evidence to support a significant relationship between TD and software security.

To summarize, the current state of research shows that the relationship between TD and software security is a rising area of study, with both empirical and theoretical contributions shed light on its significance. While some studies provide preliminary empirical evidence of a significant relationship between TD indicators and security vulnerabilities, others employ machine learning models to forecast security risks using TD metrics. Despite the promising results, research on this topic is still in its early phases, with limitations like as small sample sizes and the need for more diverse tools and datasets. Moving forward, more empirical study and the use of more powerful static analysis methods are required to fully comprehend and leverage TD as a trustworthy indicator of software security risks. In the following session, we will go over the study technique, concentrating on two large datasets that provide insights into both aspects.

Chapter 4

Research Methodology

At the core of this study lies the objective of exploring the relationship between technical debt and software security, uncovering how specific patterns and trends influence security outcomes. This section discusses the research methodologies used to create and analyze the two huge data sets required for this study, each of which provides useful insights into the dimensions under investigation. The technique is organized into four stages to ensure a methodical and detailed approach. First, data generation describes the datasets' sources and characteristics, such as their origin, structure, and data collection methods. Next, statistical analysis outlines the statistical processes used to identify patterns, trends, and relationships in data. Time series analysis then investigates temporal patterns and how trends evolve over time, using time-based data sets to achieve a more dynamic understanding. Commit analysis looks into specific event-based data, such as actions or modifications recorded, to gain a better understanding of behavior patterns and events. This structured approach aims to provide a comprehensive and comparable method for reviewing and interpreting datasets. The third step is class-level analysis, which uses the same statistical analysis as the previous project-level technique, assuring consistency and depth of understanding at a more granular level.

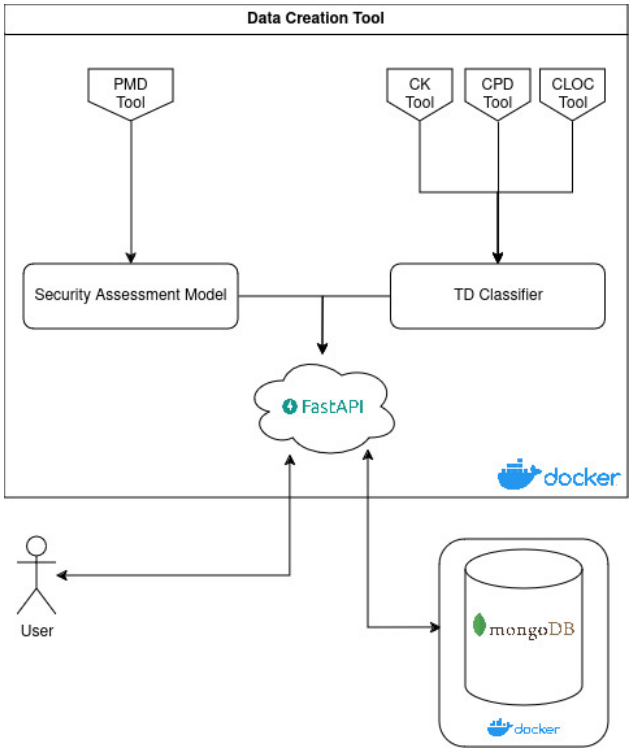

Figure 4.1: Data Creation Tool

4.1 Dataset Creation

The data collection strategy for this study employs an application that combines the TD classifier [57] with an open-source tool based on the SAM model [58] and given over a RESTful API. This solution facilitates rapid and exact data collecting and analysis for GitHub-hosted projects, with a focus on identifying technical debt and software security. The program analyzes repositories based on a 45% Java line of code (LOC) threshold. It can target recent commits, specific code releases, or long-term commit patterns. This systematic approach offers flexibility, scalability, and applicability to a wide range of research objectives, enabling both academic and industrial investigations.

The tool, which is built with Python and FastAPI, is implemented as an interactive web interface that enables several degrees of analysis. Users can start evaluations on in-

dividual projects, batches of repositories, or entire organizations by providing Git URLs, commit SHAs, and optional criteria. After initiating an investigation, the tool conducts a thorough static review, delivering the data to both the TD classifier and the SAM model to uncover technical debt and security-related findings. Results are securely kept in a MongoDB database, allowing for rapid historical data retrieval and analysis without re-execution. In addition, the system uses a containerized design called Docker to enable scalability, stability, and seamless communication between components.

Case Study: Apache Kafka

This is an example of the tool mentioned above, with a focus on Apache Kafka, a well-known project in industry and research, with numerous researchers conducting experiments on it. The analysis focuses on project-level metrics, which provide useful information on the software's development dynamics, code quality, complexity, and security posture. The PC used for the investigation has an Intel 12th Gen i5-12600 processor and 32 GB of RAM, indicating that the analysis can be performed on a typical system. Notably, the entire analysis of the Apache Kafka project for its most recent commit takes around 650 seconds.

Individual project files, such as "DataGenerator.java," provide a more detailed look at several file-level metrics, including Cyclomatic Complexity (CBO), Weighted Method Count (WMC), and the amount of lines of code. For example, "DataGenerator.java" has a CBO of 14, a WMC of 8, and a total of 39 detected concerns in several categories such as resource management and logging. These file-level metrics help comprehend specific portions of the code that may require attention, assuring the project's continuing optimization and security. The research demonstrates the tool's usefulness in detecting complexities, vulnerabilities, and potential improvements across the entire codebase, providing developers with valuable information for future development.

Table 4.1: Apache Kafka Project Level Analysis

Attribute	**Value**
Git URL	https://github.com/apache/kafka
CBO	9
WMC	22
DIT	1
RFC	26
LCOM	74.154
NPM	9
Fan-In	3
Fan-Out	9
NOC	0
NCLOC	1662
Duplicated Lines	0
Duplicated Lines Density	0
Comment Lines	611
Comment Lines Density	0.2297
Total Lines	2660
HTD	0
HTD Density	0
SAM Resource Handling Eval	0.917
SAM Resource Handling Density	0.0116
SAM Assignment Eval	0
SAM Assignment Density	0.2839
SAM Exception Handling Eval	0

Attribute	**Value**
SAM Exception Handling Density	0.0279
SAM Misused Functionality Eval	0
SAM Misused Functionality Density	0.0621
SAM Synchronization Eval	0
SAM Synchronization Density	0.0054
SAM Null Pointer Eval	1
SAM Null Pointer Density	0
SAM Logging Eval	0
SAM Logging Density	0.0202
SAM Cohesion Eval	0.9000
SAM Cohesion Norm	74.1047
SAM Coupling Eval	0.6301
SAM Coupling Norm	9.5400
SAM Complexity Eval	0.5318
SAM Complexity Norm	23.2552
SAM Encapsulation Eval	1
SAM Encapsulation Norm	0
SAM Confidentiality Eval	0.3698
SAM Integrity Eval	0.2570
SAM Availability Eval	0.4706
SAM Total Issues	530
Security Index	0.3658

Table 4.2: Apache Kafka File Level Analysis

Attribute	**Value**
File Path	contrib/hadoop-consumer/src/main/java/ kafka/etl/impl/DataGenerator.java
File Name	DataGenerator.java
Commit SHA	3d93852fda00437d71ab05c06b374adac640451a
CBO	14
WMC	8
DIT	1
RFC	30
LCOM	0
LCOM3	0.55
NPM	3
Total Methods	4
Max Nested Blocks	2
Total Variables	26
Fan-In	1
Fan-Out	14
NOC	0
Duplicated Lines	0
Comment Lines	24
NCLOC	88
Total Lines	134
Resource Handling Issues	2
Assignment Issues	19

Attribute	Value
Exception Handling Issues	5
Misused Functionality Issues	9
Synchronization Issues	0
Null Pointer Issues	0
Logging Issues	4
Total Issues	39
High TD	0
High TD Probability	0

The tables above provide a detailed overview of the Apache Kafka project, highlighting significant project- and file-level metrics that disclose critical information about its code quality, complexity, and security. These findings lay the groundwork for additional investigation, offering a clear picture of possible areas for software quality and security. Building on this, the next section digs into statistical analysis, where we will investigate the links between various metrics, discover trends, and evaluate the importance of these traits in terms of overall program performance and stability. This statistical method seeks to identify deeper correlations and insights that can inform future development strategies and enhance software maintainability.

4.2 Statistical Analysis

This section focuses on two main metrics: High Technical Debt Density (HTD) and Security Index (SI), which will be the core areas of inquiry. HTD is the proportion of projects identified as having High Technical Debt (TD), as assessed by the TD Classifier, divided by the total lines of code in the project. This measure indicates the level of technical debt

in the codebase, which provides insight into the software's quality. The Security Index (SI) examines the software's entire security posture, including its vulnerability profile and potential threats.

We will investigate the correlation between these two metrics, determining whether there is a relationship between code quality, as evaluated by HTD, and software security performance, as reflected by the Security Index. By focusing on these indicators, we want to obtain a better understanding of how technical debt affects software security and if larger levels of technical debt correlate with a poorer security posture. This investigation will provide useful insights into the connection between code quality and security, potentially improving both elements of software development.

4.2.1 Distribution Visualization

The first stage in the process is to visualize the distribution of the two key variables of interest: HTD and the Security Index. This is accomplished by showing their corresponding distributions, which aids in understanding the underlying patterns in the data. The distributions are visually evaluated to see if the data follows a normal distribution or if there is any evident skewness or anomalies. This initial visualization provides insight into the overall structure and properties of the data, which is critical for determining the following steps in the statistical analysis.

4.2.2 Normality Testing

After visualizing the distributions, the next step is to determine whether the data for both variables has a normal distribution. This is accomplished via a statistical test that determines whether the distribution of the data deviates considerably from normality. If the data passes

the normality test, it indicates that the variables follow a bell-shaped curve, allowing for the use of parametric approaches in following analyses. If the data does not have a normal distribution, non-parametric approaches are examined as an alternative. This step ensures that the relevant statistical tests are used for future investigation.

4.2.3 Correlation Analysis & Visualization

Following normality testing, the approach assesses the correlation between HTD and Security Index. Two types of correlation algorithms are used: one that assumes the data is normally distributed, and another that is acceptable for data that does not follow a normal distribution. The correlation coefficients indicate the intensity and direction of the association between two variables. By combining both methodologies, the analysis assures that any potential association is accurately recorded, regardless of how the data is distributed.

The methodology offers a visual representation of the correlations between the chosen variables. A correlation matrix will be constructed to show the link between several variables in the dataset. This visual tool emphasizes the strength and direction of correlations, with stronger links represented by brighter hues. The correlation matrix provides a thorough view of how the variables interact, making it easy to spot any noteworthy associations that warrant further examination.

4.2.4 Descriptive Statistics and Feature Selection

The procedure starts with a basic overview of the data, which is summarized using descriptive statistics to highlight key characteristics such central tendency, spread and distribution. Next, feature selection determines the most relevant predictor variable, with an emphasis on the feature that is most likely to explain the association between HTD and Security Index.

A statistical technique is used to identify the most predictive characteristic, guaranteeing that the analysis prioritizes the most important variable for the next steps.

4.2.5 Data Splitting and Model Training

The next step in this procedure is to divide the dataset into training and test sets. This phase is critical for ensuring that the model may be trained on one subset of the data and evaluated on an unknown fraction, lowering the danger of overfitting. After the data has been separated, two linear regression models are generated. One model attempts to predict the Security Index using HTD, whilst the second model does the opposite, predicting HTD using the Security Index. Both models are then trained with the training data to determine the relationship between the variables.

4.2.6 Model Evaluation and Visualization

Once the models have been trained, the technique evaluates their performance using important measures such as Mean Squared Error (MSE) and R-squared. These metrics demonstrate how well the models predict the target variables and how efficiently they explain the data's variance. The findings of the model evaluation are then utilized to decide which model makes the best predictions. In addition, scatter plots are used to display the predictions, comparing actual data points to anticipated values. This enables a visual evaluation of the models' correctness and the strength of the association between the variables.

4.2.7 Heteroscedasticity Testing

To strengthen the analysis's robustness, the technique adds a test for heteroscedasticity, specifically the Breusch-Pagan test. Heteroscedasticity occurs when the variance of errors in a regression model differs across all levels of the independent variable. Detecting heteroscedasticity is critical since it can influence the reliability of regression results. The Breusch-Pagan test is used on the regression model's residuals to determine whether the data contains this issue. If heteroscedasticity is discovered, changes to the model may be required to provide accurate findings.

4.2.8 Non-linear Correlation Analysis

This section investigates the potential nonlinear connection between the HTD and the Security Index. While linear regression methods presume a straight-line relationship, the variables may have non-linear relationships. To analyze this, different correlation approaches, such as Kendall's Tau, are employed to determine whether a nonlinear link exists. The relationship is also represented using scatter plots and non-linear regression lines, which allows for a better understanding of the complicated dynamics between the variables. This phase guarantees that both linear and nonlinear relationships are accounted for in the study.

4.2.9 Outlier Removal using IQR

This section of the process identifies and removes outliers from the dataset using the Interquartile Range (IQR) method. The IQR is determined as the difference between the third and first quartiles. Outliers are data points that are outside the range of $Q1 - 1.5 \times IQR$ (lower bound) and $Q3 + 1.5 \times IQR$ (upper bound). The method iterates over all numerical

columns in the dataset, removes any outlier values, and then outputs the minimum and maximum values for each column following the outlier removal process.

4.2.10 Correlation Grouping and Categorization

After calculating the correlations, the coefficients are classified according to predetermined thresholds. Correlations more than or equal to 0.5 are classified as high, those between 0.3 and 0.5 as moderate, and those less than 0.2 as low. The correlations are then sorted by absolute value in descending order and kept in dictionaries based on category. These findings are produced, indicating which pairs of variables have a high, moderate, or low correlation, enabling for simple detection of significant associations between variables.

4.2.11 Threshold-Based Classification and Analysis

A classification method was used to divide the dataset into two categories based on the determined mean value of the minimum and maximum Security Index after outlier removal. The classification threshold was established at 0.48, with SI values less than 0.48 designated as "low SI" and those more than or equal to 0.48 as "high SI." This classification was applied to both the HTD Density (HTD) and Non-Comment Lines of Code (NCLOC) metrics for further investigation.

To compare these two groupings, boxplots were created for the HTD variable compared to the SI categorization, giving a visual representation of the distributions within each. Spearman correlation coefficients were also determined for each group individually to analyze the correlation between SI and HTD in the "low SI" and "high SI" categories.

The findings demonstrated how the connections between these variables differed across

the two groups, providing insight into the behavioral dynamics of high and low SI segments. This classification and analysis method provided a structured approach to investigating the interplay of software metrics in various situations of the security index.

The statistical analysis provided useful insights into the relationship between numerous software measures, with a particular emphasis on the interaction between HTD Density (HTD) and the Security Index (SI). A number of rigorous stages, including correlation assessments, outlier handling, and classification-based analysis, revealed that certain patterns and correlations warrant further investigation. However, the static character of these analyses emphasizes the significance of studying how these measurements change over time. To address this, a time series analysis is required, and the results will be provided in the following sections to reveal temporal trends and dynamics in the data.

4.3 Time-Series Analysis

This section describes the approach for conducting a time series analysis on the HTD and Security Index. The major goal is to explore how these measurements evolve over time and identify any temporal patterns or trends between them. For the purposes of this study, we look at the last commit of each year and relate it to the closest tag or release on GitHub. By connecting each year's final commit with a specific release, we ensure that the data is consistent with the project's version history. This enables us to monitor the evolution of both HTD and SI over time, noting how they evolve in the context of each release.

This approach aims to supplement statistical findings by offering a dynamic perspective, allowing for the detection of time-dependent correlations that may provide more light on the interaction between code quality and security posture. Through this strategy, we hope to capture temporal trends in both metrics and gain a better understanding of how

code quality effects security outcomes throughout the course of a project's lifecycle.

4.3.1 Time Series Data Preparation and Stationary Analysis

The time series analysis process begins with data preparation, which involves categorizing indicators such as HTD, SI, and NCLOC by project and year. This approach ensures that the data is arranged for temporal analysis while also collecting yearly trends unique to each project. For stationarity analysis, the Augmented Dickey-Fuller (ADF) test is applied to each project's HTD and SI series to examine whether their statistical features stay consistent over time. Stationarity is assessed by examining ADF test results, which include test statistics, p-values, and critical values.

4.3.2 Spearman Correlation Over Time

To study the temporal link between HTD and SI, Spearman correlation coefficients are calculated for each project. This nonparametric measure sheds light on monotonic correlations between these two metrics while also accounting for potential nonlinear relationships. Correlation data are shown with p-values to demonstrate statistical significance, providing a project-specific insight of how code quality and security metrics may interact over time.

4.3.3 Visualization of Temporal Trends

Each project's HTD, SI, and NCLOC time trends are displayed. Line plots are used to show the annual progression of HTD and SI, allowing for a comparative comparison of these measures across time. A secondary plot displays the NCLOC measure to show increases in code size or complexity. These visualizations are supplemented with markers and gridlines

to improve interpretability, resulting in a full representation of the metrics' time-dependent behavior across projects.

4.3.4 Detrending and Correlation Analysis of Temporal Data

The temporal study investigates the patterns of HTD and SI by grouping data annually by project. A rolling mean is used to extract trends across time, with a three-year window size to smooth out oscillations and show underlying patterns. These trends are then subtracted from the original data to produce a detrended series, which separates short-term fluctuations from long-term changes. The detrended HTD and SI data for each project are analyzed for monotonic relationships using Spearman correlation. This approach captures both linear and nonlinear dependencies, and p-values indicate the statistical significance of the findings. Correlation coefficients provide insight into how changes in code quality (HTD) relate to short-term security measures (SI) across different projects, supplementing trend research.

Each project's detrended series are visualized to show fluctuations in HTD and SI over time. These charts help to analyze the temporal alignment or divergence of the metrics, providing a detailed understanding of how localized changes in code quality may relate to security results. Detrending the data reduces the influence of overarching trends, allowing for a focus on year-to-year fluctuations that could otherwise be buried by long-term shifts. By focusing on these short-term dynamics, the research can provide more detailed insights into how specific changes in HTD affect SI. This method is critical for evaluating the responsiveness of security results to code quality initiatives across shorter timescales, which may be overwhelmed by project growth or macro trends. Based on the insights gained from detrended correlations, this research offers the framework for identifying important periods or project-specific interventions that influence both measures. This insight can help to guide strategic decisions targeted at increasing code quality and security posture in a dynamic

software development environment.

4.3.5 Cross-Correlation Analysis Across Temporal Lags

Each project underwent cross-correlation analysis to investigate the temporal association between HTD and SI. This approach assesses the relationship between the two time series at different time lags, revealing whether changes in one measure occur before, coincide with, or follow changes in the other.

The study begins with sorting the dataset by project and year, then determining the yearly mean for HTD and SI. For each project, cross-correlation values are calculated at lag intervals ranging from -2 to +2 years to account for any lead-lag dynamics within a tolerable temporal window. For each project, bar charts are generated to show the cross-correlation values at the defined lags. These visualizations show the degree and direction of the temporal link at various time points, demonstrating whether HTD changes can forecast, respond to, or move independently of SI adjustments.

The time series analysis emphasizes the relevance of understanding the temporal dynamics of HTD and Security Index. However, in order to properly understand the causes underlying the observed patterns, the data must be examined at a more detailed level. Analyzing activity within each year, particularly at the commit level, will provide a greater understanding of the variations and events that shape these trends, laying the groundwork for the upcoming commit-level analysis.

4.4 Commit Analysis

Although the time series analysis revealed useful insights into patterns and trends, it was deemed necessary to conduct a more in-depth research into the changes that occurred throughout each period, particularly with regard to the most recent commits. To accomplish this, the most recent commits were inspected and compared to the earliest known tag or release for each time period. This method enabled the grouping of commits and their association with specific tags, resulting in a better understanding of the software development context during those periods.

The release notes connected with each tag were then reviewed to determine the changes made at that time. By analyzing the changes between distinct tags, the variations identified in the time series analysis were explained, showing major elements that may have influenced the data patterns. This in-depth examination of the commits resulted in a more complete understanding of how software changes correlated with the observed metrics, allowing for a more accurate interpretation of the time series results and providing valuable insights into the relationship between software development activities and the metrics under consideration.

4.5 Class-level Analysis

The final element of the methodology required investigating the link between the two indexes at the class level, utilizing the tool's capacity to retrieve class-level data. However, it became clear that the Security Index does not exist at the class level because it is fundamentally specified at the project level. To solve this constraint, the analysis concentrated on specific security issues that affect the overall Security Index, such as resource handling, assignment,

exception handling, misused functionality, synchronization, null pointers, and logging. The overall number of issues was given special attention because they are the main contributors to the final Security Index.

For the purposes of this analysis, it is considered that classes with more issues have inferior quality assignment handling, resource management, null pointer handling, and other security-related elements. Regarding quality, the dataset offers two metrics: High TD and High TD Probability. If the latter surpasses 50%, the class is considered as High TD by the classifier, with binary values of 0 (not High TD) and 1 (High TD).

The purpose of this analysis is to evaluate the amount of security issues inside a class to the possibility of being classed as High TD. To do this, a second statistical analysis was conducted using part of the methodology mentioned in Section 4.2. This approach seeks to identify potential links between security issues and technical debt classification at the class level.

With the methodology well stated, the focus now switches to examining the results of its application. Section 5 summarizes the conclusions of the analyses, highlighting major findings and revealing the links between the High Technical Debt Class Density, Security Index, and associated metrics. By combining insights from statistical analysis, time series analysis, commit exploration, and class-level investigations, the results provide a thorough knowledge of how these aspects interact and influence software quality and security.

Chapter 5

Results

This section delves into the findings of the research, revealing key patterns and linkages between High Technical Debt Class Density (HTD), the SI, and their underlying measures. The data are given in a systematic fashion, beginning with statistical correlations to determine the strength and significance of relationships, followed by time series trends that depict the evolution of these measurements over time. Furthermore, a thorough examination of commit histories and class-level properties sheds light on the practical ramifications of these interactions, providing a comprehensive understanding of the relationship between software quality and security.

5.1 Results of the Statistical Analysis

5.1.1 Distribution Visualization & Normality Testing

To begin, normality tests were run to determine the distributional features of the important variables, High Technical Debt Class Density (HTD) and SI. The Shapiro-Wilk test findings along with Figure 5.1 show that neither metric has a normal distribution, with test statistics of 0.7784 (p-value = 0.0000) for HTD and 0.9521 (p-value = 0.0000) for SI. These findings, with p-values less than 0.05, contradict the null hypothesis of normality for both variables. As a result, non-parametric approaches were used in future studies to compensate for the

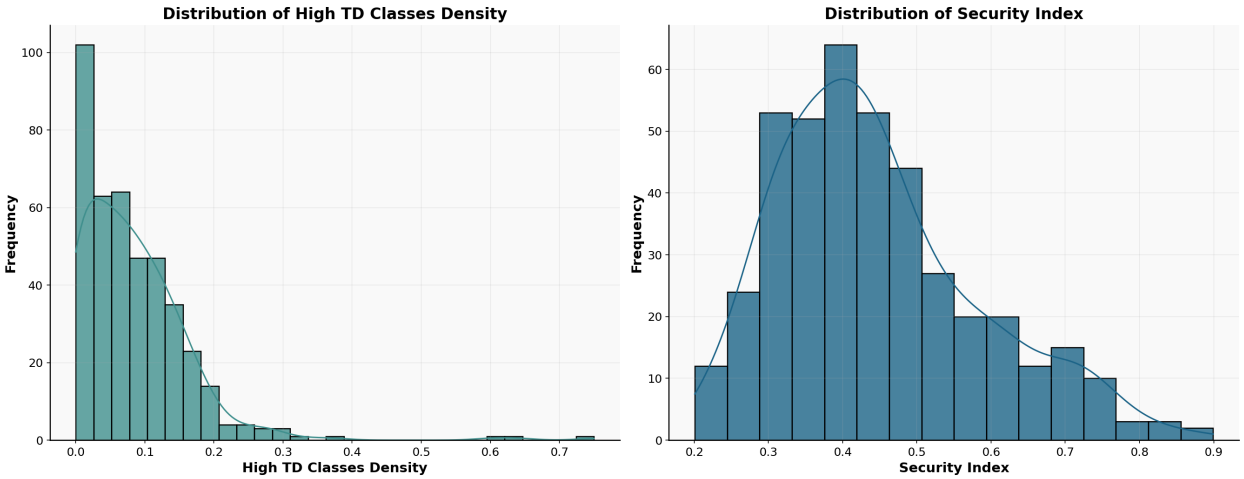

Figure 5.1: Distribution of SI and HTD

lack of normalcy and assure robust interpretations. This implies that for direct correlation analysis, we must utilize Spearman correlation, which does not rely on the data's normality. Spearman correlation is well-suited to capture monotonic connections between variables, making it an ideal choice for assessing the interaction between HTD and SI in this context.

5.1.2 Correlations of SI with Quality Metrics

The SI has varied degrees of negative association with several quality indicators, implying an inverse relationship between security concerns and certain quality characteristics. As Figure 5.3 illustrates, SI correlates adversely with measures such as CBO (-0.232), WMC (-0.263), RFC (-0.266), and Fanout (-0.233). Higher values of these metrics, which frequently imply complexity and tightly linked code, are associated with poorer security ratings. SI has a weak correlation with NOC (-0.051) and Duplicated Lines Density (-0.111), suggesting a less direct impact on software security performance.

High TD Classes have a moderate negative correlation with the SI (-0.338), indicating a link between technical debt within specific classes and lower security performance. How-

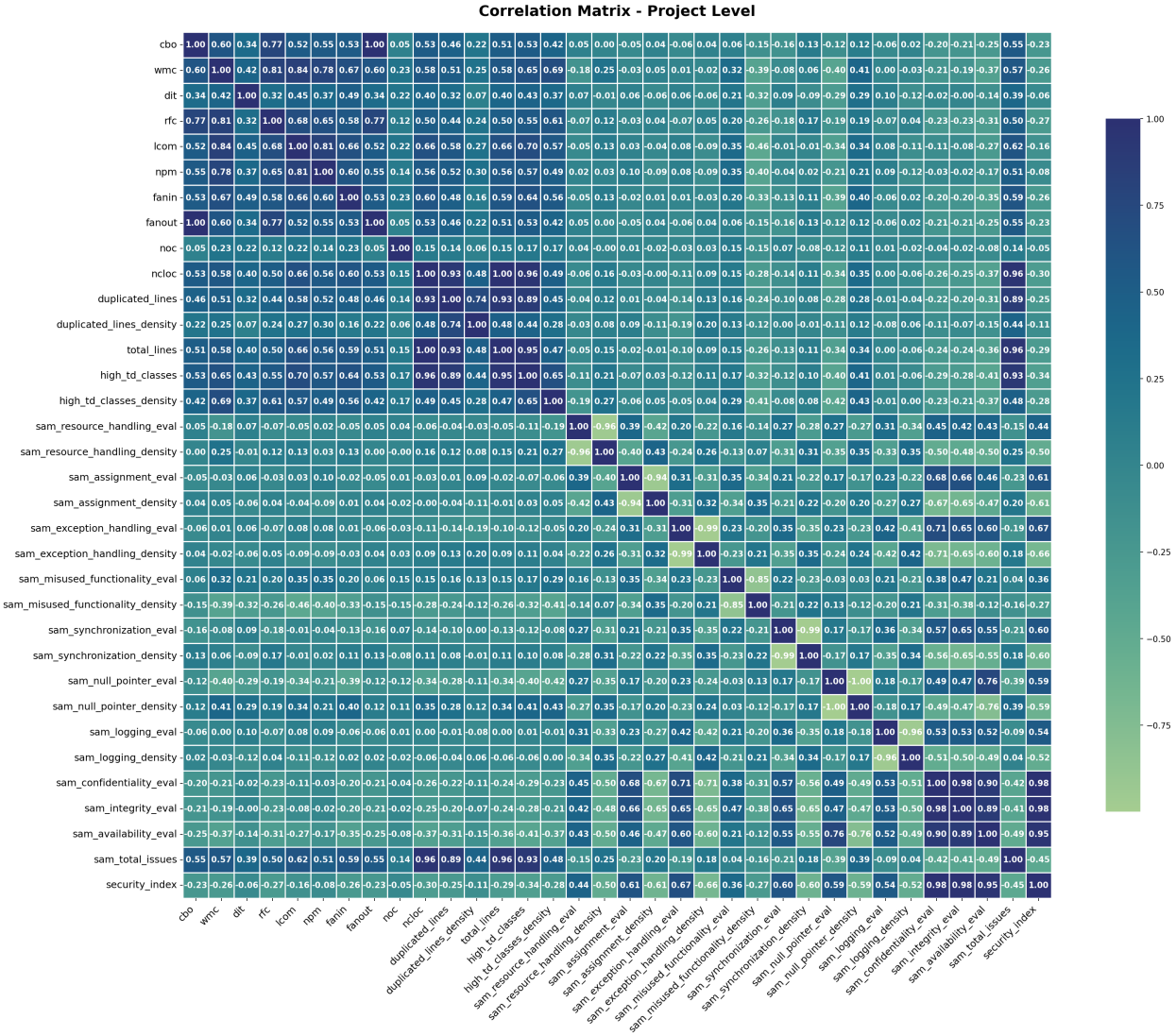

Figure 5.2: Correlation Matrix with all metrics

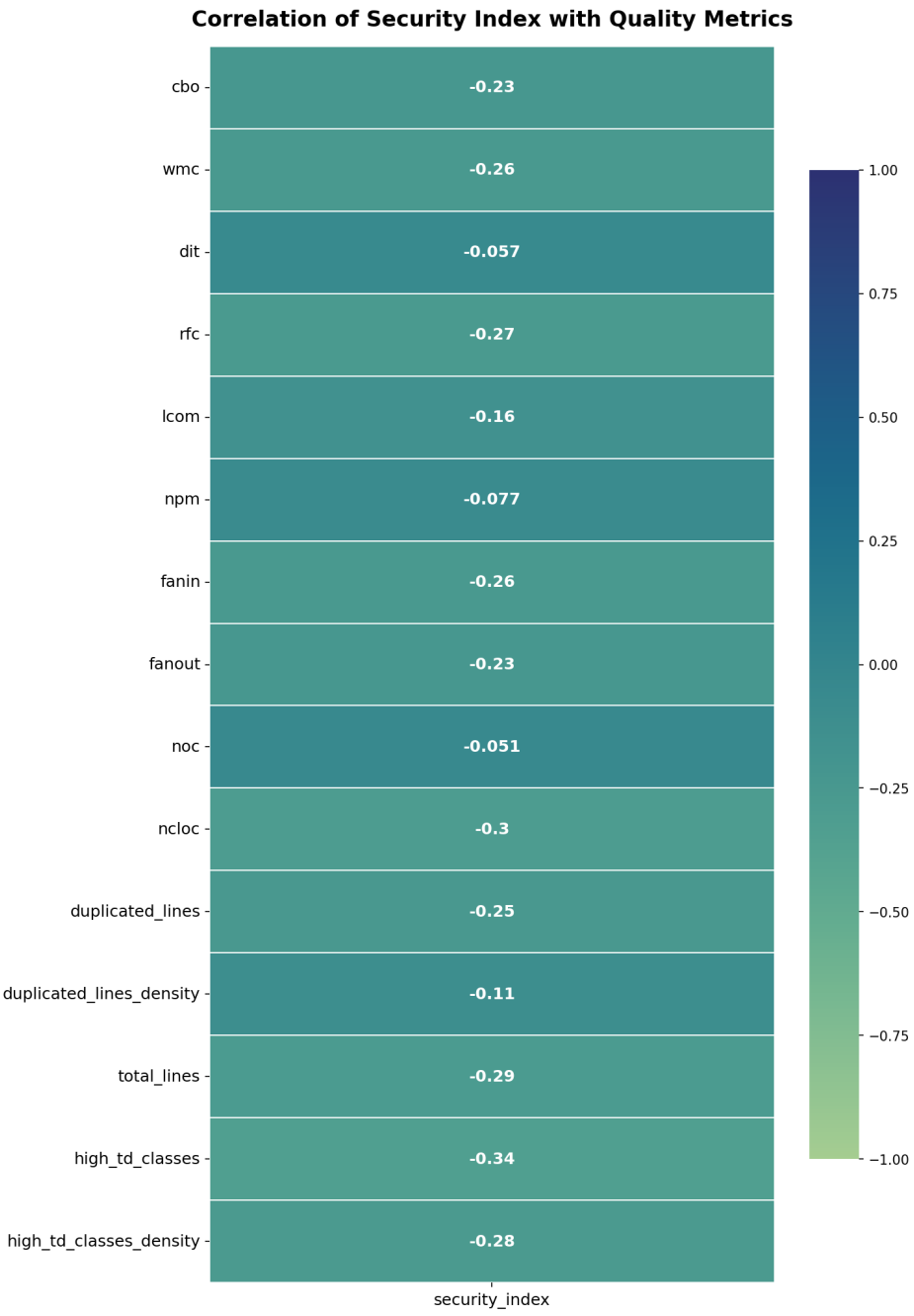

Figure 5.3: Correlations of SI with Quality Metrics

ever, this statistic does not account for project size, which may induce bias by preferring larger initiatives. To address this constraint, we use HTD Density, which normalizes the statistic by project size, resulting in a reduced but still considerable negative correlation (-0.277). This density metric provides a more balanced depiction by taking into account both the amount of technical debt and the size of the project. While the link is not as obvious as with High TD Classes alone, it provides a more solid and contextually correct view into the relationship between

5.1.3 Correlation of HTD with Security Metrics

The relationships between HTD and various security metrics can be viewed in Figure 5.4. Metrics exhibit a range of positive and negative correlations. Strong positive correlations were found between HTD and Sam Null Pointer Density (0.425), Sam Misused Functionality Eval (0.287), and Sam Resource Handling Density (0.274), suggesting that high technical debt in specific classes correlates with higher risks in these security dimensions. On the other hand, negative correlations were found with Sam Misused Functionality Density (-0.406), and Sam Null Pointer Eval (-0.423) indicating that areas with dense technical debt tend to exhibit security vulnerabilities in these aspects as well. Interestingly, Sam Total Issues (0.480) correlates positively with HTD, further emphasizing the direct impact of technical debt on security issues within a project.

5.1.4 Predicting SI & HTD

To investigate the association between HTD and the Security Index, predictive models were developed to see how well one metric predicts the other. Two models were created: one using HTD to predict the Security Index and another using the SI to predict HTD. Figure

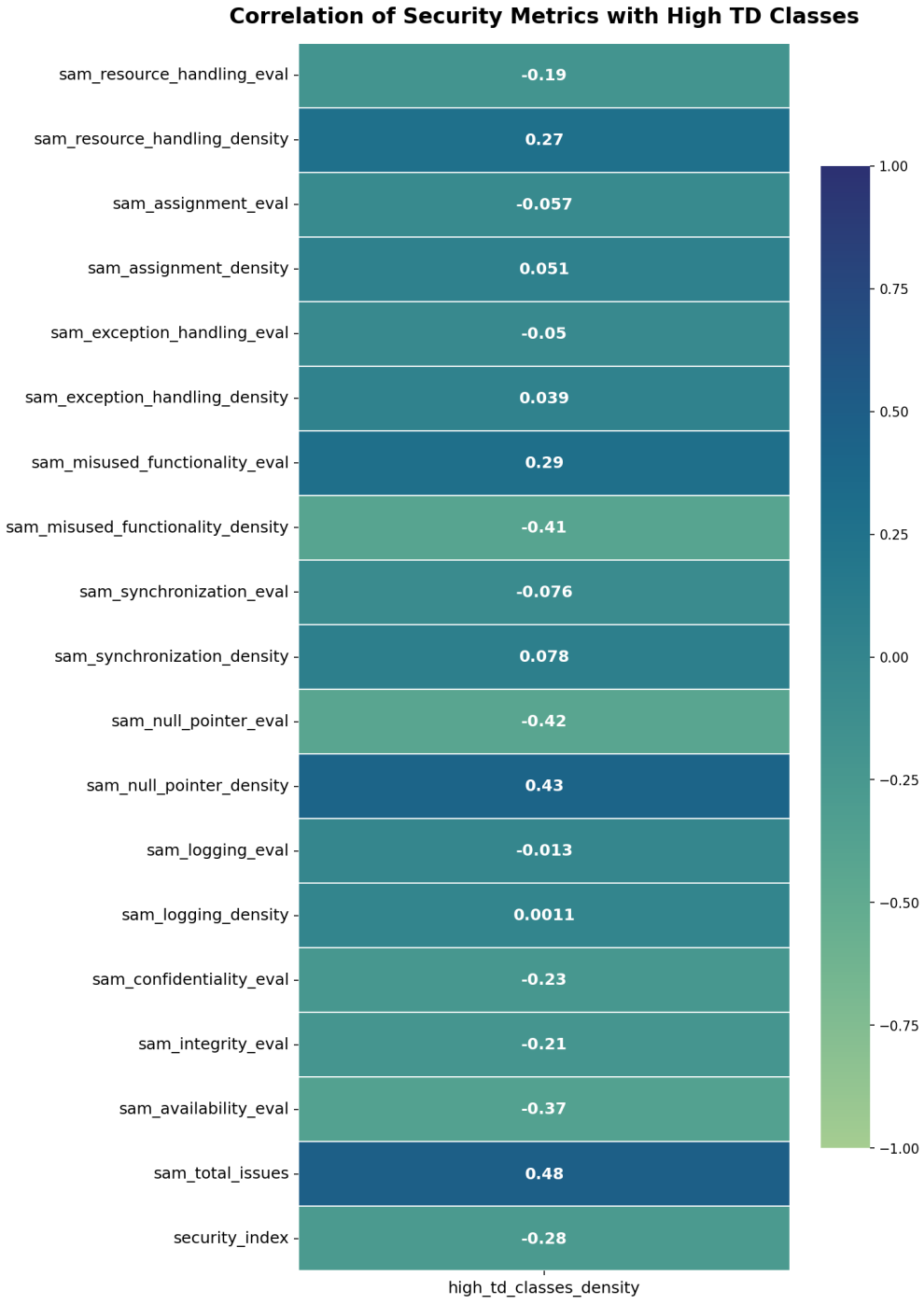

Figure 5.4: Correlation of HTD with Security Metrics

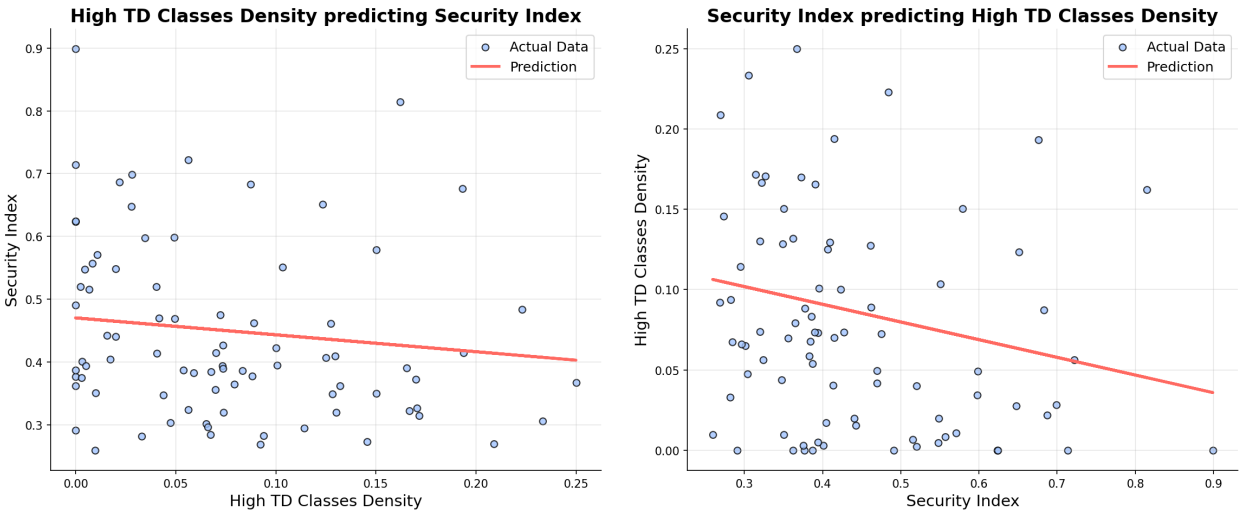

Figure 5.5: Predicting SI & HTD

5.5 illustrates these relationships.

Similarly, when the SI was employed to predict HTD, it performed poorly. The model explained just 2.26% of the variation in HTD, with an MSE of 0.0041 and a R^2 value of 0.0226. These findings indicate that SI has a limited ability to predict HTD, supporting the limited link between these metrics in the models evaluated. Both models have poor prediction accuracy, as evidenced by the plots 5.5, with points showing large dispersion from the regression lines. This indicates that, while there may be some underlying correlations between HTD and Security Index, they are not substantially linear or easily represented by these specific predictive models.

5.1.5 Heteroscedasticity Analysis

The Breusch-Pagan Test was used to determine whether or not the predictive models had heteroscedasticity. The test determines if the variance of the residuals is constant across different values of the independent variable, which is a necessary assumption in linear regression models.

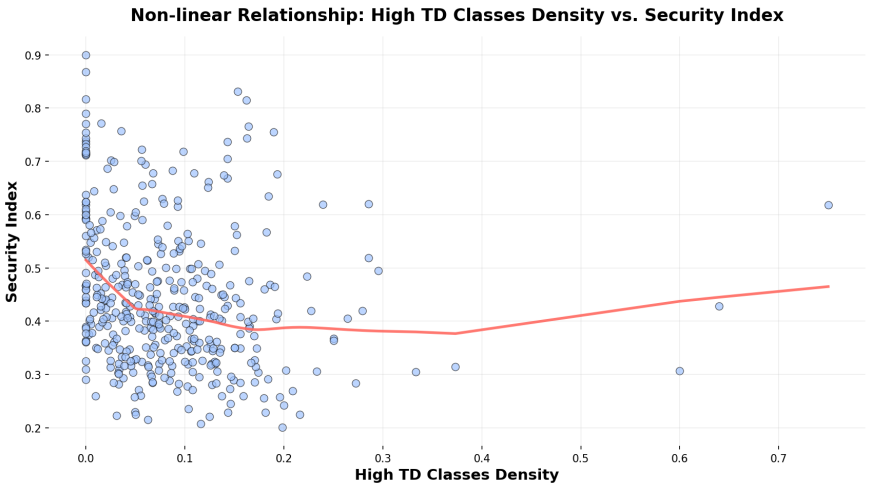

Figure 5.6: Non-Linear Regression

The results of the Breusch-Pagan test yielded a Lagrange multiplier statistic of 1.477, with a p-value of 0.224. Additionally, the F-statistic was 1.475, with a corresponding p-value of 0.225. Since the p-values for both the Lagrange multiplier and the F-statistic are greater than the typical significance level of 0.05, the test does not suggest the presence of heteroscedasticity in the models.

This suggests that the residual variance is relatively constant, and the homoscedasticity assumption is valid. As a result, heteroscedasticity is not an issue in the tested models, and the results can be regarded as reliable in terms of residual variance.

5.1.6 Non-Linear Regression

A Kendall's Tau correlation was used to investigate potential non-linear correlations between the HTD and the Security Index. Kendall's Tau statistic assesses the strength and direction of a correlation between two variables while accounting for non-linear interactions.

The computed Tau value was -0.1954, with a p-value of 0.0000, showing a statistically

significant but weak negative correlation between the two variables. Although the association is substantial, the low amplitude of the correlation shows that a non-linear model does not provide a meaningful advantage over the linear approach (see Figure 5.6).

Given the weak connection, this result validates the finding from linear models that forecasting the SI from HTD (and vice versa) is not well-suited to either linear or non-linear models, as seen in Figure 5.6. The models' overall performance remains limited, indicating that more components or more complicated models may be necessary to adequately capture the link between these indicators.

5.1.7 Correlation Grouping and Categorization

The research reveals distinct patterns in the connections between the SI, HTD, and other software indicators. The dataset was classified using the SI threshold (0.48), and the subsequent analysis revealed useful insights into the behavioral dynamics of software systems. The "low SI" (SI $<$ 0.48) and "high SI" (SI $>=$ 0.48) groups have Spearman correlation coefficients of -0.220 and -0.143, respectively, demonstrating a weak negative correlation between SI and HTD in both situations. However, the somewhat greater link in the "low SI" group indicates that the relationship between SI and HTD is more prominent in less secure settings.

Further stratification of the data using the HTD threshold (0.13) revealed more significant variability in the correlations. For projects with HTD less than 0.13, the Spearman correlation coefficient is -0.305, indicating a moderately negative correlation between SI and HTD. In contrast, in classes with HTD greater than 0.13, the correlation is low (-0.015). This significant contrast shows that SI's influence on HTD declines in environments with larger levels of technical debt, possibly indicating a saturation effect in which other factors

exceed SI's effects.

In conclusion, our findings demonstrate the subtle interplay between SI and HTD in various circumstances. While the overall negative trend between these indicators is clear, their severity differs depending on security levels and technical debt criteria. The data imply that SI has a greater impact in low technical debt settings, which provides a foundation for targeted interventions.

5.2 Results of the Time-Series Analysis

This section delves into the time series analysis of HTD and SI to investigate their temporal behavior and potential connections. By evaluating the evolution of these indicators over time, the analysis hopes to discover underlying patterns, trends, and shifts that static statistical measures may ignore. Recognizing the complexities of these processes, we determined the necessity to delve more into the findings. To gain a more complete picture, we decided to perform a second research phase and create a new dataset that included data from five industry-recognized initiatives. This expanded technique aims to validate findings and provide more solid insights into the relationship between technical debt and security.

5.2.1 Visualization of Temporal Trends

Project: Apache Dubbo

In the Dubbo project (Figure 5.7), the time series analysis of HTD and SI reveals minimal interaction between these two metrics over the observed period. As indicated by the Augmented Dickey-Fuller (ADF) tests, both HTD and SI are non-stationary, with p-values

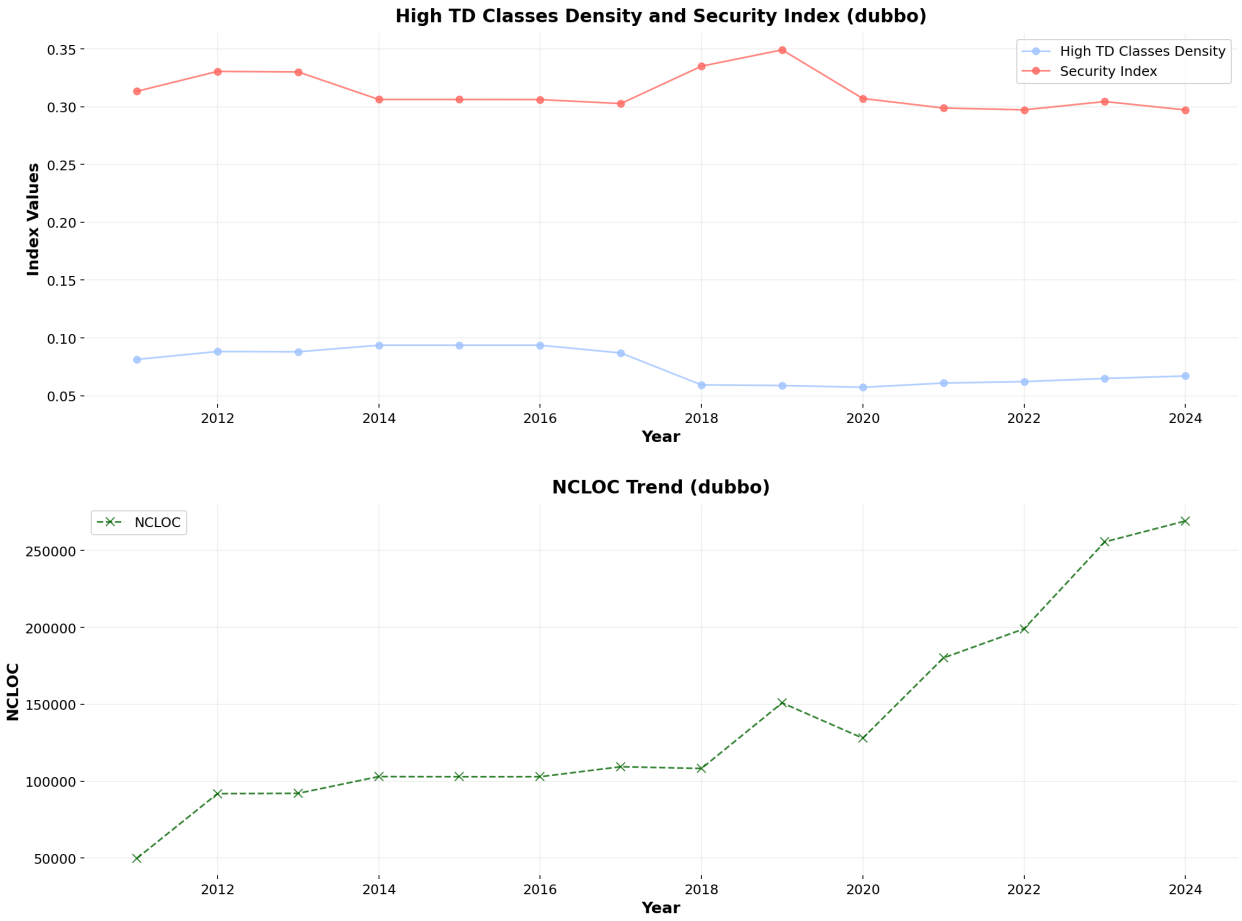

Figure 5.7: Time Series Analysis of Dubbo Project

of 0.7785 and 0.2530, respectively, confirming the absence of a fixed trend or mean in their temporal evolution. This non-stationarity is further visualized in the first subplot, where the HTD shows subtle fluctuations, while the SI exhibits a more consistent pattern at a higher value range.

The Spearman correlation analysis supports these findings, revealing a modest and statistically insignificant negative correlation (-0.0949, p-value = 0.7469) between HTD and SI. This implies that the temporal dynamics of these metrics are essentially independent, revealing different underlying variables influencing code quality and security.

Furthermore, the second subplot depicts the Non-Comment Lines of Code (NCLOC)

trend, which demonstrates a steady and large growth over time, especially after 2020. The constant growth in NCLOC contrasts with the relatively consistent patterns of HTD and SI, demonstrating that while the project's codebase size has increased significantly, the quality and security measures have not seen commensurate changes. This observation is consistent with the statistical findings, demonstrating that codebase growth does not intrinsically alter these critical software measures. These findings imply that the Dubbo project's technical debt and security posture are separated over time.

Project: Elasticsearch

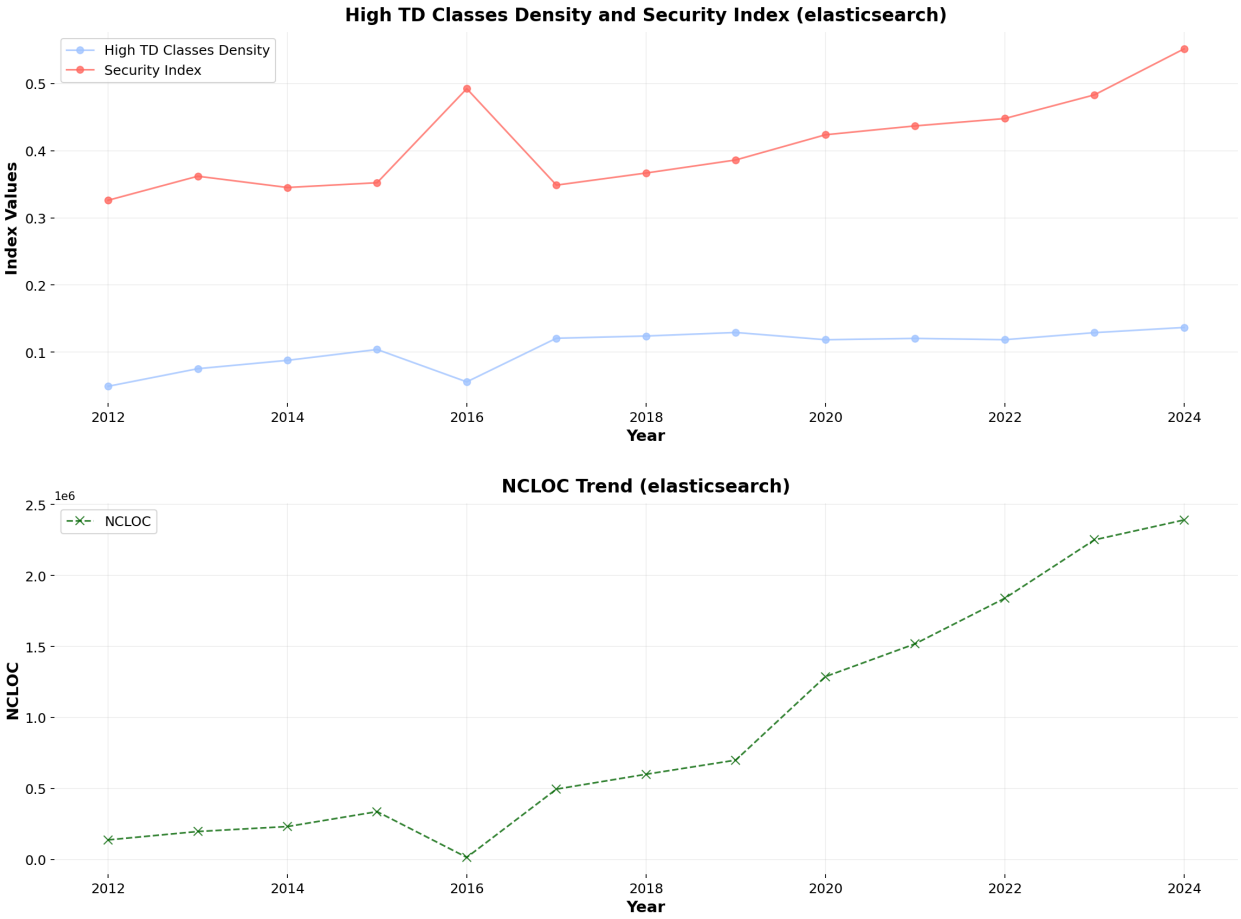

Figure 5.8: Time Series Analysis of Elasticsearch Project

The Elasticsearch project's time series analysis (Figure 5.8) identifies major patterns in the HTD and SI. The Augmented Dickey-Fuller (ADF) tests indicate non-stationarity for both metrics, with p-values of 0.1593 for HTD and 1.0000 for SI, showing a lack of a set trend or mean in their time-dependent behaviour. The first subplot 5.8 demonstrates a lack of stationarity, with SI showing a significant rising trend over time and HTD exhibiting modest oscillations within a steady range.

The Spearman correlation analysis shows a moderate positive correlation of 0.4505 between HTD and SI, however the association is not statistically significant (p-value = 0.1223). This shows a possible interdependence between these indicators, which could indicate common influences from project aspects such as quality improvements or security measures. However, the lack of statistical significance emphasizes the need for additional research to identify the nature and intensity of this association.

The second subplot displays the trend in Non-Comment Lines of Code (NCLOC), which has increased significantly throughout the investigated time, notably since 2016. This increase in codebase size contrasts with the slow dynamics of HTD and the consistent increase in SI. The expanding NCLOC shows the extent and progress of the Elasticsearch project, while the HTD and SI indicators do not indicate comparable changes, highlighting the complexities of their relationship. These data, taken together, imply that quality and security dynamics are evolving and should be investigated further.

Project: Apache Kafka

In the Kafka project (Figure 5.9), time series analysis demonstrates that HTD and SI behave differently. The Augmented Dickey-Fuller (ADF) tests show that HTD is non-stationary, with a p-value of 0.2658, showing time-dependent variability and no stable mean or trend. In comparison, the SI is stationary (p-value = 0.0003), indicating a stable pattern across

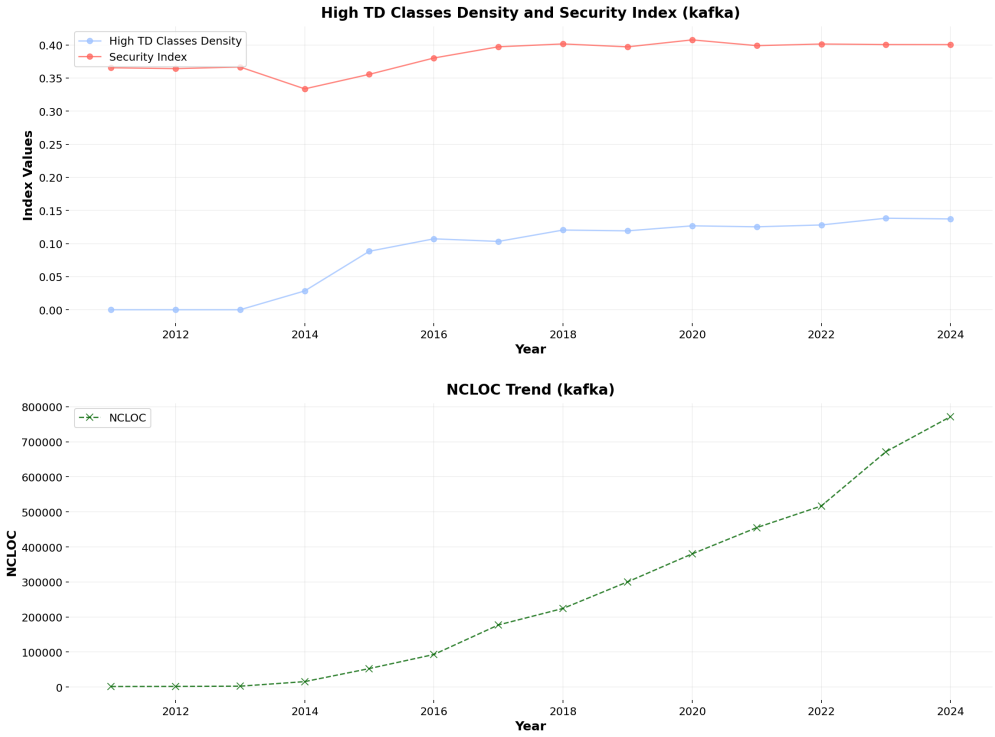

Figure 5.9: Time Series Analysis of Kafka Project

time.

The Spearman correlation analysis shows a strong positive correlation of 0.8190 between HTD and SI, which is statistically significant (p-value = 0.0003). This implies a strong link in which increased HTD levels are closely connected with improved SI. Such a significant relationship is unique among the evaluated projects and may indicate the intertwining effects of quality and security improvements in Kafka's development process.

The visual representation emphasizes these interactions. In the first subplot 5.9, the SI trend is constant at relatively high values, but HTD gradually grows over time. This co-movement strengthens the previously established favorable association. Meanwhile, the second subplot shows a significant increase in Non-Comment Lines of Code (NCLOC), indicating a fast expanding codebase. Kafka's scalability is obvious, but its HTD and SI metrics stay closely matched, implying successful management of technical debt and security

Figure 5.10: Time Series Analysis of Guava Project

concerns despite rising project complexity.

Project: Google Guava

The Guava plots (Figure 5.10) show the patterns and correlations between HTD, the Security Index, and NCLOC (Non-Comment Lines of Code) in the Guava project over time. The High TD Class Density metric has been generally steady over the years, with slight changes. Following 2016, a minor increasing trend emerges, possibly indicating a gradual increase in the density of classes with considerable technical debt. In comparison, the SId shows a significant decline around 2016, followed by a steady recovery and stabilization in later years.

This pattern could indicate a time of significant architectural or feature changes that briefly impacted security. Meanwhile, NCLOC has shown constant growth through 2016, indicating that the codebase is actively developed and expanded. Following 2016, NCLOC levels off, with a slight decline, possibly attributed to code refactoring, optimization efforts, or shifts in development priorities.

To assess the stationarity of the High TD Class Density and SI time series, the Augmented Dickey-Fuller (ADF) test was used. The HTD test produced a test statistic of -0.9083 with a p-value of 0.7852, showing non-stationarity because the p-value surpasses the 0.05 significance level. Similarly, the SI had a test statistic of -2.0833 with a p-value of 0.2513, indicating non-stationarity as well.

These findings imply that both measurements alter over time without a constant mean or variance, maybe driven by external variables like as project milestones, codebase modifications, or development processes.

A Spearman correlation study demonstrates a moderate negative connection between the High TD Class Density and the Security Index (correlation coefficient = -0.5941; p-value = 0.0152). This observation implies that when the density of high-TD classes falls, the SI improves. This inverse relationship shows the interaction between technical debt and security, emphasizing the possible benefits of reducing technical debt in order to improve software security.

Overall, these insights provide a thorough understanding of the Guava project's changing dynamics. The observed trends in the SI and High TD Class Density, combined with their negative association, point to a strong relationship between technical debt management and security improvements. The non-stationarity of both metrics emphasizes the importance of contextual factors and the necessity for adaptive techniques to successfully manage software quality and security over time.

Project: Spring Framework

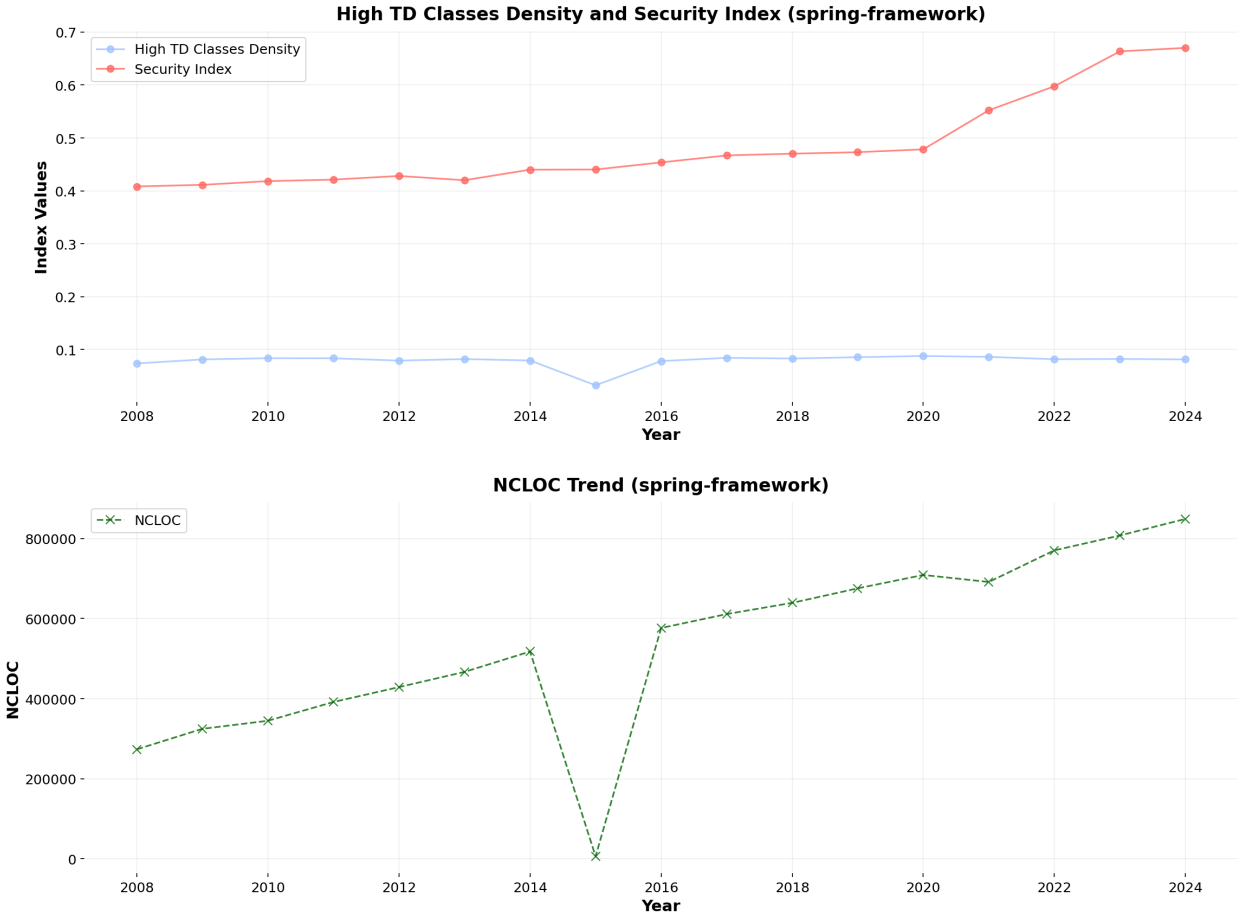

Figure 5.11: Time Series Analysis of Spring Framework Project

The Spring Framework project's investigation (Figure 5.11) focuses on the HTD and SI over time. The Augmented Dickey-Fuller (ADF) test results show that HTD has stationarity (p-value = 0.0106), indicating a continuous trend during the study period. In comparison, the SI is non-stationary (p-value = 0.9991), indicating that its values vary significantly.

The Spearman correlation coefficient of 0.3701 between HTD and SI suggests a somewhat positive correlation. However, this correlation lacks statistical significance (p-value = 0.1437). While this shows a possible link between the two metrics, the relationship is less obvious when compared to other projects, such as Kafka, which showed a bigger correlation.

Figure 5.11 depicts how these measurements change over time. Despite its relatively low values, HTD has remained consistent over time, whereas SI has been trending steadily, notably around 2020. This shows that security-related aspects are improving, either as a result of better development methods or a greater emphasis on fixing vulnerabilities.

The second chart illustrates a consistent increase in Non-Commented Lines of Code (NCLOC) over time, indicating the Spring Framework's growing size and complexity. The big decline around 2014 could indicate major refactoring or restructuring efforts that reduced code redundancy. Following 2016, the continued growth in NCLOC corresponds to the observed trends in the Security Index, indicating a likely focus on extending functionality while resolving security concerns.

Overall, while the Spring Framework indicates relatively modest and non-significant statistical correlations between HTD and the Security Index, the observed trends indicate that continued attempts to improve security may have an indirect impact on technical debt management.

5.2.2 Detrending and Correlation Analysis of Temporal Data

Project: Apache Dubbo

The Dubbo investigation focused on examining the connection between HTD and SI after removing their respective patterns. Figure 5.12 displays a detrended time series that shows fluctuations in both metrics over time. Statistical study finds a -0.3357 Spearman correlation coefficient between HTD and the Security Index. However, this link lacks statistical significance (p-value = 0.2861). These findings imply that any potential relationship between the two measurements is modest and could be due to random variation rather than a systematic tie.

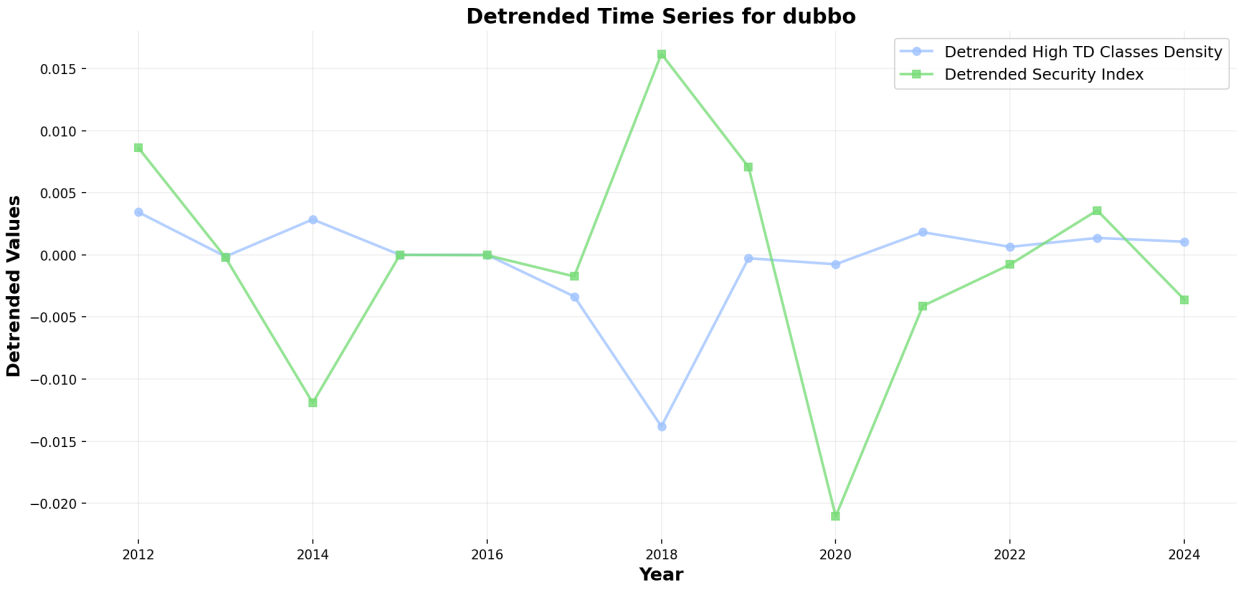

Figure 5.12: Detrended Data for Dubbo Project

The detrended data shows periodic inverse connections, particularly between 2014 and 2020, with severe dips in one metric corresponding to surges in the other. This could be due to particular trade-offs or development decisions affecting technical debt or security, rather than an overarching trend. Notably, the fluctuations stabilize slightly after 2020, potentially indicating more regular development or maintenance procedures.

Overall, the Dubbo analysis shows that, while the two metrics move in opposite directions at times, there is no substantial statistical association, indicating a weak and inconsistent link.

Building on the analysis, the cross-correlation plot (Figure 5.13) provides additional insights into the relationship between HTD and the SIacross different time lags. The prominent peaks in the negative lag region (e.g., -4 and -3) indicate that changes in HTD precede corresponding changes in the Security Index. This suggests that technical debt accumulation might influence security outcomes with a delay. Conversely, the positive lag region shows weaker correlations, implying that the SIhas limited predictive power for future variations

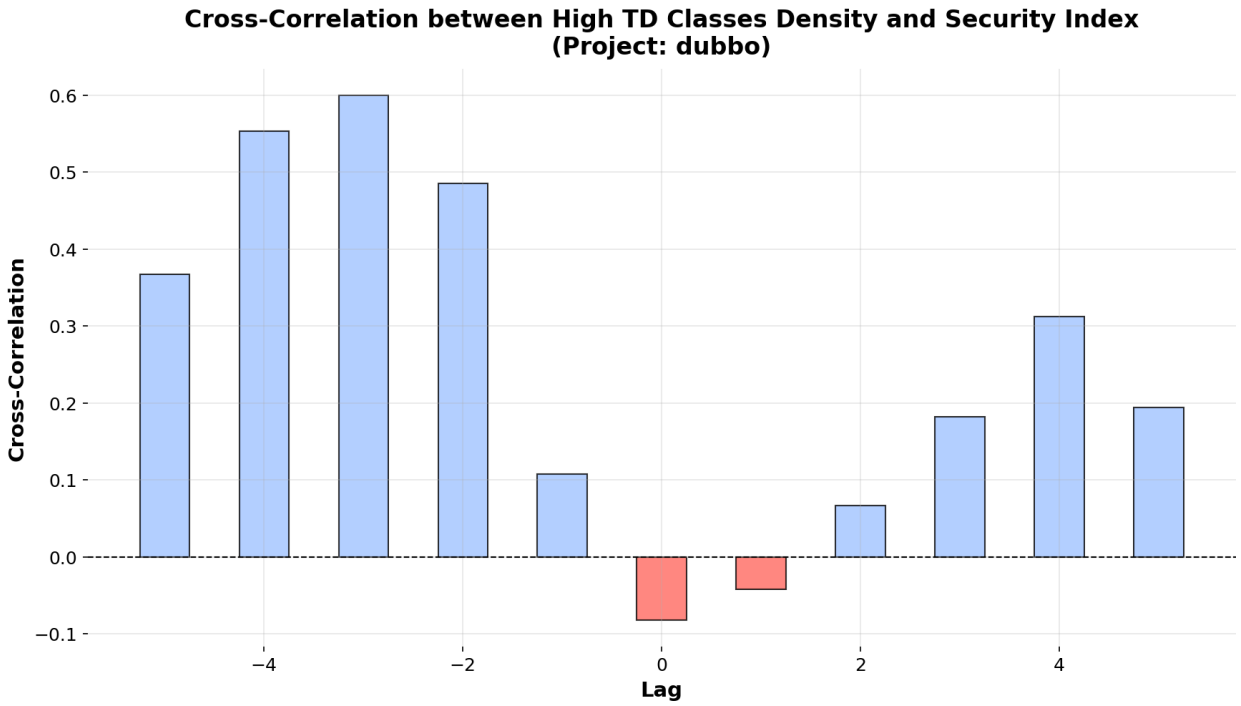

Figure 5.13: Cross-Correlation of Dubbo Project

in HTD

Project: Elasticsearch

After detrending, the Elasticsearch project examined the link between HTD and SI. Figure 5.14 shows significant changes in both metrics between 2016 and 2018. The Spearman correlation analysis finds a significant negative correlation of -0.7909 between HTD and the Security Index (p-value = 0.0037). This suggests that while one metric increases, the other tends to decrease, implying an inverse relationship. This tendency could be indicative of trade-offs between the two measures during the development phase.

The detrended time series graphic shows noticeable peaks and troughs, particularly in 2016, when the SI jumps and the HTD lowers sharply. In contrast, the years after 2018 demonstrate rather stable dynamics, with the SI progressively rising and the HTD

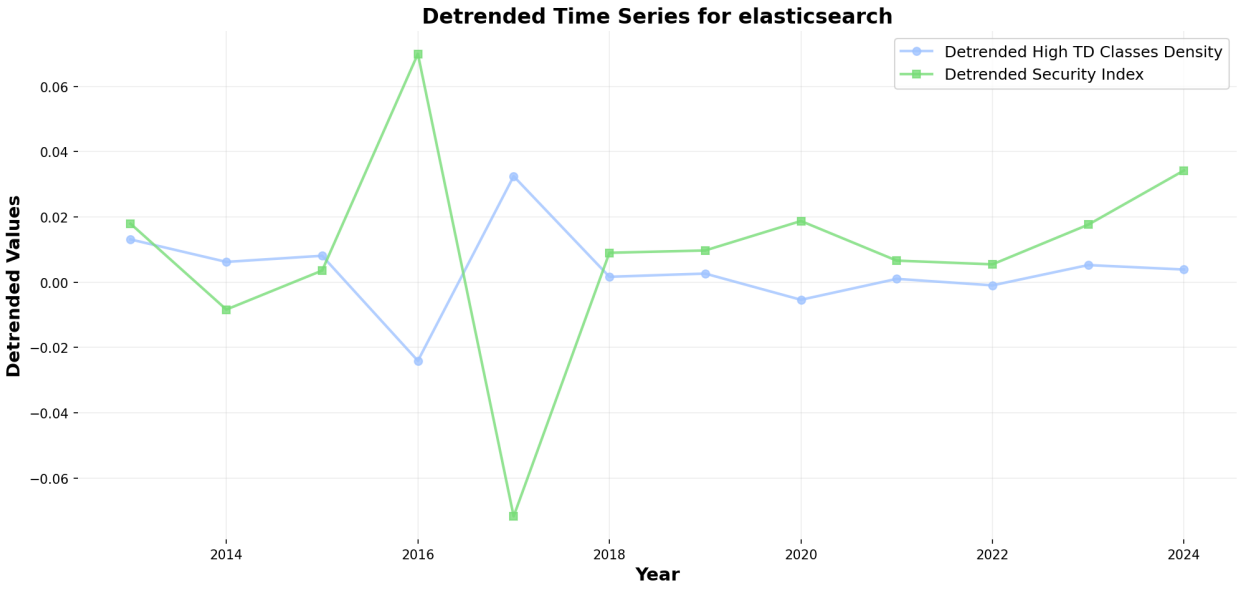

Figure 5.14: Detrended Data for Elasticsearch Project

maintaining a more regular trajectory. This could signal a shift in development priorities, with a greater emphasis on improving security measures at the risk of incurring technological debt. Overall, the Elasticsearch project demonstrates a strong inverse link between the two indicators.

The cross-correlation Figure 5.15 for the Elasticsearch project provides more information on the temporal link between HTD and the Security Index. Strong positive correlations in both negative and positive lag regions, particularly around -4 and -2, indicate that changes in HTD can affect the SI with a delay. Notably, the significant correlation at lag 0 suggests synchronous variations between the measurements, supporting the notion of a close inverse relationship. Furthermore, the significant correlations at positive lags suggest that the SI may alter HTD in following time periods. These trends point to a bidirectional interaction, with possible feedback loops in how technical debt and security concerns are addressed over time.

Figure 5.15: Cross-Correlation of Elastic Project

Project: Google Guava

The detrended study of the Guava project demonstrates significant changes in both High TD Class Density (HTD) and the Security Index (SI) over time. Figure 5.16 shows considerable decreases in SI and a small decrease in HTD from 2016 to 2018. After 2018, SI exhibits sustained recovery with a steady increasing trajectory, whilst HTD maintains a rather constant trend with minor changes. These observed trends point to an increased emphasis on upgrading security measures beyond 2018, possibly at the expense of controlling technical debt. The synchronized patterns between the two indicators point to a dynamic relationship, with trade-offs influencing overall project quality during specific development phases.

The Spearman correlation analysis shows a weak correlation between HTD and SI (-0.1964, p-value = 0.4829). Although the correlation is not statistically significant, it does indicate a minor inverse connection, with rises in one metric corresponding to declines in the

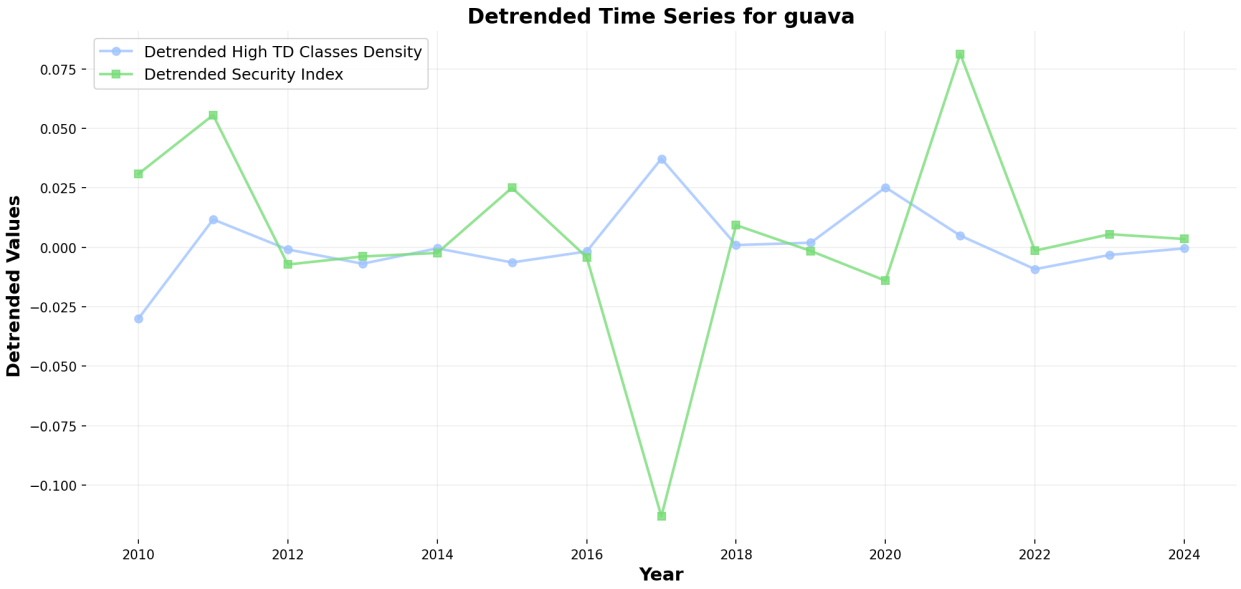

Figure 5.16: Detrended Data for Guava Project

other. This finding emphasizes the complexities of balancing security enhancements with technical debt management throughout the development lifecycle, with no clear indication of a dominant relationship between the two variables.

Figure 5.17 illustrates a cross-correlation analysis that investigates the temporal relationships between HTD and SI. Significant positive correlations with negative lags (e.g., -4 and -2) indicate that changes in HTD may occur before differences in SI, underscoring how earlier technical debt decisions can influence future security outcomes. Significant negative correlations at positive delays, on the other hand, imply the converse effect, in which SI changes may have an impact on HTD in future years. The synchronous correlation at lag 0 supports the notion of a concurrent relationship between the two measurements. Overall, the Guava project demonstrates a bidirectional dynamic between HTD and SI, with feedback loops impacting their evolution over time, even if the overall connection is poor.

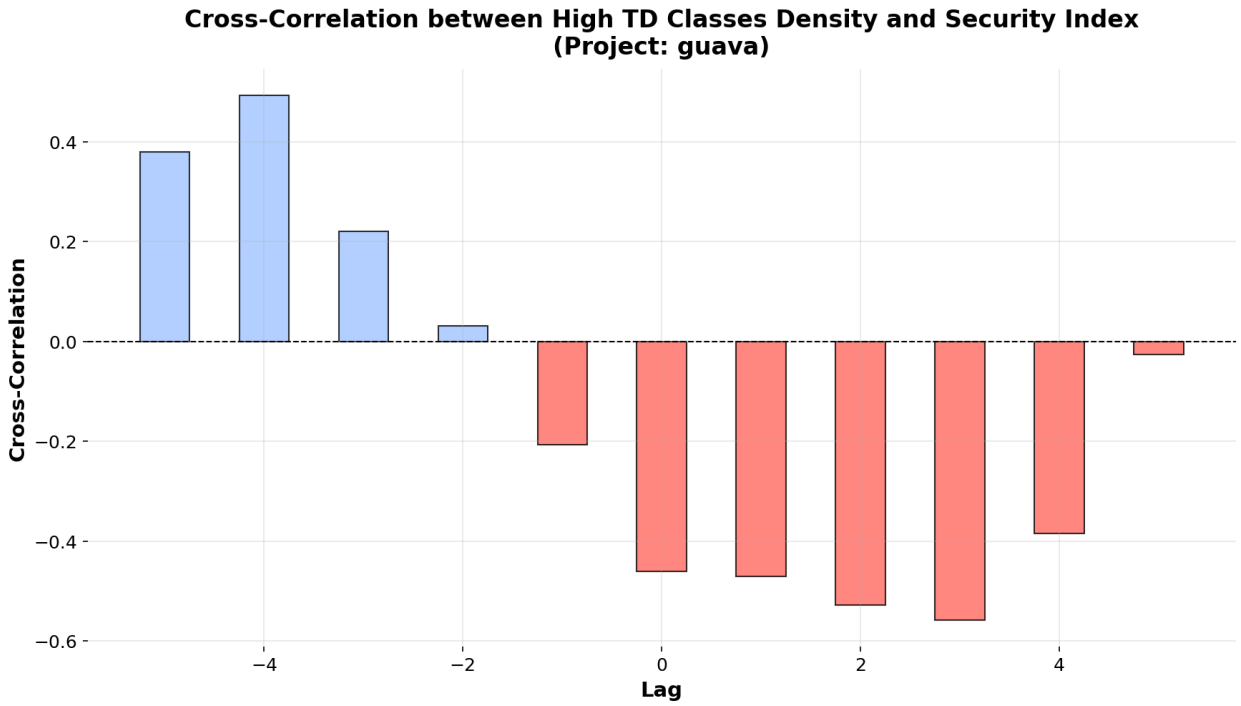

Figure 5.17: Cross-Correlation of Guava Project

Project: Apache Kafka

Figure 5.18 shows the detrended time series for two Kafka metrics: High TD Class Density (blue) and SI (green) from 2012 to 2024. By removing the overall trends from the data, we can focus on the relative variations and underlying patterns in the measurements.

The HTD exhibits great variability, peaking in 2014 and then fluctuating cyclically. The highest positive deviation is recorded in 2014, with a considerable decline in succeeding years, followed by oscillations of lower magnitude.

In contrast, the SI reaches a less noticeable high in 2014 but recovers fast the next year, staying at zero with slight changes in subsequent years. During certain eras, the SI strongly resembles the HTD (for example, 2020 and 2022), implying that their patterns may be correlated.These detrended data reveal temporal anomalies and probable connections

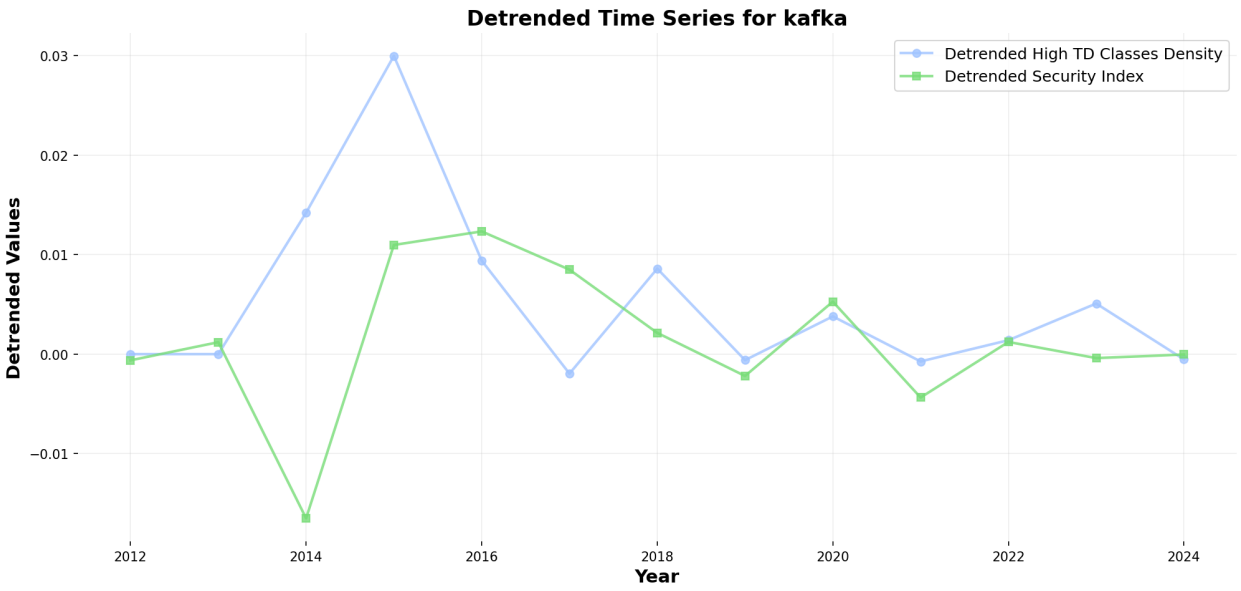

Figure 5.18: Detrended Data for Kafka Project

between Kafka's software density, complexity, and security features.

The cross-correlation Figure 5.19 for Kafka provides further information on the temporal link between HTD and the Security Index. Strong correlations with negative lags (e.g., -2 and -3) imply that changes in HTD may occur before shifts in the Security Index, indicating a potential predictive link. The significant correlation at lag 0 indicates synchronous fluctuations between the two measurements, supporting the notion of a close link in their short-term behavior.

Interestingly, the positive lag zone has moderate correlations, implying that the SI may impact subsequent fluctuations in HTD, but the effect is weaker than the inverse correlation. These findings indicate a dynamic interplay between the indicators, which could reflect a balance between attempts to control technological debt and maintain security.

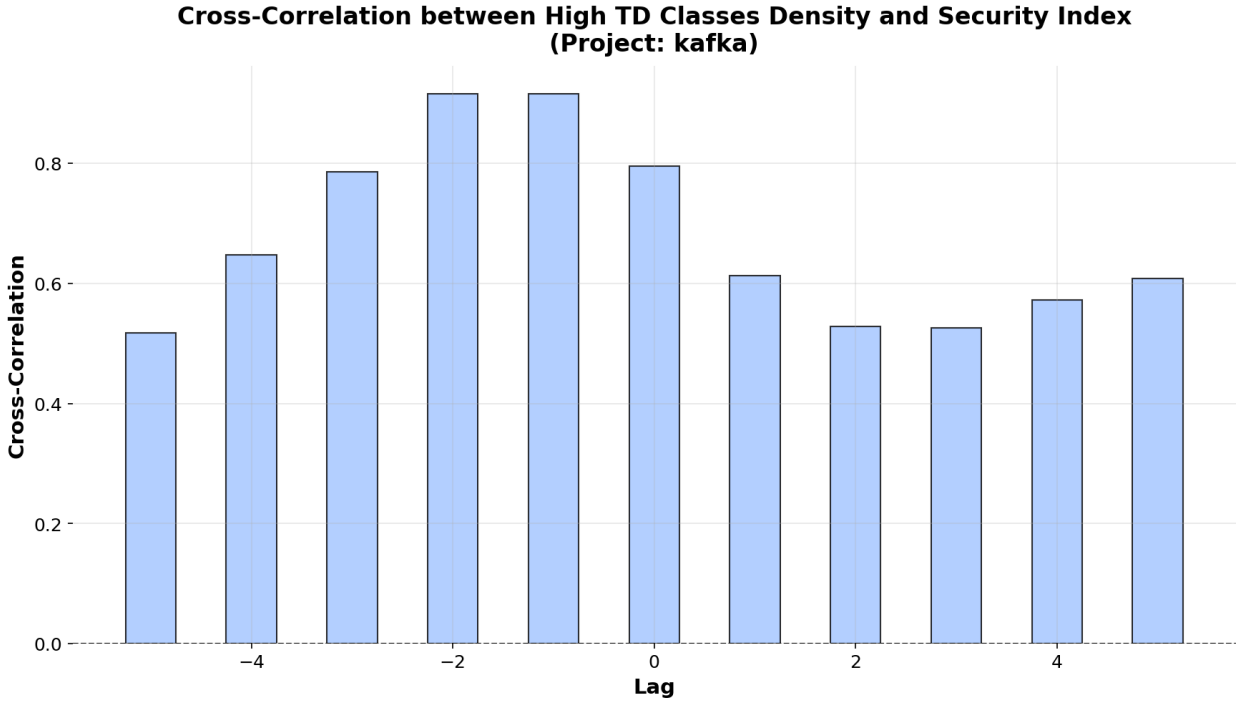

Figure 5.19: Cross-Correlation of Kafka Project

Project: Spring Framework

Figure 5.20 displays the detrended time series for two Spring Framework metrics: the HTD and the SI from 2008 to 2024. By removing the overall trends, we can look at the relative shifts and deviations in these variables. The HTD fluctuates significantly, with a noticeable high around 2016. This peak is followed by a dramatic decrease to its lowest value, after which it stabilizes with tiny oscillations around the zero line beginning in 2018.

The SI follows a different pattern, with a notable peak about 2020. Prior to this period, the SI showed steady cyclical variations, but it became very unpredictable in later years, peaking in 2022 and declining drastically in 2024. Although the two measurements follow different trends, some periods of congruence, such as around 2020, may indicate potential linkages. The dramatic changes in both metrics point to changing dynamics in the framework's complexity and security policies over time.

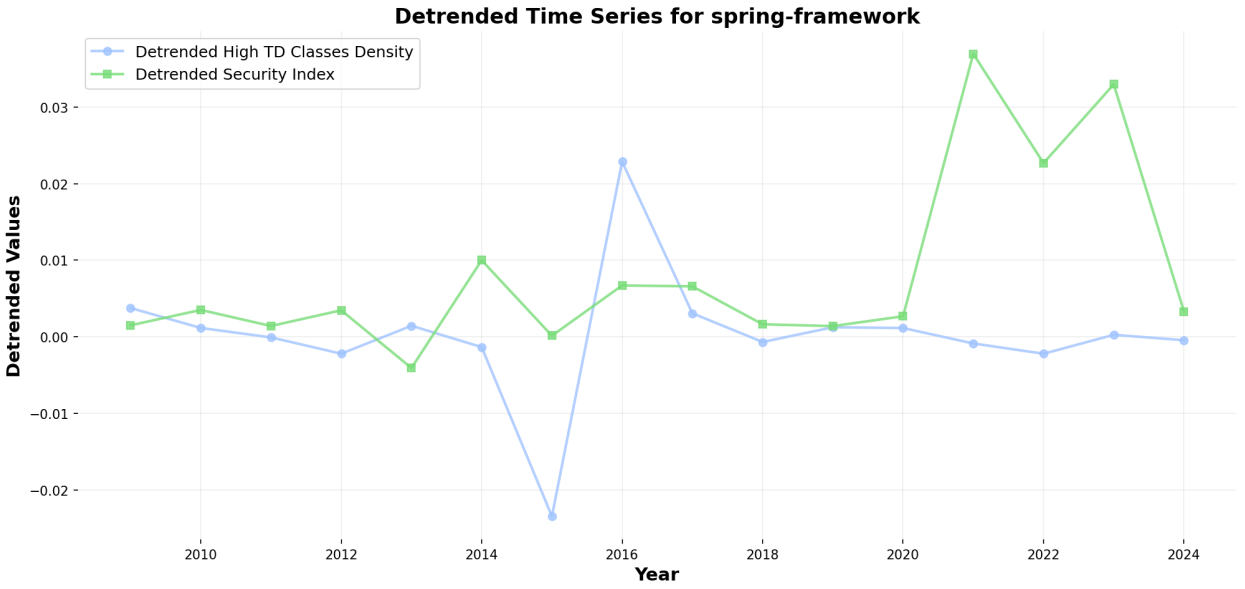

Figure 5.20: Detrended Data for Spring Framework Project

A statistical study of the detrended data reveals a Spearman correlation coefficient of -0.4621 between HTD and the Security Index. However, this link is not statistically significant (p-value = 0.1743), indicating that, while there is a mild inverse relationship, there is insufficient evidence of systematicity. This implies that shifts in the two measurements may be influenced by independent variables or random variation rather than a constant interaction.

The detrended time series also emphasizes times of localized inverse connections, such as those between 2016 and 2020, when substantial rises in one metric are accompanied by significant drops in the other. These trends could indicate development trade-offs or project-specific decisions that emphasize decreasing technical debt or improving security. However, the lack of a significant link suggests that these trade-offs do not persist over time. Future studies should concentrate on detecting extrinsic effects, such as large updates or external pressures, that could cause these variations.

Cross-correlation was used to investigate the link between HTD and the SI in the

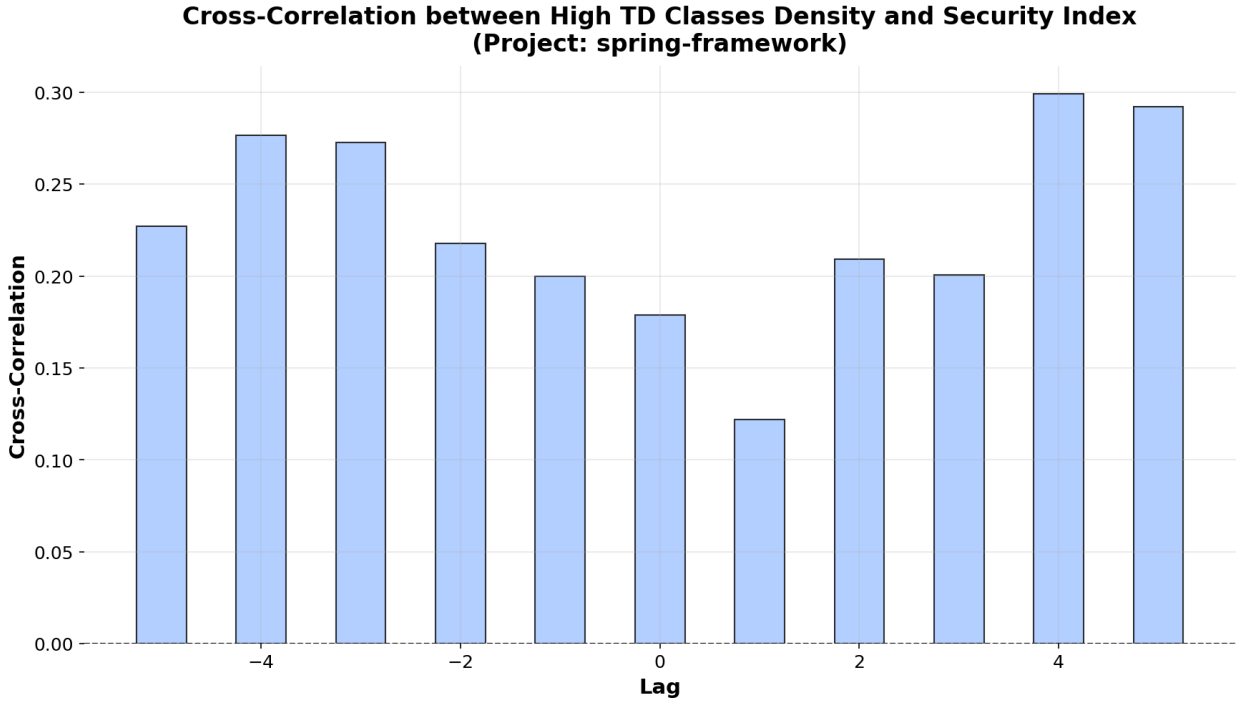

Figure 5.21: Cross-Correlation of Spring Framework Project

Spring Framework and identify potential time-lagged dependencies. Figure 5.21 shows the cross-correlation values with different time lags.

The analysis suggests that the highest positive connections occur at lags -4 and +4, with a correlation coefficient of around 0.30. This suggests that changes in one metric can influence the other after around four time periods, either forward or backward in time. Interestingly, the association is smaller at lag 0 (when the metrics are assessed concurrently), indicating a limited direct, instantaneous relationship between HTD and the Security Index. The symmetry of the cross-correlation pattern shows that the two measurements interact bidirectionally across time.

Inconsistencies occur at intermediate lags (e.g., lag ±3), where the correlation decreases slightly. These abnormalities could be the result of external or project-specific variables interfering with the predicted relationship. While the presence of positive cross-

correlation at certain lags indicates a possible time-lag dependency, the moderate inverse Spearman correlation coefficient of -0.4621 between the two metrics shows a distinct pattern. This negative association means that increases in HTD are associated with decreases in SI (or vice versa). However, the p-value of 0.1743 suggests that this inverse link is not statistically significant, implying that the oscillations in the two metrics are primarily independent or caused by random variations.

Localized patterns of inverse connections, such as those seen between 2016 and 2020, highlight potential trade-offs in which rising technical debt corresponds to increased security efforts, or vice versa. For example, in 2016, a peak in HTD coincided with a significant drop in the Security Index, whereas the opposite occurred around 2020. These findings show that the relationship between technical debt and security may be driven by project-specific decisions or development trade-offs, rather than systemic, long-term dynamics.

5.3 Results of the Commit Analysis

This section investigates the relationship between these two essential factors of software performance by analyzing numerous well-known open-source projects and their changes, including the Spring Framework, Kafka, Dubbo, and Elasticsearch. In these projects, the analysis focuses on HTD as a proxy for software quality and SI as a software security metric. By following these indicators over time, the study discovers various patterns and degrees of congruence between software quality and security, with some projects displaying a clear trade-off and others demonstrating independent variations.

The found correlations between software quality and security in the selected projects differ significantly. Elasticsearch showed a very notable inverse association, with a substantial and statistically significant negative correlation of -0.7909 after detrending. This shows

that as software quality improved, as measured by a decrease in HTD, the SI increased concurrently, implying a potential trade-off in which efforts to improve security contributed to greater technical complexity. In comparison, Dubbo had a weaker negative correlation of -0.3357, which was statistically insignificant, implying that the link between security and quality in this project is less predictable and less consistent. The study of detrended time series data for both Kafka and the Spring Framework found moderate negative correlations (-0.4621 for the Spring Framework), albeit these correlations were not statistically significant. In both cases, the alignment of peaks and troughs in the two metrics—such as a simultaneous decrease in HTD and SI during specific time periods, such as 2020—indicates that, while security and quality may appear to interact during certain phases, these interactions appear to be driven by external events or specific development priorities rather than a consistent, long-term trend.

Several factors influence the observed variation in the association between software quality and security across projects. Prioritizing development efforts during specific events, such as security vulnerabilities or public fault exposures, is a critical element. For example, vulnerabilities such as CVE-2019-17564 in Dubbo frequently result in a heightened focus on security, which may temporarily jeopardize other parts of the product, particularly quality, while resources are redirected to resolve the security issue. Similarly, dependency management has a considerable impact on shaping these connections. Upgrading dependencies to resolve security vulnerabilities, as seen in upgrades to Spring Framework 5.3.25 or Guava, frequently results in modifications that affect backward compatibility or technical debt. These upgrades may result in short-term disparities between quality and security metrics, with security improvements potentially worsening technical debt by introducing more complex dependencies or setups. Architectural changes inside a project, which are intended to improve long-term scalability or security, can sometimes result in short-term increases in technical debt. For example, Kafka's attempts to upgrade its architecture between 2020 and

2022 resulted in an increase in HTD while also preparing the system for future security and scalability improvements. This architectural reworking may cause a temporary misalignment between security and quality while developers strive to reengineer the system for future requirements, potentially adding new complications or technical debt that impair short-term quality measures.

Localized trade-offs between quality and security have also been noticed in some versions or updates. For example, in both Dubbo and Elasticsearch, various stages of development were characterized by a heavy emphasis on either security or quality, which frequently resulted in the prioritizing of one aspect over another. In many cases, efforts to resolve security vulnerabilities or improve security configurations came at the expense of increasing system complexity, resulting in increased technical debt. When the focus returned to quality improvements, such as rewriting code or lowering technical debt, security initiatives were frequently deprioritized, resulting in a brief drop in the Security Index.

The conclusions of this study emphasize the need of using balanced techniques in software evolution. Although negative correlations between software quality and security indicate potential trade-offs, these dynamics also highlight opportunities for collaboration. One major conclusion is the value of integrating quality and security initiatives. By resolving technical debt while also focusing on security improvements, projects can reap long-term advantages without sacrificing maintainability or adding unneeded complexity. This strategy necessitates careful planning and coordination among development teams to ensure that security fixes do not add unnecessary technical debt and that quality improvements do not unintentionally expose the system to new vulnerabilities.

In addition, proactive dependency control is critical for addressing both security threats and software stability. Development teams may maintain a healthy balance between quality and security by remaining on top of security vulnerabilities in dependencies

and ensuring that changes do not destabilize the system or raise technical debt. Furthermore, tracking quality and security metrics holistically throughout the software lifecycle enables more informed decision-making and better forecasting of potential trade-offs during development. This continuous monitoring allows teams to efficiently prioritize activities and solve any concerns before they worsen.

In conclusion, while the link between software quality and security is frequently inverse, differences across projects and development periods indicate that these dynamics are very context-dependent. The research emphasizes the importance of a sophisticated approach to balancing quality and security, with a focus on strategic planning, dependency management, and complete metrics tracking.

5.4 Results of the Class-Level Analysis

In this section, we will do a more in-depth analysis of the data at the class level, focusing on trends and associations unique to each class in the dataset. By breaking down aggregated metrics, we hope to uncover unique patterns, anomalies, and connections that are not visible at a higher level. This granular approach allows for a more nuanced understanding of the factors driving both total_issues (in terms of Software Security) and high_td_proba across different subsets of the data, ultimately increasing the interpretability and robustness of our prediction models.

5.4.1 Nornality Tests

The normality tests show that both total_issues and high_td_proba do not have a normal distribution. The Shapiro-Wilk, Kolmogorov-Smirnov, and D'Agostino's K^2 tests showed

statistically significant results (p-value = 0.000), rejecting the null hypothesis of normality. Similarly, for high_td_proba, all three tests generated significant results (p-value = 0.000), which further confirmed the divergence from normality. These findings imply that non-parametric approaches may be more suited for assessing these variables.

The 95% confidence interval (CI) for total_issues is 18.45 to 18.96, meaning that the true population mean for this variable is likely to lie within this range. The 95% confidence interval (CI) for high_td_proba is extremely narrow, ranging from 0.11 to 0.11, indicating that the variable's value is highly constant and near to 0.11, with minimal variability.

5.4.2 Correlation Matrix

The Spearman rank correlation matrix (Figure 5.22) illustrates complicated links between numerous software measures, including technical debt, complexity, cohesiveness, and security concerns. Results show us that there is a moderate positive correlation between the two indexes (high td probability and security issues) with a score of 0.33. Notable patterns emerge, such as substantial correlations between cohesion-based measures like LCOM and technical debt indicators (e.g., high technical debt probability), implying potential overlaps in their effects on maintainability. Furthermore, coupling metrics (CBO, RFC) have moderate to strong relationships with security-related variables such as resource handling issues and logging issue density, emphasizing the relationship between structural dependencies and security vulnerabilities. The matrix also underlines the importance of density-based measurements, as they regularly correlate with wider categories such as total methods and fan-out, providing additional insight into the systemic nature of quality and security aspects. These findings support the concept that technical and security debts are related.

The dataset's descriptive statistics show significant variety across technical debt and

Figure 5.22: Correlation matrix for Class-Level Analysis

security-related indicators, reflecting the diversity of the software projects evaluated. Metrics like CBO (mean: 8.51, max: 1783) and WMC (mean: 17.13, max: 47,551) have large distributions, showing significant variances in coupling and complexity levels between software components. Security-related density metrics, such as resource handling issue density (mean: 0.0107, max: 1.0) and exception handling issue density (mean: 0.0194, max: 1.0), have generally low averages but occasional extreme values, indicating that critical vulnerabilities exist in only a small subset of cases. The high technical debt probability (mean: 0.109, maximum: 1.0) and high technical debt class density highlight the difficulties in maintaining code quality. These findings highlight the necessity for systematic approaches to addressing both technical debt and security vulnerabilities, with an emphasis on outlier-prone areas to improve overall software reliability and maintainability.

5.4.3 Predicting Security Issues & High TD Probability

Linear regression was used to examine the correlation between high technical debt (`high_td_proba`) and total number of issues (`total_issues`). Figure 5.23 shows the regression findings, with `high_td_proba` as the predictor variable for `total_issues`. A slight positive association (R-squared = 0.385) indicates that the risk of significant technical debt can explain approximately 38.5% of the variance in total difficulties.

Figure 5.24 illustrates the reverse regression, which uses `total_issues` to predict `high_td_proba`. This study produces a comparable R-squared value and a mean squared error of 0.032.

The projected regression lines in both models show general patterns. However, the wide range of values demonstrates significant variability, emphasizing the complexity of the processes causing technical debt and its accompanying hazards. These findings highlight

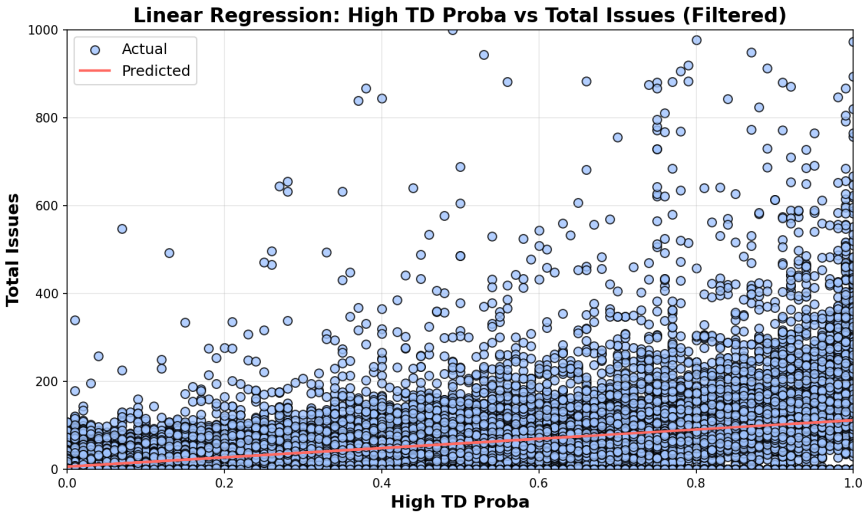

Figure 5.23: Predicting total issues with high TD probability

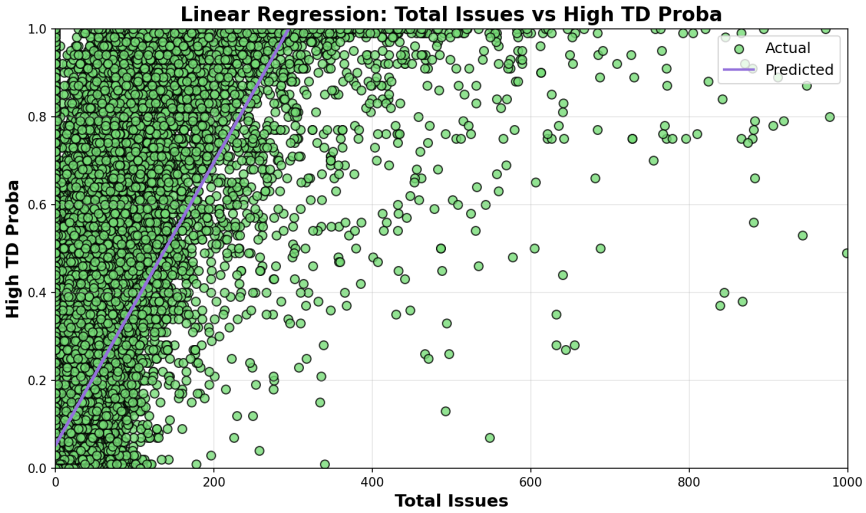

Figure 5.24: Predicting high TD probability with total issues

the linked nature of software quality measurements and provide avenues for further research using more complicated modeling methodologies.

5.4.4 Outlier removal

Following the removal of outliers, statistical examination of major software quality and security measures gives considerable insights into their distributions and variability. Metrics like coupling between objects (CBO) and weighted methods per class (WMC) have a wide range, with maximum values of 23 and 38, respectively, indicating possible hotspots of complexity. Similarly, the response for a class (RFC) reaches a maximum of 50, indicating locations where significant inter-class communication may cause maintenance issues. Notably, cohesiveness (LCOM) ranges from 0 to 37, representing different levels of class organization, but the cohesion metric LCOM3 goes from 0.0 to 1.0, indicating differing structural integrity across the codebase.

Variables such as total methods (19) and nested blocks (5) highlight the variability of the investigated system. On the security front, assignment concerns dominate with a maximum count of 32 and a density of 1.0, while other vulnerabilities, such as misused functionality, appear infrequently but noticeably. Interestingly, crucial measures such as high technical debt probabilities (high_td_proba) remain relatively low, with a maximum of 0.12, highlighting the system's resiliency. These findings establish the groundwork for further investigation into the relationship between quality and security measurements, as well as their impact on software reliability.

5.4.5 Correlations between metrics

Certain metrics have nearly perfect correlations, indicating that they cover similar features of software structure. For example, Fanout and CBO have a correlation coefficient of 0.999, indicating high overlap and maybe pointing to duplication in how these metrics quantify structural complexity. Both measures have strong connections with RFC, NCLOC, and Total Lines, highlighting their link to codebase complexity and size. Similarly, security-related metrics indicate significant correlations between issues and their corresponding densities, such as logging_issues and logging_issues_density (0.999), or null_pointer_issues and null_pointer_issues_density (0.998), which is expected given that densities are derived metrics. The excellent correlation (0.971) between Total Lines and NCLOC indicates that raw and normalized metrics of code size are in agreement.

Furthermore, Total Issues has a significant correlation with individual security measures such as Assignment Issues (0.971) and Misused Functionality Issues (0.712), demonstrating its efficacy at aggregating subcategories. Finally, the high correlation (0.890) between Total Methods and WMC indicates that the latter is significantly influenced by the amount of methods in the code.

Moderate correlations give light on the subtle interactions between measurements. High technical debt (TD) indicators, such as High TD Proba and HTD, have moderate relationships with complexity metrics like WMC, RFC, and NCLOC. This implies that technical debt is inextricably linked to both structural and logical complexity. Metrics like Fanin and Fanout show moderate correlations with Total Issues and Assignment Issues, indicating a link between dependencies and error rates. Cohesion-related measures, particularly LCOM, have moderate associations with High TD Proba (0.459) and LCOM3 (0.461), demonstrating their impact on cohesion and maintainability. Finally, Max Nested Blocks has a moderate correlation with high TD metrics, complexity measurements, and security problems, indi-

cating that deeply nested structures may present challenges in various dimensions, including maintainability and risk.

5.4.6 Correlation Grouping and Categorization

The results of the class-level study shed light on the relationship between Total Issues (Security) and High Technical Debt (TD) Probability, with diverse patterns seen at various levels. The Spearman correlation coefficient for the lower threshold dataset based on the class size (21 lines) is -0.178, showing a weak negative link with statistical significance (p-value = 0.000). This shows that for classes below the lower threshold, increases in quality metrics are correlated with a reduction in security-related concerns, albeit to a limited extent.

In comparison, the upper threshold dataset shows a substantially greater positive connection, with a Spearman coefficient of 0.757 (p-value: 0.000), which is the strongest result in the analysis. This suggests that for classes that exceed the upper threshold (21), higher-quality measures are substantially associated with improved security results. These findings highlight that at a certain quality threshold, the alignment between good class quality and increased security becomes far more obvious, emphasizing the need of maintaining high-quality standards in these situations.

Further investigation of the probabilistic and line-based metrics reveals consistent patterns. In the lower threshold (0.06) dataset, the correlation between Total Issues and High TD Probability is moderately positive (0.192, p-value: 0.000), as is the correlation with Total Issues (0.190, p-value: 0.000). These findings suggest that even at the lowest end of the quality continuum, small increases in class-level quality indicators are related with improved security performance. The correlations in the upper threshold dataset grow dramatically, with High TD Probability at 0.361 and total lines at 0.344 (both p-values: 0.000). This

supports the notion that higher-quality classes have much better security outcomes as quality standards improve.

In conclusion, the analysis shows a strong trend in which gains in class quality indicators correlate favorably with better security outcomes, especially in higher-threshold datasets. The greater associations identified at the upper level emphasize the need of meeting and maintaining high-quality requirements in order to maximize security performance. These findings call for focused quality assurance efforts, particularly for classes that exceed critical quality levels, to capitalize on their potential to improve security.

5.5 Cross-Subset Analysis

This section covers quality metrics and issue distributions for three distinct groups of classes: Bad Classes, Insecure Classes, and Large Classes. The purpose of this analysis is to find patterns and correlations between various quality parameters and their impact on software dependability and maintainability.

Figure 5.25 displays the mean values for complexity-related metrics, such as CBO, WMC, DIT, RFC, and LCOM, over three subsets. The findings show that Bad Classes and Large Classes have consistently higher values for CBO and RFC, implying that these classes are more interrelated and potentially more difficult to manage. Furthermore, the LCOM measure is much higher in both Bad and Large Classes, indicating weaker cohesiveness that may affect code readability and maintainability. In comparison, Insecure Classes have slightly lower values across the majority of complexity measures. This observation may signal that while these classes are smaller and less complex, they are nevertheless vulnerable to other influences.

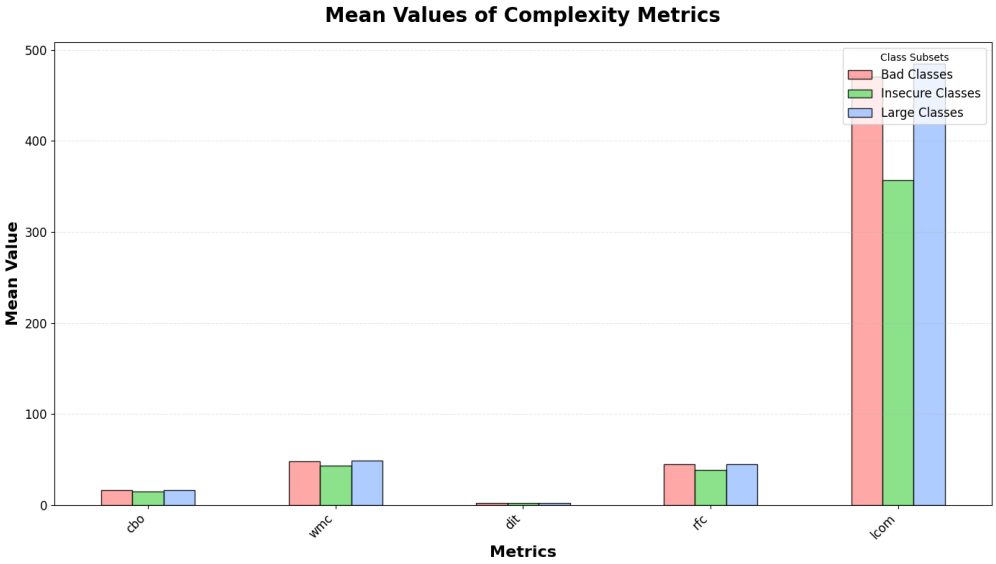

Figure 5.25: Mean Values of Complexity Metrics

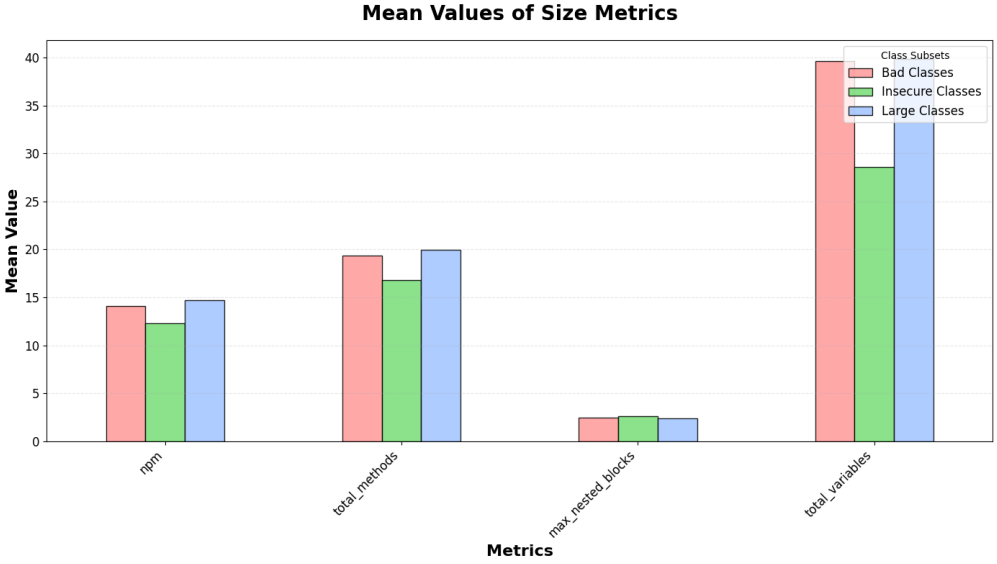

Figure 5.26: Mean Values of Size Metrics

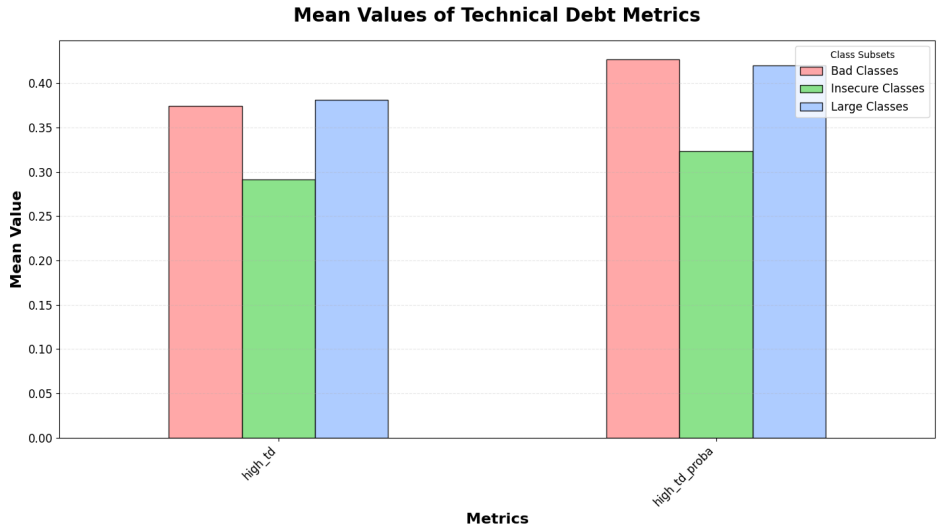

Figure 5.27: Mean Values of Technical Debt Metrics

Figure 5.26 shows a comparison of size-related metrics, such as NPM, Total Methods, Max Nested Blocks, and Total Variables. Large Classes, as expected, dominate these measures, particularly Total Variables and Total Methods, owing to their greater size and complexity. Bad Classes also have relatively high values in these metrics, supporting the link between technical debt and larger class sizes. Insecure Classes are normally smaller, but they contain a significant number of nested blocks, which may increase code complexity and contribute to security problems.

Figure 5.27 shows a comparison of High TD and High TD Probability across subsets. The findings show that Bad and Large Classes have much larger risks of technical debt than Insecure Classes. This demonstrates that technical debt is more common in larger and more sophisticated classes. Interestingly, Insecure Classes have lower probability of technical debt, indicating efforts to reduce debt, despite the fact that these classes are more vulnerable to security flaws.

Figure 5.28 depicts the distribution of several issue kinds, including Resource Handling Issues, Assignment Issues, and Exception Handling Issues, across subsets. Assignment

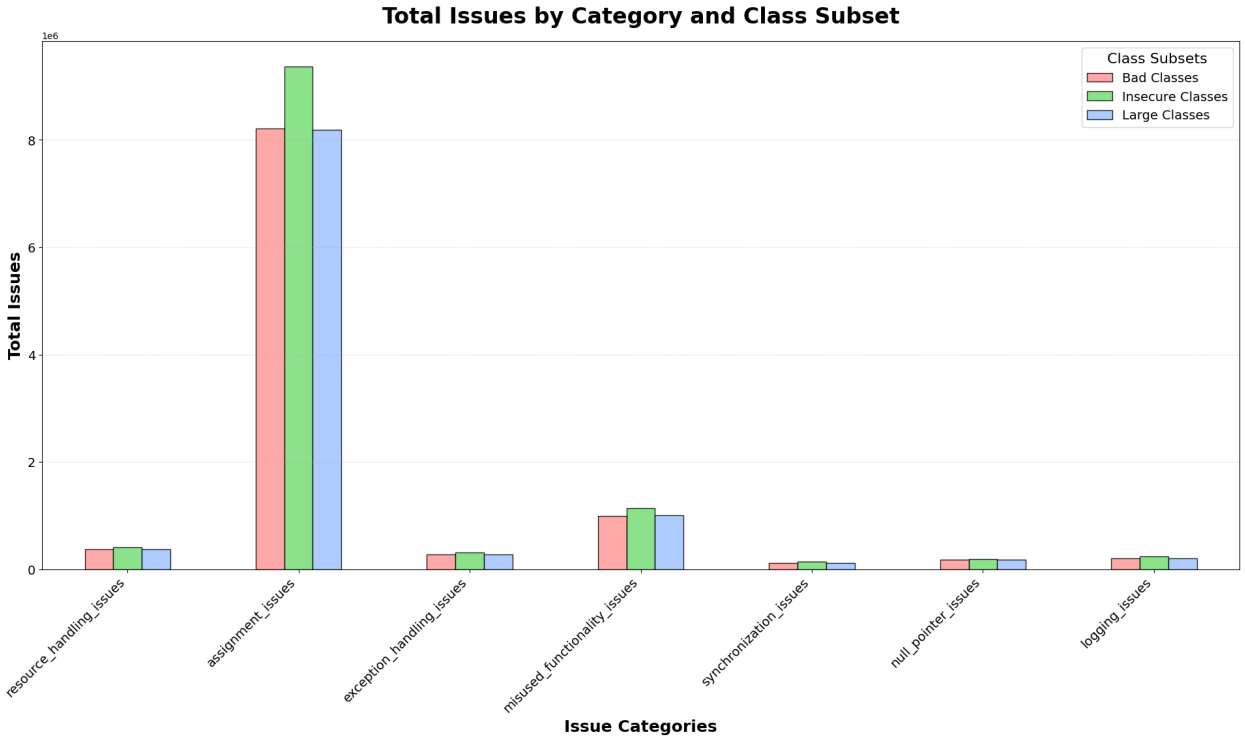

Figure 5.28: Total Security issues by Category and Class Subset

issues are the most common sort of issue across all subgroups, with Insecure Classes having the greatest count. This discovery shows potential issues with keeping accurate variable assignments in these classes, which could lead to vulnerabilities. Furthermore, Misused Functionality and Logging Issues are more common in Bad and Large Classes, most likely due to poor development techniques in larger, interconnected codebases. In contrast, Resource Handling difficulties and Synchronization Issues stay reasonably low across all subgroups, demonstrating that these sorts of difficulties are less affected by class size, complexity, or technical debt.

The study provides numerous critical findings on the relationship between technical debt, quality indicators, and security. Bad and Large Classes are more prone to higher complexity, larger sizes, and more technical debt, emphasizing the importance of targeted refactoring efforts in such classes. Insecure Classes, although being smaller and less com-

plex, have more security-related difficulties, emphasizing the significance of targeted security measures even in smaller classes. Finally, the prevalence of Assignment Issues across all subgroups demonstrates a systemic challenge in sustaining flexible assignment procedures, indicating a potential area for improvement in software development processes.

5.6 Final Results and Discussion

Analysis Level	**Metric/Variable**	**Correlation Coefficient**	**P-Value**	**Key Insight**
Project Level	HTD and Security Index	-0.2773	0.000	Weak negative correlation between HTD and SI
Project Level	sam_misused_functionality_density and HTD	-0.4057	0.000	Moderate negative correlation
Project Level	sam_null_pointer_eval and HTD	-0.4229	0.000	Strong negative correlation
Project Level (Low HTD)	HTD and Security Index	-0.305	0.000	Moderate negative association for projects with low HTD
Time-Series (Guava Project)	HTD and Security Index (Over Time)	-0.5941	0.0152	Substantial negative correlation over time
Class Level	HTD Probability and Overall Issues	0.757	0.000	Strong positive correlation for high-security issue classes
Class Level	Class Size and Security-related Technical Debt	0.344	0.000	Moderate positive correlation for large classes
Class Level	HTD Probability and Security Concerns	0.361	0.000	Moderate positive correlation for HTD probability classes

Table 5.1: Summary of Correlation Analysis Results

The findings of this study provide a thorough understanding of the correlations between High Technical Debt (HTD) metrics, the SI, and other software quality indicators at the project and class levels. The data show various notable patterns and correlations, emphasizing the intricate relationship between software quality and software security.

At the project level, the Spearman correlation matrix indicates a weak negative correlation (-0.2773) between HTD and the Security Index, meaning higher technical debt is associated with lower security performance. This pattern was further investigated in a subset analysis, in which specific measurements revealed significant connections. For instance, `sam_misused_functionality_density` and `sam_null_pointer_eval` had moder-

ate to strong negative correlations with HTD (-0.4057 and -0.4229, respectively), whereas `sam_total_issues` had a positive correlation (0.4797), indicating that an increase in total issues correlates with higher technical debt. Projects with low HTD (<0.13) show a moderately negative association between HTD and SI (-0.305), indicating a link between lower technical debt and improved security.

A time-series investigation of the Guava project revealed further insights, including a substantial negative correlation (-0.5941, p=0.0152) between HTD and the Security Index. This trend implies that reducing HTD over time can improve the overall security of the project. The temporal dynamics seen in this research highlight the significance of monitoring and lowering technical debt on a constant basis to improve security outcomes.

Class-level analysis provided important insights in various circumstances. For classes with high overall security issues (above 21), a strong positive connection (0.757, p=0.000) was found between HTD Probability and overall Issues, indicating that classes with large security difficulties have high technical debt probabilities. Similarly, for large classes (above 192 lines), a moderate positive association (0.344, p=0.000) was discovered, underlining the link between class size and security-related technical debt. Classes with a high HTD Probability (>0.06) showed a moderate positive connection (0.361, p=0.000) with security concerns, indicating that low-quality classes are more vulnerable to security flaws.

In conclusion, these findings show that technical debt and software security are inextricably intertwined, with larger levels of technical debt being associated with lower security performance across a variety of standards. The findings emphasize the need of lowering technical debt, particularly in high-security-risk classes, big classes, and over time at the project level, in order to achieve better software quality and security.

Chapter 6

Conclusions

6.1 Summary

This study clarifies the complex relationship between software security and technical debt in open-source projects, providing important insights into the ways in which these variables interact. The research found important trends, even if there was no clear association between High Technical Debt Density (HTD) and the Security Index (SI) as a whole. The difficulties of managing big, complicated software systems were highlighted by the discovery that higher codebase size and complexity were specifically associated with higher risks of technical debt and security flaws.

The findings highlight the necessity of proactively managing technical debt, especially in projects with higher accumulation risks, which are related with increased security vulnerabilities. Organizations that manage technical debt methodically can reduce possible security vulnerabilities, increase maintainability, and improve overall software quality. This paper emphasizes the interconnection of software quality and security, arguing for an integrated development approach that prioritizes both aspects in order to provide strong, secure, and dependable software systems.

6.2 Suggestions for Future Research

The study's findings suggest several intriguing paths for further research, particularly in determining the causal links between technical debt variables and software security. While this study identified associations, a better understanding of causality is required. Future research could look into how specific elements of technical debt, such as code duplication, class coupling, or inadequate documentation, directly impact security. Researchers can provide actionable insights to developers by pinpointing the precise mechanisms that drive the formation of vulnerabilities, allowing them to prioritize technical debt management solutions that effectively minimize security risks.

Another area to investigate is the growth of metrics and study contexts. This study focused on a specific set of measures and how they may be used to open-source software projects. Future study could build on this by using more technical debt and security measures to capture a broader variety of factors influencing these dimensions. Metrics relating to team dynamics, code review methods, or dependency management, for example, could provide further information about the relationship between technical debt and security. Furthermore, replicating similar studies within commercial software systems may validate and extend the findings outside the open-source ecosystem, providing a more complete knowledge of these linkages across diverse development settings.

Future research should look on the effectiveness of tailored approaches for managing technological debt and minimizing security risks. Practical research assessing the influence of tactics like as refactoring, code modularization, and the use of automated testing frameworks on technical debt and security outcomes would be extremely beneficial. Such studies could provide real information to developers and project managers about the best techniques for risk mitigation in projects with a high probability of technical debt. Future research can provide a toolkit of proven ways for enhancing code quality and security by evaluating the

performance of these interventions in various contexts.

The creation of predictive models based on technical debt measures is another interesting route. These models could assist teams predict the likelihood of security problems in software systems, allowing them to address them proactively. Incorporating variables like team size, development speed, project maturity, and code ownership dynamics into these models may improve their forecasting accuracy. Predictive modeling has the potential to improve technical debt management by allowing developers and stakeholders to foresee vulnerabilities before they occur, resulting in more secure and reliable software systems.

Finally, the creation of real-time monitoring and adaptive management systems provides a promising avenue for future study in technological debt management and security. These systems could use machine learning algorithms to continuously monitor the progression of technical debt metrics and security threats, resulting in dynamic insights and predictive analytics. By incorporating such tools into CI/CD pipelines, teams might receive instant feedback during the development process, allowing them to fix problems proactively. This strategy not only saves time and money on post-deployment fixes, but it also improves overall program quality. Furthermore, adaptive management systems may provide specialized remediation solutions depending on project-specific parameters such as resource availability, development objectives, and the present status of the codebase. By incorporating these solutions into modern agile and DevOps methods, this research direction holds the potential to revolutionize technical debt and security management in software development.

References

[1] M. Siavvas, D. Tsoukalas, M. Jankovic, *et al.*, "An Empirical Evaluation of the Relationship between Technical Debt and Software Security", en,

[2] M. Siavvas, D. Tsoukalas, M. Jankovic, D. Kehagias, and D. Tzovaras, "Technical debt as an indicator of software security risk: A machine learning approach for software development enterprises", *Enterprise Information Systems*, vol. 16, no. 5, p. 1 824 017, 2022.

[3] B. Kitchenham, "Software metrics", in *Software Reliability Handbook*, Elsevier, 1989.

[4] A. Gillies, *Software Quality: Theory and Management*. Lulu.com, 2011, ISBN: 9781446753989. [Online]. Available: https://books.google.gr/books?id=XTvpAQAAQBAJ.

[5] H. Gomaa, "Software quality attributes", in *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. Cambridge University Press, 2011, pp. 357–368.

[6] R. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* (Robert C. Martin Series). Pearson Education, 2008, ISBN: 9780136083252. [Online]. Available: https://books.google.gr/books?id=_i6bDeoCQzsC.

[7] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou, and P. Avgeriou, "Can clean new code reduce technical debt density?", *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1705–1721, 2022. DOI: 10.1109/TSE.2020.3032557.

[8] ISO/IEC, "Iso/iec 25010 - systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models", *ISO/IEC*, 2011.

[9] W. Cunningham, "The wycash portfolio management system", *ACM Sigplan Oops Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[10] N. Brown, Y. Cai, Y. Guo, *et al.*, "Managing technical debt in software-reliant systems", in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 47–52, ISBN: 9781450304276. DOI: 10.1145/1882362.1882373. [Online]. Available: https://doi.org/10.1145/1882362.1882373.

[11] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management", *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.12.027. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214002854.

[12] C. Seaman and Y. Guo, "Chapter 2 - measuring and monitoring technical debt", in ser. Advances in Computers, M. V. Zelkowitz, Ed., vol. 82, Elsevier, 2011, pp. 25–46. DOI: https://doi.org/10.1016/B978-0-12-385512-1.00002-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123855121000025.

[13] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review", *Information and Software Technology*, vol. 64, pp. 52–73, 2015, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.04.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915000762.

[14] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection", *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016, ISSN: 1573-7616. DOI: 10.1007/s10664-015-9378-4. [Online]. Available: https://doi.org/10.1007/s10664-015-9378-4.

[15] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system", in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ser. ISESE '06, Rio de Janeiro, Brazil: Association for Computing Machinery, 2006, pp. 8–17, ISBN: 1595932186. DOI: 10.1145/1159733.1159738. [Online]. Available: https://doi.org/10.1145/1159733.1159738.

[16] T. Chaikalis and A. Chatzigeorgiou, "Forecasting java software evolution trends employing network models", *IEEE Trans. Softw. Eng.*, vol. 41, no. 6, pp. 582–602, Jun. 2015, ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2381249. [Online]. Available: https://doi.org/10.1109/TSE.2014.2381249.

[17] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, *et al.*, "An overview and comparison of technical debt measurement tools", *Ieee software*, vol. 38, no. 3, pp. 61–71, 2020.

[18] SonarSource, *Sonarqube: Continuous code quality*, https://www.sonarsource.com/products/sonarqube/, Accessed: 2024-07-15, 2023.

[19] Vector, *Squore: Software quality analysis and management*, https://www.vector.com/at/en/products/products-a-z/software/squore/, Accessed: 2024-07-15, 2023.

[20] CodeMRI, *Codemri: Software analysis for better code quality*, https://codemri.com/, Accessed: 2024-07-15, 2023.

[21] SAP, *Technical debt*, https://help.sap.com/doc/saphelp_nw73ehp1/7.31.19/en-US/49/205531d0fc14cfe10000000a42189b/content.htm?no_cache=true, Accessed: 2024-07-15, 2024.

[22] Y. Cai and R. Kazman, "Dv8: Automated architecture analysis tool suites", in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019, pp. 53–54. DOI: 10.1109/TechDebt.2019.00015.

[23] SymfonyInsight, *Symfonyinsight*, https://insight.symfony.com/, Accessed: 2024-07-15.

[24] D. Tsoukalas, A. Chatzigeorgiou, A. Ampatzoglou, N. Mittas, and D. Kehagias, "Td classifier: Automatic identification of java classes with high technical debt", in *Proceedings of the International Conference on Technical Debt*, 2022, pp. 76–80.

[25] T. Amanatidis, N. Mittas, A. Moschou, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis, "Evaluating the agreement among technical debt measurement tools: Building an empirical benchmark of technical debt liabilities", *Empirical Software Engineering*, vol. 25, no. 5, pp. 4161–4204, Sep. 2020, ISSN: 1573-7616. DOI: 10.1007/s10664-020-09869-w. [Online]. Available: https://doi.org/10.1007/s10664-020-09869-w.

[26] C. Marantos, M. Siavvas, D. Tsoukalas, *et al.*, "Sdk4ed: One-click platform for energy-aware, maintainable and dependable applications", in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 981–986.

[27] G. McGraw, *Software Security: Building Security in* (Addison-Wesley professional computing series). Addison-Wesley, 2006, ISBN: 9780321356703. [Online]. Available: https://books.google.gr/books?id=HCQdypbpZXgC.

[28] S. Barnum and G. McGraw, "Knowledge for software security", *IEEE Security & Privacy*, vol. 3, no. 2, pp. 74–78, 2005.

[29] I. McAfee, *Threats to information security: Today's reality, tomorrow's agenda*, https://www.marylandnonprofits.org/wp-content/uploads/mcafee.pdf, Accessed: 2024-07-15, 2003.

[30] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford, "Security during application development: An application security expert perspective", in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.

[31] H. Anderson, "Introduction to nessus", *Retrieved from Symantec*, 2003.

[32] M. U. Aksu, E. Altuncu, and K. Bicakci, "A first look at the usability of openvas vulnerability scanner", in *Workshop on usable security (USEC)*, 2019.

[33] A. Rahman, K. R. Kawshik, A. A. Sourav, and A. Gaji, "Advanced network scanning", *American Journal of Engineering Research (AJER)*, vol. 5, no. 6, pp. 38–42, 2016.

[34] J. Shahid, M. K. Hameed, I. T. Javed, K. N. Qureshi, M. Ali, and N. Crespi, "A comparative study of web application security parameters: Current trends and future directions", *Applied Sciences*, vol. 12, no. 8, 2022, ISSN: 2076-3417. DOI: 10.3390/app12084077. [Online]. Available: https://www.mdpi.com/2076-3417/12/8/4077.

[35] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks", *Advances in neural information processing systems*, vol. 32, 2019.

[36] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?", *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[37] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction", in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[38] V. Verendel, "Quantified security is a weak hypothesis: A critical survey of results and assumptions", in *Proceedings of the 2009 Workshop on New Security Paradigms Workshop*, ser. NSPW '09, Oxford, United Kingdom: Association for Computing Machinery, 2009, pp. 37–50, ISBN: 9781605588452. DOI: 10.1145/1719030.1719036. [Online]. Available: https://doi.org/10.1145/1719030.1719036.

[39] S. Zafar, M. Mehboob, A. Naveed, and B. Malik, "Security quality model: An extension of dromey's model", *Software Quality Journal*, vol. 23, no. 1, pp. 29–54, Mar. 2015, ISSN: 0963-9314. DOI: 10.1007/s11219-013-9223-1. [Online]. Available: https://doi.org/10.1007/s11219-013-9223-1.

[40] M. Siavvas, D. Kehagias, D. Tzovaras, and E. Gelenbe, "A hierarchical model for quantifying software security based on static analysis alerts and software metrics", *Software Quality Journal*, vol. 29, no. 2, pp. 431–507, 2021.

[41] T. L. Saaty, "Decision making with the analytic hierarchy process", *International Journal of Services Sciences*, vol. 1, no. 1, pp. 83–98, 2008.

[42] W. Edwards and F. H. Barron, "Smarts and smarter: Improved simple methods for multiattribute utility measurement", *Organizational behavior and human decision processes*, vol. 60, no. 3, pp. 306–325, 1994.

[43] J. T. Force, "Security and privacy controls for information systems and organizations", National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-53, Sep. 2020, Includes updates as of 12-10-2020. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-53r5.

[44] J. Andress, *The basics of information security: understanding the fundamentals of InfoSec in theory and practice*. Syngress, 2014.

[45] S. Wagner, A. Goeb, L. Heinemann, *et al.*, "Operationalised product quality models and assessment: The quamoco approach", *Information and Software Technology*, vol. 62, pp. 101–123, 2015, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.02.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915000452.

[46] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability", in *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8.

[47] M. Siavvas, D. Kehagias, and D. Tzovaras, "A preliminary study on the relationship among software metrics and specific vulnerability types", in *2017 International Confer-*

ence on Computational Science and Computational Intelligence (CSCI), 2017, pp. 916–921. DOI: 10.1109/CSCI.2017.159.

[48] J. Walden and M. Doyle, "Savi: Static-analysis vulnerability indicator", *IEEE Security & Privacy*, vol. 10, pp. 32–39, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:25616204.

[49] J. Walden, M. Doyle, G. A. Welch, and M. Whelan, "Security of open source web applications", in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 545–553. DOI: 10.1109/ESEM.2009.5314215.

[50] S. Wagner, K. Lochmann, L. Heinemann, *et al.*, "The quamoco product quality modelling and assessment approach", in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1133–1142. DOI: 10.1109/ICSE.2012.6227106.

[51] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, "Software quality models: Purposes, usage scenarios and requirements", in *2009 ICSE Workshop on Software Quality*, 2009, pp. 9–14. DOI: 10.1109/WOSQ.2009.5071551.

[52] D. Hovemeyer and W. Pugh, "Finding bugs is easy", *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004, ISSN: 0362-1340. DOI: 10.1145/1052883.1052895. [Online]. Available: https://doi.org/10.1145/1052883.1052895.

[53] K. Rindell, K. Bernsmed, and M. G. Jaatun, "Managing security in software: Or: How i learned to stop worrying and manage the security technical debt", in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19, Canterbury, CA, United Kingdom: Association for Computing Machinery, 2019, ISBN: 9781450371643. DOI: 10.1145/3339252.3340338. [Online]. Available: https://doi.org/10.1145/3339252.3340338.

[54] K. Rindell and J. Holvitie, "Security risk assessment and management as technical debt", in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 2019, pp. 1–8. DOI: 10.1109/CyberSecPODS.2019.8885100.

[55] C. Izurieta, D. Rice, K. Kimball, and T. Valentien, "A position study to investigate technical debt associated with security weaknesses", in *2018 International Conference on Technical Debt*, Gothenburg, Sweden, 2018.

[56] C. Izurieta and M. Prouty, "Leveraging secdevops to tackle the technical debt associated with cybersecurity attack tactics", in *Proceedings of the Second International Conference on Technical Debt*, ser. TechDebt '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 33–37. DOI: 10.1109/TechDebt.2019.00012. [Online]. Available: https://doi.org/10.1109/TechDebt.2019.00012.

[57] D. Tsoukalas, A. Chatzigeorgiou, A. Ampatzoglou, N. Mittas, and D. Kehagias, *Td classifier: Automatic identification of java classes with high technical debt*, https://sites.google.com/view/ml-td-identification/home, Based on the work: Tsoukalas, D., Chatzigeorgiou, A., Ampatzoglou, A., Mittas, N., Kehagias, D. TD classifier: Automatic identification of Java classes with high technical debt., 2025.

[58] M. Siavvas, D. Kehagias, and D. e. a. Tzovaras, *Open-source tool based on the sam model*, https://sites.google.com/view/sec-model-supp, Based on the work: Siavvas, M., Kehagias, D., Tzovaras, D. et al. A hierarchical model for quantifying software security based on static analysis alerts and software metrics. Software Qual J 29, 431–507 (2021). https://doi.org/10.1007/s11219-021-09555-0, 2021.

[59] N. Brown, Y. Cai, Y. Guo, *et al.*, "Managing technical debt in software-reliant systems", in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 47–52.