Universidade do Minho
Escola de Engenharia

Ana Maria Carvalho Balsa

Scan Metrics for Static Application Security Testing (SAST)

October 2023

Scan Metrics for Static Application Security Testing (SAST)
Ana Balsa
UMinho | 2023

Universidade do Minho
Escola de Engenharia

Ana Maria Carvalho Balsa

Scan Metrics for Static Application Security Testing (SAST)

Master's Dissertation Report
Integrated Master's in Engineering and Management of Information Systems

Work performed under the supervision of
Professor Henrique Santos

October 2023

COPYRIGHT AND TERMS OF USE OF THE WORK BY THIRD PARTIES

ACKNOWLEDGEMENTS

This document is proof that I completed the five years of MIEGSI. The journey to get here was long and challenging, but with the right support, it was accomplished. Thus, I would like to recognize those who have supported me throughout this journey.

First of all, I would like to express my gratitude to Professor Henrique Santos, who kindly agreed to supervise this dissertation. His straightforward approach and empathetic nature helped me to tackle all the challenges with confidence and make the right decisions throughout this dissertation.

I am grateful to my mentor at Checkmarx, Nuno Oliveira, for his exceptional guidance, knowledge, and support from the beginning. His dedicated effort enabled me to carry out my master's thesis successfully. I also appreciate his review of my entire dissertation. I want to thank my colleagues who generously provided their time and expertise to review specific sections as well. I want to extend special thanks to my team, ACEofSpades, for their flexibility and support in meeting all the deadlines.

I want to thank my closest family members, who have always been there to celebrate my success with the same level of enthusiasm as their own. In particular, thank you, Mother, for all the support over the five years. Her unconditional love and support have been a source of strength for me. Even though she mistakenly referred to the name of my course for three consecutive years, I appreciate her support. I also want to thank my Father with "Saudade", who would have been proud of me, the Master Balsa that was being built over these years.

Finally, I want to express my gratitude to Bruno. He helped me with everything required and handled the moments when the drama queen within me emerged during stressful times. I would also like to extend my thanks to my close friends from Braga and Tabuaço, who played a crucial role in cheering me up throughout this journey, especially on those days when writing this dissertation felt like an endless task. And lastly, I want to thank my coursemates whose friendship made these five years much more enjoyable.

Thank you everyone,

Ana Balsa

ABSTRACT

Scan Metrics for Static Application Security Testing (SAST)

In today's business landscape, safeguarding sensitive data is paramount due to the growing risk of cyber threats. Despite their incredible potential, technologies like AI, 5G, and blockchain come with security challenges that need to be addressed. Security failures can result in substantial losses, emphasizing the need for a standardized definition of cybersecurity. Vulnerability scanning tools like Static Application Security Testing (SAST), integrated into CI/CD processes, help detect code vulnerabilities, enhancing overall software security. The main goal of this dissertation is to establish criteria for assessing the quality of scans performed by SAST tools, with the ultimate goal of enhancing software quality. To achieve this goal, the dissertation will explore various security testing techniques, including SAST tools, identify essential metrics and their relevance across different scan phases, develop a comprehensive formula for quantifying the overall scan quality using these metrics, create techniques for metric extraction, and finally, apply this formula to guide the decisions of Quality Assurance (QA) team during software releases. This research addresses a critical gap in evaluating the quality of SAST scans, which is essential given the increasing demand for high-quality software products. To accomplish this goal, the approach involved the development of a service named CxScanQuality, aimed at evaluating project scan quality based on SAST log files. CxScanQuality integration was planned within a platform responsible for assessing the overall quality of SAST products used by the QA team, along with integration into Continuous Integration (CI) pipelines. To assess scan quality through CxScanQuality, it was essential to identify the set of characteristics contributing to it. These characteristics were segmented into recognition coverage, DOM structure, and query execution. Based on that, raise for each component these quality factors: Coverability, Domability, and Querability. The scan quality was an aggregated metric, representing the sum of these quality factors, each with its specific impact on scan quality. Following this service's integration, the results show that the overall scan quality across the 23 languages and 160 projects is high, with an average score of 89.07%. This master's dissertation emphasizes that scan quality extends beyond result accuracy. These findings are precious for the QA team, as they provide relevant data on scan quality for all projects in the organization. These results also introduce a new method for ensuring the quality of the SAST product, ultimately contributing to enhanced software quality.

Keywords: SAST, Software Quality, SDLC, Quality Metrics.

RESUMO

Métricas de Scan para Static Application Security Testing (SAST)

No mundo empresarial atual, proteger dados sensíveis é crucial devido ao crescente perigo de ameaças cibernéticas. Tecnologias como Inteligência Artificial, 5G e *blockchain*, embora promissoras, enfrentam desafios de segurança. Falhas na área de desenvolvimento de *software* podem causar perdas monetárias significativas, destacando a necessidade de normas de cibersegurança. Ferramentas de análise estática, como o *SAST*, integradas nos processos de CI/CD, detetam vulnerabilidades no código-fonte, melhorando a segurança do *software*. Posto isto, esta dissertação tem como objetivo principal estabelecer critérios de avaliação da qualidade dos *scans* realizados por ferramentas *SAST*, com o propósito último de aprimorar a qualidade do *software*. Para alcançar este objetivo, a dissertação explorará diversas técnicas de teste de segurança, incluindo ferramentas *SAST*, identificará métricas essenciais e a sua relevância nas diferentes fases de *scan*, conceberá uma fórmula abrangente para quantificar a qualidade global dos *scans* com base nessas métricas, desenvolverá técnicas de extração dessas métricas e, por fim, aplicará essa fórmula para orientar as decisões da equipa de Garantia de Qualidade durante os lançamentos de novas versões do produto *SAST*. A ausência de literatura sobre a avaliação da qualidade dos *scans* do *SAST* foi a motivação central desta dissertação, uma vez que a qualidade destes *scans* é fundamental no contexto do crescente interesse em *software* de alta qualidade. Assim, o serviço *CxScanQuality* foi concebido para avaliar a qualidade dos *scans* com base nos *logs* fornecidos pelo *SAST* após a análise do projeto. Este serviço integra-se em dois pontos cruciais: na plataforma de avaliação geral de produtos *SAST* utilizada pela equipa de Garantia de Qualidade e nas pipelines de CI. Para avaliar a qualidade dos *scans*, identificaram-se três componentes críticas: cobertura de reconhecimento do código-fonte, estrutura *DOM* e execução de consultas por vulnerabilidades. Definiram-se fatores de qualidade para cada componente, a saber, *Coverability*, *Domability* and *Querability*. A qualidade do *scan* é uma métrica agregada, refletindo a soma ponderada desses fatores, cada um com o seu impacto específico na qualidade global do *scan*. Após a implementação deste serviço, os resultados revelaram uma média de 89.07% na qualidade global dos *scans*, um valor considerado positivo para esta métrica. Esta avaliação abrangeu 23 linguagens de programação e 160 projetos. Portanto, esta dissertação destaca que a qualidade dos *scans* de *SAST* não se limita à precisão dos resultados do *SAST*, fornecendo informações relevantes para a equipa de Garantia de Qualidade, abarcando todos os projetos da empresa. Além disso, esses resultados introduzem um novo método para garantir a qualidade dos produtos *SAST*, contribuindo para a melhoria da qualidade do *software*.

Keywords: SAST, Ciclo de Vida do Desenvolvimento de Software, Qualidade do Software.

INDEX

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

SAST	Static Application Security Testing
CI/CD	Continuous Integration and Continuous Delivery
OWASP	Open Web Application Security Project
DOM	Domain Object Model
DSR	Design Science Research
SDLC	Software Development Life Cycle
CWE	Common Weakness Enumeration
SQUARE	Systems and software Quality Requirements and Evaluation
SVT	Security Vulnerability Testing
DAST	Dynamic Application Security Testing
IAST	Interactive Application Security Testing
DevOps	Development and Operations
LOC	Lines of Code
AST	Abstract Syntax Tree
AbsInt	Abstract Interpretation
QA	Quality Assurance
API	Application Programming Interface
KPI	Key Performance Indicator
RegEx	Regular Expression

1 INTRODUCTION

Nowadays, information security is one of the most relevant areas in the business world. Companies increasingly have access to huge amounts of sensitive data. Therefore, they know the importance of securing their data, which is the key to surviving in a competitive business environment. Exploiting vulnerabilities of business systems or any other cyber attack may lead to catastrophic consequences resulting in confidential information losses, financial losses, and reputation damage.

For instance, some technology trends, such as Artificial Intelligence, green technology, 5G networks, or blockchain systems, handle large amounts of data. For example, the architecture of the 5G network adopts an open structure with the advantage of rapid and flexible expansion of network functions. However, the open architecture brings new software security issues and can become a national security issue (Chen et al., 2022). Moreover, smart contract security is an emerging research area that deals with security issues arising from the execution of smart contracts in a blockchain system. Unfortunately, some of these security issues have been reported in the media, often leading to substantial financial losses (Huang et al., 2019). As a result, the potential security failures within these technology trends can result in significant losses for the organizations involved.

That is why, as a crucial first step in mitigating vulnerabilities, defining the concept of cybersecurity becomes imperative. However, this definition is highly variable. So, it is essential to have and use standardized terminology and develop a comprehensive common understanding of what is meant by cybersecurity (Cains et al., 2022). One possible definition describes cybersecurity as the set of resources, processes, and structures to protect cyberspace and cyberspace-enabled systems from any occurrences that disrupt the rightful property rights (Craigen et al., 2014).

To reduce cybersecurity risk in software, the security community has widely adopted an approach involving a collage of techniques, tools, and methods, each addressing some aspect of the threat implications of bad code (Amoroso, 2018). The need for rapid and efficient software development pushes the demand for automation in test, build, and release phases, such as the Continuous Integration and Continuous Deployment (CI/CD) process (Liao, 2020).

Static Application Security Testing (SAST) is a tool that can be integrated into the CI/CD process, enabling developers to detect security vulnerabilities in the source code or compiled code of software. These tools can analyze the source or compiled code and detect any weaknesses that could lead to a security issue. OWASP categorizes these weaknesses into various types, such as input validation and representation (Aloraini et al., 2019).

In conclusion, the demand for high-quality software products from software stakeholders, including developers, managers, and end users, is undeniable. To show the safety of software, it is crucial to have a reliable quality

measure of SAST scans. The literature has not extensively explored the quality metrics associated with SAST scans. Therefore, this dissertation addresses this research gap and evaluates SAST scan quality.

1.1 Motivation

SAST tools play a vital role in detecting and mitigating security vulnerabilities in source code. The scan process in these tools usually involves several phases, including source code recognition, abstract representation creation, and vulnerability pattern detection. These phases have the potential to generate valuable data that can be analyzed to evaluate the quality of a scan. The higher the scan quality, the higher the confidence in the results. Achieving this quality is the main objective of the engineers involved in the development and quality assurance of SAST tools. While several ways exist to assess this quality, looking into metrics during the scans and testing phases can provide accurate insight. However, metrics written to scan logs are often inconsistent or undefined and, when defined, lack standardization, and log files lack structure.

Reviewing the several scan phases can help identify factors in each stage and contribute to understanding the overall scan quality. These factors and how to obtain them can be standardized and systematized. Furthermore, the notion of quality is not only impacted by the phases of a scan but also by other factors, such as the language being scanned. The quality standards for a language can vary based on its importance in the business domain, defined as the language's tier. A language with a higher tier is usually best supported in these tools as it will be in higher demand for customers. Therefore, the scan quality of a tier 1 language must be superior to that of a tier 2 or tier 3 language.

To determine the quality of a scan, various factors must be considered. These include the recognition coverage of the programming language, the structure of the Domain Object Model (DOM) and its syntactic and semantic relationships, and the execution of queries on the DOM without compromising the performance of the scan. By breaking down the quality of a scan into smaller components, such as Coverability, Domability, and Querability, it is possible to understand better and evaluate the properties that contribute to the overall quality of the scan.

This master's dissertation aims to build a notion of scan quality by considering the data extracted in each scan phase. It was proposed by Checkmarx, which operates in the cybersecurity market with products focused on the search for vulnerabilities in software applications.

1.2 Objectives

The main objective of this dissertation is to establish criteria for assessing the quality of scans performed by a SAST tool, with the ultimate aim of achieving high-quality software. To accomplish this, the following specific

objectives will be addressed:

- Compare and analyze various security testing techniques and deepen knowledge of software quality and security.
- Identify key metrics and their weight in each scan phase of a SAST tool.
- Research a formula to quantify overall scan quality, incorporating the metrics and their weights studied in the previous item.
- Develop ways of extracting the metrics from a scan.
- Integrate the created formula to inform software release decisions by the Quality Assurance team.

1.3 Research Method

The research methodology adopted in this dissertation is Design Science Research (DSR). According to Venable and Baskerville (2012), a widely recognized method in Information Systems focuses on inventing and evaluating new artifacts to solve specific problems. The primary goal of DSR is to create new knowledge and innovative artifacts that can positively impact and enhance the world.

The DSR involves six phases: identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication (Peffers et al., 2007). Therefore, these phases will be adapted to the following objectives:

- Identify the research problem and outline the key aspects for resolution.
- Analyze and describe various aspects of vulnerabilities, security testing, continuous integration, and delivery, software development life cycle, and software quality. Details are provided in 2.1.
- Conduct a comprehensive literature review to understand the current state of the art in software quality. Details in 2.2.
- Develop a generic scan quality metric based on data extracted from SAST tools during a scan. Details can be found in 3.2.
- Test and improve the scan quality metric and evaluate the fulfillment of all requirements. Details in 4.1.
- Analysis and discussion of the results. Details in 5.2 and 5.3.

This is an iterative process. Suppose the results achieved in one of the stages are not satisfactory. In that case, it is important to return to one of the previous stages, deepen the literature research, or revise some project decisions.

1.4 Document Structure

This master's dissertation is structured into six chapters, following the methodology adopted. The introduction, motivation, objectives, and research method were covered up to this point. Each chapter includes a "Summary" section, which provides a brief overview of its contents. The "State of the Art" chapter explores the latest developments in the field of scan quality, including explanations of fundamental concepts and a literature review. The "Proposed Approach" chapter outlines the SAST scan process, requirements, architecture, technology stack, and decisions made to achieve scan quality. The "Development" chapter presents all the steps in developing the new service to generate the scan quality and its integration into a quality platform and CI pipelines. It also includes the "Solution Validation" section that outlines which requirements were successfully achieved. Subsequently, the "Results" chapter provides an overview of analyzing and discussing the results obtained. Finally, the "Conclusion" chapter summarizes the dissertation's findings and outlines potential avenues for future work.

2 STATE OF THE ART

This section presents the latest developments in the field of scan quality. It begins with an overview of fundamental concepts and continues with a comprehensive literature review. The literature research used various paper databases, including IEEE Explorer, Google Scholar, Scopus, ACM, and ResearchGate. Search terms such as "security testing techniques and tools", "software security testing", "CI/CD", "CI/CD in software development life cycle", "software development life cycle in SAST", "software development life cycle phases", "CI/CD KPIs", "quality metrics", "software metrics", "security vulnerability testing", "vulnerability detection tool", "benchmarks for SAST", "vulnerability detection with metrics", "vulnerability discovery strategies", "software quality models", and "software quality" were used to extract relevant information. The study period was focused on articles published between 2016 and 2022, but other periods were also included if they had a significant number of citations or relevant information.

Initially, all articles found were considered. However, after evaluating the title and abstract, non-relevant articles were excluded. The introduction and conclusion were also reviewed where necessary. If the paper contained a discussion of the results section, it was also analyzed. Finally, the relevant articles were analyzed to compile the literature review.

2.1 Fundamental Concepts

This section offers a comprehensive overview of the fundamental software quality concepts. It also covers crucial topics, including definitions and practices of vulnerabilities, security testing, continuous integration/continuous delivery (CI/CD) pipelines, and the Software Development Life Cycle (SDLC). This information is crucial for understanding and improving software quality.

2.1.1 Vulnerabilities

A software vulnerability is a weakness in the source code that can be exploited by hackers to cause harm or damage, such as accessing unauthorized data or gaining administrative privileges (Pereira, 2020). To effectively categorize these vulnerabilities and identify patterns, the Common Weakness Enumeration (CWE) was introduced to group similar and related vulnerabilities (Aivatoglou et al., 2021). There are two typical categories of vulnerabilities: implementation-level bugs and design-level flaws (McGraw & Potter, 2004).

Furthermore, understanding the types and categories of vulnerabilities is essential for devising appropriate strategies to prevent their exploitation. Preventing software vulnerabilities can be achieved through a combination of best coding practices and utilizing various vulnerability detection approaches and tools (B. C. Liu et al., 2012). For example, the Security Quality Requirements Engineering (SQUARE) methodology determining and prioritizing

security requirements (Mead et al., 2005); the Open Web Application Security Project (OWASP[1]) offers secure coding practices to prevent the introduction of vulnerabilities during the development process (Pereira et al., 2019); techniques like Static Code Analysis (SCA) or penetration testing can be employed to detect and remediate vulnerabilities once the code is complete (Pereira et al., 2019).

However, even with preventive measures in place, vulnerabilities may persist. Therefore, effective detection techniques and tools are vital to identify and remediate these weaknesses.

Empirical studies play a crucial role in evaluating the efficiency and effectiveness of vulnerability detection techniques. Elder (2021) conducted empirical research to provide decision-makers in software development projects with insightful information on the efficiency and effectiveness of various vulnerability detection techniques. In the study, four techniques were used to detect vulnerabilities in Java Applications: Systematic Manual Penetration Testing (SMPT), Exploratory Manual Penetration Testing (EMPT), Dynamic Application Security Testing (DAST), and Static Application Security Testing (SAST). The choice of technique will depend on the practitioner's specific goals, as each technique detects different vulnerabilities in different phases of the Software Development Life Cycle (SDLC). Finally, the expected contributions of this study include a decision support model to assist practitioners in selecting the most appropriate technique and a dataset of vulnerabilities from two open-source projects, including information on the detection technique used, type of vulnerability, and solution.

Beba et al. (2021) presented a study of three open-source IDE (Integrated Development Environment) plugins for detecting web application security vulnerabilities in Java-based applications using static source code analysis. The authors conducted a root cause analysis of false-negative and false-positive results, identifying factors that can impact plugin performance. Based on their findings, they analyzed over 20,000 vulnerability reports across 11 categories and improved the plugins accordingly. The study also highlights the importance of conducting solid root cause analyses when evaluating security vulnerability detection approaches and tools. Without this, the authors warn that the evaluation and comparisons of these approaches could be misleading. To address this issue, Beba et al. (2021) proposed a guideline for reporting the evaluation results of such approaches.

Furthermore, benchmarking and evaluation of Security Vulnerability Testing (SVT) tools are essential to assess their performance.

Effective vulnerability discovery strategies can help practitioners identify vulnerabilities early in software development. Bhuiyan et al. (2021) conducted a study to assess the effectiveness of four strategies for discovering vulnerabilities in software: diagnostics, malicious payload construction, misconfiguration, and pernicious execution. The authors surveyed 51 practitioners to gather their perceptions of these strategies and found that the

[1]Available at: https://owasp.org/

majority agreed that all four strategies can be used to uncover vulnerabilities. The authors suggest that these results validate previous research on vulnerability discovery strategies and offer guidance for practitioners to identify latent vulnerabilities. Additionally, Bhuiyan et al. (2020) conducted a qualitative and quantitative analysis of Open Source Software (OSS) bug reports over ten years and identified the same four strategies. However, the most commonly used strategy differed between different software projects.

Moreover, Cadariu et al. (2015) presented a case study of the Vulnerability Alert Service (VAS) and its effectiveness in addressing known vulnerabilities in software systems. Through an empirical investigation, the study found that the problem of using components with known vulnerabilities is prevalent in the software industry, with an unknown number of proprietary applications affected by it. The study also found that the VAS is a valuable tool for addressing this problem in the context of external software product quality monitoring, and that means like OWASP Dependency Check [2] can be used in external software product quality assurance. However, the false positive rate may be as high as 70%. The study also conducted interviews with VAS operators and found that the information about using the connector and the overview of vulnerabilities in the application was still considered valuable.

Additionally, Hao et al. (2019) proposed a new method for constructing benchmarks to evaluate SAST tools. The method combines the benefits of real-world software and synthetic micro-benchmarks by extracting representative source code from real-world software and using it to construct test cases with reduced syntactic features and counterexamples. Therefore, the resulting benchmark will contain one original test case, one primary test case, several test case variants, and counterexamples for each vulnerability. The authors demonstrate the effectiveness of this method by applying it to an existing benchmark, generating ten groups of test cases, and evaluating two SAST tools. The resulting evaluation is more explainable, allowing a better understanding of the SAST tool's vulnerability detection capabilities.

Furthermore, Parizi et al. (2018) provided a set of requirements for selecting a candidate benchmark project to be used in the assessment of SVT and detection tools. The authors believe that by using these requirements, individuals and companies can select the most appropriate SVT tool for their projects, and its developers will be motivated to improve their tools to produce sharper results. The authors proposed a list of benchmark requirements. These requirements are intended to be used by newcomers and researchers in software security to assess the performance of off-the-shelf and newly proposed tools. The authors also demonstrate the use of these requirements by applying them to select benchmark candidates from existing open-source projects.

[2] Available at: https://owasp.org/www-project-dependency-check/

2.1.2 Security Testing

The concept of software security refers to designing software that continues to function correctly under malicious attack (McGraw, 2004). The three pillars of software security are applied risk management, software security best practices, and knowledge. The software security best practices include code review using static analysis tools, architectural risk analysis, penetration testing, security testing, abuse case development, and security requirements (McGraw, 2006). However, we will focus on security testing because it relates to software quality.

Testing is an essential component in the assurance process for safety-critical software. Demonstrating that the software meets the required specifications and standards is necessary. It is estimated that most of the effort in developing safety-critical software is dedicated to verification and validation, with testing being used to confirm that the requirements and code structures have been adequately covered (Baker & Habli, 2012).

Security testing requires two distinct approaches - testing security mechanisms to guarantee correct functioning and risk-based security testing, which involves analyzing and replicating potential attacker's methods (McGraw & Potter, 2004). Risk analysis, especially at the Software Development Life Cycle (SDLC) design level, can identify potential security problems and their impact (Verdon & McGraw, 2004).

According to the existing literature, the security testing techniques take into account code reviews, automated static analysis testing, binary code analysis, source, and binary code fault injection, risk analysis, vulnerability scanning, and penetration testing (Al-Ghamdi, 2013). Figure 1 shows each Security Testing Technique.

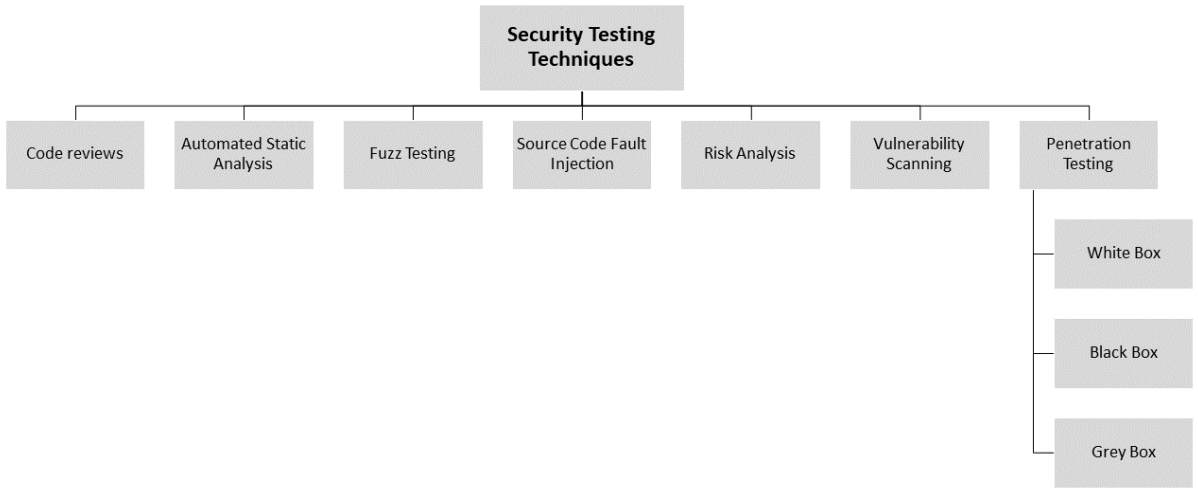


Figure 1: Security Testing Techniques

Code review manually inspects source code to identify security vulnerabilities. It is considered the most reliable technique for finding subtle security issues and is particularly useful for detecting malicious code, implementation

issues, and cryptography weaknesses.

Automated static analysis is a process of analyzing software without executing it, which involves using a static analysis tool to analyze the program's source code or binary executables. This process can be performed iteratively throughout the software's implementation to identify security flaws and potential fixes. Binary code analysis is the process of reverse engineering and analyzing binary code to identify security flaws, with binary scanning, decompilation, and disassembly being the three main techniques used.

Fuzz testing is a powerful technique for finding security-critical flaws in any software. It involves sending random invalid data to the software to see how it responds. These techniques are generally specific to a particular input type and are developed to test a specific program. They cannot be reused and are very effective in uncovering security vulnerabilities that other testing tools cannot find. Fuzz testing can be conducted at various steps during the development and testing and is a beneficial security assessment technique.

Source code fault injection is a testing technique used to induce stress in the software, simulate faults in the execution environment, and reveal safety-threatening flaws. In contrast, binary fault injection is an adjunct to security penetration testing used to monitor system call traces, inject faults into environmental resources, and simulate real-world attack scenarios to gain a complete understanding of the software system's security under all operating conditions.

Risk analysis is a process used during the design phase of software development to review security requirements and identify security risks. It involves analyzing software's potential threats and vulnerabilities and developing strategies to mitigate these risks.

Application vulnerability scanners are software security testing tools used to scan executing applications for patterns associated with known vulnerabilities, including web servers, databases, and operating systems, to detect known classes of attacks and vulnerabilities. However, they are limited to signature-based scanning. They can only detect specific vulnerabilities, so combining different test techniques to examine the software for weaknesses is essential. Penetration testing is used to test the security of a system or network to identify vulnerabilities that an attacker may exploit. It involves simulating an attack on the system and analyzing the results to find any weaknesses that could be used against it. White (e.g., SAST), black (e.g., DAST), and grey box testing (e.g., IAST) are all forms of penetration testing, but each has a different focus. White box testing focuses on the system's internal structure, black box testing focuses on the external interfaces, and grey box testing combines the two approaches. Each approach has advantages, so a comprehensive penetration testing strategy should include all three.

Sheakh (2015) outlines software testing and emphasizes its role in software reliability. The study compares three testing techniques (White box, Grey box, and Black box) and concludes that White box testing produces better results for software reliability. This study highlights the importance of software testing for ensuring the validity and structure of software for efficient performance.

2.1.3 Continuous Integration and Delivery

DevOps incorporates Continuous Integration (CI) and Continuous Delivery (CD) practices to increase the speed and efficiency of software projects (Arachchi & Perera, 2018). DevOps combines concepts and techniques to bring developers and operations teams closer to delivering software faster, more reliably, and with higher quality. It emphasizes collaboration between these two teams throughout the software development process to identify and address potential issues quickly and efficiently (Mowad et al., 2022).

The Continuous Integration stage starts with a commit, followed by a build of the modified application, which is then verified using unit tests. The tested application is deployed to a testing environment when all test cases pass. If Continuous Delivery is implemented, automated acceptance tests are executed to verify no regressions in the system's features. This step also helps to identify any errors that may occur due to a difference in the run-time environment because the testing environment is usually a server with a similar configuration to the production environment. Depending on the level of automation, this step can also involve manual testing and approval before the pipeline advances to the next stage. When all previous stages have passed, the system is automatically deployed to the production environment in the Continuous Deployment stage, where the users will have access to the new version. If a test fails or when an error occurs during build or deployment, the pipeline is automatically stopped, and developers are notified of the error. When a fix has been committed, the pipeline will re-test the entire application (Rangnau, 2020).

2.1.4 Software Development Life Cycle

Software Development Life Cycle (SDLC) is a crucial software and systems development process involving several steps, from initial feasibility studies to deployment and maintenance. There are different models for different types of software, such as back-end, service-oriented, and visual interfaces, and the most popular models are waterfall, spiral, unified, incrementing, rapid application development, v-model and w-model (Ruparelia, 2010).

The waterfall model consists of seven stages: operational analysis, operational specification, design and coding specification, development, testing, deployment, and evaluation. After the operational analysis stage, a preliminary design stage can be added before the analysis stage. At each stage, documents such as a requirements document, initial design specification, interface design specification, final design specification, test plan, and operations manual

or instructions should be produced. Quality assurance is built in by verifying each stage. During the design stage, verification is used to assess the design's suitability. During the development stage, unit and integration testing is performed. During the validation stage, system testing is conducted. After each stage, a feedback loop allows for revisiting previous stages if needed. The model helps create complex software such as relational databases, compilers, and secure operating systems (Ruparelia, 2010).

Software engineers should incorporate security from the start of the software design procedure. Examining and assessing can result in the early recognition of mistakes or issues, quick implementation of appropriate solutions, and, most of all, the manufacture of the safest software item. Addressing, analyzing, and lessening security hazards will enable the creation of totally secure systems surveillance, shielding essential assets, enabling effective security decision-making regulations, developing practical security regulations, and providing relevant data for future forecasting (Alenezi & Almuairfi, 2019).

When discussing risk, it refers to an uncertain event with a certain probability of happening and a certain amount of impact or loss (Alenezi & Almuairfi, 2019). Risk management is a process that helps us identify, address, and avoid risks before they can cause damage. We need to be aware of two broad categories of risks - proactive and reactive. Proactive risks might happen in the future, while reactive risks occur after the software has been released. We can also divide risks into two more categories - systematic and unsystematic. Systematic risks are from external sources like hacking and viruses, while unsystematic risks are unique to the company, such as data loss or misuse and human error (Chowdhury & Arefeen, 2011). Risk management helps us identify, avoid, and minimize the risks so that our software can be secure (Alenezi & Almuairfi, 2019).

The Software Development Life Cycle is a crucial concept software engineers use to produce high-quality software. It involves many steps like Requirements, Architecture and Design, Implementation, Verification, and Release & Maintenance (Alenezi & Almuairfi, 2019), as shown in Figure 2.

In the Requirements phase, software engineers have discussions with users to understand what the user wants the software to do. They use different methods, such as studying existing software and systems, interviewing users, and asking questions to collect information. This information is used to create and ensure the software meets quality standards. Misuse cases are also used to make sure the software is secure. They help identify possible attacks and how the system should respond to them (Alenezi & Almuairfi, 2019).

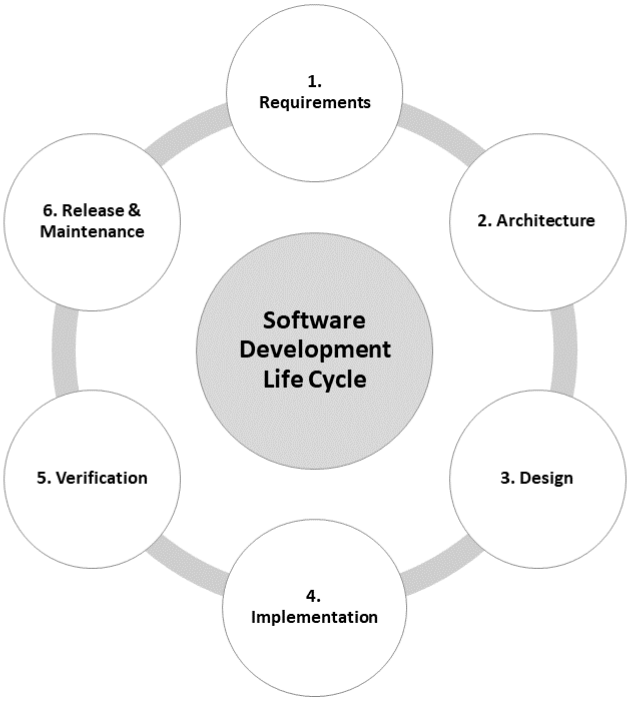


Figure 2: Software Development Life Cycle

The architecture phase involves creating and documenting a high-level structure for a software system. This is analogous to the construction of a building's architecture. Documentation ensures that all stakeholders understand the design, enabling early decisions on high-level design and the reuse of design elements between projects (Imran et al., 2016). Software architects work with project managers, discuss architecturally relevant requirements with stakeholders, formulate software architecture, and evaluate and communicate the architecture. This involves four core activities that are iteratively performed at various software development life cycle stages. Architectural Analysis entails understanding the system's operating environment and determining what is needed to function. Architectural Synthesis is the creation of the architecture, while Architecture Evaluation assesses how well the architecture meets the desired requirements. Lastly, Architecture Evolution deals with maintaining and adapting the architecture to new requirements and environmental changes (Alenezi & Khellah, 2015).

The Design phase combines all the knowledge from the requirements and analysis and uses it to create the design. Different designs, such as object-oriented and functional designs, and various tools, such as data dictionaries, entity-relationship diagrams, and flow diagrams, can all be used in this stage. Addressing the security-related issues in the early phases of SDLC can help reduce the time and cost spent on system security software (Kaur et al., 2018).

During the coding phase, the software development team writes program code using the appropriate programming language and creates error-free executable programs. Integrating software with databases, libraries, and other programs may also be necessary. The team needs expert-level programming skills to produce a high-quality software

product. The coding phase follows standard software engineering practices, such as using version control systems like Git or SVN (stands for Subversion) (Payer, 2019) and adhering to strict coding standards. The code should also go through a formal review process, either manual or automated, as part of the continuous integration process to ensure it meets standards and does not disrupt existing code (Assal & Chiasson, 2018). Various tools, such as GitHub and Gerrit, can aid in this process, and it is essential to evaluate and choose the one that best fits the development process.

The Testing (Verification) phase is a crucial step in the software development process. This phase involves thoroughly testing new commits and releases for functionality and security. Various testing methodologies, such as unit testing, ensure the software is free of mistakes or errors and has high quality (Kaur et al., 2018). Testing experts perform testing at different code levels, including program, object-oriented, module, and product testing at both dynamic and static levels. Security testing is distinct from functional testing as it focuses on abstract properties that are not easily testable. Automatic security testing can be done through fuzz testing, penetration testing, symbolic execution, and formal verification to increase the likelihood of identifying vulnerabilities (Payer, 2019). A dedicated security response team must address any discovered threats or vulnerabilities. This team prioritizes and manages the response to any issues, including changes to the software environment.

The Release and Maintenance phase is the final stage in the Software Development Life Cycle. This phase includes deploying the software and maintaining it over time. A life cycle maintenance approach plans how the software will be updated and modified to meet the changing needs of users and clients. The phase includes installing the software and its components in the production environment and developing a strategy to update and patch the software to address any discovered flaws or vulnerabilities. Ensuring the security of updates is a crucial challenge, requiring a mechanism that can securely check for updates while minimizing the load on update servers (Payer, 2019). During the maintenance phase, the software is further optimized, and new features are added over time.

In summary, it is essential to integrate security best practices throughout the software development cycle by conducting risk assessment at the start of the process, continuing to assess risk throughout each phase of the cycle, and effectively managing any issues or errors that are detected to meet all requirements for secure software development and satisfy clients and users (Alenezi & Almuairfi, 2019).

2.1.5 Software Quality

Software Quality Engineering (Kan, 2002) is focused on enhancing the methods used to ensure software quality. In the software industry, it is widely accepted to utilize software quality models for managing the quality of software systems (Ihirwe et al., 2022). A quality model can be defined as a group of sub-characteristics and their interrelationships that form the foundation for establishing quality standards and assessing quality. Such models

are typically used for specifying quality requirements and evaluating the quality of software products throughout the software development life cycle. However, due to the diverse perspectives encountered throughout the software development process, there can be disagreements regarding what constitutes software quality (Ihirwe et al., 2022).

Software quality has several definitions in the literature. However, it can be divided into two categories (Yang et al., 2012). The IEEE standard offers two possible definitions of software quality (Radatz, 1990). The first definition is the degree to which a system, component, or process meets specified requirements. In contrast, the second definition is the degree to which a system, component, or process meets customer or user needs or expectations. These definitions highlight the importance of meeting technical requirements and satisfying customer needs regarding software quality (Yang et al., 2012).

According to Yang et al. (2012), some well-known quality models are McCall, Boehm, FURPS, and Dromey. ISO 9126-1 is an international standard based on those quality models and was updated in 2011 to ISO 25010, also known as Systems and software Quality Requirements and Evaluation (SQuaRE). The revised standard includes seven new sub-characteristics: functional completeness, capacity, user error protection, accessibility, availability, modularity, and reusability.

The McCall model, proposed in 1977 (McCall et al., 1977), is one of the earliest models for evaluating software quality. It defines product quality from three perspectives: Product operation, Product revision, and Product transition. Under each perspective, quality attributes are defined as a hierarchy of quality factors, quality criteria, and quality metrics. Factors are user-oriented, and criteria are software-oriented. The model does not include system or design elements.

The Boehm model, presented in 1978, is a hierarchical model that aims to qualitatively define software quality by a given set of attributes and metrics. The model is similar to the McCall model and includes high-level characteristics that address a software buyer's three main questions: as-is utility, maintainability, and portability. The intermediate-level characteristic includes Boehm's seven quality factors: portability, reliability, efficiency, usability, testability, understandability, and flexibility. The primitive characteristics provide the foundation for defining quality metrics, with 17 attributes describing the product quality. The model is structured in a hierarchy, with each level contributing to the overall quality level (Singh & Kannojia, 2013).

The FURPS model was originally presented by Grady (1992). FURPS stands for Functionality, Usability, Reliability, Performance, and Supportability. Functionality refers to the software's features and capabilities, while Usability focuses on the ease of use and user experience. Reliability refers to the software's ability to operate without failure, and performance measures the software's speed and efficiency. Supportability encompasses the

software's ability to be maintained, tested, and evolved. The FURPS model is widely used as a checklist to evaluate software requirements and ensure that all quality aspects are considered during development.

Dromey's software product quality model proposes that software does not directly manifest quality attributes but rather exhibits properties that contribute to them (Dromey, 1995). The model consists of three key elements: quality attribute, product component, and quality carrying product property. This model focuses on the properties that must be built into the software product to achieve the desired quality attributes. While the model does not guide how to achieve product properties, it does consider quality definitions (Nistala et al., 2019).

In conclusion, selecting an appropriate software quality model is critical to ensure that a company's software products meet the necessary quality standards. Various software quality models are available. However, based on multiple literature reviews, ISO 9126 appears to be the most suitable for software quality engineering, as it provides a comprehensive framework for defining and measuring software quality attributes. Therefore, Table 1 provides an overview of the Software Quality Models mentioned above.

Table 1: Software Quality Models Comparison adapted from Singh and Kannojia (2013)

Criteria/Factors/Goal		McCall	Boehm	Dromey	FURPS	ISO 9126 & 25010
Quality Attributes	Correctness	X	X			
	Reliability	X	X	X	X	X
	Integrity	X	X			
	Usability	X	X	X	X	X
	Efficiency	X	X	X		X
	Maintainability	X	X	X		X
	Testability	X				
	Interoperability	X				
	Flexibility	X	X			
	Reusability	X	X			
	Portability	X	X	X		X
	Clarity		X			
	Modifiability		X			
	Documentation		X			
	Resilience		X			
	Understandability		X			
	Validity		X			
	Functionality			X	X	X
	Generality		X			
	Economy		X			
	Compatibility					X
	Performance				X	
	Supportability				X	
	Security					X

2.2 Related Work

This section is divided into two subsections. The first subsection (2.2.1) reviews relevant literature and scientific papers. The second subsection (2.2.2) explores the state of the art in Checkmarx, explicitly focusing on existing methods for measuring the quality of a scan.

2.2.1 General Related Work

This section covers various crucial topics studied in software engineering and quality. One of the primary focuses is software quality evaluation, with three articles dedicated to exploring software quality evaluation frameworks. Additionally, one article delves into the effectiveness of different software testing frameworks about software quality. Another significant contribution in this section is the introduction of the SourceAudit Tool for Software Quality Management. Furthermore, two articles evaluate specific metrics for object-oriented software, while a final article explores software metrics for software security. This section provides a comprehensive overview of recent research related to software quality, offering a solid foundation for the context and focus of this work.

According to Kato et al. (2022), Key Performance Indicators (KPIs) can ensure the quality of software products in DevOps. The authors propose a method to visualize and control the quality of software products in DevOps using quality characteristics as KPIs. The authors suggest that by categorizing the quality ensured by the CI/CD pipeline testing into quality characteristics, it becomes possible to visualize and control the quality of the released system using a quality model. They proposed the KPIs to define software quality goals before the project starts. This involves managing the results of tests in each pipeline and comparing them with past builds. The test pyramid is recommended as a valuable technique to prioritize tests in the CI/CD pipeline, with unit tests at the bottom, service tests in the middle, and UI tests at the top. Furthermore, the CI/CD pipeline delivers minimum quality faster, and test engineers can spend more time on test design and automation. Automated tests become the acceptance tests for manual testing, allowing for more efficient and complex manual testing. The authors recommend incorporating Agile processes, clarifying the prioritized quality characteristics in each sprint plan, and using the SQuaRE model to build up quality efficiently. In conclusion, the study provides a method for visualizing and controlling the quality of software products in DevOps using quality characteristics as KPIs and prioritizing tests using the test pyramid. It highlights the importance of incorporating Agile processes and using SQuaRE to build quality efficiently.

The researchers Gu et al. (2015) investigate methods that can accurately predict software defects using metrics. While many studies have investigated the best techniques for building defect prediction models, few have explored the optimal number of software metrics. The authors propose a network-based approach using software metrics to predict defects. The unique approach identifies relationships among features using a network based on the correlation between components. The authors used metrics specific to Object-Oriented languages: CK Metrics (a type of software metric) and two different association coefficients (Maximal Information Coefficient and Pearson correlation coefficient) to measure the similarity between features. Then, they choose a representative set of metrics by adjusting the correlation threshold. This subset selection approach is validated using a Poisson regression model, and the results show that using metrics networks is an efficient way to build a defect prediction model compared to using the complete set of metrics.

Medeiros et al. (2017) provides insights into the relationship between software security vulnerabilities and software metrics in software development. The study aims to determine if software metrics can distinguish vulnerable code from non-vulnerable code. To achieve this goal, the authors perform an exploratory analysis on a dataset of software metrics and security vulnerabilities reported in five commonly used software projects. The study includes an examination of the correlation between software metrics and the number of vulnerabilities, as well as an analysis of the interdependency between software metrics. The results show a strong correlation between several project-level metrics and the number of reported vulnerabilities, indicating that software metrics can provide valuable insights into software security. The study also found that a group of metrics can accurately distinguish between vulnerable and non-vulnerable code. Still, the best subset of metrics may vary from one software system to another. The results also emphasize that considering metrics at different architectural levels, such as the function, file, and project, can provide a more comprehensive understanding of software security. In conclusion, the study highlights the potential of software metrics in detecting and preventing software security vulnerabilities and offers valuable insights for future work in this area.

M. Liu et al. (2014) developed an appraisal target system for evaluating software product quality following ISO/IEC 9126 standards. The authors devised a framework based on a hierarchical tree structure, which divided software quality into multiple levels. The authors proposed a three-layer system, "characteristic-sub characteristic-indicator", where each quality characteristic was further divided into sub-characteristics and refined into relevant indicators. These indicators were derived primarily from software engineering experience, encompassing code error rate and CPU utilization metrics. The appraisal target system, the key to software quality metrics, was built upon five principles: Compatibility, Comprehensiveness, Effectiveness, Operability, and Commonality. To determine the weight of the indicators, the authors employed the entropy and distance matrix methods. The researchers also outlined the implementation of the software quality metrics, providing a step-by-step process for assessing the quality of a software product using the appraisal target system. This measurement process involved five stages: Goal Determination, Indicators Reduction, Data Collection, Weight Determination, and Result Calculation. In conclusion, the study highlights a framework where indicators were presented to assess the software product quality.

Nakai et al. (2016) proposed a comprehensive software quality evaluation framework based on the Systems and software Quality Requirements and Evaluation (SQuaRE) series, specifically ISO/IEC 25022:2016 and ISO/IEC 25023:2016. The framework aims to provide a concrete set of quality metrics for software products, addressing key quality characteristics such as functional suitability, maintainability, and user satisfaction. Comprising two components, namely "Product Quality" and "Quality in Use", it encompasses 47 product metrics and 18 quality-in-use metrics, covering more than 50% of the metrics initially defined in the SQuaRE series. The framework requires information from various sources, including documents, user tests, and questionnaires, to measure and evaluate

the quality metrics. Furthermore, the authors conducted a case study demonstrating the framework's usefulness in a commercial software product context. The case study yielded promising results, indicating the framework's ability to assess and improve software quality. Most metrics evaluated in the study achieved a 100% measurement rate, which strongly shows high quality for the target product. These findings reinforce the notion that the framework can serve as a valuable tool for software stakeholders, empowering them to evaluate and improve the quality of their products effectively.

Bakota et al. (2014) presents a description of SourceAudit tool, a software quality management tool that addresses the challenge of software erosion and the resulting decrease in internal quality over time. Software erosion leads to higher development, testing, and operational costs. The tool measures source code maintainability based on the ISO/IEC 25010 standard and the ColumbusQM model, offering a holistic view of software quality and warning about maintainability decline. It enables benchmark management, quality model management, and certification views, allowing users to assess and monitor software quality effectively. The tool benefits managers and software developers, providing a comprehensive understanding of software quality, and improving source code quality and development team performance.

The authors Lincke et al. (2008) evaluated a set of object-oriented software metrics using ten different software metric tools. The findings revealed that calculating the same metrics by various tools resulted in varying values, indicating a lack of consistency in metrics interpretation and implementation. The study emphasizes the need to consider the tool-dependent nature of metrics when making decisions. Moreover, the research examined the implications of these variations on client analyses, explicitly focusing on maintainability assessment using a software quality model based on ISO 9126. The study raises concerns about the reliability and comparability of metrics results, cautioning software engineers to be careful in relying solely on metrics for quality assessment. The authors also suggest further research to explore additional metrics and programming languages, expand the scope of analyzed software systems and revise metrics definitions to reduce ambiguity.

The authors Mladenova (2020) focus on the importance of software quality metrics in evaluating and assessing the quality of software development. It emphasizes the need for well-selected metrics to provide a comprehensive view of the development process and enable risk assessment and evaluation. The paper explores various quality models and metrics, including project, process, and product metrics. It also highlights the significance of metrics related to size and complexity, performance, maintenance, security, and software testing. The article concludes by suggesting the development of a comprehensive tool for quality checks and measurements to aid organizations in tracking their development, costs, risks, and defects.

The research opportunity presented in this dissertation lies in developing a novel tool that focuses on software quality metrics to evaluate the quality of scans conducted by Static Application Security Testing (SAST) tools. While

the existing literature has extensively explored software quality metrics in various domains, the specific area of scan quality assessment for SAST tools remains relatively unexplored. By creating a dedicated tool that retrieves a scan quality metric, this research addresses the need for comprehensive evaluation and assessment of SAST scans, ultimately enhancing the effectiveness of security testing in software development. This opportunity opens avenues for investigating and defining appropriate scan quality characteristics and designing effective metrics and measurement techniques to analyze log files from SAST tools.

2.2.2 Specific Related Work

Scan Coverage metric is the existing method used to assess scan quality. This metric considers the ratio of good and bad files among the total files. Figure 3 illustrates the *Scan Coverage* and its corresponding indicators in the log file. These indicators include:

- Total Files - This represents the files recognized by each language´s processor in the CxSAST Engine.
- Good Files - Files without parsing exceptions or parsing errors are considered good files.
- Partially Good Files - If a file encounters parsing exceptions or parsing errors but still successfully parses, it is considered partially good.
- Bad Files - Files that fail to parse entirely are categorized as bad files.

```
---------------------------
Total files                   153
Good files:                   152
Partially good files:         1
Bad files:                    0
Parsed LOC:                   10856
Good LOC:                     10855
Bad LOC:                      1
Number of DOM Objects:        47132
Scan coverage:                99.78%
Scan coverage LOC:            99.99%
```

Figure 3: Example of Scan Coverage

Scan Coverage metric is computed during the SAST scan process, and its value is expressed as a percentage retrieved from the log file (see Figure 3). This metric is represented by the following formula:

$$\text{Scan Coverage} = \left(\frac{GoodFiles}{Totalfiles} + \frac{PartiallyGoodFiles}{TotalFiles} \times 0.66 \right) \times 100$$

However, the simplicity of this metric, which relies on the file count, presents particular challenges. These issues are linked to the need for more differentiation in the $ScanCoverage$ values across various projects. In the majority of the cases, this percentage approximates 100%. Such uniformity makes it difficult to establish priorities, estimate task completion times, and obtain relevant information regarding the location of problems. The lack of granularity in the $ScanCoverage$ metric limits effective management for Product Managers, Team Leaders, and Developers. It also limits the ability of QA teams to make informed decisions related to software releases.

Considering these limitations, this master's dissertation proposes a more sophisticated metric to address these challenges and provide enhanced insights into scan quality.

2.3 Summary

This dissertation's "State of the Art" section begins by exploring fundamental concepts, including vulnerabilities, security testing, continuous integration and delivery, software development life cycle, and software quality. It discusses the significance of identifying vulnerabilities in software systems and emphasizes the importance of conducting thorough security testing to ensure robustness. The section also covers the concepts of continuous integration and delivery, highlighting their role in streamlining the software development process. It also examines the software development life cycle and emphasizes the need for quality considerations throughout the process.

Following the discussion of fundamental concepts, the section provides an overview of related work in the master's research area. It reviews existing literature and studies that have explored topics relevant to scan metrics for SAST tools. By examining prior research, this review helps to identify gaps in the literature and establishes the foundation for the research conducted in this dissertation.

Overall, the "State of the Art" section provides a comprehensive summary of fundamental concepts and related work, laying the groundwork for the subsequent chapters of the dissertation and highlighting the research opportunity to develop scan quality metrics for SAST tools.

3 PROPOSED APPROACH

This section outlines the overall strategy to achieve quality on a SAST tool scan. To comprehensively address the issue, examining the SAST scan process is essential. The output of this process serves as a fundamental component in accurately establishing a scan quality metric. Then, the overview of the proposed approach and all related information will be presented, such as requirements for the new tool implementation, architecture design, technology needed, and the decisions taken to address the problem.

3.1 SAST Scan Process and Core Concepts

The SAST scan plays an essential role in achieving the notion of scan quality. Once the scan is initiated, the SAST tool performs an in-depth examination of the project's source code. SAST builds a logical graph of the code's elements and flows. The scanning process involves examining the lines from the source code of a project written in a specific programming language (or a combination of several) to uncover potential security vulnerabilities and flaws. The scan operates on two key aspects:

- **Syntactic Recognition:** Ensuring that the code has the syntax rules of a programming language, verifying correctness and compliance with guidelines.
- **Semantic Recognition:** The scan analyzes the code's meaning and logic to understand how different components interact, identifying potential security issues.

Besides these two key elements, SAST has an extensive list of hundreds of pre-configured queries for known security vulnerabilities for each programming language. SAST provides scan results either as static reports or in an interactive interface that enables tracking runtime behavior per vulnerability through the code and provides tools and guidelines for remediation.

The input to SAST's scanning and analysis is the source code, so no building or compiling is required, and no libraries need to be available. The consumer must submit a project written in a supported programming language. The SAST engine performs a static analysis of the source code, and then there are two options. The first option is for the consumer to run a specific query to identify a particular vulnerability in their project. Alternatively, the consumer can opt for comprehensive analysis by running all the available queries for the programming language used in their project. The second option will be used in further analysis. Figure 4 illustrates the flow of this process.

In addition to the results being shown at the end of the scan and exported as a JSON file, a log file is generated. It captures essential insights about the overall scanning process, such as information about the machine where the SAST engine is running, scan phases, any errors encountered, and other relevant details that give us insights into the analyses performed by the tool.

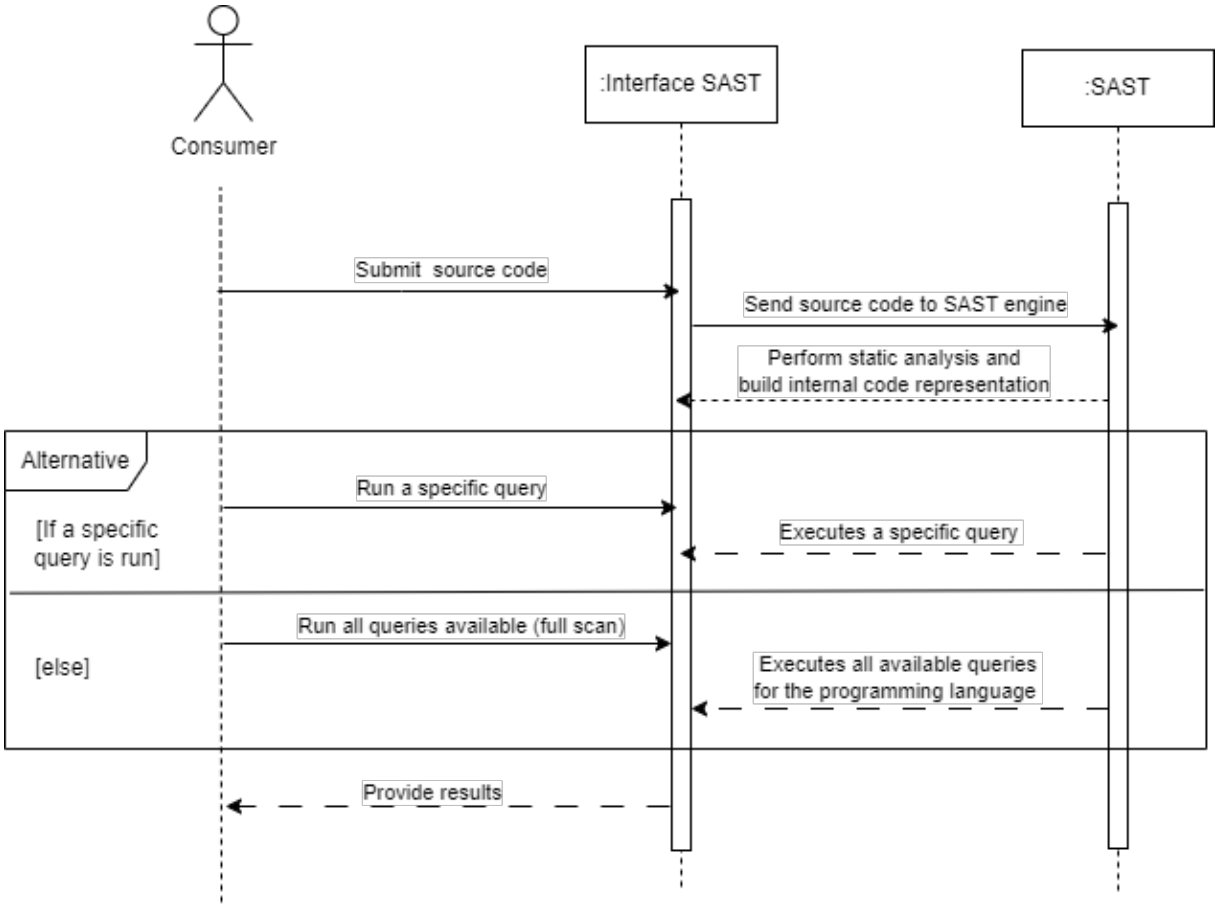


Figure 4: Sequence Diagram for SAST Scanning Process

The log file is the main object to build the notion of a scan quality for this master's dissertation. By carefully examining the log file, multiple scanning phases were identified. It involves many stages like Parsing, Resolver, Abstract Interpretation (AbsInt), and Querying, as shown in Figure 5.

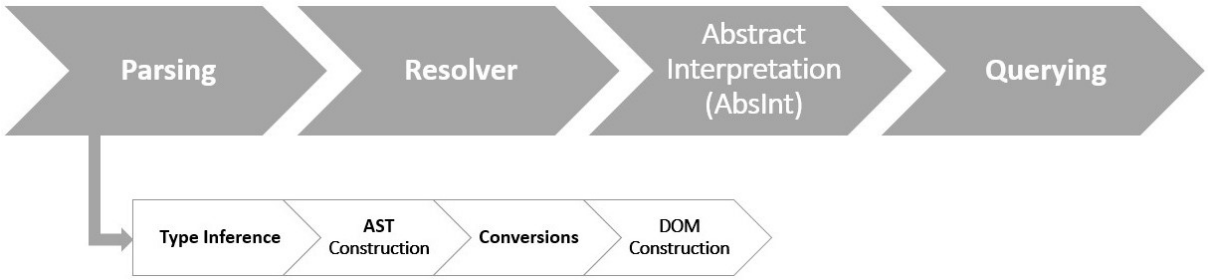


Figure 5: Key Phases in a Standard Project Scanning Process

The Parsing stage involves the syntactic and semantic recognition process. It is the most critical phase because it is where the semantic representation model (Domain Object Model, or DOM for short, henceforth) is built. The better the DOM is created, the more vulnerabilities are potentially detected. This stage also consists of 4 more substages: Type Inference, Abstract Syntax Tree (AST) Construction, Conversions, and DOM Construction.

The Type Inference substage is responsible for inferring data types from navigation on the parse tree. The AST Construction is responsible for traversing the parse tree generated by Antlr to create an abstract syntax tree. The Conversions include all the transformations required to "fit" the AST data structures in the abstractions of the DOM representation (e.g., object restructuring into multiple variables). The DOM Construction is responsible for building the DOM with all the necessary nodes mapped from AST into the DOM tree.

The Resolver stage is a processing stage executed immediately after parsing a language and is responsible for relating symbol instances to definitions. Like Parsing, it also processes each language separately, working on the symbols of that language.

Some stages are common to all scanned projects. These stages are AbsInt and Querying. In contrast, Parsing and Resolver are executed for each programming language identified in the project.

The AbsInt (Abstract Interpretation) stage is responsible for abstracting the behavior of the source code project by simplifying its values into abstract representations.

The Querying stage is the final phase of the scan process. It includes the results summary for each query executed during the scan. This summary presents the query's name, the number of results for each, its severity, CWE code, and execution time.

To summarize, the quality of a scan relies on how well the programming language is recognized, the structure of the DOM and its syntactic and semantic relations, as well as executing queries on the DOM without compromising the scan's performance.

3.2 Overview of the Proposed Approach

The proposed approach for building the notion of scan quality is to develop a service that can give the scan quality of a project based on the SAST log file. This service, CxScanQuality, will be integrated into an internal Quality Assurance (QA) platform (*SK* for short) to assess the overall SAST product quality. Additionally, it will be integrated into Continuous Integration (CI) pipelines. *SK* is managed by the QA team, providing access to valuable project-related data and enhancing their decision-making capabilities.

To determine the quality of a scan using CxScanQuality, it is important to identify the components that contribute to scan quality. By analyzing the individual components - recognition coverage, DOM structure, and query execution - a scan is considered to have good quality if all of its components exhibit good quality. As a result, the following quality factors are relevant: Coverability, Domability, and Querability. The scan quality is measured as an aggregate

metric that combines all of the quality factors, with each factor having a varying impact on the overall scan quality. The definitions of each factor are provided below:

- **Coverability:** Refers to the ability of the scan to recognize the syntax of the programming language used in the source code. Measuring the Coverability of a scan is measuring the Parsing coverage.
- **Domability:** Refers to the ability to translate what was syntactically recognized into a semantic structure in the DOM and its respective semantic relations. To measure the Domability in SAST, we need to measure the quality of each scan phase that contributes to the DOM Construction and its semantic relations. These stages are Type Inference, AST Construction, Conversions, DOM Construction, Resolver, and AbsInt. However, not all stages are equally crucial to the final scan.
- **Querability:** Refers to the capability of executing all queries defined by CxSAST, which are related to the source project's scope, on the generated DOM. To determine the Querability, we need to evaluate the success of each query's execution and the number of exceptions encountered during this process. The queries can be categorized based on their severity, such as High, Medium, Low, and Information (stands for Info), and each category has a corresponding weight based on its impact on the quality perception of the scan.

The quality factors for measuring the scan quality are shown in Table 2, with corresponding indicators extracted from log files at different stages of the scan. To measure Coverability, the number of recognized lines during Parsing and the total lines of code are taken into account. For Domability, the indicators are the number of exceptions thrown during each affected phase. Querability is measured by successful and unsuccessful queries, categorized by severity, executed during the scan, as well as query exceptions.

The implementation of the indicators in Table 2 requires modifications to the Engine Log service for log file analysis. Other services already use this service, which is responsible for displaying metrics from a log file. This adaptation is necessary for computing the quality factors into CxScanQuality. Then, integration of CxScanQuality into CI/CD pipelines and the *SK* platform becomes essential. In summary, extracting data from log file analysis enhances the precision and depth of scan quality assessment.

Table 2: Scan Quality Framework

Scan Quality		
Coverability	**Domability**	**Querability**
Parsed LOC	Exceptions Parsing	Total Success High queries
LOC	Exceptions TypeInference	Total Failed High queries
	Exceptions AST Construction	Total Success Medium queries
	Exceptions Conversions	Total Failed Medium queries
	Exceptions DOM Construction	Total Success Low queries
	Exceptions Resolver	Total Failed Low queries
	Exceptions AbsInt	Total Success Info queries
		Total Failed Info queries
		Total Success queries
		Total Failed Info queries
		Exceptions queries

3.3 Requirements

To design the CxScanQuality service, a set of requirements has been formulated to ensure that the outcomes align with the expectations and objectives of the stakeholders. Table 3 presents a comprehensive breakdown of each requirement, encompassing its unique identification (ID), the corresponding category, a descriptive summary, and the specific service into which it has been integrated. These categories are classified into two distinct dimensions: Non-Functional, regarding the performance characteristics the system should exhibit, and Functional, encompassing the features that the system is expected to provide.

These requirements are essential for successfully developing and evaluating the CxScanQuality. Consider Requirement 7 as an illustrative case in Table 3. This requirement falls under the Functional category and is described as follows: "The solution must be integrated into the CI pipelines". This means that the CxScanQuality service must be integrated into CI pipelines. However, it is important to note that this requirement does not specify an "Applicable Service" due to the complexity involved in the integration process. This is because modifying multiple services makes listing each in this context impractical. However, the "Development" chapter explains this process in detail.

Table 3: Requirements

ID	Requirement Category	Requirement Description	Applicable Service
1	Non-Functional	The solution must offer a comprehensive assessment of the scan quality for a project.	CxScanQuality
2	Functional	Provide an object containing relevant information for each programming language in the scanned source code. The relevant data are the names and number of exceptions encountered during the various stages of the scanning process.	Engine Log
3	Functional	Provide an object detailing the counts of successful and unsuccessful queries for each severity level and the overall totals for successful and unsuccessful queries.	Engine Log
4	Functional	The system should supply an object, including information on Lines of Code (LOC) and Parsed Lines of Code.	Engine Log
5	Functional	The solution must calculate and present the quality factor in percentages.	CxScanQuality
6	Functional	The solution must be integrated into the quality platform, which is *SK*.	SK platform
7	Functional	The solution must be integrated into the CI pipelines.	Not Applicable
8	Functional	Implement a POST request to the Engine Log API to retrieve indicators related to each quality factor.	CxScanQuality

3.4 Architecture Design

The architecture design of the CxScanQuality service is shown in Figure 6 to aid in developing the proposed approach.

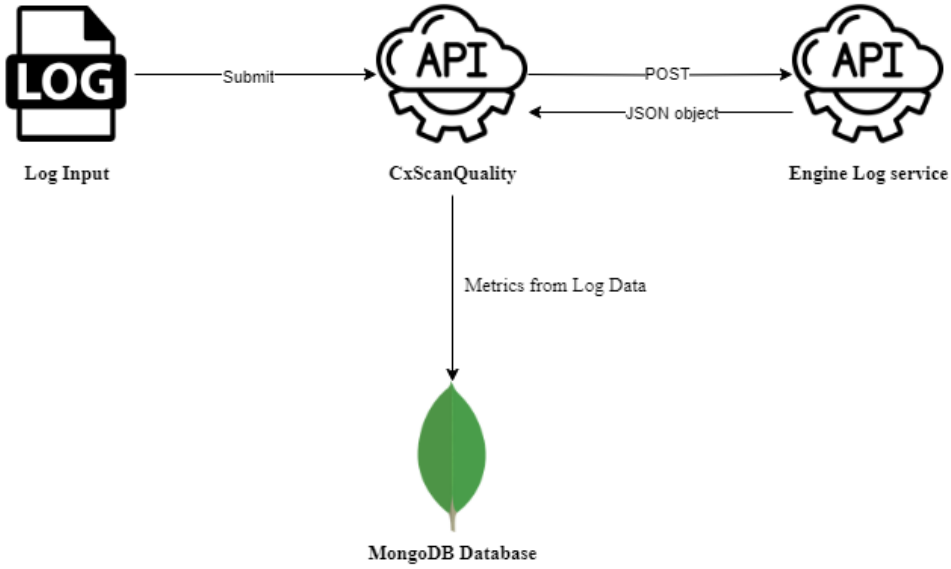


Figure 6: Architecture Overview

Log Input - A log file generated from a project scanned by the SAST must be submitted to CxScanQuality to obtain a scan quality score.

CxScanQuality - This API is responsible for receiving the log file and analyzing it using the indicators provided by the Engine Log service. To achieve this, CxScanQuality initiates a POST request to the Engine Log service, incorporating the log input in the request. The goal is to obtain all the necessary indicators for calculating the Scan Quality metric, including its three components: Coverability, Domability, and Querability.

Engine Log service - This API is responsible for receiving the log file and providing the log's raw data in JSON format.

MongoDB Database - Utilized as a MongoDB database, this system stores metrics derived from log data acquired through CxScanQuality. A new MongoDB collection must be established within the existing MongoDB database to accommodate this storage.

3.5 Technology Stack

Following the requirements and architecture design, the technologies used to implement CxScanQuality are represented in Figure 7. Furthermore, this section explores the chosen technologies and explains the reasons for their selection.

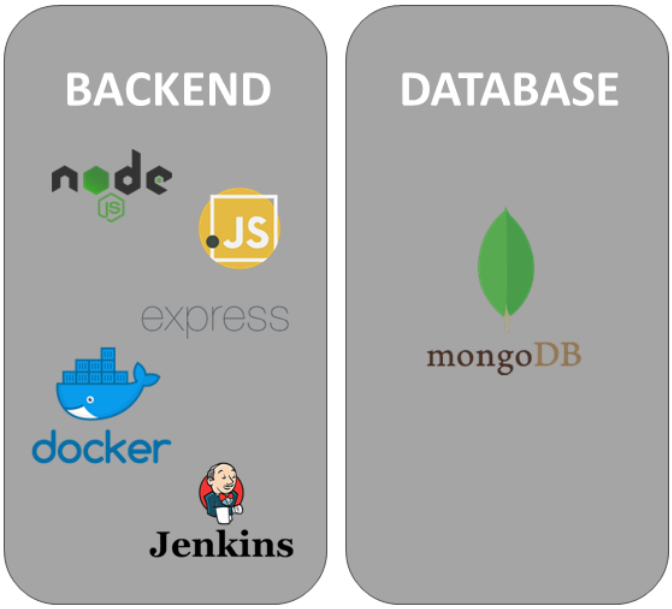


Figure 7: Technology Stack

For the development of CxScanQuality, the programming language used was *JavaScript*. *JavaScript* is a versatile and multipurpose language utilized for web development and across various domains, including server-side programming with *Node.js*. Its advantages extend beyond web development, resulting in cost-efficiency through multipurpose use and streamlined maintenance. *JavaScript* also seamlessly integrates with other technologies like databases, APIs, and third-party services. Additionally, *Node.js* [3], an open-source, cross-platform runtime environment for executing *JavaScript* code, was employed. The QA team chose these services because they manage similar ones, allowing for more straightforward maintenance and support.

Express [4] is a lightweight and adaptable web application framework for Node.js, offering a comprehensive range of features for building applications that can be used on both web and mobile platforms. It facilitates the rapid development of Node-based web applications and APIs by simplifying common tasks such as routing, middleware integration, request handling, and response management.

[3] Available at: https://nodejs.org/en

[4] Available at: https://expressjs.com/

Docker [5] is a containerization platform that enables developers to package applications and their dependencies into portable containers. These containers can run consistently across various environments, making developing, testing, and deploying software applications easier. This technology facilitates the integration of CxScanQuality into both the *SK* platform and CI pipelines.

Jenkins [6] is an open-source automation server widely used for building, deploying, and automating software projects. It provides a robust platform for Continuous Integration (CI) and Continuous Delivery (CD), helping development teams automate repetitive tasks, test code changes, and streamline the software development and deployment processes. This technology is applied in conjunction with *Docker* to push a *Docker* image of CxScanQuality into the Amazon Elastic Container Registry (AWS ECR). This step aims to enhance the integration of CxScanQuality into both the *SK* platform and CI pipelines.

MongoDB [7] is a widely adopted, open-source NoSQL (Not only SQL) database management system that stores data in a flexible, schema-less format known as BSON (Binary JSON). It is designed for high scalability, performance, and ease of development. *MongoDB* is classified as a document database, making it highly suitable for managing large volumes of structured and unstructured data, particularly in modern web and mobile applications. This technology enables the storage of all the metrics developed in CxScanQuality for various projects.

3.6 Design Decisions

Developing a robust scan quality metric requires careful consideration of various design decisions. These decisions play an important role in ensuring the accuracy and reliability of the metric. The following design choices were made during the development process:

1. **Handling Undefined Scan Stages:** In cases where certain scan stages lacked well-defined boundaries, exceptions occasionally exceeded the established limits of scan phases. To address this, a systematic approach was adopted. Any exceptions arising from such scenarios were associated with the previous scan stage. This decision maintains a coherent flow of exception handling while preserving the integrity of the scan phases.

2. **Standardizing Unnamed Exceptions:** In cases where log entries indicated errors (ERROR) or warnings (WARN) without explicitly specifying the exception's name or when extraction of the actual name was impossible, a default naming convention was implemented. The decision to assign default names, such as

[5] Available at: https://www.docker.com/

[6] Available at: https://www.jenkins.io/

[7] Available at: https://www.mongodb.com/

"ERRORUnknownException" or "WARNUnknownException" ensures consistency in exception labeling. This enhances clarity and simplifies subsequent analysis.

3. **Uniform Identification of Common Engine Stages:** Certain engine stages were typical across different languages throughout the scan process. To represent these stages cohesively and facilitate the identification of language-specific exceptions within them, they were labeled as "Common" for the respective language names.

The design decisions mentioned above relate to the developments in Engine Log service, which ensures effective log data management. These considerations collectively contribute to the robustness and consistency of the scan quality metrics design.

3.7 Summary

This section discusses a proposed approach to achieving the quality of the CxSAST tool scan. It outlines the process and core concepts of SAST scanning and introduces the "Scan Quality" concept. It also outlines the requirements, architecture design, technology stack, and decisions made.

The proposed approach aims to build the notion of scan quality using a service called CxScanQuality, which analyzes SAST log files. The scan quality is divided into three quality factors: Coverability, Domability, and Querability. Each factor contributes to overall scan quality, measuring various indicators from log files.

This approach has the potential to enhance the accuracy and effectiveness of SAST tool scans, providing insights for the QA team to make informed decisions about the quality of scanned projects.

4 DEVELOPMENT

This chapter covers all the steps related to the development process. The development process includes creating a new service (CxScanQuality) and updating an existing service (Engine Log service). To address this, topics discussed here include methods for data extraction from a scan, configuration of Coverability, Domability, and Querability. It also encompasses a statistical analysis, the integration process of CxScanQuality, and the solution validation.

4.1 Data Extraction from a Scan

The data extraction process from a scan involves updating the Engine Log service to compute the Scan Quality metric. This section outlines the steps and methodologies employed to extract valuable data from the log files generated during the scanning process.

4.1.1 Identifying Scan Phases in Log Files

The first step in data extraction from a scan involves identifying the scan phases within the log files. This task was straightforward due to the patterns in each scan phase's log entries. When a scan stage starts, the log entry follows the pattern "Engine Phase (Start): *StageName*". Conversely, when a stage concludes, the design changes to "Engine Phase (End): *StageName*". Figure 8 illustrates these patterns.

In certain stages, such as Parsing and Resolver, details related to the language name are added to the pattern. For example, the log entries for these stages appear as "Engine Phase (Start): *StageName LanguageName*" and "Engine Phase (End): *StageName LanguageName*". Substages within the Parsing phase exhibit a slight variation in their entries, with the pattern for the start being "Entering *substageParsingName* of *LanguageName*" and for the end, "Exiting *substageParsingName* of *LanguageName*". Figure 9 illustrates these patterns.

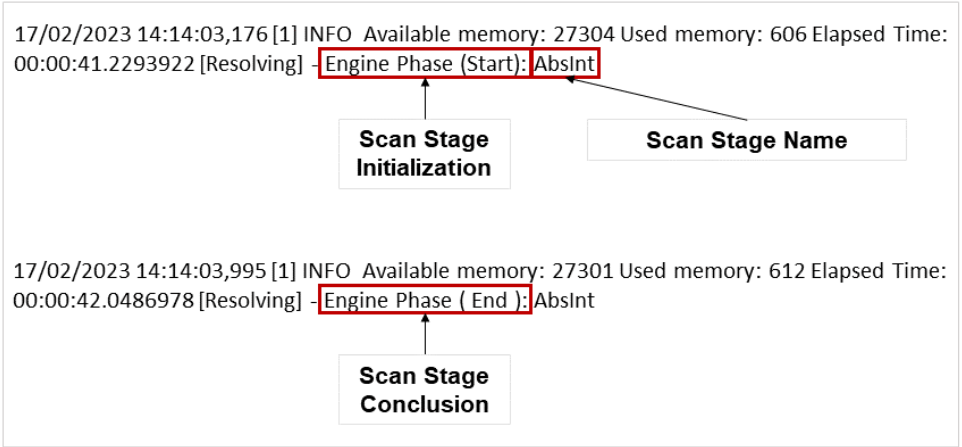


Figure 8: Illustrating a Log Entry from the AbsInt Stage

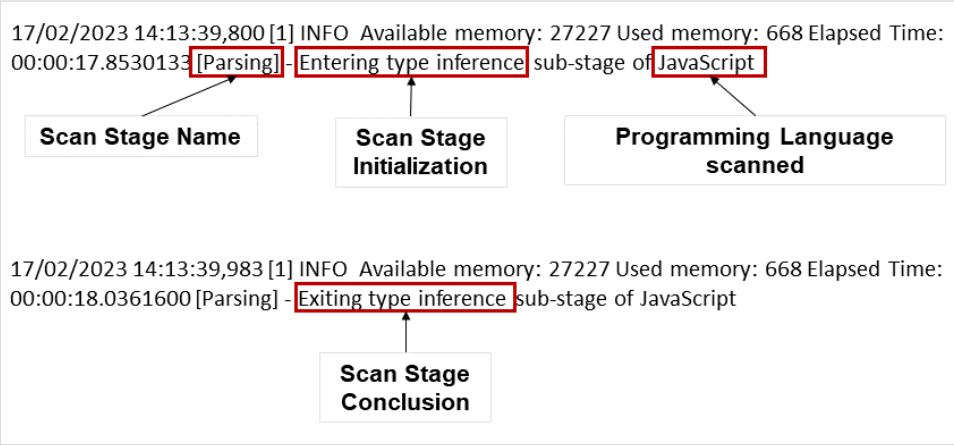


Figure 9: Illustrating a Log Entry from the Type Inference Stage

4.1.2 Analyzing Common Exceptions

Once the various scan phases were identified, a comprehensive analysis was conducted to determine the exceptions that occurred most frequently throughout the scanning process.

Table 4: Overview of Exceptions Across Scan Stages

Scan Stage	Exception Names
Parsing	"Antlr4.Runtime.NoViableAltException" "Antlr4.Runtime.InputMismatchException" "unexpected token" "Unexpected type for the Abstract Syntax Tree (No Definition)" "unexpected char: '#' (No Definition)" "NullReferenceException" "ArgumentNullException" "InvalidOperationException" "InvalidCastException" "ApplicationExceptions" "OverflowException" "ArgumentException"
Resolver	"RegexParseException" "NullReferenceException" "ArgumentException"
Querying	"BasicDomIteratorManager"

This analysis was based on scanning log files from every project in the *SK* database to identify exceptions using Checkmarx's internal tool. Table 4 presents the results, listing all exceptions found during the analysis.

Our analysis reveals that the Parsing and Resolver stages had the most frequent occurrences of exceptions. Among the different substages of the Parsing stage, we found that DOM Construction, Type Inference, and AST Construction had the highest occurrence of exceptions.

Analyzing exceptions in the scanning process is a crucial step as it enables the measurement of their impact on the quality of the scan. For instance, an exception like 'Antlr4.Runtime.NoViableAltException' can significantly affect the scan quality more than an 'ArgumentNullException.' ANTLR (ANother Tool for Language Recognition), which is a powerful parsing tool, plays a critical role in the DOM Construction of the source code and, thus, has a significant impact on scan quality.

4.1.3 Updating the Engine Log Service

Once the necessary modifications are applied (further details provided in sections 4.2, 4.3, 4.4, and 4.5), the next step is to update the Engine Log service. Figure 10 illustrates how the Postman software is employed to send an HTTP request (specifically, a POST request) to the service. Postman was chosen as the testing tool for the Engine Log service. The output of this request is a JSON object.

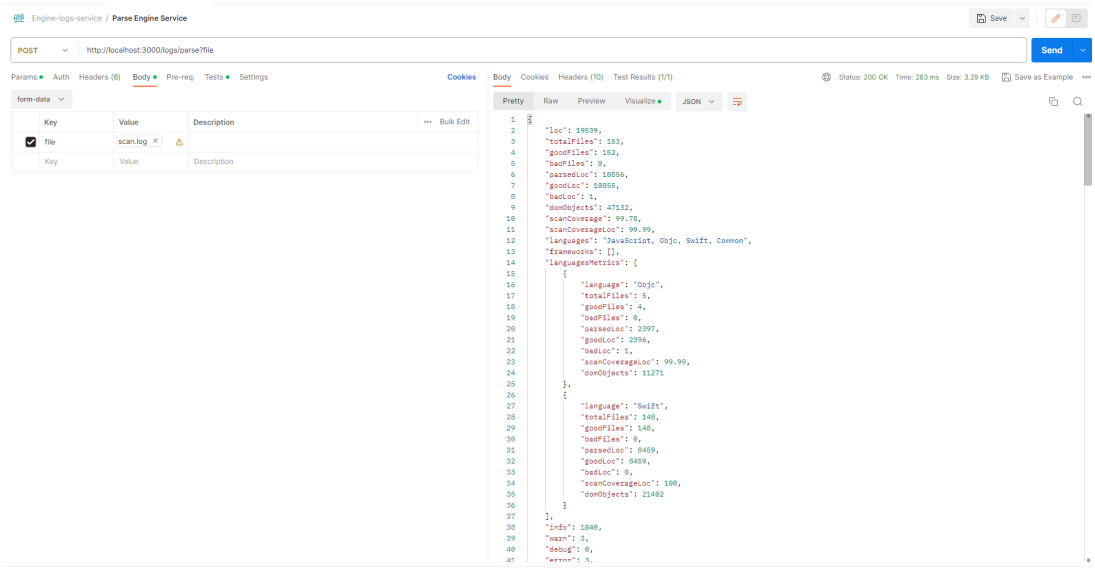


Figure 10: Engine Log Service Output

The Engine Log service, when receiving the request, generates a JSON response containing various log metrics. In total, this service collects 39 metrics. Here, all the key metrics from the JSON object are presented: *loc*, *totalFiles*, *goodFiles*, *badFiles*, *parsedLoc*, *goodLoc*, *badLoc*, *domObjects*, *scanCoverage*, *scanCoverageLoc*,

languages, *frameworks*, *languagesMetrics*, *info*, *warn*, *debug*, *error*, *tested*, *results*, *exceptions*, *antlrExceptions*, *queryCount*, *generalQueryCount*, *maxMemory*, *scanDuration*, *slowestQuery*, *slowestGeneralQuery*, *absintDuration*, *ast2domDuration*, *flowDuration*, *queryDuration*, *parseDuration*, *parseLanguagesDuration*, *ast2domLanguagesDuration*, *preprocessingLanguagesDuration*, *antlrParsingLanguagesDuration*, *typeInferenceLanguagesDuration*, *astLanguagesDuration* and *conversionsLanguagesDuration*

Two new metrics, *querabilityThesis* and *domabilityThesis*, were introduced to measure the quality factors of Querability and Domability. These metrics can be used to calculate the scores for Querability and Domability. Below is the expected JSON structure for the quality factor of Domability.

```
{
    "domabilityThesis": {
        "javaScript": [
            {
                "engineStage": "Parsing",
                "exceptionsList": [
                    {
                        "exceptionName": "RecognitionException",
                        "count": 1
                    },
                    {
                        "exceptionName": "Json.JsonReaderException",
                        "count": 1
                    }
                ],
                "total": 2
            }
        ]
    }
}
```

Here is an example JSON object that shows the metrics for measuring Querability.

```
{
    "querabilityThesis": {
        "totalSuccessQueries": 344,
        "totalInsuccessQueries": 0,
        "highQueries": {
            "success": 67,
```

```
            "insuccess": 0
        },
        "mediumQueries": {
            "success": 107,
            "insuccess": 0
        },
        "lowQueries": {
            "success": 129,
            "insuccess": 0
        },
        "infoQueries": {
            "success": 41,
            "insuccess": 0
        }
    }
}
```


4.2 Configuring Coverability

During the initial phase of the scan, Coverability is the first quality factor that the scan retrieves information on. This information does not include the Coverability score, but it consists of the necessary calculation indicators. To configure Coverability, two key metrics, namely LOC and $ParsedLOC$, need to be extracted from the Engine Log service. These metrics are already available and require no further updates, as they are obtained through a RegEx pattern-matching process.

The concept of Coverability is related to recognizing the programming language's syntax used in the scanned project. Coverability is expressed using the formula:

$$Coverability = \frac{ParsedLOC}{LOC} \times 100 \tag{4.1}$$

Formula (4.1) assesses the Parsing coverage of the project, where $ParsedLOC$ represents lines recognized during Parsing, and LOC includes all lines of code in the scanned project.

It is essential to exclude exceptions in the scan log related to Parsing issues from the Coverability formula. The count of unrecognized Lines of Code already addresses this handling. A Parsing exception indicates a failure to recognize certain LOC. In thread abort timeout exceptions cases, the entire file failed recognition, making the count of unrecognized LOC equal to the total number of LOC in the file.

4.3 Configuring Domability

To configure Domability, the creation of new indicators was necessary to generate the Domability JSON object (see JSON 4.1.3). Table 5 provides the required regular expressions.

Table 5: Scan Stages and Corresponding RegEx Patterns

Scan Stages	RegEx Pattern
Parsing	*/Engine Phase \(Start\): Parsing (\w+)\n([\s\S]*?)\[Resolving\] - Engine Phase \(End \): Parsing \1/g*
Type Inference	*/\[Parsing\] - Entering type inference sub-stage of (\w+)\n([\s\S]*?)\[Parsing\] - Exiting type inference sub-stage of \1/g*
AST Construction	*/\[Parsing\] - Entering AST sub-stage of (\w+)\n([\s\S]*?)\[Parsing\] - Exiting AST sub-stage of \1/g*
Conversions	*/\[Parsing\] - Entering conversions sub-stage of (\w+)\n([\s\S]*?)\[Parsing\] - Exiting conversions sub-stage of \1/g*
DOM Construction	*/\[Parsing\] - Entering AST to DOM sub-stage of (\w+)\n([\s\S]*?)\[Parsing\] - Exiting AST to DOM sub-stage of \1/g*
Resolver	*/Engine Phase \(Start\): Resolver (\w+)\n([\s\S]*?)\[Resolving\] - Engine Phase \(End \): Resolver \1/g*
AbsInt	*/Engine Phase \(Start\): AbsInt Resolver\n([\s\S]*?)\[Resolving\] - Engine Phase \(End \): AbsInt Client Resolver/g*

To accommodate decision 1 (Handling Undefined Scan Stages) in section 3.6, it was essential to develop RegEx patterns where the endpoint of each RegEx did not mark the 'End' or 'Exiting' entry, but rather the entrance where the next stage or substage starts.

The concept of Domability is related to DOM Construction. Domability is represented using the following formula:

$$\text{Domability} = (0.6Q_{\text{Parsing}} + 0.2Q_{\text{DOM Construction}} + 0.15Q_{\text{Resolver}} + 0.05Q_{\text{AbsInt}}) \times 100 \tag{4.2}$$

Where:

$$Q = e^{-0.009 \times (\text{Number\_Of\_Exceptions})}$$

In formula (4.2), the variable Q_{Parsing} represents the sum of exceptions in the Parsing, Type Inference, AST Construction, and Conversions stages. The variable $Q_{\text{DOM Construction}}$ represents the exceptions that occur in the DOM Construction stage. The variable Q_{Resolver} represents the sum of exceptions in the Resolver stages, and the variable Q_{AbsInt} represents the sum of exceptions in the AbsInt stages.

The coefficients assigned to different components in the Domability formula have an explanation. Each attribute may hold varying importance when assessing scan quality. These weights reflect this importance and contribute to the overall Domability score. The coefficients are determined based on factors such as the attribute's relevance to the impact on scan quality and the importance of the attribute's content. The weight assignment process involves analyzing scan log files to understand the impact on the overall scan quality and collaborating with domain experts to validate attribute importance.

Based on research conducted by Checkmarx, the Parsing stage is the one that takes more time in most languages, with an average impact accounting for approximately 50% of the total scan time. This is because the Parsing stage encompasses all the steps involved in constructing the DOM, and the accuracy of results relies on the structure of the DOM. Therefore, exceptions occurring in this phase substantially impact result quality from the client's perspective, justifying a higher weight of 0.6. The remaining coefficients are attributed hierarchically to assess their impact on scan quality, with 0.2 for the DOM Construction stage, 0.15 for Resolver, and 0.05 for AbsInt.

To understand why an exponential function is the most appropriate choice for Domability, it is important to consider how exceptions affect the resulting Domability score. The decision to use an exponential decay function, represented by $Q = e^{-0.009 \times (\text{Number\_Of\_Exceptions})}$, was based on the fact that an increase in the number of exceptions will result in a decrease in the scan quality, and therefore, Domability should also decrease. The graph in Figure 11 shows exactly that. The more exceptions, the lower the value of Q. It also shows that as the number of exceptions grows, the value of Q will decrease slowly towards 0, meaning that after a certain number of exceptions, the quality is already bad, and it is harder to make it worse. On the opposite, the quality decays faster when the number of exceptions is smaller. This conveys the real-world perception of quality degradation whenever a small amount of things start to fail. Therefore, It is crucial to consider the number of exceptions present and their potential impact on the overall Domability score. The negative exponent in the function ($k = -0.009$) results in an exponential decay that appropriately mirrors this expected scenario.

The $k = -0.009$ was tested on a range of exceptions from 0 to 4200, which is the maximum number of exceptions seen. This also confirmed that the k value is appropriate. Moreover, the weighted average approach to calculate Domability allows for adjustments based on the importance of the scan stages.

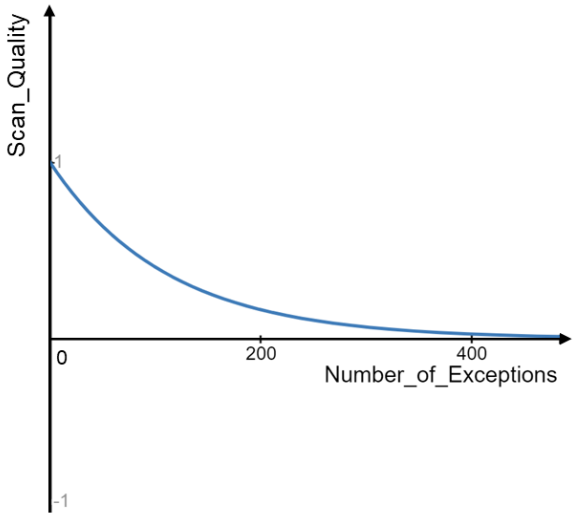


Figure 11: Representative Graph for k Constant Generated by Desmos

As an example, Figure 11 demonstrates an exponential decay function that represents the equation $y = e^{-0.009 \times x}$. The X-axis of Figure 11 ranges between 0 and 400, while the Y-axis ranges between -1 and 1. It is worth noting that this graph has no roots.

4.4 Configuring Querability

To configure Querability, it is necessary to set up each indicator to generate the Querability JSON object (see JSON 4.1.3). Table 6 presents the required regular expressions.

The concept of Querability is related to query execution, and it is represented using the following formula:

$$\text{Querability} = (0.7(0.5W_{\text{high}} + 0.3W_{\text{medium}} + 0.15W_{\text{low}} + 0.05W_{\text{info}}) + 0.3Q_{\text{querying}}) \times 100 \quad (4.3)$$

Where:

$$Q = e^{-0.009 \times (\text{Number\_Of\_Exceptions\_Query\_Stage})}$$

$$W = \left(\frac{\#\text{SuccessedQueries}}{\#\text{TotalQueries}} \right) \times 100$$

In formula (4.3), the first part pertains to the weights assigned to different query severity levels: high, medium, low, and information attributes. These weights ($0.5W_{\text{high}}$+$0.3W_{\text{medium}}$+$0.15W_{\text{low}}$+$0.05W_{\text{info}}$) determine the contribution of each severity to the overall Querability score based on the notion that higher severity queries play a higher and more critical role in the severity posture of an application. The second part of the formula involves the

weight (0.3) assigned to exceptions (Q_{querying}) that occur during the Querying stage. This weight reflects the significance of handling querying exceptions to maintain a high level of Querability.

Table 6: Querability Indicators and Corresponding RegEx Patterns

Indicators	RegEx Pattern
Querying	*/\[Resolving\] - Exiting FinalizeDOMNodesProperties\(\)\n([\s\S]*?) - Engine Phase \(End \): Querying/g*
SuccessHighQueries	*/Query - ([^\s]+)\s+Severity: (High)\s+(success)\s+Results:\s+(\d+)/img*
SuccessMediumQueries	*/Query - ([^\s]+)\s+Severity: (Medium)\s+(success)\s+Results:\s+(\d+)/gm*
SuccessLowQueries	*/Query - ([^\s]+)\s+Severity: (Low)\s+(success)\s+Results:\s+(\d+)/img*
SuccessInfoQueries	*/Query - ([^\s]+)\s+Severity: (Information)\s+(success)\s+Results:\s+(\d+)/img*
FailedHighQueries	*/Query - ([^\s]+)\s+Severity: (High)\s+(FAILED!)\s+Results:\s+(\d+)/img*
FailedMediumQueries	*/Query - ([^\s]+)\s+Severity: (Medium)\s+(FAILED!)\s+Results:\s+(\d+)/gm*
FailedLowQueries	*/Query - ([^\s]+)\s+Severity: (Low)\s+(FAILED!)\s+Results:\s+(\d+)/img*
FailedInfoQueries	*/Query - ([^\s]+)\s+Severity: (Information)\s+(FAILED!)\s+Results:\s+(\d+)/img*

4.5 Configuring Scan Quality

All configuration belonging to Scan Quality was centralized within the CxScanQuality service. This service initiates a POST request to the Engine Log service, retrieving the essential data for computing the Scan Quality metric. Scan Quality is represented using the following formula:

$$\text{Scan Quality} = 0.4 \cdot \text{Coverability} + 0.4 \cdot \text{Domability} + 0.2 \cdot \text{Querability} \tag{4.4}$$

The output from CxScanQuality is presented as a JSON result for each project under analysis, as exemplified below:

```
{
    "ScanQuality":
    {
        "Percentage": <value>
        "Coverability": <value>
        "Domability": <value>
        "Querability": <value>
    }
}
```

In this output, all the values are expressed in percentages. The "ScanQuality" object provides the following information:

- `Percentage`: The overall scan quality score is in this field.
- `Coverability`: This field is the Coverability score.
- `Domability`: This field is the Domability score.
- `Querability`: This field is the Querability score.

The assignment of weights to each of the mentioned components was informed by domain experts to validate their influence on the overall scan quality. Moreover, statistical analysis contributes to enhancing this assessment.

4.6 Statistical Analysis

This section discusses applying a statistical method for calculating the coefficients of the Scan Quality formula considering the quality of the scan results in terms of True Positives, False Positives, True Negatives, and False Negatives, i.e., their accuracy. During this statistical analysis, five projects were excluded due to insufficient information regarding accuracy.

A linear regression analysis was conducted using the JASP software on 160 projects to assess the statistical support for the weights assigned to the variables Coverability, Domability, and Querability in the Scan Quality formula. The dependent variable was accuracy, derived from the proportion of True Positives and True Negatives in the total results. The independent variables included Coverability, Domability, and Querability.

Linear regression is a widely used technique for exploring the relationships between variables and predicting outcomes based on these relationships. In our case, we employed it to understand whether the assigned weights accurately reflect the influence of Coverability, Domability, and Querability on Scan Quality.

The model was structured as follows:

$$\text{Scan Quality} = \beta_0 + \beta_1 \cdot \text{Coverability} + \beta_2 \cdot \text{Domability} + \beta_3 \cdot \text{Querability} + \varepsilon, \tag{4.5}$$

where β_0 is the intercept, β_1, β_2, and β_3 are the coefficients for Coverability, Domability, and Querability respectively, and ε represents the error term.

However, the outcome of this analysis indicates that the model failed to adequately fit the data, as evidenced by an R-squared value of 0.008 (Fig.12). This R-squared suggests that the proposed model can explain only a tiny proportion of the variance in Scan Quality. This indicates that the model's predictions deviate significantly from the actual data points.

Results ▾

Linear Regression

Model Summary - Accuracy

Model	R	R^2	Adjusted R^2	RMSE
H_0	0.000	0.000	0.000	12.790
H_1	0.090	0.008	−0.004	12.818

ANOVA

Model		Sum of Squares	df	Mean Square	F	p
H_1	Regression	321.772	3	107.257	0.653	0.582
	Residual	39430.765	240	164.295		
	Total	39752.537	243			

Note. The intercept model is omitted, as no meaningful information can be shown.

Coefficients

Model		Unstandardized	Standard Error	Standardized	t	p
H_0	(Intercept)	93.135	0.819		113.744	< .001
H_1	(Intercept)	197.071	151.091		1.304	0.193
	Querability	−1.092	1.518	−0.047	−0.719	0.473
	Coverability	0.044	0.038	0.075	1.166	0.245
	Domability	0.013	0.050	0.017	0.267	0.790

Figure 12: Linear Regression Data

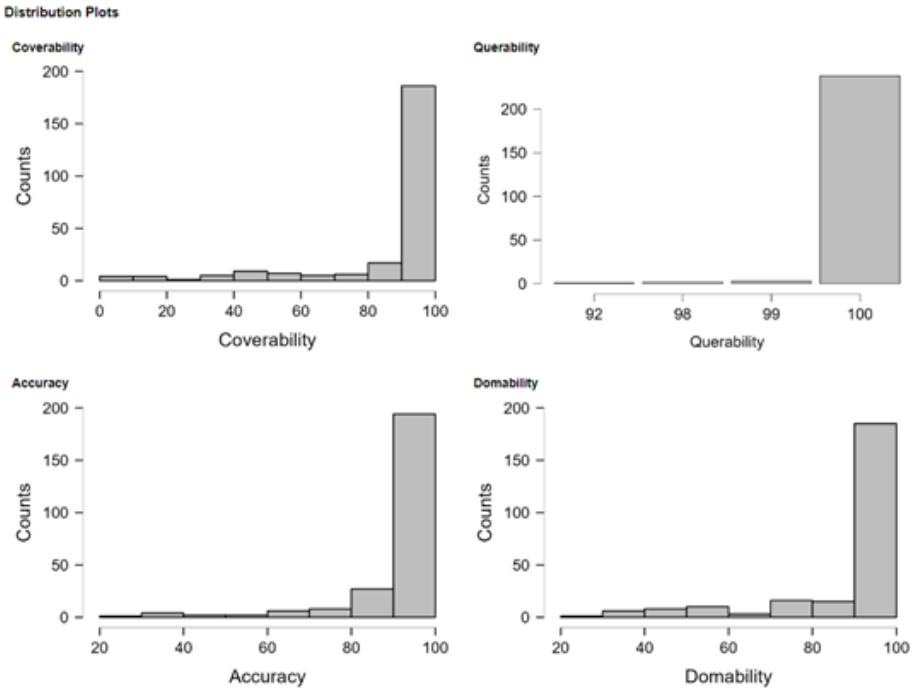


Figure 13: Distribution Plots

While there could be several reasons for this model fit failure, one prominent explanation might be the asymmetric and limited variability within the dataset. Across all variables, data points tend to hover around 90% to 100% (Fig.13). This restricted range of variation could contribute to the inability to construct a meaningful linear regression model.

4.7 Scan Quality Metric Integration

Integrating the Scan Quality metric as a new Key Performance Indicator (KPI) is a significant improvement for Checkmarx's QA team in monitoring the quality of the CxSAST product. Scan Quality encompasses sub-KPIs, including Querability, Domability, and Coverability, alongside over 50 other metrics, collectively evaluating the overall SAST performance. Figure 14 illustrates the initial set of over 50 metrics, while Figure 15 showcases the scenario following this integration.

The importance of the Scan Quality KPI extends beyond the QA team, impacting all stakeholders. This KPI enables real-time assessment of scan quality for each version of the SAST product. This real-time assessment is made possible due to Checkmarx's agile approach, with CI playing a key role in maintaining product quality.

Real-time assessment of Scan Quality impacts various organizational roles. Thanks to the CI process, developers, QA teams, and Product Managers gain immediate insights into scan quality with each pull request a developer generates. This ensures that Scan Quality is continuously monitored and improved throughout the development life cycle.

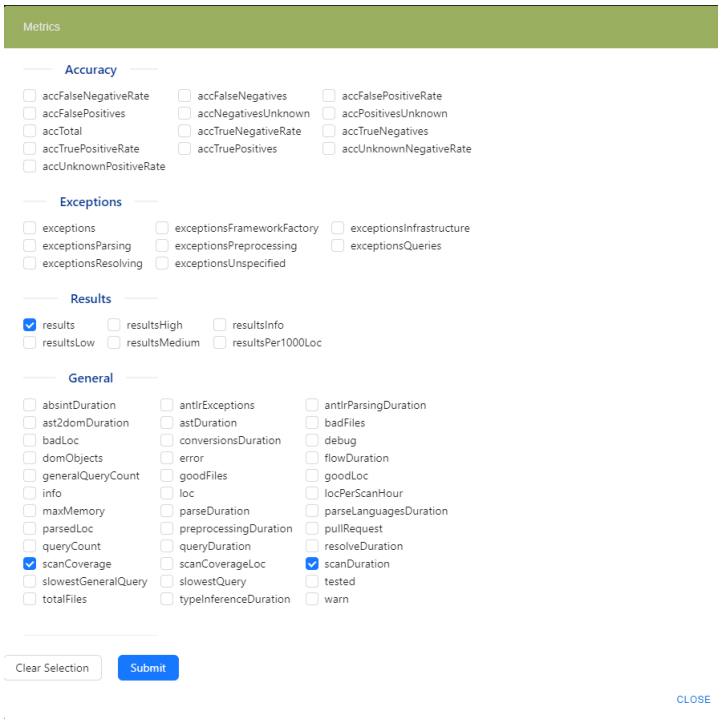


Figure 14: Initial Core Metrics for Quality Assessment

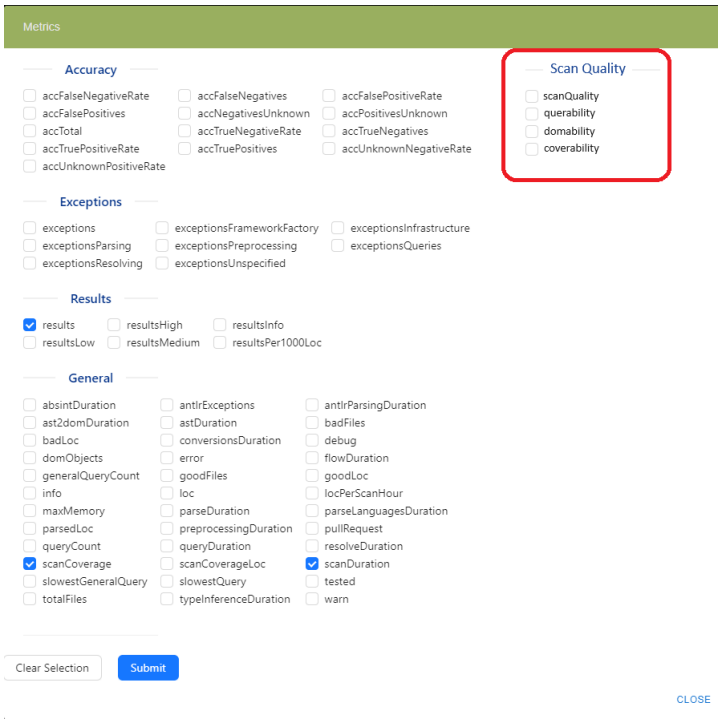


Figure 15: Improved Core Metrics for Quality Assessment After Integration

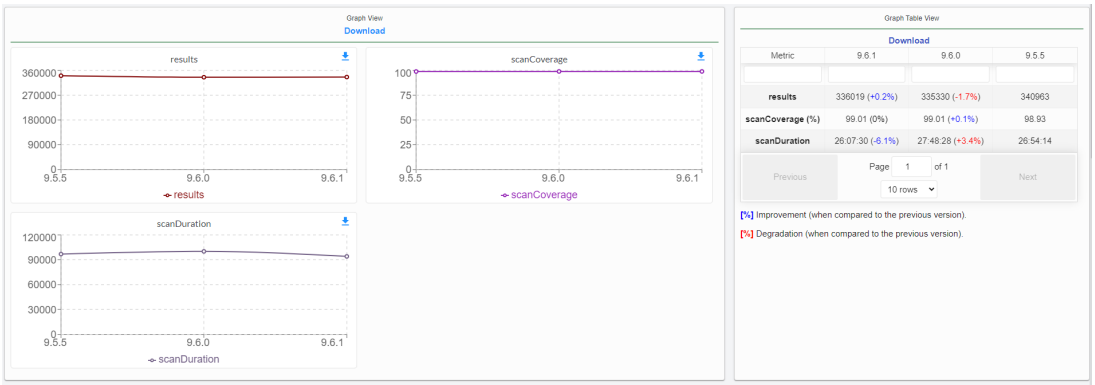


Figure 16: Initial Dashboard of *SK* Platform

Figure 17: Prototype of Scan Quality Integration in *SK* Platform

As a part of this master's dissertation, one of the primary objectives is to integrate the Scan Quality metric and its quality factors into the *SK* platform. The ultimate goal is to integrate this into the CI process. Figures 16 and 17 have been provided to demonstrate the before-and-after scenario when a user selects the metrics from Figures 14 and 15 within the *SK* dashboard. It is important to note that the values displayed in these figures are prototypes and do not represent actual data. In the following subsections, we will discuss the integration steps and considerations in detail.

4.7.1 Plan for Scan Quality Metric Integration

The successful integration of the Scan Quality Metric service (CxScanQuality) depends on two phases: Publishing and Deployment. In the Publishing phase, the objective is to make CxScanQuality a service accessible to other services, like the QA platform (*SK*). In contrast, the Deployment phase relies on utilizing a Container Orchestration Tool, simplifying containerized applications' management in a production environment. The workflow of these phases is elucidated in Figure 18.

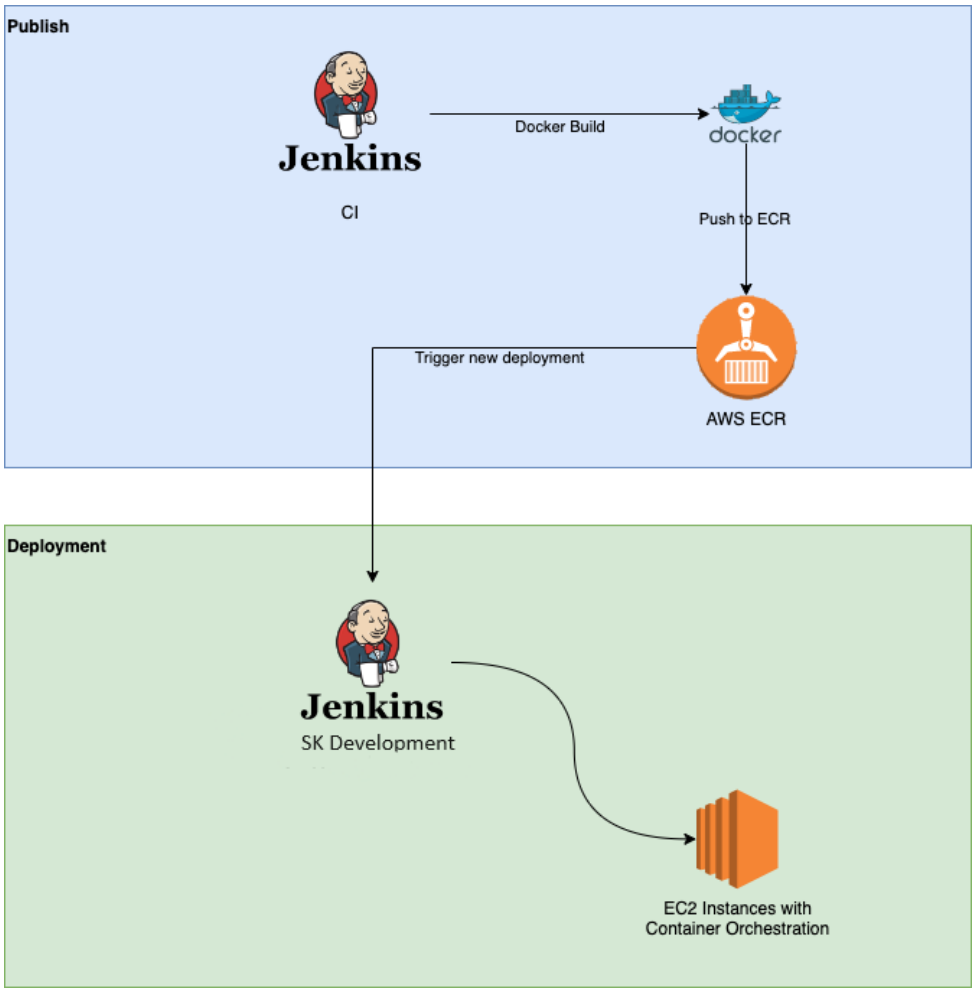


Figure 18: Scan Quality Metric Integration Phases

Figure 18 represents the entire Publishing and Deployment process workflow. In general, Jenkins plays a crucial role in this process, with a dedicated job responsible for building the Docker image of the CxScanQuality service and pushing it to the Amazon Elastic Container Registry (AWS ECR). Once available in AWS ECR, the deployment of the software is triggered whenever there is a new image or update. The deployment logic is hosted on EC2 Instances with Container orchestration software. This process will be explained in the subsequent subsections.

After completing the two phases, namely, Publish and Deployment, for the *SK*, the next step in the process is to integrate the CxScanQuality into the company's Continuous Integration (CI) pipeline. However, before the CxScanQuality service can be effectively implemented within the CI environment, a thorough analysis of the data from the CxScanQuality's database is required. This analysis requires accumulating data over 3 to 5 months, spanning at least two versions of SAST. This extended data collection period is essential to ensure that the service can only be integrated into the company's pipelines once it reaches a more stable version, thus guaranteeing its reliability and performance within the CI system. Figure 19 simplifies the workflow of this process, and each step is explained below:

1. A Jenkins job is triggered.

2. The CI projects run on the SAST engine.

3. Upon completion of each project, the "ci-sk-updater" tool is executed. It manages any necessary tasks, including logs or other functions. This tool acts as a wrapper, invoking the "logs-uploader-cli."

4. The "logs-uploader-cli" is responsible for sending the logs from each project to any service utilized by the *SK* platform.

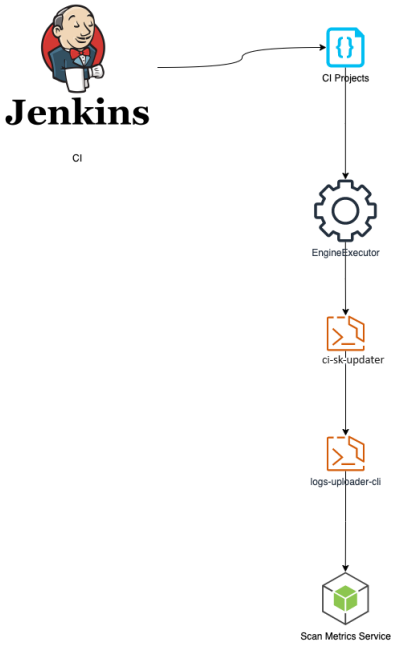


Figure 19: Continuous Integration Workflow

Integrating the Scan Quality Metric service (CxScanQuality) requires a multi-phased approach that covers both the Publishing and Deployment stages. It is crucial to integrate CxScanQuality into the CI pipeline to improve the quality and security of the company's software development processes.

4.7.2 Publish Phase

In the Publish phase, the creation of two complementary files is essential, requiring the following steps:

1. **Jenkinsfile:** In this type of pipeline, the definition of the pipeline is included within the code and is referred to as a declarative pipeline. Consequently, a pipeline with four stages, namely "build", "ECR", "Clean-up", and "Deploy", is created. The first stage is responsible for building the Container image of CxScanQuality. A tag, such as "latest," is assigned during this build command, which is necessary for subsequent steps. The ECR stage sends the Docker image to the Amazon Elastic Container Registry (AWS ECR). This pipeline exclusively

handles building the Docker image of the service and sending it to AWS ECR. This tool enables registering Docker images of existing services, making them accessible to those who need to use them. The "Clean Up" stage performs cleanup on the Docker image of the service. The "Deploy" stage is triggered whenever a newer image is available in AWS ECR, sending this information to another job responsible for deploying the *SK*. Additionally, it is essential to configure the Jenkins platform with the CxScanQuality service. During the configuration, specific information is required, such as indicating the repository where the CxScanQuality service is hosted, credentials for the repository, and the path to the Jenkinsfile within the repository.

2. **Dockerfile:** This file is responsible for creating the CxScanQuality image used by the Jenkinsfile. The Dockerfile contains instructions that detail how to configure and set up the container environment. These instructions include installing software packages, copying files into the container, defining environment variables, and executing specific commands.

In summary, Jenkins automates and orchestrates the software development pipeline, including building and deploying applications. Dockerfile defines the containerized environment in which the application runs. Combining these tools allows for efficient and consistent software development and deployment processes.

4.7.3 Deployment Phase

In the "Deployment" phase, it is necessary to modify a configuration file for the Container orchestration tool used in the *SK* Deployment process. It is important to modify the following information:

- **Name of the New Service:** Specify the new service's name.
- **AWS ECR Image Name:** This contains the link to the specific service in AWS ECR.
- **Port Definition:** The port number is typically an increment 1 to the previously defined service.
- **Environment File (if necessary):** Include the necessary environment file.
- **Policy Definitions (if necessary):** Define policies, such as the restart time in case of application failures.
- **Network Definition:** Specify the network to which the container of the CxScanQuality service belongs, typically "production."

In summary, this phase involves modifying a specific file with essential information mentioned previously. These configurations are indispensable for ensuring the smooth and secure deployment of *SK*, facilitating the CxScanQuality seamless integration.

4.8 Solution Validation

To ensure that the proposed solution is valid, it is important to evaluate how well it aligns with established requirements. The important features and functionalities that support the solution's effectiveness were initially explained in section Requirements. Table 7 provides an overview of the requirements that have been achieved, creating a strong validation framework.

The last column shows whether or not the requirement was achieved/implemented. The 'Y' represents Yes and 'N' No. The requirement with ID 4 has not been implemented because it already exists in the Engine Log service.

Table 7: Requirements Validation

ID	Requirement Category	Requirement Description	Applicable Service	Implemented?
1	Non-Functional	The solution must offer a comprehensive assessment of the scan quality for a project.	CxScanQuality	Y
2	Functional	Provide an object containing relevant information for each programming language in the scanned source code. The relevant data are the names and number of exceptions encountered during the various stages of the scanning process.	Engine Log	Y
3	Functional	Provide an object detailing the counts of successful and unsuccessful queries for each severity level and the overall totals for successful and unsuccessful queries.	Engine Log	Y
4	Functional	The system should supply an object including information on Lines of Code (LOC) and Parsed Lines of Code.	Engine Log	N
5	Functional	The solution must calculate and present the quality factor in percentages.	CxScanQuality	Y
6	Functional	The solution must be integrated into the quality platform, which is *SK*.	SK platform	Y
7	Functional	The solution must be integrated into the CI pipelines.	Not Applicable	Y
8	Functional	Implement a POST request to the Engine Log API to retrieve indicators related to each quality factor.	CxScanQuality	Y

4.9 Summary

Throughout this chapter, we delved into the implementation of scan quality metrics, covering technical details, strategic decision-making, and a thorough understanding of the approach taken to enhance the evaluation of scan quality. Furthermore, we have highlighted the requirements that were met.

5 RESULTS

The following section presents the outcome of computing the scan quality, detailing the assessment of Domability, Coverability, and Querability concerning the Scan Quality metric. Through a comprehensive examination, the results shed light on the effectiveness of the metrics in quantifying key aspects of scan quality.

The results analysis includes 160 projects spanning 23 distinct programming languages. It is pertinent to emphasize that each programming language exhibits varying levels of support, categorized into three tiers:

- **Tier 1** - Characterized by the highest level of support.
- **Tier 2** - Constituting intermediate support levels.
- **Tier 3** - Reflecting the lowest level of support among the languages under examination.

Following the analysis of the results, the discussion subsection critically analyzes the implications and significance of the findings, while the summary encapsulates the fundamental insights derived from this evaluation.

5.1 Data Analysis and Visualization Using PowerBI

To derive meaningful insights from the dataset generated by the CxScanQuality service, Microsoft PowerBI was employed as the chosen Business Intelligence (BI) tool. PowerBI was chosen based on factors such as accessibility and robust capabilities in transforming raw data into informative visualizations, all without incurring additional costs.

The data analysis approach in PowerBI consisted of several crucial steps, highlighting the accuracy and comprehensiveness of the analysis. Two tables with information for all 160 projects were imported into the PowerBI environment. The first table contained information for all projects, which can be seen in ANNEX I. The second table contained only the project name, programming language, and tier. Since each project had more than one programming language, it was necessary to split each language for each cell and make multiple entries for each project's name.

As a result, the following data transformations were carried out:

- **Grouping by Language:** The data was grouped by language and tier to gain insights into the distribution of projects across various programming languages. The total number of projects associated with each language was calculated, enabling comprehension of the prevalence of different programming languages within the dataset.
- **Filtering Rows:** To better understand the impact of the Scan Quality on each tier, the programming language was filtered by tier into separate tables. Each table includes the programming language, the average scan quality, and the total number of projects.

- **Removing Blank Spaces:** To ensure data accuracy and avoid duplication, blank spaces from the programming language cells were removed. This minor but essential transformation prevented erroneous language counting and provided a more precise representation of language usage.

Once the data was appropriately prepared, the powerful visualization capabilities of PowerBI were employed to generate the charts illustrated in Figures 20 and 21. Using PowerBI, along with the execution of data transformation and visualization, valuable insights were extracted from the dataset generated by the CxScanQuality service.

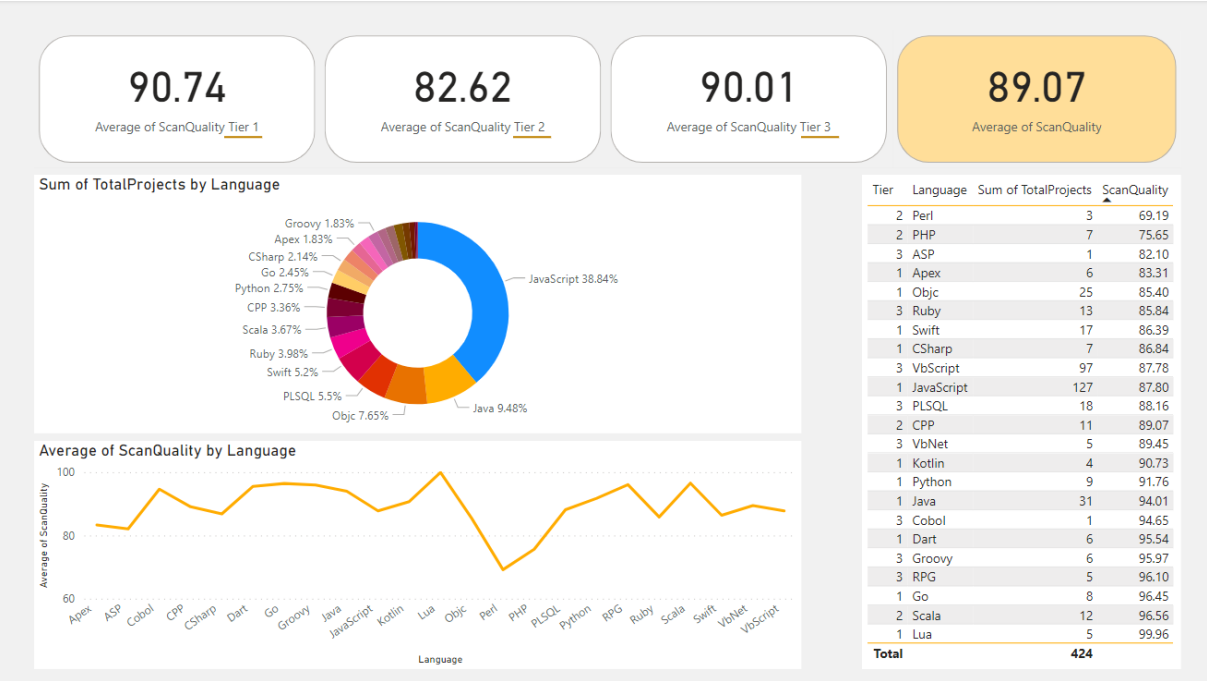
5.2 Results Analysis

To fully grasp the importance of Scan Quality for each project, it is crucial to understand that this metric expresses a quantifiable measure as a percentage. This metric is the output generated by CxScanQuality mentioned in section Configuring Scan Quality. This metric derives its value from Coverability, Domability, and Querability. Coverability and Domability each contribute 40% to the overall scan quality. This equivalence underscores their parallel impact on the scan's overall quality. In contrast, Querability contributes 20%, indicating its somewhat lesser but still meaningful influence on the scan's quality evaluation.

Each analyzed project contributed to constructing a dataset comprising 27 columns, as elaborated in ANNEX I. These columns include information such as scanned language, maximum Tier, exceptions encountered in each scan phase, total exceptions, LOC, parsed LOC, the total number of results (vulnerabilities), counts of successful and unsuccessful queries per severity, and the implemented metrics, namely Coverability, Domability, Querability, and Scan Quality.

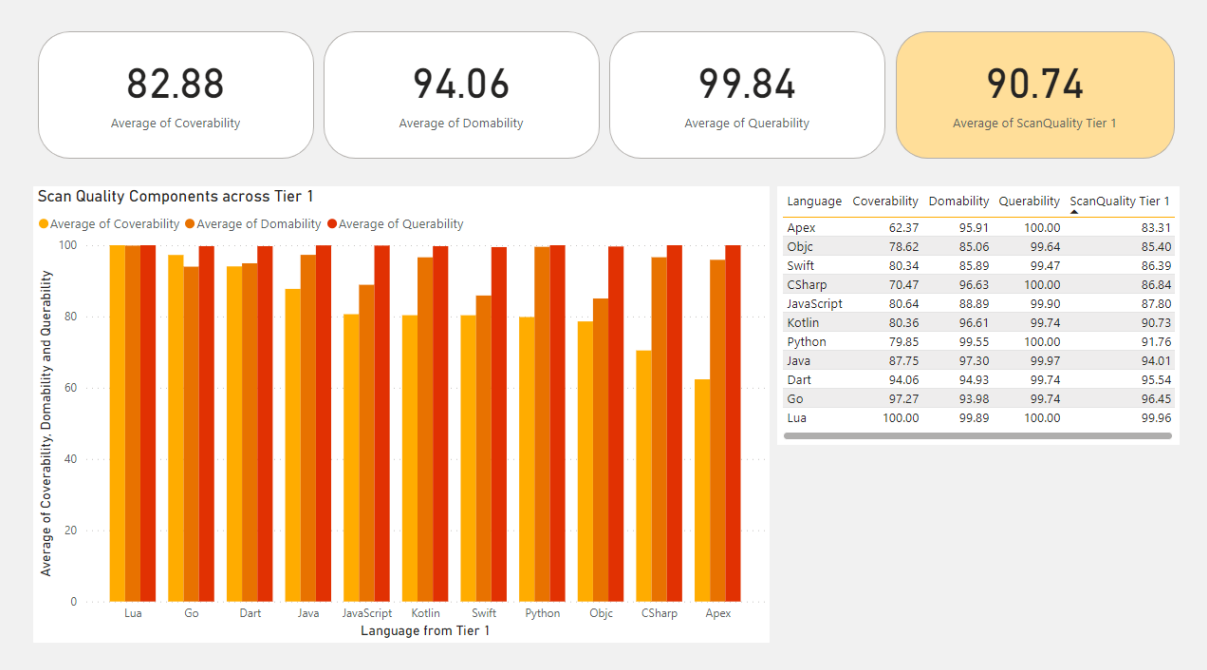
In Figure 20, a dashboard with multiple components is presented to provide a comprehensive view of scan quality across all projects in the dataset. The donut chart visually illustrates the distribution of projects across different programming languages, while the line chart showcases the average scan quality for each programming language. The table provides detailed numerical values corresponding to the charts. Note that the count exceeds 160 as many projects used multiple languages. The cards display different tiers' average scan quality, with an overall average of 89.07%.

To focus on tier-1 programming languages, Figure 21 presents another dashboard, highlighting a detailed analysis of these languages due to their high priority and support level. The clustered column chart reveals the scan quality factors, Coverability, Domability, and Querability for tier-1 programming languages. The cards display the average value of each quality factor, while the table showcases the average value visualized in the clustered column chart.



Tier	Language	Sum of TotalProjects	ScanQuality
2	Perl	3	69.19
2	PHP	7	75.65
3	ASP	1	82.10
1	Apex	6	83.31
1	Objc	25	85.40
3	Ruby	13	85.84
1	Swift	17	86.39
1	CSharp	7	86.84
3	VbScript	97	87.78
1	JavaScript	127	87.80
3	PLSQL	18	88.16
2	CPP	11	89.07
3	VbNet	5	89.45
1	Kotlin	4	90.73
1	Python	9	91.76
1	Java	31	94.01
3	Cobol	1	94.65
1	Dart	6	95.54
3	Groovy	6	95.97
3	RPG	5	96.10
1	Go	8	96.45
2	Scala	12	96.56
1	Lua	5	99.96
Total		**424**	

Figure 20: Dashboard for Scan Quality Analysis



Language	Coverability	Domability	Querability	ScanQuality Tier 1
Apex	62.37	95.91	100.00	83.31
Objc	78.62	85.06	99.64	85.40
Swift	80.34	85.89	99.47	86.39
CSharp	70.47	96.63	100.00	86.84
JavaScript	80.64	88.89	99.90	87.80
Kotlin	80.36	96.61	99.74	90.73
Python	79.85	99.55	100.00	91.76
Java	87.75	97.30	99.97	94.01
Dart	94.06	94.93	99.74	95.54
Go	97.27	93.98	99.74	96.45
Lua	100.00	99.89	100.00	99.96

Figure 21: Dashboard for Top-tier Analysis

Table 8: Summary of Projects Analyzed Data

	Project 1	Project 2	Project 3	Project 4
Language	JS, VbScript, ObjC, Swift	JS, PHP	JS, Apex	Lua
LOC	188159	646102	6039	9388
Parsed LOC	155883	308417	1030	9388
Exceptions_Parsing	501	99	49	0
Exceptions_TypeInference	2	11	0	0
Exceptions_AST_Construction	5	1	0	0
Exceptions_Conversions	0	0	0	0
Exceptions_DOM_Construction	119	49	0	0
Exceptions_Resolver	66	0	0	0
Exceptions_AbsInt	0	0	0	0
Exceptions_Queries	0	0	0	0
Total_Exceptions	693	160	49	0
Success_High	69	64	48	15
Insuccess_High	0	0	0	0
Success_Medium	111	128	93	40
Insuccess_Medium	0	0	0	0
Success_Low	132	94	93	52
Insuccess_Low	0	0	0	0
Success_Info	111	128	93	40
Insuccess_Info	0	0	0	0
Total_Results	348	5182	150	173
Coverability	82.85%	47.74%	17.06%	100%
Domability	20.76%	54.96%	78.6%	100%
Querability	100%	100%	100%	100%
Scan Quality	**61.44%**	**61.08%**	**58.26%**	**100%**
Scan Coverage (old method)	**97.18%**	**99.98%**	**100%**	**100%**

To gain a comprehensive understanding of the scan quality within the tier-1 projects, we will analyze four projects in more detail. By concentrating our efforts on a smaller, representative subset, we can explain the value of the Scan Quality metric. We based the project selection on the following criteria:

- **Number of Scan Exceptions:** The project selection criteria included the number of exceptions encountered during the scan, allowing for the identification of each scan phase's impact on the Domability score.
- **Scan Quality Score:** Selected projects had both high and low scan quality scores to ensure a diverse representation. Above-average projects had scores close to 100% or even 100%.
- **Programming Language Used by the Project:** Based on the distribution observed in Figure 20, JavaScript emerges as the predominant programming language with 127 projects, followed by Java and

Objective-C. Consequently, three selected projects used JavaScript in combination with other languages.

Table 8 presents the outcomes derived from 4 out of 160 projects that are well-suited for in-depth examination. Project 1 was selected because it combines programming languages from tier 1, which includes JavaScript, Objective-C, Swift, and the large number of exceptions that should impact the Domability score. Project 2, on the other hand, due to its high number of exceptions and Coverability score, makes it a suitable project for comparative analysis with Project 1. Project 3 was selected due to its low Scan Quality score among tier 1 projects. In contrast, Project 4 was chosen for a comparative analysis with Project 3, as it has the highest Scan Quality score and is a Lua project.

5.3 Discussion

This section delves into the key findings of this master's dissertation. The main objective of this dissertation was to gain a comprehensive understanding of the scan quality performed by a SAST tool. An extensive investigation of the overall scan quality of different programming languages was carried out to achieve this objective, as illustrated in Figure 20. Figure 21 demonstrates the overall scan quality for tier-1 projects. Table 8 provides a detailed analysis of four specific projects.

Let us begin by comparing Project 1 and Project 2. In terms of Coverability, Project 1 had a LOC count of 188,159 and 155,883 of Parsed LOC, resulting in a Coverability of 82.85%. On the other hand, Project 2 had a LOC count of 646,102 and 308,417 Parsed LOC, achieving a Coverability of 47.74%.

Regarding Domability, when we look at the *Total_Exceptions* indicator, it is clear that Project 1 has a significantly higher value, at 693, compared to the 160 exceptions observed in Project 2. Project 2's exceptions are concentrated in the Parsing stage, with 99 exceptions, followed by 11 in the Type Inference stage, 1 in AST Construction, and 49 in DOM Construction exceptions. In contrast, Project 1 presents exceptions spread across multiple stages, with 501 in Parsing, 2 in Type Inference, 5 in AST Construction, 119 in DOM Construction, and 66 in Resolver. The Domability value is higher for Project 2, which is 54.96% than Project 1, 20.76%, for the following three reasons:

1. Project 1 has a higher number of exceptions during the Parsing stage than Project 2. In the Domability formula, Q_{Parsing} is calculated by adding the exceptions that occur during the Parsing, Type Inference, AST Construction, and Conversions stages. Since Project 1 has more exceptions in these stages (except for the Type Inference stage) than Project 2, it directly impacts 60% of the Domability score.

2. Project 1 has more exceptions in DOM Construction than Project 2. DOM Construction is the second stage with more impact on the Domability score, which is 20%.

3. Project 1 has additional exceptions in the Resolver stage, while Project 2 has no exceptions in this stage. The Resolver stage impacts 15% of the Coverability score.

Regarding Querability, both Projects 1 and 2 achieved a 100% score, indicating that no query failures of any severity category (high, medium, low, or informational) were observed. However, when it comes to Scan Quality, Project 1 scored higher at 61.44% compared to Project 2 at 61.08%. It is necessary to note that both projects' Scan Quality scores were below the Tier 1 average of 90.74%.

Let us compare Project 3 and Project 4. Project 3 had a low Scan Quality score of 58.26% due to a coverability score of only 17.06%, indicating that only a tiny portion of LOC were parsed and analyzed effectively. Furthermore, Project 3's domability score was 78.6% because all 49 exceptions were related to the Parsing stage, which carries a high weight and significantly affected the Scan Quality score. However, Project 3 had a Querability score of 100%. In contrast, Project 4 scored 100% for all quality factors, unlike Project 3.

The dashboard in Figure 20 shows that the average Scan Quality for all languages across the three tiers is 89.07%. Tier 1 languages have a higher Scan Quality of 90.74%, while Tier 2 languages have a lower Scan Quality of 82.62% (below the average). Tier 3 languages have a greater Scan Quality of 90.01% than Tier 2, despite having lower support levels. This value is because most Tier 3 projects include some Tier 1 languages. On the other hand, most Tier 2 projects only have Tier 2 languages, with a few exceptions that may include the use of Tier 1 languages. The dashboard in Figure 21 highlights that Tier 1 languages such as *Python*, *Java*, *Dart*, *Go*, and *Lua* have exceeded the average scan quality, demonstrating their alignment with the established quality standards of their tier. However, six other languages have fallen below the average, indicating the need for additional efforts to improve scan quality. The language with the lowest Scan Quality is *Apex*.

Table 8 provides a comparison between the old and new methods of evaluating the scan quality of SAST. The old method used to rely on the Scan Coverage metric, which was assumed by the QA team to determine the scan quality. However, this metric presented almost 100% homogeneity across the majority of projects, as we can see from the values of Scan Coverage across the four projects: Project 1 - 97.18%, Project 2 - 99.98%, Project 3 - 100%, and Project 4 - 100%.

The new method of evaluating scan quality relies on the Scan Quality metric by the CxScanQuality service. Unlike Scan Coverage, Scan Quality allows us to identify the possible location of the problem, the first step to improve a specific project, and the priority projects and languages. This method has several benefits. Firstly, it empowers developers by allowing them to identify and address areas with lower scores, which makes troubleshooting and optimization easier. Secondly, it enables management roles, such as Product Managers, Directors, or Team Leaders, to make informed decisions by providing valuable information on the criticality of

projects and programming languages. This knowledge helps prioritize interventions and allocate resources efficiently.

During this master's dissertation, we came across several limitations. One of the main limitations we encountered was related to the weighting assigned to the Coverability, Domability, Querability, and Scan Quality formulas. We attempted to use statistical techniques to determine the optimal solution. However, unfortunately, we were unable to find any correlation between the accuracy of project results and their Scan Quality score.

Another limitation is the challenge of accurately naming exceptions that occur due to information gaps in the log files. To address this issue, we relied on the *domabilityThesis* field within the Engine Log service. This field helped to identify exception names by examining WARN or ERROR log entries. We established two new conventions for these scenarios, resulting in default exception names: "WARNUnknownExceptions" and "ERRORUnknownException".

In summary, Scan Quality is a more effective metric for evaluating scan quality than Scan Coverage, which only provides a value based on files recognized by SAST. Its benefits are numerous and make it an essential metric for both developers and management roles. These findings shed light on the significance of understanding and improving scan quality, ultimately contributing to enhanced security in software development practices.

5.4 Summary

This chapter delves into the findings obtained from this master's dissertation and provides a detailed discussion. To analyze these results, we used Microsoft PowerBI as a BI tool. Using this tool helped us gain deeper insights into the data extracted by the CxScanQuality service. However, it was necessary to perform specific data transformations to enhance the accuracy of the information. This analysis used a dataset of 160 projects from 23 different programming languages. The main focus of this analysis was to evaluate the expected scan quality across different programming languages and, in particular, Tier 1 programming languages. Four projects from Tier 1 were selected to illustrate distinct scenarios of Scan Quality scores.

6 CONCLUSION

In this master's dissertation, the primary objective was to establish a notion of scan quality conducted by SAST tools, mainly focusing on the data derived from each scanning phase. These phases yield crucial data points for evaluating the quality of scans, encompassing aspects such as exceptions generated, the number of parsed files, the volume of queries executed during the scan, which produces vulnerability findings, and the LOC identified, among other relevant metrics. Consequently, this master's dissertation has delineated Scan Quality into three distinct components - Coverability, Domability, and Querability. Each component contributes to the overall scan quality with varying degrees of impact. These components are quantifiable, and their contributions to Scan Quality are formalized within a comprehensive Scan Quality formula.

Furthermore, associated indicators have been developed to assess the relative significance of each component. In pursuit of these objectives, the CxScanQuality has been created, enabling the assignment of a Scan Quality score to log files generated from scanned projects. Before the development of this approach, obtaining such information required manual inference examining each log file individually, and the concept of Scan Quality remained largely unexplored within the field.

For this master's dissertation, five objectives were outlined. Below is explained how each one of them was successfully achieved:

1. Compare and analyze various security testing techniques and deepen knowledge of software quality and security. To tackle this objective, an extensive study and literature review were conducted. This objective involved an in-depth analysis of over 50 articles sourced through relevant keywords such as "software metrics", "software quality models", "quality metrics", "CI/CD", "KPIs", and "security vulnerability testing".

2. Identify key metrics and their weight in each scan phase of a SAST tool. To identify critical metrics and their respective weights in each scan phase of a SAST tool, a comprehensive analysis of numerous log files generated by CxSAST was undertaken. This analysis focused on pinpointing the start and end of each scan phase within the log files and extracting relevant information that directly impacts scan quality.

3. Research a formula to quantify overall scan quality, incorporating the metrics and their weights studied in the previous item. An initial version of the formula was proposed to formulate a comprehensive formula to quantify overall scan quality, including the metrics and weights studied in the previous objective. Subsequently, this version underwent substantial refinement, particularly in adjusting each indicator's coefficients within the formula, to determine a more precise means of assessing scan quality. Additionally, specific improvements were made at the indicator level due to limitations in the available information in the log files.

4. Develop ways of extracting the metrics from a scan. To achieve this objective, regular expressions were created to efficiently extract the necessary indicators for each component of scan quality. Furthermore, a

new service, CxScanQuality, was introduced alongside updates to an existing service, the engine log service.

5. Integrate the created formula to inform software release decisions by the Quality Assurance (QA) team. The final objective, involving the application of the formulated scan quality formula to inform software release decisions by the QA team, has also been accomplished. This was achieved by successfully integrating CxScanQuality into the QA platform and Continuous Integration (CI) pipelines, facilitating informed and data-driven QA decision-making.

Design Science Research (DSR) served as the guiding methodology for this study. DSR is a systematic approach focusing on creating innovative solutions to real-world problems. It combines the rigor of academic research with the practicality of solving practical issues. In this dissertation, DSR played a crucial role in enabling the creation of tangible solutions (CxScanQuality), which directly confronts the challenge of assessing the quality of SAST scans.

The proposed solution's findings are based on analyzing 160 projects across 23 distinct programming languages. To visualize the large dataset, we utilized the Microsoft PowerBI tool. We selected four projects from top-tier languages to serve as illustrative examples based on criteria such as the number of exceptions, the Scan Quality score, and the programming language used in each project.

The overall scan quality across the 23 languages is high, with an average score of 89.07%. This contribution holds particular significance for the QA team, as it provides them with relevant data on scan quality for all projects in the organization. This data empowers developers and decision-making for Management roles and offers a new way to ensure the quality of the Static Application Security Testing (SAST) product.

Furthermore, this work extends its implications to encompass organizational decision-makers. They can now incorporate another factor to consider into their decision-making processes. For instance, they could require that a new SAST version achieve a specific Scan Quality score before it is released or that a new version can not be released if the new Scan Quality is worse than the previous.

In conclusion, given the successful fulfillment of the study's requirements and objectives, the methodology's demonstrated effectiveness, and the promising results obtained, this dissertation contributes theoretically and practically to the domains of software security and software quality. The knowledge generated throughout this research provides a solution capable of establishing the scan quality of a SAST tool.

6.1 Future Work

Throughout the development of this master's dissertation, some interesting ideas came up. However, they could not be pursued owing to contextual constraints and the limitations of available time. Below are some of these ideas:

- **Real-time Scan Quality Integration:** In real-time, incorporate Scan Quality calculations into the log file during the scanning process. Currently, scan quality is determined after scanning, but the goal is to modify the log file directly to include a scan quality score.
- **CxScanQuality Integration in CI Pipelines:** This idea involves populating the CxScanQuality database with more data from two versions of the SAST product. This process typically takes three to four months to complete, as gathering and analyzing data between these versions is necessary.
- **Customize Domability formula:** Adapt the Domability formula to match the programming language used in each scanned project. Some languages have better support for specific stages of this formula. For that reason, the formula can be adapted for each case to have a suitable score.

REFERENCES

Aivatoglou, G., Anastasiadis, M., Spanos, G., Voulgaridis, A., Votis, K., & Tzovaras, D. (2021). A tree-based machine learning methodology to automatically classify software vulnerabilities. In *2021 ieee international conference on cyber security and resilience (csr)* (pp. 312–317).

Alenezi, M., & Almuairfi, S. (2019). Security risks in the software development lifecycle. *International Journal of Recent Technology and Engineering*, *8*(3), 7048–7055.

Alenezi, M., & Khellah, F. (2015). Evolution impact on architecture stability in open-source projects. *International Journal of Cloud Applications and Computing (IJCAC)*, *5*(4), 17.

Al-Ghamdi, A. (2013). A survey on software security testing techniques. *International Journal of Computer Science and Telecommunications*, *4*(4), 14–18.

Aloraini, B., Nagappan, M., German, D. M., Hayashi, S., & Higo, Y. (2019). An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software*, *158*, 110427.

Amoroso, E. (2018). Recent progress in software security. *IEEE Software*, *35*(2), 11–13.

Arachchi, S., & Perera, I. (2018). Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 moratuwa engineering research conference (mercon)* (pp. 156–161).

Assal, H., & Chiasson, S. (2018). Security in the software development lifecycle. In *Soups@ usenix security symposium* (pp. 281–296).

Baker, R., & Habli, I. (2012). An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, *39*(6), 787–805.

Bakota, T., Hegedűs, P., Siket, I., Ladányi, G., & Ferenc, R. (2014). Qualitygate sourceaudit: A tool for assessing the technical quality of software. In *2014 software evolution week-ieee conference on software maintenance, reengineering, and reverse engineering (csmr-wcre)* (pp. 440–445).

Beba, S., Karlsen, M. M., Li, J., & Zhang, B. (2021). Critical understanding of security vulnerability detection plugin evaluation reports. In *2021 28th asia-pacific software engineering conference (apsec)* (pp. 275–284).

Bhuiyan, F. A., Murphy, J., Morrison, P., & Rahman, A. (2021). Practitioner perception of vulnerability discovery strategies. In *2021 ieee/acm 2nd international workshop on engineering and cybersecurity of critical systems (encycris)* (pp. 41–44).

Bhuiyan, F. A., Rahman, A., & Morrison, P. (2020). Vulnerability discovery strategies used in software projects. In *Proceedings of the 35th ieee/acm international conference on automated software engineering workshops* (pp. 13–18).

Cadariu, M., Bouwers, E., Visser, J., & van Deursen, A. (2015). Tracking known security vulnerabilities in proprietary software systems. In *2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner)* (pp. 516–519).

Cains, M. G., Flora, L., Taber, D., King, Z., & Henshel, D. S. (2022). Defining cyber security and cyber security risk within a multidisciplinary context using expert elicitation. *Risk Analysis*, *42*(8), 1643–1669.

Chen, S.-J., Pan, Y.-C., Ma, Y.-W., & Chiang, C.-M. (2022). The impact of the practical security test during the software development lifecycle. In *2022 24th international conference on advanced communication technology (icact)* (pp. 313–316).

Chowdhury, A. A. M., & Arefeen, S. (2011). Software risk management : Importance and practices. In (p. 2078–5828). Citeseer.

Craigen, D., Diakun-Thibault, N., & Purse, R. (2014). Defining cybersecurity. *Technology Innovation Management Review*, *4*(10).

Dromey, R. (1995). A model for software product quality. *IEEE Transactions on Software Engineering*, *21*(2), 146–162.

Elder, S. (2021). Vulnerability detection is just the beginning. In *2021 ieee/acm 43rd international conference on software engineering: Companion proceedings (icse-companion)* (pp. 304–308).

Grady, R. B. (1992). *Practical software metrics for project management and process improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Gu, S., Kim, S. Y., Jeong, H.-h., & Sohn, K.-A. (2015). Constructing and exploiting software metrics networks for software quality assessment. In *2015 5th international conference on it convergence and security (icitcs)* (pp. 1–5).

Hao, G., Li, F., Huo, W., Sun, Q., Wang, W., Li, X., & Zou, W. (2019). Constructing benchmarks for supporting explainable evaluations of static application security testing tools. In *2019 international symposium on theoretical aspects of software engineering (tase)* (pp. 65–72).

Huang, Y., Bian, Y., Li, R., Zhao, J. L., & Shi, P. (2019). Smart contract security: A software lifecycle perspective. *IEEE Access*, *7*, 150184–150202.

Ihirwe, F., Di Ruscio, D., Gianfranceschi, S., & Pierantonio, A. (2022). Assessing the quality of low-code and model-driven engineering platforms for engineering iot systems. In *2022 ieee 22nd international conference on software quality, reliability and security (qrs)* (p. 583-594). doi: 10.1109/QRS57517.2022.00065

Imran, M., Alghamdi, A. A., & Ahmad, B. (2016). International journal of computer science and mobile computing software engineering: Architecture, design and frameworks. *International Journal of Computer Science and Mobile Computing*, *5*, 801-815.

Kan, S. H. (2002). *Metrics and models in software quality engineering* (2nd ed.). USA: Addison-Wesley Longman Publishing Co., Inc.

Kato, D., Shimizu, A., & Ishikawa, H. (2022). Quality classification for testing work in devops. In *Proceedings of the 14th international conference on management of digital ecosystems* (pp. 156–162).

Kaur, J., Alka, R., & Khan, A. (2018). Major software security risks at design phase. *ICIC Express Lett Int J Res Surv*.

Liao, Q. (2020, 10). Modelling ci/cd pipeline through agent-based simulation. *2020 IEEE International Symposium*

on Software Reliability Engineering Workshops (ISSREW), 155-156. doi: 10.1109/ISSREW51248.2020.00059

Lincke, R., Lundberg, J., & Löwe, W. (2008). Comparing software metrics tools. In *Proceedings of the 2008 international symposium on software testing and analysis* (p. 131–142). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1390630.1390648

Liu, B. C., Shi, L., Cai, Z., & Li, M. (2012). Software vulnerability discovery techniques: A survey. *2012 Fourth International Conference on Multimedia Information Networking and Security*, 152-156.

Liu, M., Tan, L., Yu, M., & Wang, Q. (2014). Research on appraisal target system of software product quality metrics and evaluation. In *2014 10th international conference on reliability, maintainability and safety (icrms)* (pp. 401–406).

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality. volume-iii. preliminary handbook on software quality for an acquisition manager* (Tech. Rep.). Fort Belvoir, VA: Defense Technical Information Center.

McGraw, G. (2004). Software security. *IEEE Security & Privacy*, *2*(2), 80–83.

McGraw, G. (2006). Software security: Building security in. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 6. doi: 10.1109/ISSRE.2006.43

McGraw, G., & Potter, B. (2004). Software security testing. *IEEE Security & Privacy*, *2*, 81-85.

Mead, N., Hough, E., & Stehney II, T. (2005). Security quality requirements engineering technical report. *Tech. Rep. CMU/SEI-2005-TR-009*.

Medeiros, N., Ivaki, N., Costa, P., & Vieira, M. (2017). Software metrics as indicators of security vulnerabilities. In *2017 ieee 28th international symposium on software reliability engineering (issre)* (pp. 216–227).

Mladenova, T. (2020). Software quality metrics–research, analysis and recommendation. In *2020 international conference automatics and informatics (icai)* (pp. 1–5).

Mowad, A. M., Fawareh, H., & Hassan, M. A. (2022). Effect of using continuous integration (ci) and continuous delivery (cd) deployment in devops to reduce the gap between developer and operation. In *2022 international arab conference on information technology (acit)* (pp. 1–8).

Nakai, H., Tsuda, N., Honda, K., Washizaki, H., & Fukazawa, Y. (2016). A square-based software quality evaluation framework and its case study. In *2016 ieee region 10 conference (tencon)* (p. 3704-3707). doi: 10.1109/TENCON.2016.7848750

Nistala, P., Nori, K. V., & Reddy, R. (2019). Software quality models: A systematic mapping study. In *2019 ieee/acm international conference on software and system processes (icssp)* (pp. 125–134).

Parizi, R. M., Qian, K., Shahriar, H., Wu, F., & Tao, L. (2018). Benchmark requirements for assessing software security vulnerability testing tools. In *2018 ieee 42nd annual computer software and applications conference (compsac)* (Vol. 1, pp. 825–826).

Payer, M. (2019). *Software security: Principles, policies, and protection.* HexHive Books, April.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45-77. doi: 10.2753/MIS0742-1222240302

Pereira, J. D. (2020). Techniques and tools for advanced software vulnerability detection. In *2020 ieee international symposium on software reliability engineering workshops (issrew)* (pp. 123–126).

Pereira, J. D., Campos, J. R., & Vieira, M. (2019). An exploratory study on machine learning to combine security vulnerability alerts from static analysis tools. In *2019 9th latin-american symposium on dependable computing (ladc)* (pp. 1–10).

Radatz, J. (1990). *Ieee standard glossary of software engineering terminology* (Vol. 12) (No. 610121990). IEEE.

Rangnau, T. (2020). Remco v. buijtenen, frank fransen, and fatih turkmen. continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In *2020 ieee 24th international enterprise distributed object computing conference (edoc)* (pp. 145–154).

Ruparelia, N. B. (2010). Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, *35*(3), 8–13.

Sheakh, T. (2015, 05). A comparative study of software testing techniques viz. white box testing black box testing and grey box testing. *International Journal of Allied Practice, Research and Review Website: www.ijaprr.com (ISSN 2350-1294)*, *Vol. II, Issue V, p.n. 01-08, 2015*, Vol. II, Issue V, p.n. 01-08, 2015.

Singh, B., & Kannojia, S. P. (2013). A review on software quality models. In *2013 international conference on communication systems and network technologies* (p. 801-806). doi: 10.1109/CSNT.2013.171

Venable, J., & Baskerville, R. (2012). Eating our own cooking: Toward a more rigorous design science of research methods. *Electronic Journal of Business Research Methods*, *10*(2), pp141–153.

Verdon, D., & McGraw, G. (2004). Risk analysis in software design. *IEEE Security & Privacy*, *2*(4), 79–84.

Yang, H., Zheng, S., Chu, W. C.-C., & Tsai, C.-T. (2012). Linking functions and quality attributes for software evolution. In *2012 19th asia-pacific software engineering conference* (Vol. 1, pp. 250–259).

ANNEX I

Table 9: General Information (1)

Project_Name	Language	Max_Tier
30DaysofSwift	JavaScript, Objc, Swift	1
Aerial	JavaScript, Objc, Swift	1
akka-main	Java, JavaScript, Scala	2
Alamofire	JavaScript, Objc, Swift	1
Angular2-master	Java, JavaScript, VbScript, Groovy, Scala	3
angular-cosmosdb-master	JavaScript, VbScript	3
Angular-Full-Stack-master	JavaScript, VbScript, PHP	3
AngularJS-CafeTownsend-master	JavaScript, VbScript	3
AngularJS-ColorGame-master	JavaScript, VbScript	3
AngularJS-info-cars	JavaScript, Apex, VbScript	3
AngularJS-java-server-midi-master	JavaScript, VbScript	3
AspNuke08052934_52934_lines	JavaScript, VbScript, ASP, PLSQL	3
asptemplate131487_1487_lines	JavaScript, VbNet, VbScript	3
ATE754	Cobol	3
backboneboilerplate-gh-pages-JSv2	JavaScript, VbScript	3
BBS400-main	RPG	3
Benchmark-master	Java, JavaScript, VbScript	3
Benchmark-master_JS	CSharp, JavaScript, VbScript	3
BitTorrent-5.0.9_ReWrite	JavaScript, VbScript, Python	3
blueblog	Java, JavaScript, VbScript	3
BookStoreJava_21403_lines	Java	1
BookStoreVBDotnet_22163_lines	VbNet	3
botpress-botpress-v12.19.2-35	JavaScript, VbScript, Ruby	3
breakableflask-master	Python	1
CAF-master	RPG	3
catiline-gh-pages	JavaScript, VbScript	3

Table 10: General Information (2)

Project_Name	Language	Max_Tier
celestia_1.6.0_198563lines	CPP, JavaScript, VbScript	3
Chocobozzz-PeerTube-v3.1.0-47	JavaScript, VbScript	3
ci-brakeman	Go	1
climbers_master	Objc	1
CSharp_Rewrite_Cosmos_181269lines	CSharp, JavaScript, VbScript	3
CSharp_Rewrite_log4net_72646_lines	CSharp, JavaScript, VbScript	3
CSharp_Rewrite_northwindtraders_aspnetcore	CSharp, JavaScript, VbScript	3
CSharp_Rewrite_Rainbow_209794_lines	CSharp, JavaScript, VbScript, PLSQL	3
CxPlayGround-Mobile-master	Java, JavaScript, Objc, Kotlin, Swift, Dart	1
CxPlayGround-Web	Java, JavaScript, VbScript, Groovy	3
dart-pad-master	Dart	1
dashboard-master	JavaScript, VbScript, PHP	3
deplate_30457_lines	JavaScript, VbScript, Ruby	3
df17-ant-to-sfdx-master	JavaScript, Apex	1
Dollar_Bets_master	Objc	1
drone-master_458434_lines_goV2	JavaScript, VbScript, PLSQL, Go	3
eclipse-theia-theia-v1.11.0-55	JavaScript, VbScript	3
emule4_172454_lines	CPP, JavaScript	2
evans_goV2	Go	1
finch-master	Scala	2
fintrospect-master	Scala	2
firefox-ios (Project 1)	JavaScript, VbScript, Objc, Swift	3
flame-main	Dart	1
FlightGear_1.0.0_230815lines	CPP, JavaScript, VbScript	3
flutter-master	Dart	1
flutter-quill-master	Dart	1

Table 11: General Information (3)

Project_Name	Language	Max_Tier
FusionChartsFree28681_28681_lines-JSv2	JavaScript, VbScript, Ruby, PLSQL	3
gatsby	JavaScript, VbScript, Ruby	3
GitterMobile-master	JavaScript, Objc	1
gizmo-master_11638_lines_goV2	JavaScript, Go	1
gogs_master_goV2	JavaScript, VbScript, PLSQL, Go	3
groove-dl-master_ReWrite	JavaScript, VbScript, Python	3
groovywebconsole-master	JavaScript, VbScript, Groovy	3
habitica-develop	CSharp, JavaScript, VbScript	3
hermes-master	JavaScript, Python	1
hooligram-client-develop	JavaScript, Objc	1
hugo-master_50147_lines_goV2	JavaScript, VbScript, Go	3
iFixit_iOS_master	Objc	1
Inuendo-master	RPG	3
jade_agents-master	Java, JavaScript, VbScript	3
jade-master-JSv2	JavaScript, VbScript	3
jasperreports_4_0_0_proje_350576_lines	Java, JavaScript, VbScript	3
Java11and12NewFeatures	Java, JavaScript, VbScript	3
jboard_jspff	Java, JavaScript, VbScript, PLSQL	3
jetspeed_1_6_206226_lines_jspff	Java, JavaScript, VbScript, PLSQL	3
juice-shop-4.2.1	JavaScript, VbScript, PHP	3
knockout_js_samples_master-JSv2	JavaScript, VbScript	3
leaky-angular-master	JavaScript, VbScript	3
LightningWorkingApp_ReWrite (Project 3)	JavaScript, Apex	1
lila-master	Java, JavaScript, VbScript, Scala	3
lua_samples-master	Lua	1
luaSample-master	Lua	1

Table 12: General Information (4)

Project_Name	Language	Max_Tier
lwc_goat	JavaScript, Apex, VbScript	3
mingw32_make	CPP	2
MISRA_C_celestia	CPP, JavaScript, VbScript	3
MISRA_C_FlightGear	CPP, JavaScript, VbScript	3
MISRA_C_mingw32	CPP	2
MISRA_C_UltimateToolbox93	CPP, JavaScript, VbScript	3
mleung_feather	JavaScript, VbScript, Ruby	3
MSDN	JavaScript, VbNet, VbScript, PLSQL	3
MTOS_4_38_en_390199_lines	JavaScript, VbScript, Perl	3
MySQLdb1-master_ReWrite	JavaScript, VbScript, Python	3
NetNewsWire	JavaScript, VbScript, Objc, Swift	3
nodejs-mysql-native-master-JSv2 (Project 4)	JavaScript, PLSQL	3
node-mongodb-native-1.4-JSv2_ReWrite	JavaScript, VbScript, Python	3
OSSILE-master	RPG	3
owasp-top10-salesforce-master	JavaScript, Apex	1
pebble_jspff	Java, JavaScript, VbScript	3
Pebble_Spring_Example	Java	1
perl_5_16_0_898436_lines	JavaScript, VbScript, Perl	3
personalblog_jspff	Java, JavaScript	1
PHP_Rewrite_AlegroCart_1.2.5_125254_lines-JSv2	JavaScript, VbScript, PHP, PLSQL	3
PHP_Rewrite_Sylius_1.2	JavaScript, VbScript, PHP	3
PHP_Rewrite_symfony-master(Project2)	JavaScript, PHP	2
play-groovy-master	Java, JavaScript, VbScript, Groovy	3
pokerBuddyApp-master	JavaScript, Ruby	3
polynote-master	Java, JavaScript, VbScript, Scala	3
Probabilistic-Programming	JavaScript, VbScript, Python	3
Programming_In_Lua-master	Lua	1

Table 13: General Information (5)

Project_Name	Language	Max_Tier
PSLQL_OSCOMMAND	PLSQL	3
Public-Corona-Lua-master(project 4)	Lua	1
quasar-dev	JavaScript, VbScript, Ruby	3
qxmpp_0_2_0_20478_lines	CPP, JavaScript, VbScript	3
railsgoat-master	JavaScript, VbScript, Ruby	3
react-datasheet-master	JavaScript, Ruby	3
react-native-elements-next	JavaScript, Ruby	3
react-security	JavaScript, VbScript, PHP, PLSQL	3
react-swipeable-views-master	JavaScript, VbScript, Ruby	3
rico2143347_43347_lines	JavaScript, VbNet, VbScript, PLSQL	3
riverpod-master	Dart	1
rmux	JavaScript, Go	1
roller_jspff	Java, JavaScript, VbScript, PLSQL	3
RxSwift	JavaScript, Objc, Swift	1
Scala_ReWrite_apache-samza-0.10.0-src	Java, JavaScript, VbScript, Scala	3
Scala_ReWrite_atlas-master	Java, JavaScript, VbScript, Scala	3
Scala_ReWrite_casbah-master-JSv2	JavaScript, VbScript, Scala	3
Scala_ReWrite_finagle-develop	Java, JavaScript, VbScript, PLSQL, Scala	3
Scala_ReWrite_miniboxing-plugin-wip	Java, Scala	2
shadowsocks-android-master	Java, JavaScript, VbScript, Groovy, Kotlin	3
singularity_master_goV2	JavaScript, Go	1
snipsnap	Java, JavaScript, VbScript	3
soundcloud-redux-master	JavaScript	1
Spring_Rewrite_AwesomeNotes	Java, JavaScript, VbScript, Kotlin	3
Spring_Rewrite_compass	Java, JavaScript, VbScript	3
Spring_Rewrite_SpringBootGoat	Java, JavaScript, VbScript, PLSQL	3
Swift_Rewrite_Cards-master	JavaScript, Objc, Swift	1
Swift_Rewrite_CVCalendar-master	JavaScript, Objc, Swift	1
Swift_Rewrite_ImagePickerTrayController	JavaScript, Objc, Swift	1
Swift_Rewrite_Maria-master	JavaScript, Objc, Swift	1

Table 14: General Information (6)

Project_Name	Language	Max_Tier
Swift_Rewrite_SwiftGoat-master	JavaScript, VbScript, Objc, Swift	3
Swift_Rewrite_Swift	Objc, Swift	1
Swift_Rewrite_SwiftLanguageWeather	JavaScript, Objc, Swift	1
Swift_Rewrite_Swift-Master_Of_Swift	JavaScript, VbScript, Objc, Swift	3
Swift_Rewrite_SwiftMessages-master	JavaScript, Objc, Swift	1
Swift_Rewrite_Xniffer-master	JavaScript, Objc, Swift	1
tables-3.0.0_ReWrite	JavaScript, VbScript, Python	3
testcodav3_79566_lines_ReWrite	Apex	1
testVbNet	JavaScript, VbNet, VbScript	3
tiny-dnn-master_cpp11	CPP, JavaScript, VbScript	3
trape-master	JavaScript, VbScript, Python	3
Twidere-Android-master-Kotlin	Java, JavaScript, VbScript, Groovy, Kotlin	3
twitter-finagle	Java, JavaScript, VbScript, PLSQL, Scala	3
UltimateToolbox93_src_322262_lines	CPP, JavaScript, VbScript	3
Verademo-Dotnet	CSharp, JavaScript, VbScript	3
Vertical-Fill-master	Lua	1
vuetify-master	JavaScript, VbScript, Ruby	3
vulnerable-app-master	JavaScript, VbScript	3
vux-2	JavaScript, VbScript, Ruby	3
WebGoat_5.4_117234_lines_jspff	Java, JavaScript, VbScript, PLSQL	3
webmin_1_570_220564_lines	JavaScript, VbScript, Perl	3
WeiPulse_OpenSource	JavaScript, VbScript, Objc	3
wikihow_iphone_app_master	JavaScript, VbScript, Objc	3
WordPress_iOS_develop	JavaScript, VbScript, Objc	3
xmlservice-master	RPG	3

Table 15: Exceptions Information (1)

Project_Name	Exceptions_Parsing	Exceptions_TypeInference	Exceptions_AST_Construction	Exceptions_Conversions	Exceptions_DOM_Construction	Exceptions_Resolver	Exceptions_AbsInt	Exceptions_Queries	Total_Exceptions
30DaysofSwift	3	0	0	0	1	0	0	0	4
Aerial	0	0	0	0	0	0	0	0	0
akka-main	0	0	4	0	7	1	0	0	12
Alamofire	4	0	0	0	3	1	0	2	10
Angular2-master	0	8	33	0	0	0	0	0	41
angular-cosmosdb-master	0	0	0	0	0	0	0	0	0
Angular-Full-Stack-master	0	0	0	0	0	1	0	0	1
AngularJS-CafeTownsend-master	0	0	0	0	0	0	0	0	0
AngularJS-ColorGame-master	0	0	0	0	0	0	0	0	0
AngularJS-info-cars	0	0	0	0	0	0	0	0	0
AngularJS-java-server-midi-master	0	0	0	0	0	0	0	0	0
AspNuke08052934_52934_lines	124	1	0	0	0	0	0	0	125
asptemplate131487_1487_lines	0	0	0	0	0	0	0	0	0
ATE754	28	0	0	0	0	0	0	0	28
backboneboilerplate-gh-pages-JSv2	0	2	0	0	0	0	0	0	2
BBS400-main	3	0	0	0	0	0	0	0	3
Benchmark-master	15	0	0	0	0	0	0	0	15
Benchmark-master_JS	0	0	0	0	0	0	0	0	0
BitTorrent-5.0.9_ReWrite	0	0	0	0	5	0	0	0	5
blueblog	0	0	0	0	0	0	0	0	0
BookStoreJava_21403_lines	0	0	0	0	0	0	0	0	0
BookStoreVBDotnet_22163_lines	2	0	0	0	0	0	0	0	2
botpress-botpress-v12.19.2-35	0	58	12	0	0	0	0	0	70
breakableflask-master	0	0	0	0	0	0	0	0	0
CAF-master	143	0	0	0	0	0	0	0	143
catiline-gh-pages	0	2	0	0	0	0	0	0	2

Table 16: Exceptions Information (2)

Project_Name	Exceptions_Parsing	Exceptions_TypeInference	Exceptions_AST_Construction	Exceptions_Conversions	Exceptions_DOM_Construction	Exceptions_Resolver	Exceptions_AbsInt	Exceptions_Queries	Total_Exceptions
celestia_1.6.0_198563lines	262	0	4	0	0	0	0	0	266
Chocobozzz-PeerTube-v3.1.0-47	0	108	37	0	0	0	0	0	145
ci-brakeman	0	0	0	0	0	0	0	0	0
climbers_master	61	0	0	0	0	0	0	0	61
CSharp_Rewrite_Cosmos_181269lines	0	2	0	0	0	0	0	0	2
CSharp_Rewrite_log4net_72646_lines	0	42	0	0	0	0	0	0	42
CSharp_Rewrite_northwindtraders_aspnetcore	0	0	0	0	0	0	0	0	0
CSharp_Rewrite_Rainbow_209794_lines	0	0	0	0	0	0	0	0	0
CxPlayGround-Mobile-master	22	0	0	0	0	0	0	0	22
CxPlayGround-Web	0	3	0	0	0	0	0	0	3
dart-pad-master	0	0	0	0	0	0	0	0	0
dashboard-master	0	0	0	0	1	0	0	0	1
deplate_30457_lines	1	2	0	0	0	0	0	0	3
df17-ant-to-sfdx-master	0	4	2	0	0	0	0	0	6
Dollar_Bets_master	1	0	0	0	0	0	0	0	1
drone-master_458434_lines_goV2	0	0	1	0	14	0	0	8	23
eclipse-theia-theia-v1.11.0-55	0	16	1	0	0	0	0	0	17
emule4_172454_lines	121	0	2	0	0	0	0	0	123
evans_goV2	0	0	0	0	0	0	0	0	0
finch-master	0	0	0	0	2	0	0	0	2
fintrospect-master	0	0	0	0	0	0	0	0	0
firefox-ios (Project 1)	501	2	5	0	119	66	0	0	693
flame-main	0	0	0	0	0	0	0	0	0
FlightGear_1.0.0_230815lines	0	0	1	0	0	0	0	0	1
flutter-master	44	0	0	0	0	0	0	6	50
flutter-quill-master	0	0	0	0	0	0	0	0	0

Table 17: Exceptions Information (3)

Project_Name	Exceptions_Parsing	Exceptions_TypeInference	Exceptions_AST_Construction	Exceptions_Conversions	Exceptions_DOM_Construction	Exceptions_Resolver	Exceptions_AbsInt	Exceptions_Queries	Total_Exceptions
FusionChartsFree28681_28681_lines-JSv2	0	0	0	0	0	0	0	0	0
gatsby	0	182	7	0	0	0	0	0	189
GitterMobile-master	63	0	0	0	0	0	0	0	63
gizmo-master_11638_lines_goV2	0	0	0	0	0	0	0	0	0
gogs_master_goV2	151	1	0	0	0	0	0	0	152
groove-dl-master_ReWrite	0	0	0	0	0	0	0	0	0
groovywebconsole-master	0	0	0	0	0	0	0	0	0
habitica-develop	0	7	0	0	0	0	0	0	7
hermes-master	0	0	0	0	0	0	0	0	0
hooligram-client-develop	0	0	0	0	0	0	0	0	0
hugo-master_50147_lines_goV2	0	0	0	0	0	0	0	0	0
iFixit_iOS_master	20	0	0	0	0	0	0	0	20
Inuendo-master	0	0	0	0	0	0	0	0	0
jade_agents-master	0	0	0	0	0	0	0	0	0
jade-master-JSv2	32	99	1	0	0	0	0	0	132
jasperreports_4_0_0_proje_350576_lines	1	0	0	0	0	0	0	0	1
Java11and12NewFeatures	8	0	0	0	1	0	0	0	9
jboard_jspff	2	0	0	0	0	0	0	0	2
jetspeed_1_6_206226_lines_jspff	2	5	0	0	0	0	0	0	7
juice-shop-4.2.1	0	0	0	0	0	0	0	0	0
knockout_js_samples_master-JSv2	0	0	0	0	0	0	0	0	0
leaky-angular-master	0	0	0	0	0	0	0	0	0
LightningWorkingApp_ReWrite (Project 3)	49	0	0	0	0	0	0	0	49
lila-master	0	0	0	0	5	0	0	0	5
lua_samples-master	1	0	0	0	0	0	0	0	1
luaSample-master	0	0	0	0	0	0	0	0	0

Table 18: Exceptions Information (4)

Project_Name	Exceptions_Parsing	Exceptions_TypeInference	Exceptions_AST_Construction	Exceptions_Conversions	Exceptions_DOM_Construction	Exceptions_Resolver	Exceptions_AbsInt	Exceptions_Queries	Total_Exceptions
lwc_goat	0	0	0	0	0	0	0	0	0
mingw32_make	21	0	1	0	0	0	0	0	22
MISRA_C_celestia	262	0	4	0	0	0	0	0	266
MISRA_C_FlightGear	0	0	1	0	0	0	0	0	1
MISRA_C_mingw32	21	0	1	0	0	0	0	0	22
MISRA_C_UltimateToolbox93	124	0	0	0	0	0	0	0	124
mleung_feather	14	0	0	0	0	2	0	0	16
MSDN	1	0	0	0	0	5	0	0	6
MTOS_4_38_en_390199_lines	202	2	0	0	0	0	0	0	204
MySQLdb1-master_ReWrite	0	0	0	0	0	0	0	0	0
NetNewsWire	35	3	2	0	10	0	0	2	52
nodejs-mysql-native-master-JSv2 (Project 4)	0	0	0	0	0	0	0	0	0
node-mongodb-native-1.4-JSv2_ReWrite	0	2	0	0	0	0	0	0	2
OSSILE-master	0	0	0	0	0	0	0	0	0
owasp-top10-salesforce-master	0	0	0	0	0	0	0	0	0
pebble_jspff	6	0	0	0	0	0	0	0	6
Pebble_Spring_Example	0	0	0	0	0	0	0	0	0
perl_5_16_0_898436_lines	1659	0	0	0	0	0	0	0	1659
personalblog_jspff	0	0	0	0	0	0	0	0	0
PHP_Rewrite_AlegroCart_1.2.5_125254_lines-JSv2	122	0	1	0	75	0	0	0	198
PHP_Rewrite_Sylius_1.2	39	4	0	0	8	0	0	0	51
PHP_Rewrite_symfony-master(Project2)	99	11	1	0	49	0	0	0	160
play-groovy-master	0	0	0	0	0	0	0	0	0
pokerBuddyApp-master	0	0	0	0	0	0	0	0	0
polynote-master	0	0	0	0	9	0	0	0	9

Table 19: Exceptions Information (5)

Project_Name	Exceptions_Parsing	Exceptions_TypeInference	Exceptions_AST_Construction	Exceptions_Conversions	Exceptions_DOM_Construction	Exceptions_Resolver	Exceptions_AbsInt	Exceptions_Queries	Total_Exceptions
Probabilistic-Programming	0	0	0	0	0	0	0	0	0
Programming_In_Lua-master	0	0	0	0	0	0	0	0	0
PSLQL_OSCOMMAND	0	0	0	0	0	0	0	0	0
Public-Corona-Lua-master(project 4)	0	0	0	0	0	0	0	0	0
quasar-dev	0	42	0	0	0	0	0	0	42
qxmpp_0_2_0_20478_lines	16	0	0	0	0	0	0	0	16
railsgoat-master	61	0	0	0	0	0	0	0	61
react-datasheet-master	0	0	0	0	0	0	0	0	0
react-native-elements-next	0	2	0	0	0	0	0	0	2
react-security	0	0	0	0	0	0	0	0	0
react-swipeable-views-master	0	2	0	0	0	0	0	0	2
rico2143347_43347_lines	10	50	12	0	0	0	0	0	72
riverpod-master	0	0	0	0	0	0	0	0	0
rmux	0	0	0	0	0	0	0	0	0
roller_jspff	14	0	0	0	0	0	0	0	14
RxSwift	179	23	28	0	64	0	0	34	328
Scala_ReWrite_apache-samza-0.10.0-src	1	0	0	0	0	0	0	0	1
Scala_ReWrite_atlas-master	0	0	0	0	0	0	0	0	0
Scala_ReWrite_casbah-master-JSv2	0	0	1	0	0	0	0	0	1
Scala_ReWrite_finagle-develop	0	0	0	0	2	27	0	0	29
Scala_ReWrite_miniboxing-plugin-wip	0	0	0	0	2	0	0	0	2
shadowsocks-android-master	0	0	0	0	0	0	0	0	0
singularity_master_goV2	1	0	0	0	0	0	0	0	1
snipsnap	4	4	0	0	0	0	0	0	8
soundcloud-redux-master	0	0	0	0	0	0	0	0	0
Spring_Rewrite_AwesomeNotes	0	0	0	0	0	0	0	0	0

Table 20: Exceptions Information (6)

Project_Name	Exceptions_Parsing	Exceptions_TypeInference	Exceptions_AST_Construction	Exceptions_Conversions	Exceptions_DOM_Construction	Exceptions_Resolver	Exceptions_AbsInt	Exceptions_Queries	Total_Exceptions
Spring_Rewrite_compass	0	0	0	0	0	0	0	0	0
Spring_Rewrite_SpringBootGoat	1	0	0	0	0	0	0	0	1
Swift_Rewrite_Cards-master	0	0	0	0	0	0	0	0	0
Swift_Rewrite_CVCalendar-master	4	2	2	0	2	0	0	0	10
Swift_Rewrite_ImagePickerTrayController	0	0	0	0	0	0	0	0	0
Swift_Rewrite_Maria-master	0	0	0	0	0	0	0	0	0
Swift_Rewrite_SwiftGoat-master	73	0	0	0	1	0	0	0	74
Swift_Rewrite_Swift	0	0	0	0	1	0	0	0	1
Swift_Rewrite_SwiftLanguageWeather	0	0	0	0	0	0	0	0	0
Swift_Rewrite_Swift-Master_Of_Swift	77	0	0	0	0	0	0	0	77
Swift_Rewrite_SwiftMessages-master	0	0	1	0	0	0	0	0	1
Swift_Rewrite_Xniffer-master	0	0	0	0	0	0	0	0	0
tables-3.0.0_ReWrite	0	4	0	0	0	0	0	0	4
testcodav3_79566_lines_ReWrite	0	0	0	0	0	0	0	0	0
testVbNet	29	0	0	0	0	5	0	0	34
tiny-dnn-master_cpp11	19	0	1	0	12	1	0	0	33
trape-master	0	0	0	0	0	0	0	0	0
Twidere-Android-master-Kotlin	1	0	1	0	10	0	0	4	16
twitter-finagle	0	0	0	0	4	0	0	0	4
UltimateToolbox93_src_322262_lines	124	0	0	0	0	0	0	0	124
Verademo-Dotnet	0	0	0	0	0	0	0	0	0
Vertical-Fill-master	0	0	0	0	0	0	0	0	0

Table 21: Exceptions Information (7)

Project_Name	Exceptions_Parsing	Exceptions_TypeInference	Exceptions_AST_Construction	Exceptions_Conversions	Exceptions_DOM_Construction	Exceptions_Resolver	Exceptions_AbsInt	Exceptions_Queries	Total_Exceptions
vuetify-master	0	40	7	0	1	0	0	0	48
vulnerable-app-master	0	0	0	0	0	0	0	0	0
vux-2	0	67	4	0	0	0	0	0	71
WebGoat_5.4_117234_lines_jspff	13	0	0	0	0	0	0	0	13
webmin_1_570_220564_lines	541	0	0	0	0	0	0	0	541
WeiPulse_OpenSource	25	0	0	0	0	0	0	0	25
wikihow_iphone_app_master	0	0	0	0	0	0	0	0	0
WordPress_iOS_develop	621	3	0	0	0	0	0	0	624
xmlservice-master	7	0	0	0	0	0	0	0	7

Table 22: LOC and Total Results Information (1)

Project_Name	LOC	Parsed_LOC	Total_Results
30DaysofSwift	19539	10856	211
Aerial	35764	22195	31
akka-main	494098	494080	3801
Alamofire	34910	34221	64
Angular2-master	422766	405413	799
angular-cosmosdb-master	13217	559	3
Angular-Full-Stack-master	19181	2109	106
AngularJS-CafeTownsend-master	1403	1204	3
AngularJS-ColorGame-master	760	759	9
AngularJS-info-cars	1033	621	3
AngularJS-java-server-midi-master	217	216	1
AspNuke08052934_52934_lines	63804	61100	1496
asptemplate131487_1487_lines	1467	726	8
ATE754	8551	8551	30
backboneboilerplate-gh-pages-JSv2	41940	41847	223
BBS400-main	7118	7118	359
Benchmark-master	366449	364484	27350
Benchmark-master_JS	105902	102745	1085
BitTorrent-5.0.9_ReWrite	82933	82930	206
blueblog	4626	4431	276
BookStoreJava_21403_lines	24517	24517	5069
BookStoreVBDotnet_22163_lines	7864	6556	577
botpress-botpress-v12.19.2-35	155851	154804	476
breakableflask-master	243	243	25
CAF-master	45055	45055	997
catiline-gh-pages	17797	17718	114

Table 23: LOC and Total Results Information (2)

Project_Name	LOC	Parsed_LOC	Total_Results
celestia_1.6.0_198563lines	192136	183157	4929
Chocobozzz-PeerTube-v3.1.0-47	166435	165628	665
ci-brakeman	109883	109811	729
climbers_master	52677	52445	155
CSharp_Rewrite_Cosmos_181269lines	140229	138550	1063
CSharp_Rewrite_log4net_72646_lines	208386	80401	543
CSharp_Rewrite_northwindtraders_aspnetcore	20424	7649	54
CSharp_Rewrite_Rainbow_209794_lines	301275	234617	5386
CxPlayGround-Mobile-master	12946	8331	549
CxPlayGround-Web	4221	3930	110
dart-pad-master	15607	15607	73
dashboard-master	51978	30107	164
deplate_30457_lines	30696	30387	26
df17-ant-to-sfdx-master	5797	5773	225
Dollar_Bets_master	27827	5886	13
drone-master_458434_lines_goV2	272394	261377	3303
eclipse-theia-theia-v1.11.0-55	230924	230762	173
emule4_172454_lines	168872	167336	4171
evans_goV2	84828	84827	425
finch-master	5719	5719	3
fintrospect-master	6294	6294	2
firefox-ios (Project 1)	188159	155883	348
flame-main	50639	50639	118
FlightGear_1.0.0_230815lines	238487	234094	11897
flutter-master	1351024	1351024	564
flutter-quill-master	18541	18541	16

Table 24: LOC and Total Results Information (3)

Project_Name	LOC	Parsed_LOC	Total_Results
FusionChartsFree28681_28681_lines-JSv2	76631	34822	359
gatsby	117449	105081	243
GitterMobile-master	17994	16694	44
gizmo-master_11638_lines_goV2	11649	11559	75
gogs_master_goV2	161824	147615	572
groove-dl-master_ReWrite	865	863	25
groovywebconsole-master	4728	4727	51
habitica-develop	371278	162438	931
hermes-master	16615	16613	42
hooligram-client-develop	13037	4993	36
hugo-master_50147_lines_goV2	52556	51570	365
iFixit_iOS_master	29850	17671	148
Inuendo-master	12345	12345	16
jade_agents-master	195675	195112	10895
jade-master-JSv2	7478	4125	20
jasperreports_4_0_0_proje_350576_lines	1799156	1749597	11150
Java11and12NewFeatures	2066	2057	280
jboard_jspff	89670	75815	1927
jetspeed_1_6_206226_lines_jspff	305325	281301	7114
juice-shop-4.2.1	19038	15377	124
knockout_js_samples_master-JSv2	13857	13299	82
leaky-angular-master	12164	1474	30
LightningWorkingApp_ReWrite (Project 3)	6039	1030	150
lila-master	148386	146289	248
lua_samples-master	3242	3242	63
luaSample-master	727	727	18

Table 25: LOC and Total Results Information (4)

Project_Name	LOC	Parsed_LOC	Total_Results
lwc_goat	15618	1605	50
mingw32_make	35050	34327	2158
MISRA_C_celestia	192136	183157	29423
MISRA_C_FlightGear	238487	234094	41571
MISRA_C_mingw32	35050	34327	8085
MISRA_C_UltimateToolbox93	323231	322922	41767
mleung_feather	50844	46730	145
MSDN	88582	84306	983
MTOS_4_38_en_390199_lines	373959	335526	6013
MySQLdb1-master_ReWrite	7701	4472	8
NetNewsWire	106188	91305	247
nodejs-mysql-native-master-JSv2 (Project 4)	3173	3173	24
node-mongodb-native-1.4-JSv2_ReWrite	58639	58169	357
OSSILE-master	14867	14854	53
owasp-top10-salesforce-master	638	556	49
pebble_jspff	53546	49894	2372
Pebble_Spring_Example	354	354	9
perl_5_16_0_898436_lines	765937	480567	11466
personalblog_jspff	24277	13237	1313
PHP_Rewrite_AlegroCart_1.2.5_125254_lines-JSv2	178143	165848	2421
PHP_Rewrite_Sylius_1.2	262028	244642	1748
PHP_Rewrite_symfony-master(Project2)	646102	308417	5182
play-groovy-master	2980	2647	68
pokerBuddyApp-master	17948	8397	398
polynote-master	44900	17889	13
Probabilistic-Programming	184048	1765	12
Programming_In_Lua-master	1246	1246	7

Table 26: LOC and Total Results Information (5)

Project_Name	LOC	Parsed_LOC	Total_Results
PSLQL_OSCOMMAND	4025	1682	26
Public-Corona-Lua-master(project 4)	9388	9388	173
quasar-dev	166306	141865	377
qxmpp_0_2_0_20478_lines	53212	52292	365
railsgoat-master	87365	84959	863
react-datasheet-master	18500	8987	25
react-native-elements-next	27369	26756	21
react-security	13904	826	32
react-swipeable-views-master	27139	26616	30
rico2143347_43347_lines	77697	66694	516
riverpod-master	48655	48655	15
rmux	5762	5748	69
roller_jspff	99244	92070	4160
RxSwift	99883	98549	80
Scala_ReWrite_apache-samza-0.10.0-src	50802	48077	581
Scala_ReWrite_atlas-master	32324	32316	22
Scala_ReWrite_casbah-master-JSv2	288471	287629	902
Scala_ReWrite_finagle-develop	148294	146996	1884
Scala_ReWrite_miniboxing-plugin-wip	72952	72952	921
shadowsocks-android-master	10181	8685	70
singularity_master_goV2	85769	80563	336
snipsnap	445588	81090	6194
soundcloud-redux-master	20818	7696	20
Spring_Rewrite_AwesomeNotes	5595	4024	126
Spring_Rewrite_compass	1217321	1204701	7756
Spring_Rewrite_SpringBootGoat	2049	1975	112
Swift_Rewrite_Cards-master	4447	3910	15
Swift_Rewrite_CVCalendar-master	5317	5119	20
Swift_Rewrite_ImagePickerTrayController	1684	1294	7
Swift_Rewrite_Maria-master	4092	2954	5

Table 27: LOC and Total Results Information (6)

Project_Name	LOC	Parsed_LOC	Total_Results
Swift_Rewrite_SwiftGoat-master	18341	9005	91
Swift_Rewrite_Swift	2232	2193	14
Swift_Rewrite_SwiftLanguageWeather	1709	1428	13
Swift_Rewrite_Swift-Master_Of_Swift	62602	58912	598
Swift_Rewrite_SwiftMessages-master	4924	3749	16
Swift_Rewrite_Xniffer-master	1379	1158	17
tables-3.0.0_ReWrite	232167	142206	300
testcodav3_79566_lines_ReWrite	79566	79566	1385
testVbNet	100501	100091	1345
tiny-dnn-master_cpp11	137537	126724	1540
trape-master	14710	14618	120
Twidere-Android-master-Kotlin	95731	95592	213
twitter-finagle	179200	178349	803
UltimateToolbox93_src_322262_lines	323231	322922	3578
Verademo-Dotnet	5720	5709	85
Vertical-Fill-master	122	122	12
vuetify-master	127057	95118	350
vulnerable-app-master	6217	5831	17
vux-2	158243	75519	179
WebGoat_5.4_117234_lines_jspff	85910	57344	4768
webmin_1_570_220564_lines	476118	411694	13763
WeiPulse_OpenSource	85160	85086	589
wikihow_iphone_app_master	15840	15107	59
WordPress_iOS_develop	156214	145823	616
xmlservice-master	49068	49058	17

Table 28: Query Information (1)

Project_Name	Success_High	Insuccess_High	Success_Medium	Insuccess_Medium	Success_Low	Insuccess_Low	Success_Info	Insuccess_Info
30DaysofSwift	67	0	107	0	129	0	107	0
Aerial	67	0	107	0	129	0	107	0
akka-main	82	0	198	0	232	0	198	0
Alamofire	67	0	107	0	129	0	107	0
Angular2-master	95	0	248	0	319	0	248	0
angular-cosmosdb-master	46	0	81	0	74	0	81	0
Angular-Full-Stack-master	66	0	132	0	97	0	132	0
AngularJS-CafeTownsend-master	46	0	81	0	74	0	81	0
AngularJS-ColorGame-master	46	0	81	0	74	0	81	0
AngularJS-info-cars	50	0	97	0	96	0	97	0
AngularJS-java-server-midi-master	46	0	81	0	74	0	81	0
AspNuke08052934_52934_lines	63	0	105	0	114	0	105	0
asptemplate131487_1487_lines	59	0	110	0	114	0	110	0
ATE754	5	0	2	0	2	0	2	0
backboneboilerplate-gh-pages-JSv2	46	0	81	0	74	0	81	0
BBS400-main	3	0	3	0	7	0	3	0
Benchmark-master	70	0	160	0	208	0	160	0
Benchmark-master_JS	62	0	134	0	134	0	134	0
BitTorrent-5.0.9_ReWrite	61	0	112	0	97	0	112	0
blueblog	70	0	160	0	208	0	160	0
BookStoreJava_21403_lines	24	0	79	0	134	0	79	0
BookStoreVBDotnet_22163_lines	13	0	29	0	40	0	29	0
botpress-botpress-v12.19.2-35	53	0	105	0	108	0	105	0
breakableflask-master	15	0	31	0	23	0	31	0
CAF-master	3	0	3	0	7	0	3	0
catiline-gh-pages	46	0	81	0	74	0	81	0

Table 29: Query Information (2)

Project_Name	Success_High	Insuccess_High	Success_Medium	Insuccess_Medium	Success_Low	Insuccess_Low	Success_Info	Insuccess_Info
celestia_1.6.0_198563lines	75	0	153	0	123	0	153	0
Chocobozzz-PeerTube-v3.1.0-47	46	0	81	0	74	0	81	0
ci-brakeman	13	0	32	0	21	0	32	0
climbers_master	16	0	15	0	34	0	15	0
CSharp_Rewrite_Cosmos_181269lines	62	0	134	0	134	0	134	0
CSharp_Rewrite_log4net_72646_lines	62	0	134	0	134	0	134	0
CSharp_Rewrite_northwindtraders_aspnetcore	62	0	134	0	134	0	134	0
CSharp_Rewrite_Rainbow_209794_lines	67	0	144	0	142	0	144	0
CxPlayGround-Mobile-master	114	0	247	0	312	0	247	0
CxPlayGround-Web	81	0	206	0	292	0	206	0
dart-pad-master	5	0	23	0	23	0	23	0
dashboard-master	66	0	132	0	97	0	132	0
deplate_30457_lines	53	0	105	0	108	0	105	0
df17-ant-to-sfdx-master	48	0	93	0	93	0	93	0
Dollar_Bets_master	16	0	15	0	34	0	15	0
drone-master_458434_lines_goV2	64	0	123	0	103	0	123	0
eclipse-theia-theia-v1.11.0-55	46	0	81	0	74	0	81	0
emule4_172454_lines	73	0	149	0	120	0	149	0
evans_goV2	13	0	32	0	21	0	32	0
finch-master	14	0	42	0	27	0	42	0
fintrospect-master	14	0	42	0	27	0	42	0
firefox-ios (Project 1)	69	0	111	0	132	0	111	0
flame-main	5	0	23	0	23	0	23	0
FlightGear_1.0.0_230815lines	75	0	153	0	123	0	153	0
flutter-master	5	0	23	0	23	0	23	0
flutter-quill-master	5	0	23	0	23	0	23	0

Table 30: Query Information (3)

Project_Name	Success_High	Insuccess_High	Success_Medium	Insuccess_Medium	Success_Low	Insuccess_Low	Success_Info	Insuccess_Info
FusionChartsFree28681_28681_lines-JSv2	58	0	115	0	116	0	115	0
gatsby	53	0	105	0	108	0	105	0
GitterMobile-master	60	0	92	0	105	0	92	0
gizmo-master_11638_lines_goV2	57	0	109	0	92	0	109	0
gogs_master_goV2	64	0	123	0	103	0	123	0
groove-dl-master_ReWrite	61	0	112	0	97	0	112	0
groovywebconsole-master	57	0	127	0	158	0	127	0
habitica-develop	62	0	134	0	134	0	134	0
hermes-master	59	0	108	0	94	0	108	0
hooligram-client-develop	60	0	92	0	105	0	92	0
hugo-master_50147_lines_goV2	59	0	113	0	95	0	113	0
iFixit_iOS_master	16	0	15	0	34	0	15	0
Inuendo-master	3	0	3	0	7	0	3	0
jade_agents-master	70	0	160	0	208	0	160	0
jade-master-JSv2	46	0	81	0	74	0	81	0
jasperreports_4_0_0_proje_350576_lines	70	0	160	0	208	0	160	0
Java11and12NewFeatures	70	0	160	0	208	0	160	0
jboard_jspff	75	0	170	0	216	0	170	0
jetspeed_1_6_206226_lines_jspff	75	0	170	0	216	0	170	0
juice-shop-4.2.1	66	0	132	0	97	0	132	0
knockout_js_samples_master-JSv2	46	0	81	0	74	0	81	0
leaky-angular-master	46	0	81	0	74	0	81	0
LightningWorkingApp_ReWrite (Project 3)	48	0	93	0	93	0	93	0
lila-master	84	0	202	0	235	0	202	0
lua_samples-master	15	0	40	0	52	0	40	0
luaSample-master	15	0	40	0	52	0	40	0

Table 31: Query Information (4)

Project_Name	Success_High	Insuccess_High	Success_Medium	Insuccess_Medium	Success_Low	Insuccess_Low	Success_Info	Insuccess_Info
lwc_goat	50	0	97	0	96	0	97	0
mingw32_make	29	0	72	0	49	0	72	0
MISRA_C_celestia	0	0	0	0	0	0	0	0
MISRA_C_FlightGear	0	0	0	0	0	0	0	0
MISRA_C_mingw32	0	0	0	0	0	0	0	0
MISRA_C_UltimateToolbox93	0	0	0	0	0	0	0	0
mleung_feather	53	0	105	0	108	0	105	0
MSDN	64	0	120	0	122	0	120	0
MTOS_4_38_en_390199_lines	56	0	97	0	87	0	97	0
MySQLdb1-master_ReWrite	61	0	112	0	97	0	112	0
NetNewsWire	69	0	111	0	132	0	111	0
nodejs-mysql-native-master-JSv2 (Project 4)	49	0	87	0	79	0	87	0
node-mongodb-native-1.4-JSv2_ReWrite	61	0	112	0	97	0	112	0
OSSILE-master	3	0	3	0	7	0	3	0
owasp-top10-salesforce-master	48	0	93	0	93	0	93	0
pebble_jspff	70	0	160	0	208	0	160	0
Pebble_Spring_Example	24	0	79	0	134	0	79	0
perl_5_16_0_898436_lines	56	0	97	0	87	0	97	0
personalblog_jspff	68	0	156	0	205	0	156	0
PHP_Rewrite_AlegroCart_1.2.5_125254_lines-JSv2	71	0	142	0	105	0	142	0
PHP_Rewrite_Sylius_1.2	66	0	132	0	97	0	132	0
PHP_Rewrite_symfony-master(Project2)	64	0	128	0	94	0	128	0
play-groovy-master	81	0	206	0	292	0	206	0
pokerBuddyApp-master	51	0	101	0	105	0	101	0
polynote-master	84	0	202	0	235	0	202	0
Probabilistic-Programming	61	0	112	0	97	0	112	0
Programming_In_Lua-master	15	0	40	0	52	0	40	0

Table 32: Query Information (5)

Project_Name	Success_High	Insuccess_High	Success_Medium	Insuccess_Medium	Success_Low	Insuccess_Low	Success_Info	Insuccess_Info
PSLQL_OSCOMMAND	5	0	10	0	8	0	10	0
Public-Corona-Lua-master(project 4)	15	0	40	0	52	0	40	0
quasar-dev	53	0	105	0	108	0	105	0
qxmpp_0_2_0_20478_lines	75	0	153	0	123	0	153	0
railsgoat-master	53	0	105	0	108	0	105	0
react-datasheet-master	51	0	101	0	105	0	101	0
react-native-elements-next	51	0	101	0	105	0	101	0
react-security	71	0	142	0	105	0	142	0
react-swipeable-views-master	53	0	105	0	108	0	105	0
rico2143347_43347_lines	64	0	120	0	122	0	120	0
riverpod-master	5	0	23	0	23	0	23	0
rmux	57	0	109	0	92	0	109	0
roller_jspff	75	0	170	0	216	0	170	0
RxSwift	67	0	107	0	129	0	107	0
Scala_ReWrite_apache-samza-0.10.0-src	84	0	202	0	235	0	202	0
Scala_ReWrite_atlas-master	84	0	202	0	235	0	202	0
Scala_ReWrite_casbah-master-JSv2	60	0	123	0	101	0	123	0
Scala_ReWrite_finagle-develop	89	0	212	0	243	0	212	0
Scala_ReWrite_miniboxing-plugin-wip	38	0	121	0	161	0	121	0
shadowsocks-android-master	21	0	34	0	50	0	34	0
singularity_master_goV2	57	0	109	0	92	0	109	0
snipsnap	70	0	160	0	208	0	160	0
soundcloud-redux-master	44	0	77	0	71	0	77	0
Spring_Rewrite_AwesomeNotes	88	0	198	0	234	0	198	0
Spring_Rewrite_compass	70	0	160	0	208	0	160	0
Spring_Rewrite_SpringBootGoat	75	0	170	0	216	0	170	0
Swift_Rewrite_Cards-master	67	0	107	0	129	0	107	0
Swift_Rewrite_CVCalendar-master	67	0	107	0	129	0	107	0

Table 33: Query Information (6)

Project_Name	Success_High	Insuccess_High	Success_Medium	Insuccess_Medium	Success_Low	Insuccess_Low	Success_Info	Insuccess_Info
Swift_Rewrite_ImagePickerTrayController	67	0	107	0	129	0	107	0
Swift_Rewrite_Maria-master	67	0	107	0	129	0	107	0
Swift_Rewrite_SwiftGoat-master	69	0	111	0	132	0	111	0
Swift_Rewrite_Swift	23	0	30	0	58	0	30	0
Swift_Rewrite_SwiftLanguageWeather	67	0	107	0	129	0	107	0
Swift_Rewrite_Swift-Master_Of_Swift	69	0	111	0	132	0	111	0
Swift_Rewrite_SwiftMessages-master	67	0	107	0	129	0	107	0
Swift_Rewrite_Xniffer-master	67	0	107	0	129	0	107	0
tables-3.0.0_ReWrite	61	0	112	0	97	0	112	0
testcodav3_79566_lines_ReWrite	4	0	16	0	22	0	16	0
testVbNet	59	0	110	0	114	0	110	0
tiny-dnn-master_cpp11	75	0	153	0	123	0	153	0
trape-master	61	0	112	0	97	0	112	0
Twidere-Android-master-Kotlin	21	0	34	0	50	0	34	0
twitter-finagle	89	0	212	0	243	0	212	0
UltimateToolbox93_src_322262_lines	75	0	153	0	123	0	153	0
Verademo-Dotnet	62	0	134	0	134	0	134	0
Vertical-Fill-master	15	0	40	0	52	0	40	0
vuetify-master	53	0	105	0	108	0	105	0
vulnerable-app-master	46	0	81	0	74	0	81	0
vux-2	53	0	105	0	108	0	105	0
WebGoat_5.4_117234_lines_jspff	75	0	170	0	216	0	170	0
webmin_1_570_220564_lines	56	0	97	0	87	0	97	0
WeiPulse_OpenSource	26	0	27	0	39	0	27	0
wikihow_iphone_app_master	26	0	27	0	39	0	27	0
WordPress_iOS_develop	62	0	96	0	108	0	96	0
xmlservice-master	3	0	3	0	7	0	3	0

Table 34: Coverability, Domability, Querability and Scan Quality Information (1)

Project_Name	Coverability	Domability	Querability	Scan Quality
30DaysofSwift	55.56	98.22	100	81.512
Aerial	62.06	100	100	84.824
akka-main	100	96.52	100	98.608
Alamofire	98.03	97.21	99.46	97.988
Angular2-master	95.9	81.49	100	90.956
angular-cosmosdb-master	4.23	100	100	61.692
Angular-Full-Stack-master	11	99.87	100	64.348
AngularJS-CafeTownsend-master	85.82	100	100	94.328
AngularJS-ColorGame-master	99.87	100	100	99.948
AngularJS-info-cars	60.12	100	100	84.048
AngularJS-java-server-midi-master	99.54	100	100	99.816
AspNuke08052934_52934_lines	95.76	59.48	100	82.096
asptemplate131487_1487_lines	49.49	100	100	79.796
ATE754	100	86.63	100	94.652
backboneboilerplate-gh-pages-JSv2	99.78	98.93	100	99.484
BBS400-main	100	98.4	100	99.36
Benchmark-master	99.46	92.42	100	96.752
Benchmark-master_JS	97.02	100	100	98.808
BitTorrent-5.0.9_ReWrite	100	99.12	100	99.648
blueblog	95.78	100	100	98.312
BookStoreJava_21403_lines	100	100	100	100
BookStoreVBDotnet_22163_lines	83.37	98.93	100	92.92
botpress-botpress-v12.19.2-35	99.33	71.96	100	88.516
breakableflask-master	100	100	100	100
CAF-master	100	56.57	100	82.628
catiline-gh-pages	99.56	98.93	100	99.396

Table 35: Coverability, Domability, Querability and Scan Quality Information (2)

Project_Name	Coverability	Domability	Querability	Scan Quality
celestia_1.6.0_198563lines	95.33	45.48	100	76.324
Chocobozzz-PeerTube-v3.1.0-47	99.52	56.27	100	82.316
ci-brakeman	99.93	100	100	99.972
climbers_master	99.56	74.65	100	89.684
CSharp_Rewrite_Cosmos_181269lines	98.8	98.93	100	99.092
CSharp_Rewrite_log4net_72646_lines	38.58	81.11	100	67.876
CSharp_Rewrite_northwindtraders_aspnetcore	37.45	100	100	74.98
CSharp_Rewrite_Rainbow_209794_lines	77.87	100	100	91.148
CxPlayGround-Mobile-master	64.35	89.22	100	81.428
CxPlayGround-Web	93.11	98.4	100	96.604
dart-pad-master	100	100	100	100
dashboard-master	57.92	99.82	100	83.096
deplate_30457_lines	98.99	98.4	100	98.956
df17-ant-to-sfdx-master	99.59	96.85	100	98.576
Dollar_Bets_master	21.15	99.46	100	68.244
drone-master_458434_lines_goV2	95.96	97.09	97.92	96.804
eclipse-theia-theia-v1.11.0-55	99.93	91.49	100	96.568
emule4_172454_lines	99.09	59.83	100	83.568
evans_goV2	100	100	100	100
finch-master	100	99.64	100	99.856
fintrospect-master	100	100	100	100
firefox-ios (Project 1)	82.85	20.76	100	61.444
flame-main	100	100	100	100
FlightGear_1.0.0_230815lines	98.16	99.46	100	99.048
flutter-master	100	80.38	98.42	91.836
flutter-quill-master	100	100	100	100

Table 36: Coverability, Domability, Querability and Scan Quality Information (3)

Project_Name	Coverability	Domability	Querability	Scan Quality
FusionChartsFree28681_28681_lines-JSv2	45.44	100	100	78.176
gatsby	89.47	50.95	100	76.168
GitterMobile-master	92.78	74.03	100	86.724
gizmo-master_11638_lines_goV2	99.23	100	100	99.692
gogs_master_goV2	91.22	55.28	100	78.6
groove-dl-master_ReWrite	99.77	100	100	99.908
groovywebconsole-master	99.98	100	100	99.992
habitica-develop	43.75	96.34	100	76.036
hermes-master	99.99	100	100	99.996
hooligram-client-develop	38.3	100	100	75.32
hugo-master_50147_lines_goV2	98.12	100	100	99.248
iFixit_iOS_master	59.2	90.12	100	79.728
Inuendo-master	100	100	100	100
jade_agents-master	99.71	100	100	99.884
jade-master-JSv2	55.16	58.29	100	65.38
jasperreports_4_0_0_proje_350576_lines	97.25	99.46	100	98.684
Java11and12NewFeatures	99.56	95.65	100	98.084
jboard_jspff	84.55	98.93	100	93.392
jetspeed_1_6_206226_lines_jspff	92.13	96.34	100	95.388
juice-shop-4.2.1	80.77	100	100	92.308
knockout_js_samples_master-JSv2	95.97	100	100	98.388
leaky-angular-master	12.12	100	100	64.848
LightningWorkingApp_ReWrite (Project 3)	17.06	78.6	100	58.264
lila-master	98.59	99.12	100	99.084
lua_samples-master	100	99.46	100	99.784
luaSample-master	100	100	100	100

Table 37: Coverability, Domability, Querability and Scan Quality Information (4)

Project_Name	Coverability	Domability	Querability	Scan Quality
lwc_goat	10.28	100	100	64.112
mingw32_make	97.94	89.22	100	94.864
MISRA_C_celestia	95.33	45.48	100	76.324
MISRA_C_FlightGear	98.16	99.46	100	99.048
MISRA_C_mingw32	97.94	89.22	100	94.864
MISRA_C_UltimateToolbox93	99.9	59.66	100	83.824
mleung_feather	91.91	92.63	100	93.816
MSDN	95.17	98.8	100	97.588
MTOS_4_38_en_390199_lines	89.72	49.57	100	75.716
MySQLdb1-master_ReWrite	58.07	100	100	83.228
NetNewsWire	85.98	80.14	99.46	86.34
nodejs-mysql-native-master-JSv2 (Project 4)	100	100	100	100
node-mongodb-native-1.4-JSv2_ReWrite	99.2	98.93	100	99.252
OSSILE-master	99.91	100	100	99.964
owasp-top10-salesforce-master	87.15	100	100	94.86
pebble_jspff	93.18	96.85	100	96.012
Pebble_Spring_Example	100	100	100	100
perl_5_16_0_898436_lines	62.74	40	100	61.096
personalblog_jspff	54.52	100	100	81.808
PHP_Rewrite_AlegroCart_1.2.5_125254_lines-JSv2	93.1	50.02	100	77.248
PHP_Rewrite_Sylius_1.2	93.36	79.36	100	89.088
PHP_Rewrite_symfony-master(Project2)	47.74	54.96	100	61.08
play-groovy-master	88.83	100	100	95.532
pokerBuddyApp-master	46.79	100	100	78.716
polynote-master	39.84	98.44	100	75.312
Probabilistic-Programming	0.96	100	100	60.384
Programming_In_Lua-master	100	100	100	100

Table 38: Coverability, Domability, Querability and Scan Quality Information (5)

Project_Name	Coverability	Domability	Querability	Scan Quality
PSLQL_OSCOMMAND	41.79	100	100	76.716
Public-Corona-Lua-master(project 4)	100	100	100	100
quasar-dev	85.3	81.11	100	86.564
qxmpp_0_2_0_20478_lines	98.27	91.95	100	96.088
railsgoat-master	97.25	74.65	100	88.76
react-datasheet-master	48.58	100	100	79.432
react-native-elements-next	97.76	98.93	100	98.676
react-security	5.94	100	100	62.376
react-swipeable-views-master	98.07	98.93	100	98.8
rico2143347_43347_lines	85.84	71.39	100	82.892
riverpod-master	100	100	100	100
rmux	99.76	100	100	99.904
roller_jspff	92.77	92.9	100	94.268
RxSwift	98.66	38.81	92.09	73.406
Scala_ReWrite_apache-samza-0.10.0-src	94.64	99.46	100	97.64
Scala_ReWrite_atlas-master	99.98	100	100	99.992
Scala_ReWrite_casbah-master-JSv2	99.71	99.46	100	99.668
Scala_ReWrite_finagle-develop	99.12	96.41	100	98.212
Scala_ReWrite_miniboxing-plugin-wip	100	99.64	100	99.856
shadowsocks-android-master	85.31	100	100	94.124
singularity_master_goV2	93.93	99.46	100	97.356
snipsnap	18.2	95.83	100	65.612
soundcloud-redux-master	36.97	100	100	74.788
Spring_Rewrite_AwesomeNotes	71.92	100	100	88.768
Spring_Rewrite_compass	98.96	100	100	99.584
Spring_Rewrite_SpringBootGoat	96.39	99.46	100	98.34
Swift_Rewrite_Cards-master	87.92	100	100	95.168
Swift_Rewrite_CVCalendar-master	96.28	95.48	100	96.704
Swift_Rewrite_ImagePickerTrayController	76.84	100	100	90.736
Swift_Rewrite_Maria-master	72.19	100	100	88.876

Table 39: Coverability, Domability, Querability and Scan Quality Information (6)

Project_Name	Coverability	Domability	Querability	Scan Quality
Swift_Rewrite_SwiftGoat-master	49.1	70.93	100	68.012
Swift_Rewrite_Swift	98.25	99.82	100	99.228
Swift_Rewrite_SwiftLanguageWeather	83.56	100	100	93.424
Swift_Rewrite_Swift-Master_Of_Swift	94.11	70	100	85.644
Swift_Rewrite_SwiftMessages-master	76.14	99.46	100	90.24
Swift_Rewrite_Xniffer-master	83.97	100	100	93.588
tables-3.0.0_ReWrite	61.25	97.88	100	83.652
testcodav3_79566_lines_ReWrite	100	100	100	100
testVbNet	99.59	85.56	100	94.06
tiny-dnn-master_cpp11	92.14	87.93	100	92.028
trape-master	99.37	100	100	99.748
Twidere-Android-master-Kotlin	99.85	97.21	98.94	98.612
twitter-finagle	99.53	99.29	100	99.528
UltimateToolbox93_src_322262_lines	99.9	59.66	100	83.824
Verademo-Dotnet	99.81	100	100	99.924
Vertical-Fill-master	100	100	100	100
vuetify-master	74.86	79.13	100	81.596
vulnerable-app-master	93.79	100	100	97.516
vux-2	47.72	71.67	100	67.756
WebGoat_5.4_117234_lines_jspff	66.75	93.38	100	84.052
webmin_1_570_220564_lines	86.47	40.46	100	70.772
WeiPulse_OpenSource	99.91	87.91	100	95.128
wikihow_iphone_app_master	95.37	100	100	98.148
WordPress_iOS_develop	93.35	40.22	100	73.428
xmlservice-master	99.98	96.34	100	98.528