RUHR-UNIVERSITÄT BOCHUM

RUB

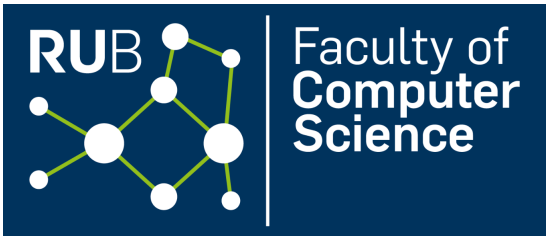
Mining of Security Features in Software Systems: How Are They Implemented in Practice?

Yasser Hekal

RUHR-UNIVERSITÄT BOCHUM

Master Thesis – March 31, 2025
Chair of Software Engineering.

Supervisor: Prof. Dr. Thorsten Berger
Advisor: M.Sc. Kevin Hermann



Abstract

Ensuring the security of software systems is becoming increasingly difficult as modern applications grow in complexity and scale. A central challenge lies in identifying and tracing the security features embedded within a codebase—features that are essential for safeguarding sensitive information and maintaining system integrity. This fragmentation poses a significant challenge for maintaining security, tracing vulnerabilities, and applying patches efficiently. To address this, developers need reliable strategies to identify, understand, and evaluate the security features embedded in real-world systems.

This thesis addresses this open challenge by proposing a structured methodology for identifying, locating, and analyzing security features in open-source Java projects. Through an iterative and tool-assisted process, this work utilizes a refined keyword-based search, and a custom-developed detection tool for library-based features. A mining study of six real-world, security-relevant repositories from domains such as healthcare, finance, and infrastructure—areas where security is particularly critical, reveals patterns of security feature distribution, modular reuse, and deviations from best practices. The study further categorizes features under a security taxonomy and evaluates their implementation against known security guidelines.

The results of this work demonstrate that security features are rarely centralized, often span multiple architectural layers, and tend to combine library-based mechanisms with custom logic. While established frameworks provide strong foundations, gaps remain in areas such as traceability, structural consistency, and alignment with best practices. These insights emphasize the importance of modular design, improved annotation and detection tooling, and greater developer support for secure implementation. This work contributes actionable guidance and empirical findings for future research and development of secure, maintainable software systems.

Contents

List of Acronyms

API	Application Programming Interface
CSRF	Cross-Site Request Forgery
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
ET	Exploring Traces
FPGA	Field Programmable Gate Array
HAnS	Help Annotating Software
HCS	Highly Configurable Software
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
LDA	Linear Discriminant Analysis
LSI	Latent Semantic Indexing
MFA	Multi-Factor Authentication
MuTT	MultiThreaded Tracer
NVD	National Vulnerability Database
OTP	One-time password
OWASP	Open Worldwide Application Security Project
SCA	Static Code Analysis
SDLC	Software Development Lifecycle
SEFF	SEcurity Feature Framework
TOTP	Time-based one-time password
URL	Uniform Resource Locator

1 Introduction

The security of software systems is a critical aspect of their overall integrity and reliability. Within a software system, various features are implemented, some of which are specifically designed to achieve security objectives [1]. Security features refer to specific functionalities or mechanisms integrated into software systems to protect against vulnerabilities, unauthorized access, and data breaches [2]. Software features in general may be implemented across various layers of the software, including application logic, communication protocols, and data handling processes. Features may sometimes be implemented in a scattered manner across the codebase, with various components and modules containing security measures [3, 4]. This nature is due to the incremental and collaborative development processes typical of software systems, where different contributors may implement security features independently without a centralized or uniform approach [5].

The complexity of locating and identifying security features within a software system raises significant concerns about the overall security posture [6]. If some of these security features are scattered across various components, as is sometimes the case with other functional features [3, 4], it could lead to inconsistent application of security measures. This inconsistency complicates the management and maintenance of security features, potentially resulting in vulnerabilities that could compromise the entire system [7]. We believe that if that scattered nature of security features is significant and detectable, it may have adverse side effects on the overall security integrity of the software system. Therefore, it is crucial to explore and uncover the distribution of these features to ensure they are consistently applied, function cohesively to provide a robust defense mechanism, and that their propagation throughout the entire software system can be traced in the event of discovered vulnerabilities. The aim of this thesis is to discover the implementation and nature of security features in practice and to investigate whether they exhibit similar scattering patterns as general features and to assess the security implications of such distribution on the system.

The increasing complexity of software systems has made the task of ensuring robust security more challenging than ever. A key aspect of maintaining a secure software environment is the ability to accurately locate and analyze security features embedded within the system [6]. These features, which may either be sourced from established security frameworks or custom-implemented by developers, play a critical

role in the system's overall security posture. Security must be considered at every stage of software development [8]. However, the wide variety of security threats and the corresponding countermeasures make it challenging for developers to select and implement appropriate security features. While security standards provide guidance, they are often too abstract to be directly applicable to specific security implementations or too focused on low-level details to be practical for developers [8].

1.1 Motivation

The motivation for this thesis is rooted in the significance and criticality of security features and the sensitive information they protect within software systems, which urge the need to locate and analyze security features within software systems. Understanding whether a feature is derived from a security framework or custom-implemented by developers is crucial for assessing the system's security posture. Moreover, identifying how these features are distributed within the code and how they interact with other components can reveal potential weaknesses and guide improvements in security practices. This research aims to provide a structured approach to addressing these challenges, ultimately contributing to the development of more secure software systems.

A critical aspect of this investigation is not only understanding the origin and distribution of security features but also evaluating their effectiveness in the context of the entire software system. The challenge of translating security standards into practical, effective implementations drives this thesis. Developers must not only select appropriate security features but also ensure their seamless integration into the software architecture. Additionally, it is essential to assess whether these features work cohesively to achieve the intended security goals. This work aims to explore how security features, whether derived from frameworks or custom-built, are applied in practice, and how their application can be optimized to more effectively address security threats.

1.2 Research Questions

The challenge is to locate and identify security features , as well as discovering their implementation in practice and their nature in the complex landscapes of software systems. In order to address this challenge, several research questions have been formulated, the answers to which will form the foundation of this work.

RQ: *How are security features of software systems implemented in practice?*

To investigate this question, a structured methodology is employed to locate, identify, and analyze security features within software systems. The investigation involves exploring their implementation, distribution, and interactions across the software architecture. The methodology ensures a robust evaluation of both custom-implemented features and those derived from frameworks, with the ultimate goal of enhancing security practices in software systems. This overarching question is further divided into five sub-questions..

RQ1: *How can security features be identified and located within the code of software systems?*

The process begins with the use of three approaches: `SonarQube`[1], `Semgrep`[2] and a manual script using security-related keyword lists[3] to systematically identify security features across the codebase and then eventually annotating these features with Help Annotating Software (HAnS) annotations. This structured approach ensures comprehensive tagging of features, enabling precise localization and facilitating deeper insights into their interactions and overall contribution to security objectives.

RQ2: *What security features exist in a software system?*

Features in software systems, including security-specific functionalities like encryption or access control, are identified using a combination of keyword searches and pattern recognition. These techniques leverage well-documented libraries and frameworks to classify features effectively, laying the groundwork for understanding their role and significance in the system's architecture. .

RQ3: *What security features are custom implemented by developers, and What are integrated from security frameworks and libraries?*

Beside categorizing the security features, those who are custom implemented by developers and those who are imported from security libraries and frameworks in the security feature locating and Identification phase. This comparative analysis is conducted to distinguish between custom-implemented security features and those derived from established frameworks such as Spring Security[4] or OWASP ESAPI[5]. This step evaluates the integration and alignment of these features with best practices, assessing their adequacy in addressing security challenges.

[1] `https://www.sonarsource.com/`
[2] `https://github.com/semgrep/semgrep`
[3] `https://github.com/danielmiessler/SecLists`
[4] `https://spring.io/projects/spring-security`
[5] `https://owasp.org/www-project-enterprise-security-api/`

RQ4: *How are security features distributed across the software architecture, and what patterns can be observed in their distribution?*

This question examines the scattering and distribution of security features within the codebase. Static analysis tools and HAnS annotations provide a detailed view of how these features are dispersed, offering insights into architectural patterns and their impact on the software system's overall integrity and security

RQ5: *How secure are security features implemented, and do they follow the best practices and security standards?*

To assess the effectiveness and security robustness of identified security features, an implementation-level analysis will be conducted on a selected subset of features. The selection focused on those that address critical security goals, such as authentication, access control, and secure data handling, as they are among the most impactful and security-sensitive areas. Each chosen feature was evaluated semantically to determine whether it fulfills its intended security purpose and whether it follows recognized best practices and industry standards, such as those outlined by OWASP. This evaluation aims to answer RQ5 and provide insights into how securely these features are implemented in real-world software systems.

1.3 Contributions

In this thesis, the following key contributions are made to the field of software engineering security, particularly within the domain of security features in software systems:

Exploration of Security Feature Nature: This research identifies and explores the nature of security features within software systems, shedding light on their characteristics, distribution, and interaction with other components. This foundational understanding addresses critical gaps in the literature on feature-specific security practices.

Enhanced Traceability: The study enhances the traceability of security features across various software layers, providing a clearer understanding of how these features are distributed and integrated. By leveraging structured methods and existent tools such as `Sonarqube`, `Semgrep`, HAnS annotations and customized scripts, this contribution facilitates better navigation and analysis of complex software architectures.

Systematic Methodology Development: A systematic methodology is developed to locate security features effectively, utilizing HAnS annotations alongside

supplementary analysis tools. This methodology introduces an innovative approach to identifying security-related elements, significantly improving precision and efficiency and including contextual considerations in security feature analysis.

Impact Analysis: The research provides a comprehensive analysis of the impact of the complex nature of security features on the overall security of software systems. It examines how these features influence system integrity, identifying potential vulnerabilities and offering insights to mitigate them.

Metrics and Insights for Modularity: The thesis introduces metrics and findings that evaluate the modularity and integration of security features within the broader landscape of software functionalities. These metrics support the assessment of feature cohesion and the refinement of security measures.

Guidelines for Best Practices: Practical guidelines are presented to ensure consistent implementation of security features, aligned with best practices and established security standards. These recommendations assist developers in maintaining uniformity and reliability across the codebase. This would provides actionable guidelines for effectively integrating security measures into existing security features, enabling smoother adoption and implementation of robust defenses in real-world software systems.

1.4 Organization of this Thesis

This thesis is structured to provide a comprehensive and systematic overview of the study. **Chapter 1** introduces the research problem, motivation, and objectives. **Chapter 2** presents the necessary background on security features and traceability. **Chapter 3** details the methodology, outlining the overall mining workflow and each phase of the process. **Chapter 4** describes the implementation of the developed tools and techniques used to locate, annotate, and categorize security features. **Chapter 5** presents the study results, including the evaluation of each locating method, feature categorization, distribution analysis, and comparisons with best practices. **Chapter 6** discusses the broader implications, limitations, and validity threats of the study, and outlines potential directions for future work. Finally, **Chapter 7** concludes the thesis with a summary of the main findings.

2 Background

This chapter provides a glimpse of related work and an overview of security features, highlighting their critical role in software systems. It discusses the challenges developers face in implementing and analyzing security features and outlines the tools and techniques used to enhance their traceability. Finally, the chapter introduces Helping Annotate Software (HAnS) as a key tool employed in this thesis for annotating and managing security features.

2.1 Overview of Security Features

Features were defined in many ways in the literature, for Berger et al. a feature is an abstract representation of functionality that characterizes the capabilities or behaviors of software systems [9]. More precisely, a feature can be defined as "a logical unit of behavior specified by a set of functional and non-functional requirements"[10]. Features may also represent characteristics distinguishing a system from others in a family of related systems [11]. Alternatively, features can be described as user-visible aspects of a system [12, 13] or as aspects that provide added value to a customer [14].

Hermann et al. [6] developed a taxonomy of 68 functional security features through a systematic literature review (see Fig. 2.1), aiming to bridge the gap between high-level security standards and practical implementation. Their objective was to assist developers in selecting and applying security measures more effectively by mapping these features to recognized security standards such as `ISO/IEC 27000`, Common Criteria, `NIST SP800-53`, and the `NIST Cybersecurity Framework`. Additionally, they examined the implementation of these security features across 21 popular security frameworks, helping developers align their security practices with established standards. In this thesis, I will utilize the taxonomy developed by Hermann et al. [6] in the **Categorizing Security Features** process. This will help organize and classify the security features identified within the software systems, enabling a structured analysis of their implementation and distribution.

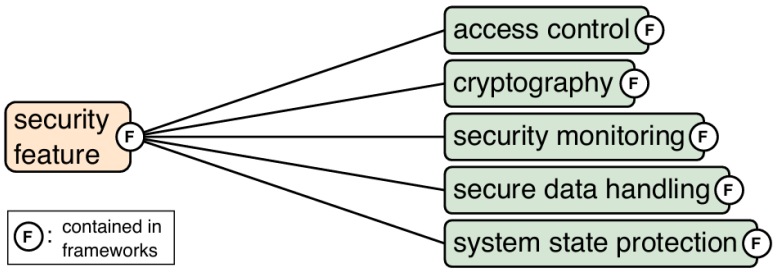


Figure 2.1: Top-level of Security Features Taxonomy [6]

In the context of this thesis, we specifically consider security features, which are functionalities designed to address security issues by preventing attacks or fulfilling security requirements [1]. These features must be carefully planned, even at the architectural level, as missing or improperly implemented security features can result in severe vulnerabilities and huge impact in software systems [15, 16]. Security features can realize both functional and non-functional requirements. However, this thesis focuses on functional security features, which are concrete security measures manifesting in the codebase to address specific functional requirements of a software system[6]. Examples include encryption mechanisms, authentication protocols, and access control features. These features are critical for maintaining the integrity, confidentiality, and availability of a system, underscoring their essential role in secure software development.

Blythe et al. (2019) examined the security features of consumer Internet of Things (IoT) devices by analyzing user manuals and support pages to assess how security is communicated to consumers [17]. Their study highlighted that IoT devices often lack built-in security measures, leaving users vulnerable to cyber threats such as hacking and large-scale cybercrime attacks. Furthermore, even when security features are available, consumers do not always utilize them effectively. A key finding was the lack of transparency from manufacturers, who provide minimal publicly available information about device security, making market surveillance difficult. The authors advocate for government intervention to establish clearer security guidelines and ensure that security information is readily accessible to consumers.

Druyer et al. (2015) explored security features in modern Field Programmable Gate Array (FPGA)s, assessing how vendors like Altera, Microsemi, and Xilinx implement security mechanisms [18]. The study categorized security threats, including IP protection, data confidentiality, and denial-of-service risks, and compared vendor-specific security measures. Their findings highlighted the strengths and weaknesses of current FPGA security implementations, focusing on hardware-level security rather than software security frameworks.

Lingham et al. (2019) examined the integration of security features in software development, emphasizing the importance of incorporating security at each phase of the Software Development Lifecycle (SDLC) [19]. The study discussed common vulnerabilities arising from neglecting security early in development and proposed methodologies to embed security practices, including automated security testing and developer training.

While previous studies have explored security features in IoT devices, FPGAs, and software development processes, my work differs in its focus on identifying, annotating, and tracking security features within software repositories. Unlike Blythe et al. (2019) [17], which examines security features utilization by consumers in IoT devices, my study investigates their actual presence and usage in software codebases. Compared to Druyer et al. (2015) [18], which focuses on hardware security in FPGA, my research targets software security features traceability and implementation. Furthermore, while Lingham et al. (2019) [19] emphasizes security best practices in SDLC, my work provides a systematic approach to locating and analyzing security features within existing software systems and their implementations best practices.

2.2 Locating and Identifying Security Features

Locating and identifying security features in software systems has long been a challenging and cumbersome task. Security features are often scattered across various components of a software system, making their identification labor-intensive and error-prone[3, 4]. Traditional approaches to feature location rely on manual efforts, where developers must trace security-related code fragments across the entire codebase. However, the lack of standardized documentation and the evolving nature of software systems exacerbate these difficulties [6].

Automated feature location techniques have been proposed to address these issues, including static and dynamic analysis methods. Static analysis techniques leverage lexical and syntactic analysis to identify feature-related code elements, while dynamic approaches rely on execution traces and user interactions to infer feature locations [20]. Despite their potential, these techniques often produce high false-positive rates, require large training datasets, and struggle with fine-grained feature identification [20], which is a fact that we will encounter and verify it in this work (more in section 5.1.1).

2.3 Challenges in Security Features Implementation, Maintenance and Analysis

Software engineers and programmers, while skilled in software development, are not inherently security experts [21]. The specialized knowledge required to implement robust security features is often beyond their typical training, leaving gaps in security practices [21]. This challenge is compounded by the usability issues associated with security libraries and frameworks [22]. Research has shown that these tools, while powerful, can be unintuitive and overly complex, leading to improper or incomplete integrations [22]. The result is a higher likelihood of vulnerabilities introduced during development, stemming from both human error and the lack of accessible, developer-friendly tools [1].

Khwaja et al. [2] proposed a SEcurity Feature Framework (SEFF) aimed at enhancing the security of software systems by integrating security features directly into programming languages. While their work focuses on security feature coverage within programming languages and identifies gaps in this area, my thesis differs by concentrating on the identification and analysis of security features within software systems as a whole, rather than focusing on the programming languages themselves. This thesis seeks to explore how security features are distributed and implemented across the broader software architecture, contributing to overall system security.

Maintaining and analyzing security features pose additional hurdles, particularly due to the complex nature of these features within software systems. Security features are often scattered across various components, making their identification and tracking laborious and error-prone [5, 20]. This scattering, combined with insufficient traceability mechanisms, complicates the task of locating, updating, or analyzing security features.

A common approach to security analysis in software systems involves the use of static and dynamic analysis tools, such as `SonarQube`[1], `Semgrep`[2]. These tools scan codebases to detect vulnerabilities, misconfigurations, and code quality issues. However, their fundamental limitation is that they do not process codebases as a structured collection of features but rather analyze them as an undifferentiated whole. This lack of feature-level awareness makes them ineffective for tracking, maintaining, or analyzing security features in a structured manner. For instance, while `SonarQube` provides valuable insights into security weaknesses, it does not facilitate the identification or traceability of security features within a system. Similarly, `Semgrep`

[1] `https://www.sonarsource.com/`
[2] `https://github.com/semgrep/semgrep`

enables pattern-based security rule detection, but it does not assist in understanding how security features are implemented or distributed within a codebase.

Dynamic analysis tools, such as `Burp Suite`[3], focus on runtime security testing, identifying vulnerabilities through simulated attacks. While these tools play a crucial role in security validation, they do not address security feature traceability and maintenance within a software system. Without explicit annotations or feature-aware tracking mechanisms, static and dynamic analyzers fail to provide developers with the necessary insights into how security features evolve, interact, and contribute to overall system security.

During security incidents or audits, the inability to quickly locate these features can result in significant delays and resource expenditure [2]. The process of manually tracing features across a codebase is not only time-intensive but also prone to oversight, further elevating the risks associated with vulnerabilities [3].

The challenges in implementation, maintenance, and analysis translate directly into increased costs for organizations [7]. Prolonged vulnerability resolution times, higher maintenance overheads, and potential damage from security breaches underscore the importance of addressing these issues effectively [23]. These challenges necessitate improvements in developers security trainings and awareness, the design of usable security libraries, and the adoption of annotating tools that facilitate effective traceability and management of security features [6].

2.4 Tools and Techniques for Features Traceability

Feature traceability involves mapping features to their corresponding code components or fragments, facilitating tasks such as program comprehension, maintenance, and debugging. Several tools and techniques have been developed to assist in this process. While extensive research has been conducted on feature location in software systems, most studies, such as those by Rubin and Chechik [5] and Krüger et al. [24], focus on general features rather than the specific challenges of locating security features, such as their distribution within the software system, critical importance, and interaction across system layers. These works highlight various techniques for identifying features but do not address these complexities. Additionally, Krüger et al. [25] emphasize traceability for general features, and Berger et al. [9] explore feature identification in large-scale systems, yet neither directly considers the risks

[3] `https://portswigger.net/burp`

posed by misconfigured or poorly implemented security features.

Passos et al. [26] introduce a feature-oriented perspective on software evolution, arguing that managing changes at the feature level can significantly enhance software traceability, maintenance, and evolution. Their approach focuses on automatic traceability, analysis, and recommendations to manage evolving software systems. They propose that features provide a common ground for all stakeholders, allowing for better communication and understanding of changes over time. While their work presents a strong case for feature-aware software evolution, it does not specifically address security features, which require additional considerations such as their different nature, their compliance with security best practices, vulnerability mitigation, and risk assessment.

This thesis aims to fill this gap by focusing specifically on locating and analyzing security features, addressing their technical implementation, distribution, and security risks. Unlike previous research that considers feature traceability in a broad sense, this work explores how security features evolve, interact, and contribute to overall system security, ensuring they can be effectively maintained and adapted as software evolves.

FeatRacer

`FeatRacer` is a tool that combines proactive feature recording and automated feature location in a way that allows developers to proactively and continuously record features and their locations during development to address the challenges of locating features in codebases [20]. It uses embedded code annotations and a machine-learning recommender to help developers continuously record features and reminds them of missed annotations. `FeatRacer` enables fine-grained traceability and significantly outperforms traditional methods like Latent Semantic Indexing (LSI) and Linear Discriminant Analysis (LDA), reducing false positives and improving feature identification. Evaluations across multiple open-source projects highlight its effectiveness in simplifying and enhancing feature traceability. `FeatRacer` records and tracks features during development, making it effective for real-time feature management but not for post-development security feature analysis. My approach identifies, annotates, and classifies security features after development, which is a more complex, error-prone and harder task for developers.

MuTT and ET Tool Suite

MultiThreaded Tracer (MuTT) and Exploring Traces (ET) Tool Suite is a pair of tools designed to trace multi-threaded, event-driven Java programs, aiding in feature location and debugging. MuTT requires no instrumentation to generate traces, while ET allows developers to browse and analyze specific parts of the trace graphs [27]. A case study on Eclipse demonstrated that this suite significantly improves feature location productivity, with ET effectively complementing MuTT in tracing and narrowing focus areas. MuTT and ET focus solely on tracing execution paths but do not identify, classify, or analyze security features. My approach goes beyond tracing by locating, annotating, and categorizing security features, examining their distribution, and architectural impact within the software system.

Test2Feature

`Test2Feature`[4] is a tool designed to address test-to-feature traceability challenges in Highly Configurable Software (Highly Configurable Software (HCS)) [28]. It links test cases to features by analyzing the source code of annotated HCSs, providing outputs such as feature-related code lines, test-related code lines, and the mapping between test cases and features. Relying solely on static code analysis, `Test2Feature` simplifies tasks like regression testing, feature management, and HCS evolution, filling gaps left by traditional variability model-based approaches or tools that only link test cases to code lines. Although `Test2Feature` improves traceability by linking tests to feature implementations, it does not focus on security features or their contextual impact on software systems. My approach not only identifies security features but also examines their architectural distribution and adherence to security best practices, providing a more comprehensive analysis of security-related elements within a codebase.

2.5 Embedded Feature Annotations

Embedded feature annotations are in-code markers that label and document specific functionalities directly within the source code, greatly enhancing program comprehension and feature traceability [29]. They allow developers to systematically track which code fragments implement particular features, thereby facilitating maintenance and evolution in complex systems. In addition, these annotations provide a common language for stakeholders, improving communication and coordination during development [29]. Several tools have adopted this approach; for example, HAnS

[4] `https://github.com/willianferrari/Test2Feature`

leverages embedded annotations to record and trace features, while other Java-based tools such as `FeatureHouse`[5] and `Featureous`[6] also support feature traceability. I chose HAnS among these options because it provides an integrated plugin that seamlessly fits into popular development IDEs, making it the optimal choice for my approach. In my method, HAnS will serve as the primary annotating tool to mark security features, enabling an enhanced traceability, quick navigation and seamless labeling of their implementation, distribution, dependencies, and overall impact on the software architecture.

2.5.1 Helping Annotate Software (HAnS)

HAnS[7] (Helping Annotate Software) is an IDE plugin integrated into JetBrains tools, designed to support developers in annotating software assets with features [30]. The plugin enables developers to record feature locations directly during the coding process, providing functionalities such as code completion and syntax highlighting to streamline the task. Using HAnS, features can be mapped to various software assets, including files, folders, and specific code fragments. For instance, as shown in Fig. 2.2, developers can map an entire file to a feature using `.feature-to-file` files or assign a feature to an entire folder and its sub-elements with `.feature-to-folder` files.

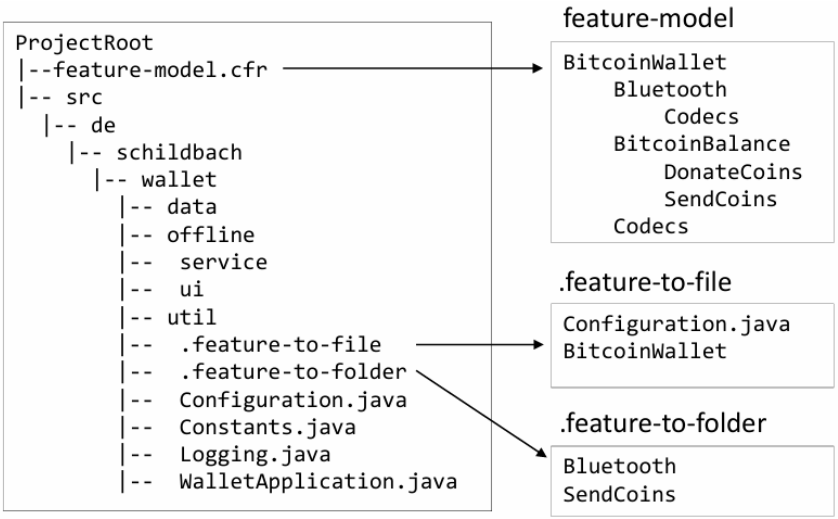


Figure 2.2: Feature model, feature-to-file, and feature-to-folder Mappings Example [29]

[5] `https://github.com/joliebig/featurehouse`
[6] `https://featureous.org/`
[7] `https://github.com/isselab/HAnS`

For more granular mapping, features can be assigned within a file using code annotations, which serve as additional descriptive comments in the code. These annotations come in two main forms: inline annotations, such as `&line[FeatureName]`, for assigning a single feature to a specific line of code, and block annotations, such as `&begin[FeatureName]` - `&end[FeatureName]`, to map an entire code block to a feature like the example in figure 2.3 .

```
// &begin[getCookieValue]
static String getCookieValue(Cookie cookie) throws UnsupportedEncodingException {
    return URLDecoder.decode(cookie.getValue(), enc: "utf-8"); //&line[decode]
}
// &end[getCookieValue]
```

Figure 2.3: Feature Block Annotation Example

HAnS also provides tools to define and manage the hierarchical relationships between features within a project. This structure is captured in a `.feature-model` file, located at the project root. This file is a critical component of HAnS, ensuring that features are well-defined and consistently applied throughout the project. As a core component of this study, HAnS will play a pivotal role in annotating security features, tracking their usage across codebases and representing their distribution among software system components. This functionality is essential for achieving the research objectives of this work.

2.6 Security Features Taxonomy

To systematically classify and analyze security features, this study adopts a taxonomy-based approach inspired by the work of Hermann et al. [6]. This taxonomy was developed through a systematic literature review and an empirical analysis of security frameworks to establish a structured representation of security features in software systems. Unlike abstract security principles, security features in this taxonomy are defined as functional components that directly address security concerns by preventing attacks, enforcing security policies, or ensuring system integrity. These features manifest in the source code as concrete security mechanisms such as authentication, encryption, or access control implementations.

The taxonomy is structured into five main security feature categories, each representing a distinct aspect of security functionality. These categories were derived by reviewing existing security standards, security frameworks, and best practices, ensuring that they capture real-world security feature implementations. Figure 2.1

illustrates the top-level security feature categories, presenting a hierarchical overview of security functionalities within the taxonomy.

Defining subcategories of security features provides a more granular classification that captures the diversity of security implementations. Each main security feature category is further divided into subcategories, ensuring a structured and systematic approach to analyzing security features across different software systems. These subcategories were identified based on common security practices in software development and patterns observed in security frameworks. By structuring security features at this level, this task directly addresses **RQ2**. Through this classification, the study provides a comprehensive understanding of the various security-related functionalities present in modern software systems, enabling a more effective evaluation of their role, implementation, and impact.

For instance, within the "Secure Data Handling" category, subcategories such as data validation, data sanitization, secure storage, and retention control define different security aspects related to handling sensitive information. These subcategories (see Fig. 2.4) reflect various techniques used to protect data integrity, confidentiality, and secure processing within a system.

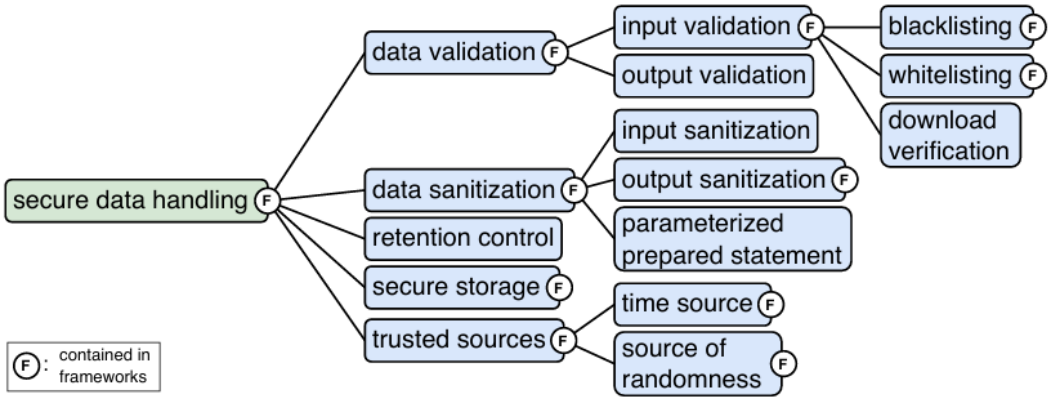


Figure 2.4: Sub-features of the security feature Secure Data handling [6]

2.7 Related Work

Mining Software Vulnerability Characteristics

Li et al. proposed a vulnerability mining algorithm to extract essential characteristics of software vulnerabilities using data mining techniques [31]. Their method processes vulnerability reports from open-source databases like `CVE`, `CWE`, and `NVD`,

identifying patterns and textual indicators that describe vulnerabilities. They utilize text mining techniques to categorize vulnerabilities into essential and non-essential groups, aiming to enhance understanding of software security risks. Their results demonstrate a recall of approximately 70% and a precision of 60%, highlighting improvements over traditional manual vulnerability classification. However, this approach primarily focuses on vulnerability detection and classification from databases rather than directly identifying and classifying security features within software repositories and analyzing them.

Mining Security Changes in `FreeBSD`

Mauczka et al. conducted a repository mining study on `FreeBSD`[8], focusing on security-related code changes over time [32]. Their method applies a lexical approach to classify security commits based on commit message keywords, filtering security-related modifications from `FreeBSD`'s version control history. The study explores the evolution of security changes, linking them to security advisories and assessing their distribution across software modules. While their approach provides insights into historical security trends, it does not attempt to identify, classify, or analyze security features themselves. Instead, it focuses on tracking security-related commits, which may not fully capture the presence or structural role of security features in software architectures.

Mining Security-Sensitive Operations in Legacy Code

Ganapathy et al. introduced a static analysis approach that mines security-sensitive operations in legacy code using concept analysis [33]. Their method identifies idiomatic resource manipulations - termed fingerprints - that indicate security-sensitive operations, such as access control checks. The approach is semi-automated, requiring a domain expert to refine mined results, and is evaluated on three real-world systems. While effective in pinpointing security-sensitive operations, the study focuses on retroactively securing legacy systems rather than understanding how security features are implemented and distributed across modern software architectures.

While previous studies have explored aspects of security-related repository mining, no prior research has specifically focused on mining security features within software systems. Existing works have either analyzed vulnerability reports [31], tracked security-related commit messages without feature-level insights [32] or identified security-sensitive operations in legacy code for policy enforcement [33]. In

[8] `https://www.freebsd.org/`

contrast, this study is the first of its kind to systematically mine security features, rather than vulnerabilities or security changes, by directly locating and annotating them within source code. By employing a semi-automated approach that leverages security-related keywords and HAnS annotations, this study not only identifies security features but also analyzes their distribution, categorizes them as custom or framework-based, explores their interactions with other components, and assesses their adherence to best practices. This feature-centric perspective fills a critical gap in security research, providing a new foundation for understanding how security features are structured and maintained in modern software architectures.

3 Methodology

This chapter describes the study design and methodology adopted for mining security features in software systems. It details the repository selection process, the tools and technologies used (`SonarQube`, `Semgrep`, and custom scripts), and the systematic approach for locating, identifying and annotating security features. Furthermore, it elaborates on the categorization of these features, their functional goals, and the analysis of their distribution and impact on the overall software architecture.

3.1 Mining Study Workflow Design

The methodology for this thesis is designed to locate, identify, categorize, and analyze security features within software systems. This process involves several key phases, each aimed at providing a comprehensive understanding of how security features are implemented, distributed, and their impact on the overall software architecture. The overview of the methodology workflow of this thesis can be found in figure 3.1.

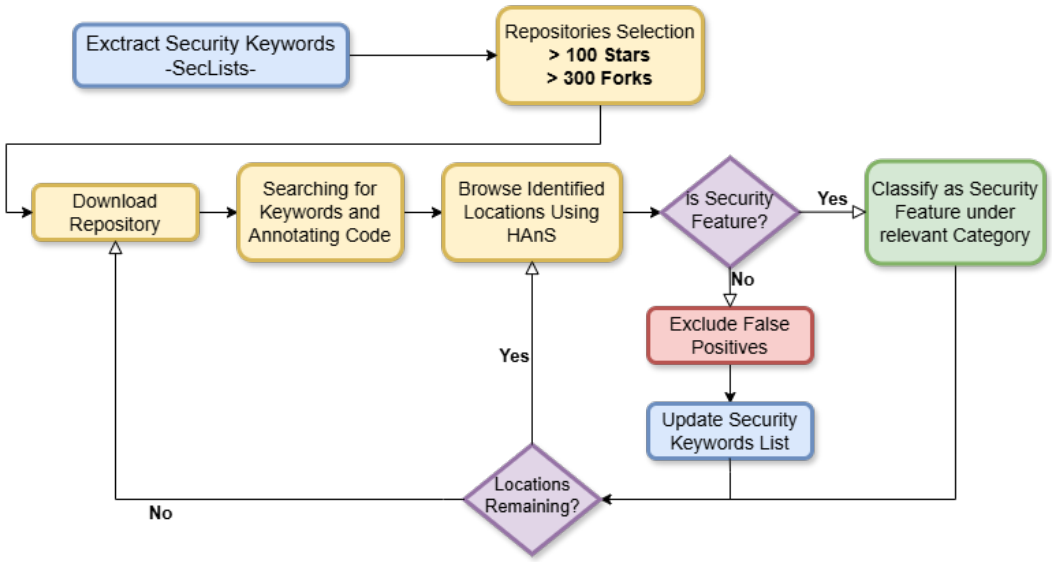


Figure 3.1: Mining Study Workflow Design

3.2 Repositories Selection Process

The repository selection process is a critical phase in this mining study, designed to ensure that the selected software repositories effectively serve the research objectives and yield meaningful insights into the nature of security features in software systems. This study employs a set of well-defined criteria to filter and select repositories, balancing quality, relevance, and diversity to support a comprehensive analysis.

A total of 6 repositories were targeted for this study. This number is justified by the qualitative nature of the research, where an overly large dataset might dilute the depth of analysis, while too few repositories would limit the generalizability of the findings. The chosen programming language is `Java`, one of the most widely used languages in software development, particularly for large-scale and enterprise-level systems [34], making it an ideal choice for observing the implementation and distribution of security features.

The selection criteria also include a minimum of 100 stars and a relative high number of forks on `GitHub` (more than 300 Forks), reflecting the repositories' popularity and widespread usage within the developer community. This criterion ensures that the selected projects are not only well-maintained but also actively used, thereby increasing the likelihood of encountering well-implemented security features. To capture potential variations in security feature implementation, repositories of varying sizes and complexities were selected. This diversity allows the study to explore how factors such as project scale and architectural complexity influence the design and distribution of security features. Additionally, only the latest stable release versions of the repositories were included to ensure that the analysis is based on the most secure and stable codebase versions, minimizing the impact of unresolved bugs or incomplete implementations.

Native security projects that serve explicit security purposes and goals, which predominantly consist of security features, were deliberately excluded from the selection process. This exclusion ensures that the focus remains on security features embedded within software systems developed for general purposes other than security, such as healthcare, banking, financial services, and data storage. These domains were chosen due to their inherent need to handle sensitive and protected information, making them more likely to incorporate diverse and sophisticated security features within their codebases. This repository selection process lays a solid foundation for the subsequent phases of this mining study, ensuring that the selected datasets are both relevant and representative, thereby enabling a thorough investigation into the implementation, distribution, and contextual usage of security features in real-world software systems.

3.3 Locating and Identifying Security Features

Identifying security features in software systems is inherently challenging due to their often scattered nature across multiple components and the lack of standardized documentation [3, 4, 6]. While automated techniques like static and dynamic analysis exist, they often suffer from high false-positive rates and difficulties in detecting fine-grained features—limitations further examined in this study (see Section 5.1.1). In response to these challenges, this study adopts an iterative process for locating and identifying security features. Rather than applying a one-size-fits-all methodology, I experiment with various tools and approaches, leveraging the insights gained to determine the most effective strategies for accurate feature identification. This adaptive strategy involves refining and adjusting the feature identification process based on the accuracy and effectiveness of the results, ensuring continuous improvement and gradually enhancing both efficiency and precision. By adopting this methodology, this part of the study aims to answer how can security features be identified and located within the code of software systems (**RQ1**). Through systematic refinement, the approach ensures that security feature identification is comprehensive, reliable, and scalable. The following sections will detail the exact methodology used in this study, including the tools and techniques applied to optimize the identification and classification of security features within software repositories.

In the subsequent sections, I will provide a detailed explanation of the tools and technologies utilized in the locating and identifying security features process, elaborating on their functionalities, implementation and desired outcome within this study.

3.3.1 SonarQube

`SonarQube` is a widely used static analysis tool designed to assess code quality, detect vulnerabilities, and enforce coding standards in software systems. It operates by scanning source code without executing it, identifying potential issues such as security vulnerabilities, code smells, and bugs. In software security, `SonarQube` plays a crucial role in detecting misconfigurations, weak implementations of security mechanisms, and adherence to best practices, making it a powerful tool for ensuring secure software development.

One of `SonarQube`'s interesting offered capabilities is `Security Hotspots`, which highlight sections of code that require developer review due to potential security risks. Unlike critical vulnerabilities that require immediate remediation, `security hotspots` indicate areas where security-sensitive code implementations exist but may not necessarily be exploitable unless misused. These hotspots are flagged when

SonarQube detects weak implementations of security-related methods, such as improper authentication mechanisms, inadequate encryption, or insecure input handling.

In the context of this study, Security Hotspots may provide valuable insights into the locations of security features within software repositories. Since they flag any security-relevant implementation that may pose a risk, they serve as indicators of security feature presence, allowing for systematic identification and classification of security features across different codebases. By leveraging SonarQube's hotspot detection, this study enhances the accuracy of locating security features, ensuring that even potentially insecure or misconfigured implementations are considered during analysis. To integrate SonarQube into this security feature mining study, the following structured deployment methodology was followed:

Repository Setup

The selected repositories were cloned and built locally to allow for comprehensive source code analysis, since repositories can not be scanned by SonarQube unless they were build first.

IDE Integration with SonarQube for IDE

SonarQube for IDE [1], a lightweight plugin for IntelliJ IDE, was used to perform real-time static analysis within the development environment, providing an immediate review of potential security hotspots before deeper analysis.

SonarQube Deployment via Docker

A SonarQube server was set up in a Docker container, running locally to enable continuous scanning of repositories while storing and managing analysis results effectively.

API-Based Data Extraction

To extract and filter relevant security feature data, a Python script was developed to interact with the SonarQube API, retrieving findings related to security hotspots and storing the extracted information in JSON-formatted files for further processing and classification.

[1] https://plugins.jetbrains.com/plugin/7973-sonarqube-for-ide

Filtering and Structuring the Findings

The extracted security hotspots were then filtered and categorized based on their relevance to security features, ensuring that only meaningful security-related code fragments were considered in the final dataset.

3.3.2 Semgrep

`Semgrep` is a lightweight static analysis tool designed for fast, pattern-based code scanning. Unlike traditional static analysis tools, `Semgrep` provides a simplified and highly customizable approach to detecting security vulnerabilities and enforcing best practices in codebases. It operates by matching predefined or custom security rules against source code, allowing developers to identify potential vulnerabilities, insecure coding patterns, and misconfigurations without requiring complex configurations.

Similar to `SonarQube`, `Semgrep` is used in software security to identify weaknesses in security-relevant code, such as improper authentication, missing input validation, and the use of weak cryptographic functions. However, `Semgrep` stands out for its usability and flexibility, offering a more user-friendly experience with minimal setup requirements. It integrates directly with `Python`, allowing security scans to be performed through simple scripts, making it a lightweight yet powerful alternative to traditional static analysis tools.

Given its ease of use and integration capabilities, `Semgrep` was deployed in this study using a script-based approach to facilitate efficient security feature mining and another source for it. The methodology consists of the following steps:

Scanning with Python

A `Python` script was developed to automate the security analysis process. This script takes only the URL of the mined repository as input, clones the repository, and executes `Semgrep` security scans based on predefined security-related patterns.

Filtering and Processing the Findings

The output of `Semgrep` includes various security warnings and detected patterns. The script processes these results to filter only the relevant security-related findings that contribute to security feature identification.

Exporting Data in JSON Format

The filtered security findings are then structured and stored in a JSON file, ensuring that the extracted security feature data is well-organized and readily available for further analysis and classification.

3.3.3 Keyword-Based Search

While `SonarQube` and `Semgrep` provide valuable insights into security weaknesses, they have a fundamental limitation, that is, they primarily detect security features with vulnerabilities or improper implementations. This means that well-implemented security features, which do not raise warnings or trigger security hotspots, remain undetected by these tools. As a result, relying solely on static analysis tools would create a biased and incomplete picture of security feature distribution within the studied repositories.

To overcome this blind spot and ensure a more comprehensive and holistic approach, a keyword-based search was developed to expand the security feature identification process beyond vulnerability detection. This search is designed to systematically scan repositories for security-related implementations, regardless of their correctness or vulnerability status.

The keyword-based search was designed to operate efficiently with minimal input requirements while ensuring thorough and scalable security feature mining. The methodology consists of the following steps:

Automated Repository Processing

Similar to the previous tools, the script takes only the URL of the mined repository as input, automating the cloning and file analysis process.

Input Filtering

To reduce false positives, the script implements context-based filtering to ensure that the identified keywords correspond to actual security feature implementations rather than generic or unrelated code snippets. For instance, it ignores comments within the source code to prevent irrelevant matches. Additionally, testing functions are excluded, as they are not considered security-related within the scope of this study. The script also filters out any pre-existing HAnS-annotated features, as these have already been identified and labeled.

Data Structuring and Storage

The extracted security-related findings are stored in a structured JSON format, facilitating further categorization, annotation, and analysis.

3.3.3.1 Security Keywords and Rules

The security features considered in this study were structured based on the security features taxonomy proposed by Hermann et al. [6]. This taxonomy provides a systematic classification of security features, ensuring that the keyword selection process aligns with established security feature categories. A detailed description of this taxonomy was provided in Section 2.6.

Each sub-category within the taxonomy was assigned a set of relevant security-related keywords, which were extracted from `SecLists`, a well-known repository compiling security-related keyword lists from various sources[2]. An example of this keywords-features mapping can be found in figure 3.2. By leveraging `SecLists`, this study ensures that security feature identification relies on widely recognized and comprehensive keyword datasets, improving detection accuracy and completeness. And by mapping the security keywords to security feature categories and sub-categories, we investigate what features exist in a software system ((**RQ2**).

Figure 3.2: Keywords-Features Mapping Examples

Given the complexity and variability of security features, the keyword list is continuously refined through an iterative improvement process. The goal is to enhance

[2] `https://github.com/danielmiessler/SecLists`

precision and minimize false positives by dynamically adjusting the keyword set based on real-world validation. This process consists of the following adjustments:

Eliminating Keywords with High False Positive Rates

If a keyword frequently identifies irrelevant code segments, such as generic logging functions or unrelated system calls, it is removed from the list to enhance accuracy.

Adding Newly Observed Keywords and Patterns

If recurrent security-related terms or code patterns are identified during output validation, they are incorporated into the keyword list to improve detection capabilities.

By employing this adaptive keyword-based approach, this study ensures a comprehensive yet precise method for identifying and locating security features within software repositories. The iterative refinement process guarantees continuous improvement, making security feature detection more reliable and scalable over time.

3.3.3.2 Embedded Feature Annotations

To enhance the efficiency of validating and navigating security feature locations, the Help Annotating Software (HAnS) plugin[3] was used in this study and its annotations format was generated by the script, enabling structured feature labeling directly within the source code. This allows for a more systematic approach to identifying and reviewing potential security feature locations while leveraging HAnS's seamless compatibility with `IntelliJ` IDE[4] for efficient navigation.

In this study, HAnS annotations are automatically inserted into the source code at positions where potential security features are detected. When the script identifies one or more security-related keywords within a particular line of code, it flags that line as a potential security feature location. However, to avoid redundancy and clutter, the script ensures that only a single HAnS annotation is used per line, even if multiple security keywords are detected. This streamlined approach prevents excessive tagging while maintaining clear and structured annotations.

[3] `https://plugins.jetbrains.com/plugin/22759-hans`
[4] `https://www.jetbrains.com/idea/`

By using these annotations, the validation process becomes more efficient and less time-consuming. Instead of manually searching through extensive source code files, the HAnS plugin in `IntelliJ` IDE provides immediate access to annotated security features, allowing for quick navigation and systematic review. The structured annotation format makes it easier to assess potential security feature locations, ensuring that validation can be conducted in an organized and scalable manner. This method not only improves the accuracy of the security feature identification process but also significantly reduces the effort required for manual inspection and validation. An example to a feature model can be seen in the following figure.

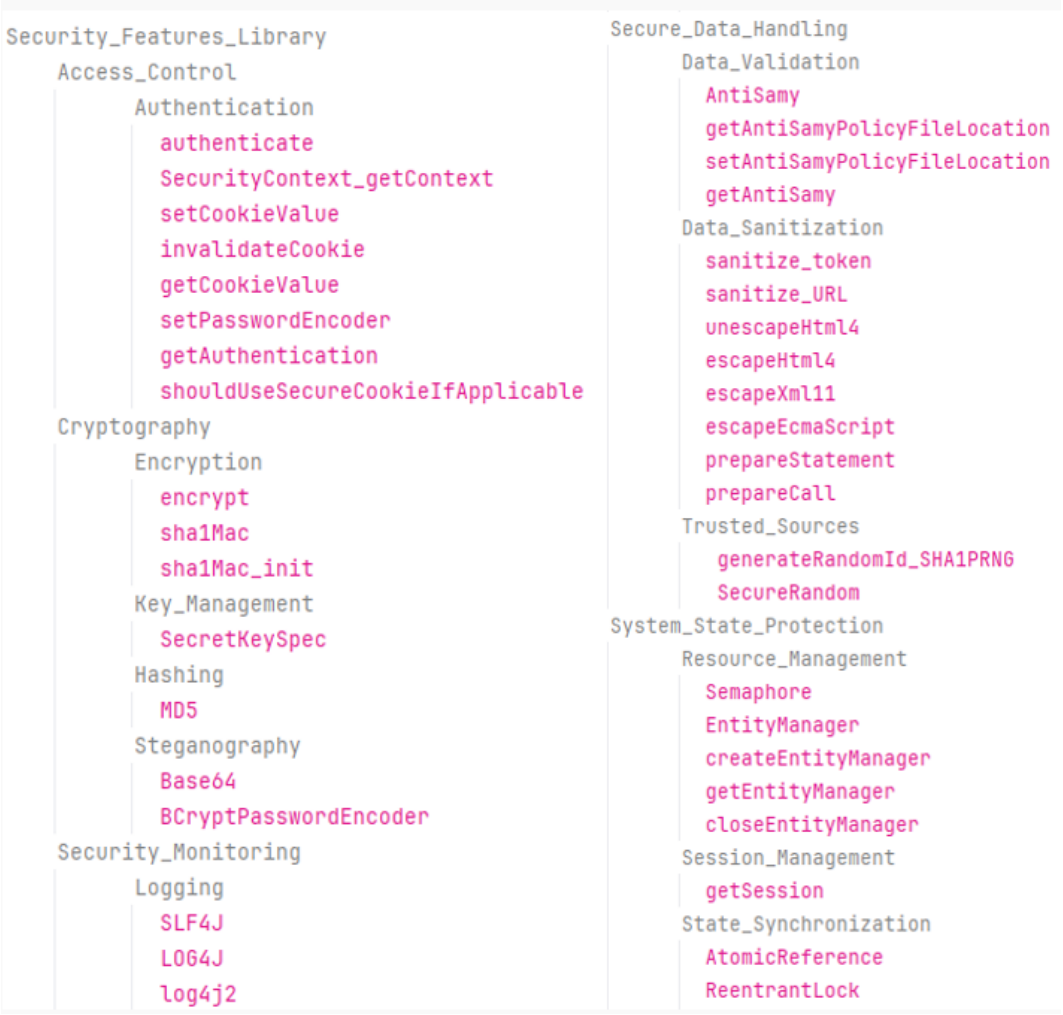


Figure 3.3: Security Feature Model Example

3.3.3.3 Security Features Sources

While the categorization of security features is primarily structured based on the security feature taxonomy, an equally important aspect of this process is identifying the origin of these features within the software repositories. This classification is based on labeling whether the identified security features are imported from external security frameworks or libraries or manually developed by the software engineers themselves. Distinguishing between these two categories provides valuable insights into how security mechanisms are integrated into real-world software systems.

A key assumption in this study is that security features derived from frameworks or libraries should be easier to locate compared to custom-developed ones. This assumption arises from the fact that framework-based security features typically use standardized API calls and follow consistent naming conventions, making them more traceable through automated detection methods. In contrast, custom security features, which are manually implemented by developers, may exhibit greater variability in their naming, structure, and integration, complicating their identification. The validity of this assumption and its impact on security feature traceability will be further discussed in the Study Results (Section 5.2).

Library-Based Security Features

To systematically identify library-based and framework-based security features, this study incorporates an external tool developed by my colleague [5], which specializes in detecting security features imported from frameworks or libraries. This tool works by analyzing security libraries and frameworks usages within the source code and mapping them to known security-related frameworks and libraries. By leveraging this tool, the study aims to automate the detection process for imported security features, ensuring a structured and scalable approach to classifying them. Meanwhile, custom security features—those written manually by developers—will be identified through context-based keyword matching and manual validation to ensure accuracy. Classifying security features discovers which security features are custom implemented by developers, and which are integrated from security frameworks and libraries (**RQ3**).

[5] `https://github.com/David0x03/security-feature-mining-study/tree/main/SecurityFeatureMiningStudy`

3.4 Security Features Analysis

The final stage of this mining study focuses on analyzing security features beyond their identification and categorization. This phase aims to evaluate their effectiveness, distribution, and impact on software security. The analysis consists of two key aspects: exploring the architectural distribution of selected security features and their influence on system security and semantic assessment of them.

3.4.1 Exploring Security Features Distribution

Beyond assessing individual security features, it is crucial to examine how they are distributed across the software architecture. This analysis part focuses on analyzing the distribution patterns, scattering degrees and nesting depth of security features, leveraging insights gathered from HAnS annotations and provided traceability data.

By exploring security feature distribution, this study can help to understand whether security measures are centralized in specific software modules or dispersed across multiple components. This architectural analysis is essential for assessing the effectiveness of security feature placement and an approach to discover how are security features distributed across the software architecture, and to see what patterns can be observed in their distribution (**RQ4**).

3.4.2 Features Implementation Analysis

To assess the effectiveness and security robustness of identified security features, a implementation analysis will be conducted on a selected subset of features. The selection will be based on the criticality of the security goal they aim to fulfill, ensuring that the most impactful and security-sensitive features are examined. This analysis will involve a semantic evaluation of feature implementations, determining whether they successfully fulfill their intended security purpose and whether they are implemented following best security practices (**RQ5**).

4 Implementation

This chapter focuses on the implementation details of the methods and tools developed in this thesis. It explains the process of security feature identification, enhancements in traceability techniques, and the integration of Static Code Analysis (SCA) tools like `SonarQube` and `Semgrep`. Additionally, it discusses the challenges encountered during implementation and the proposed solutions to address these challenges.

4.1 Security Features Locating and Identification Process

This section describes the steps taken to locate and identify security features in the selected software projects using different tools and techniques.

4.1.1 Using SCA Tools in Locating and Identification Process

The process began with Static Code Analyzers (SCA) to identify potential security feature locations. These tools highlight code patterns linked to security risks, offering a useful—but limited—starting point for further analysis.

4.1.1.1 SonarQube

The implementation process began by installing the official `SonarQube` Docker image from Docker Hub[1] and setting up the `SonarQube` instance locally. Then the `SonarQube` for IDE plugin[2] was also configured within `IntelliJ IDEA`, ensuring direct integration with the `SonarQube` server for real-time feedback. A prerequisite for `SonarQube` analysis is that the `Java` source files must be successfully built prior to scanning. This requirement applies regardless of the project's build system—whether `Maven` or `Gradle`—as `SonarQube` relies on the compiled class files to conduct accurate static analysis.

[1] `https://hub.docker.com/_/sonarqube`
[2] `https://plugins.jetbrains.com/plugin/7973-sonarqube-for-ide`

Once the setup was complete, the analysis phase commenced. The analysis parameters were configured to include only Java files, ensuring that irrelevant files were excluded from the scan. This was achieved using `SonarQube CLI` commands, which also required defining a unique project key and authentication token for each project within the `SonarQube` instance. An example of the `CLI` command used for initiating the analysis is presented in Listing 1.

```
sonar-scanner.bat ^
  -D"sonar.projectKey=OpenRefine" ^
  -D"sonar.sources=." ^
  -D"sonar.java.binaries=main/target/classes" ^
  -D"sonar.inclusions=**/*.java" ^
  -D"sonar.host.url=http://localhost:9000" ^
  -D"sonar.token=sqp_0f812d36e4788b8c69bab294713c314f65fdeef3"
```

Listing 1: Running `SonarQube` on a Project Example

After successfully analyzing the codebase, the findings were automatically saved within the `SonarQube` instance and could be easily navigated via its web interface. Among these findings, the Security Hotspots were of particular interest, as they highlight areas in the code that may indicate potential security feature locations. These hotspots could be browsed and reviewed in detail through the `SonarQube` dashboard, as illustrated in the following figures.

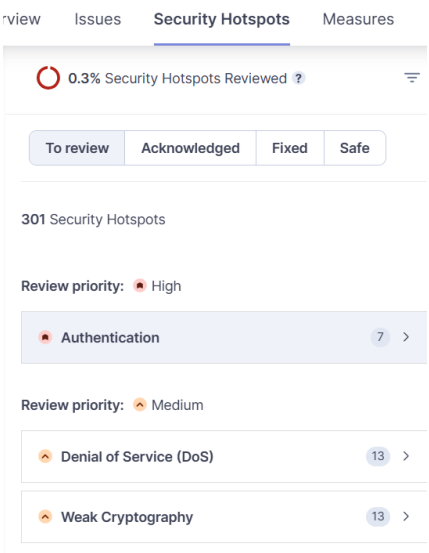


Figure 4.1: Security Hotspots Categories

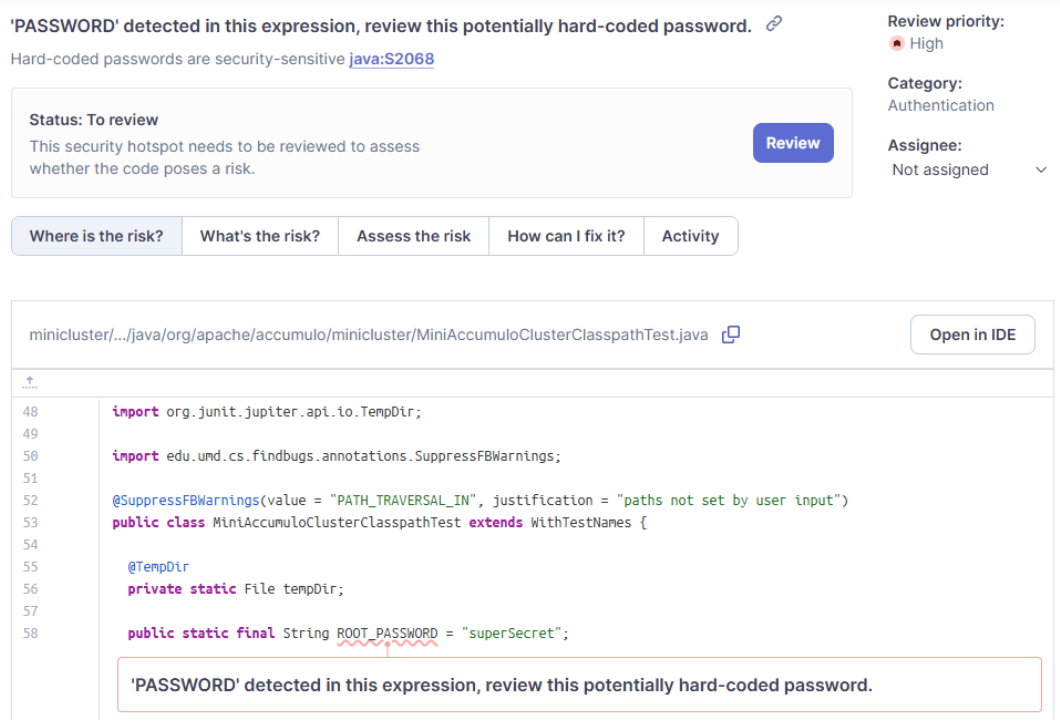


Figure 4.2: Security Hotspots Navigation

To facilitate structured data analysis, I developed a custom Python script that interacts with the `SonarQube` API. This script retrieves the findings and organizes them into a structured JSON format, aggregating security hotspots at the file level and extracting relevant attributes. These attributes include the file name and path, location lines, security category, description, severity, and recommended best practices. This structured format ensured that the data was easily accessible for subsequent analysis. An example of the results format is shown below.


```
"projectName": "accumulo",
"fileName": "PasswordTokenTest.java",
"filePath": "core/src/test/java/org/apache/accumulo/core/client/security/tokens/PasswordTokenTest.java",
"Security Category": "auth",
"Description": "Hard-coded passwords are security-sensitive",
"Location": "Line 43",
"Severity": "HIGH",
"Best Practices": "Recommended Secure Coding Practices. Store the credentials in a configuration file that
```


Figure 4.3: SonarQube Findings Example

4.1.1.2 Semgrep

The setup of `Semgrep` proved to be simpler and easier compared to `SonarQube`. Unlike `SonarQube`, `Semgrep` did not require the software projects to be built prior to analysis. Additionally, its Docker container could be easily executed within the Python script, eliminating the need for setting up a dedicated analysis instance beforehand. This streamlined approach was more flexible.

The developed Python script was designed to be straightforward and easy to use. It accepts the GitHub repository link as input, automatically initiates the `Semgrep` Docker container, and begins the analysis process. As with `SonarQube`, the analysis parameters were configured to focus exclusively on Java files, ensuring that irrelevant files were excluded. For performance optimization and storage efficiency, each repository was temporarily cloned locally, analyzed, and then deleted after the results were successfully generated.

The findings were structured to capture the most relevant and informative attributes, including the file path, exact code location line, the corresponding source code, description, severity level, security category, CWE, and reference links. This comprehensive structure ensured that each detected security-related element was clearly documented and easily traceable. The results were saved in a JSON file format, providing a consistent and structured dataset for subsequent analysis and validation. An example of the structured JSON output from the `Semgrep` analysis is presented in the following figure.


```
"File Path": "/src/minicluster/src/main/java/org/apache/accumulo/minicluster/MiniAccumuloRunner.java",
"Location": "line 253",
"Code": "    try (ServerSocket shutdownServer = new ServerSocket(shutdownPort)) {",
"Description": "Detected use of a Java socket that is not encrypted. As a result, the traffic could be read by an attacker
"Severity": "MEDIUM",
"Category": "security",
"Technology": [
    "java"
],
"CWE": [
    "CWE-319: Cleartext Transmission of Sensitive Information"
],
"References": [
    "https://owasp.org/Top10/A02_2021-Cryptographic_Failures"
]
```


Figure 4.4: Semgrep Findings Example

4.1.2 Keyword-Based Search

The Keyword-Based Search process is implemented using a `Python` script [3], designed to semi-automate the process of detecting potential security features within software codebases. The script initiates by receiving the repository URL as input, after which it clones the repository locally. Unlike the temporary approach used in the `Semgrep` process, these repositories are cloned permanently to support further steps, including annotation and classification of the identified security features.

Upon cloning, the script automatically creates a local directory named after the project name, which is extracted directly from the provided repository URL. This directory serves as the storage space for the cloned repository, ensuring that it is readily available for subsequent analysis stages.

The entire process of cloning and setting up the local directory structure is encapsulated within the `clone_repository` function, as shown in the following listing:

```python
import os
import subprocess

def clone_repository(repo_link):
    """Clone the repository into a 'repos' directory with the project name."""
    # Extract project name from the repo URL
    project_name = os.path.basename(repo_link).replace(".git", "")
    repos_dir = os.path.join(os.getcwd(), "repos")
    project_dir = os.path.join(repos_dir, project_name)

    # Create the 'repos' directory if it doesn't exist
    os.makedirs(repos_dir, exist_ok=True)
    if os.path.exists(project_dir):
        print(f"Repository '{project_name}' already exists in 'repos'.")
    else:
        try:
            subprocess.run(["git", "clone", repo_link, project_dir], check=True)
            print(f"Cloned '{project_name}' into 'repos'.")
        except subprocess.CalledProcessError as e:
            raise RuntimeError(f"Failed to clone repository: {e}")
    return project_dir, project_name
```

Listing 2: `clone_repository` Function

[3] `https://github.com/David0x03/security-feature-mining-study/tree/main/SecurityKeywordsBasedSearchTool`

4.1.2.1 Input Filtering

The input filtering process is a critical step in ensuring that the keyword-based search yields accurate and relevant results, minimizing false positives and enhancing the quality of the identified security features. The filtering process was designed for four primary components, each addressing specific challenges encountered during the initial implementation phase.

The first filtering component was comments within the source code. Comments frequently contain keywords that overlap with security-related terms, which could result in false-positive matches. Since comments do not represent executable code or actual security implementations, including them would significantly reduce the precision of the identification process. This decision was taken by preliminary experiments conducted on three different projects, where it became evident that comments were a major source of irrelevant matches. Excluding comments, therefore, ensured that the analysis remained focused solely on actual code implementations, enhancing the overall accuracy and reliability of the results.

The second filtering component was test-related code, as it does not represent functional security features within the scope of this study. The goal of the analysis is to focus on real, operational security features, not their corresponding test functions. Moreover, security-related tests were not categorized within the adopted security features taxonomy [6], further justifying their exclusion. This filtering approach ensures that the identified features are functional components contributing to the system's security, rather than auxiliary test cases.

The third filtering component addressed previously annotated code lines. If the keyword-based script is executed multiple times on the same codebase to annotate newly added keywords or if the script is executed on a preivously annotated codebase, the already annotated features could be redundantly processed, leading to duplicate annotations. Additionally, once the findings from features imported from security frameworks and libraries were annotated (more in subsection 4.2.1), it is essential to avoid re-annotating them during subsequent scans. This filtering approach also accommodates scenarios where a user may wish to identify only new, previously undetected security features within the codebase. By excluding already annotated lines, the script ensures that only unique and valuable insights are generated during each iteration.

The fourth filtering component involved the exclusion of `import` statements. While `imports` may indicate the usage of security libraries and frameworks, they do not necessarily confirm the implementation of security features within the codebase.

An imported security library might not be actively used, or its imported components may not represent security features directly relevant to the study. Therefore, excluding import statements prevents the generation of misleading or irrelevant matches, focusing the analysis on actual security feature implementations within the code.

Finally, a fundamental filter was applied to restrict the search process to `Java` files only, excluding all other file types. Since `Java` is the primary programming language targeted in this study, this ensures the search remains focused and consistent with the study's objectives.

4.1.2.2 Security Keywords and Rules

The Security Keywords were defined and structured within a JSON file, serving as the primary database for the Keyword-Based Search process. The structure of this file was directly aligned with the taxonomy of functional security features by Hermann et al. [6], using its main categories and subcategories as the foundational mapping. Under each subcategory, a set of security-related keywords was defined to facilitate accurate feature identification (as previously illustrated in Figure 3.2). This structured approach ensured that the keyword search process was systematic, organized, and aligned with the broader security feature classification framework.

The keyword sets were designed to be iteratively refined and continuously updated throughout the searching process. With each iteration of the search, new insights were gained, leading to adjustments in the keyword lists. Notably, new keywords were frequently added during the manual validation of the identified possible security feature locations. During this process, it was observed that many other security features were implemented within dedicated security-related classes or files, which contained patterns or terms not initially included in the keyword list. When such keywords were identified, they were immediately added to the corresponding subcategory in the JSON file, ensuring that future iterations could detect similar security features in other codebases.

In addition to adding new keywords, existing keywords were continuously refined to enhance detection precision. Some keywords required adjustments to better align with the coding patterns observed during validation, ensuring that the script could more accurately capture true security feature implementations and reduce the occurrence of false positives. Conversely, keywords that consistently led to irrelevant matches or exhibited a high false-positive rate were eliminated from the database. This process of iterative refinement ensured that the keyword-based search remained relevant, precise, and adaptive to the complexities of real-world codebases.

After the Keyword-Based Search script was executed, it provided initial output insights to guide the subsequent validation process. This included the total number of potential security feature locations, the number of files where keyword matches were found, how many features have been successfully identified by the automated tool (Subsection. 3.3.3.3), and an overview of the top 10 keyword matches. Each of these matches was presented alongside the category and subcategory under which it falls, offering the user an early indication of the types of security features likely to be encountered in the codebase. This preliminary insight allowed for better preparation and focus during manual validation, ensuring a more efficient review process. An example of this initial output insight is presented in Figure 4.5.

```
Enter the repository URL: https://github.com/openmrs/openmrs-core
Cloning into 'C:\Users\Yasser Hekal\SecFeatFinder\repos\openmrs-core'...
Updating files: 100% (1771/1771), done.
Cloned 'openmrs-core' into 'repos'.
Results saved to openmrs-core_PotentialFeatLocations.json
.feature-model file created at: C:\Users\Yasser Hekal\SecFeatFinder\repos\openmrs-core\.feature-model
Number of Features Identified by the Automated Tool: 5 Features
Number of Security Feature Possible Locations: 257
Number of Files with Matches: 91

Top 10 Keyword Matches:
Access Control : Authorization : Role: 112 matches (43.58%)
Secure Data Handling : Data Validation : Validator: 57 matches (22.18%)
System State Protection : State Synchronization : ReentrantLock: 25 matches (9.73%)
Cryptography : Steganography : encode(: 9 matches (3.50%)
Access Control : Authentication : Cookie: 8 matches (3.11%)
Security Monitoring : Automated Response : AlertService: 7 matches (2.72%)
Cryptography : Steganography : Base64.: 6 matches (2.33%)
Cryptography : Signature : verify: 5 matches (1.95%)
Cryptography : Steganography : decode(: 4 matches (1.56%)
Secure Data Handling : Data Sanitization : StringEscapeUtils: 4 matches (1.56%)
```

Figure 4.5: Initial Output Insights Example

4.1.2.3 Embedded Feature Annotations

The Keyword-Based Search script was designed to automatically generate Embedded Feature Annotations in a format that can be directly used by HAnS at the identified possible security feature locations. This approach aimed to facilitate seamless navigation and efficient validation of potential security features within the codebase. By leveraging the HAnS annotation structure, the validation process became more structured, allowing for quick identification and manual inspection of flagged code segments.

Upon executing the script, a file named `.feature-model` was automatically created

within the project's root directory. This file served as the central annotation reference, where all potential security feature locations were documented. Each identified location was assigned a generic naming convention, using the format `PosX`, where `X` represents a sequential identifier. This naming ensured a simple yet organized approach to managing and reviewing potential feature locations.

Once annotated, these positions could be easily accessed and navigated using the HAnS plugin within `IntelliJ IDEA`, enabling the validator to quickly move through the flagged locations for manual assessment. This process significantly streamlined the validation phase, reducing the time and effort required to manually locate and review potential security features. An example of the generated `.feature-model` file and how the annotated positions facilitate navigation and validation is illustrated in Figure 4.6.

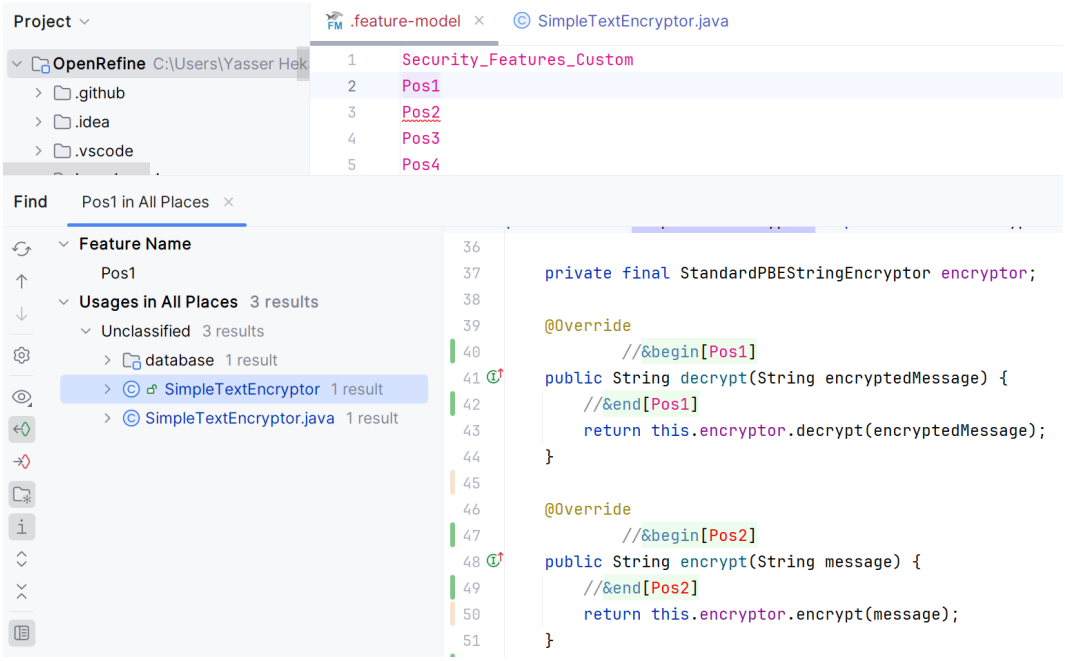


Figure 4.6: "`.feature-model`" File Example

4.2 Security Features Classification and Labeling

Once the `.feature-model` file is generated with the initial listing of possible security feature locations, the validation and classification process begins. The primary objective of this phase is to map and label the identified security features based on their source and functional classification, ensuring a structured and comprehensive overview of the features within each project.

To facilitate this, the `.feature-model` file is organized into three main categories:

`Security_Features_Custom:` This category includes security features that are manually developed by the software developers within the project. These features represent custom implementations that address specific security needs.

`Security_Features_Library:` This category covers security features that are imported from external security libraries or frameworks. These features leverage predefined security functionalities provided by established frameworks, ensuring standardized and reliable security implementations.

`Security_Features_Library_Tool:` This category will be explained in detail in the next subsection, as it pertains to specific tool-assisted detections.

```
Security_Features_Custom
    Access_Control
        Authentication
            LoginCommand
                processCookiesWithPrefix
                removeUsernamePasswordCookies
                removeOwnerOnlyConsumerCookies
                getCookieValue
                setCookie
                removeCookie
            getFreshToken
            validToken
            setPassword
    Cryptography
        Encryption
            SimpleTextEncryptor
            encrypt
            decrypt
        Hashing
            MD5
    Secure_Data_Handling
        Data_Validation
            tamperWithToken
        Data_Sanitization
            sanitizeCookieKey
            langs_directory_escape
    System_State_Protection
        Resource_Management
            createThreadPool
        Miscellaneous
            Content_Security_Policy
```

```
Security_Features_Library
    Access_Control
        Authentication
            encryptor_setPassword
    Cryptography
        Encryption
            encryptor_setAlgorithm
            encryptor_encrypt
            encryptor_decrypt
        Hashing
            md5Hex
        Steganography
            URLDecoder_decode
    Secure_Data_Handling
        Data_Sanitization
            escapeHtml4
            escapeXml10
            escapeXml11
            escapeJava
            unescapeJava
            escapeJson
            escapeCsv
            escapeEcmaScript
        Trusted_Sources
            SecureRandom
```

Figure 4.7: Security Features Classification and Labeling Example

Under each of these three divisions, the standard security feature taxonomy is ap-

plied, organizing features according to their respective categories and subcategories. This hierarchical structure ensures consistency and traceability throughout the classification process. A real-world example illustrating the structure and organization of the first two divisions, `Security_Features_Custom` and `Security_Features_Library`, is presented in Figure 4.7.

During manual validation, each identified feature possible location is carefully reviewed and classified under the appropriate category and subcategory. This step ensures that each feature is accurately mapped according to its functional purpose and source. After completing the validation, any categories or subcategories where no security features were identified are removed from the final mapping to enhance clarity and readability. This results in a concise and focused overview, emphasizing only the relevant security features present in the codebase.

4.2.1 Features Imported from Security Frameworks and Libraries

This section explains the third and final division of the `.feature-model` file, namely `Security_Features_Library_Tool`. This division specifically encompasses security features identified using an external tool developed by my colleague[4]. The primary purpose of this tool is to automatically detect security features that are imported from a predefined set of security libraries and frameworks, as listed in Table 4.1.

Table 4.1: List of Security Frameworks and Libraries Covered by the Tool

Framework/Library	**Reference**
Apache Shiro	`shiro.apache.org/`
Bcrypt	`github.com/patrickfav/bcrypt`
Bouncy Castle	`bouncycastle.org/`
Google Tink	`developers.google.com/tink`
Java SE	`oracle.com/java/technologies/javase/`
OWASP ESAPI	`owasp.org/www-project-enterprise-security-api/`
Pac4j	`pac4j.org/`
Spring Security	`spring.io/projects/spring-security`

The tool operates by scanning the codebase and identifying API calls and import statements linked to the specified libraries and then tracing functions and methods used from these libraries and frameworks. However, it is important to note that the tool's role is limited to locating the positions of these features within the

[4] `https://github.com/David0x03/security-feature-mining-study/tree/main/SecurityFeatureMiningStudy`

code. It does so by executing a `CLI` command, which is provided in Listing 3. The tool then outputs its findings in a structured JSON format without directly annotating the code. An example of the generated results is presented in Figure 4.8.

```
locate PROJECT_DIR --mappings MAPPINGS_DIR
```

Listing 3: Locating Features Command


```
"path": "src\\main\\java\\org\\traccar\\api\\signature\\CryptoManager.java",
"apiCalls": [
  {
    "line": 53,
    "api": "java.security.Signature.getInstance",
    "features": [
      "Signature"
    ]
  },
  {
    "line": 54,
    "api": "java.security.Signature.initSign",
    "features": [
      "Signature"
    ]
```


Figure 4.8: Results from Located Features Example

Given that the tool does not perform automatic annotation, it became necessary to integrate its output into the main keyword-based search script. This integration ensured that the identified security features were automatically annotated using the HAnS format within the .feature-model file before initiating the manual validation process. This approach significantly reduced manual effort, as it prevented redundant work by excluding features that had already been accurately detected and annotated by the tool.

Furthermore, it is important to highlight that security features imported from frameworks or libraries not listed in Table 4.1 would still be identified through the standard Keyword-Based Search process. Once identified, these features would be classified under the `Security_Features_Library` division within the `.feature-model` file, ensuring comprehensive coverage of all security features, regardless of the source of their library or framework. Additionally, these newly identified libraries and frameworks would be added to the list of frameworks recognized by the tool, enabling them to be automatically detected in future analyses and thereby simplifying the identification process in subsequent projects.

5 Study Results

This chapter presents the findings of the study, including the evaluation of the distribution of security features, the automated features locating tool, and security features metrics that assess their role in the broader software system landscape. Insights on modular design, traceability, and a detailed evaluation of the research questions are also provided to summarize the outcomes of the study.

5.1 Security Features Locating and Identification Process Evaluation (RQ1)

5.1.1 SCA Tools Results

The results generated by `SonarQube` and `Semgrep` during the security feature locating and identification process were, overall, limited in effectiveness. The input set for this evaluation consisted of six repositories, including two Apache projects, two Eclipse projects, and two additional standalone projects. These repositories were selected to represent diverse and widely used Java-based software systems.

The contrast in outcomes between the two tools was significant. `SonarQube` produced a total of `1,584` Security Hotspots, representing potential security feature locations across the six repositories. In stark comparison, `Semgrep` yielded only `16` possible locations in total. Notably, `Semgrep` failed to detect any results in two of the six repositories, highlighting inconsistencies in its detection capabilities across different project structures and codebases.

Following manual validation and annotation, the average false positive rate for `SonarQube` was determined to be `90.74%`, while that of `Semgrep` was `12.50%`. Although `Semgrep`'s false positive rate appears relatively low, this result is less meaningful given the extremely small number of locations identified by `Semgrep`—an average of only three findings per project. The limited number of initial detections significantly reduces its value in a broader feature identification context.

A detailed comparison of findings and false positive rates for both tools across the six repositories is presented in Table 5.1.

Table 5.1: Findings of Security Feature Detection Using `SonarQube` and `Semgrep`

Project	SonarQube (SF/L)	Semgrep (SF/L)	SonarQube FP Rate	Semgrep FP Rate	Lines of Code (Java)
Apache Accumulo	**10**/301	**3**/3	96.68%	0.00%	424K
Apache ActiveMQ	**8**/584	**4**/4	98.63%	0.00%	484K
Eclipse Milo	**6**/30	**1**/1	80.00%	0.00%	171K
Eclipse Tradista	**3**/505	0/0	99.41%	N/A	100K
Teammates	**5**/23	**4**/8	78.26%	50.00%	133K
OpenRefine	**9**/105	0/0	91.43%	N/A	102K

Note: SF/L = Security Features Confirmed / Potential Locations

SCA Tools Limitation

The most significant limitation observed in the use of SCA tools, such as `SonarQube` and `Semgrep`, lies in their inherent main task toward problem detection. These tools are primarily designed to identify security-related issues or weaknesses, which means their detection mechanisms are triggered only when there is something misconfigured, vulnerable, or suspicious in the code. As a result, any well-implemented and correctly integrated security feature is likely to go unnoticed, since it does not raise any red flags within the predefined rule sets of these tools.

Moreover, an important distinction became clear during the evaluation: many of the findings highlighted by the tools were not actual security features, but rather general-purpose functionalities that introduced potential security risks. Two prominent examples observed across the projects include the detection of hardcoded IP addresses, which were flagged as security-sensitive, and the presence of enabled debug functionalities, which were interpreted as risky from a security standpoint. While these findings are valid in the context of code quality or secure coding practices, they do not represent security features by design or intent.

As a result of analyzing the output and attempting to use it to support the Locating and Identifying Security Features process, it became evident that SCA tools did not provide the level of support expected. Their inability to detect secure, functional security features—which are often essential for enforcing authentication, access control, data protection, and similar goals—renders them ineffective as standalone tools

for the purpose of this study. This blind spot fundamentally limits their utility in locating security features in software systems code bases.

5.1.2 Input Set Projects

The input set for this analysis included six repositories, each selected based on their primary functional purpose and relevance to security-sensitive operations, as listed in Table 5.2. These systems represent real-world applications that handle protected or critical data, making them strong candidates for identifying a wide spectrum of security features.

OpenRefine is a data-cleaning and transformation tool commonly used in data science workflows, often processing large datasets that may include sensitive information. **Traccar** is a GPS tracking platform used in transportation and fleet management, involving real-time location data. **LibrePlan** is an enterprise-grade project management system that typically handles organizational and personnel data. **OpenMRS** is an open-source platform used in clinical environments, managing electronic medical records and health-related information. **Broadleaf Commerce** is a retail-oriented e-commerce framework, processing personal and payment data. Lastly, **Java NATS** is a lightweight messaging system used in distributed cloud infrastructure, where message confidentiality and system integrity are essential.

Table 5.2: Application Domains of the Selected Projects

Project	Field of Use	Lines of Code (Java)
OpenRefine	Data Processing / Data Science	102K
Traccar	Transportation / GPS Tracking	87K
LibrePlan	Project Management / Enterprise	182K
OpenMRS	Health / Medical Sector	140K
Broadleaf Commerce	E-commerce / Retail Sector	207K
Java NATS	Cloud Infrastructure / Messaging Systems	63K

5.1.3 Keyword-Based Search Results

In contrast to the limitations encountered with SCA tools, the Keyword-Based Search approach yielded notably better results in terms of locating and identifying security features across all six analyzed repositories. The process proved to be more targeted and effective, as it focused directly on the presence of security-related constructs based on a continuously evolving list of keywords derived from `SecLists`[1]

[1] `https://github.com/danielmiessler/SecLists`

and corresponding to the security features taxonomy in [6].

On average, the keyword-based method enabled the identification of `111` security features per project, reflecting both the precision and scalability of the approach. The iterative nature of the identification process played a crucial role in these results. As new projects were analyzed, the security keyword list was refined and expanded, incorporating patterns and terms from newly discovered features. This continuous feedback loop enriched the detection process and allowed for increased feature coverage in subsequent projects and also minimized needed time and effort for the results validation process. A detailed breakdown of the findings per project is presented in Table 5.3, illustrating the effectiveness and adaptability of the keyword-based strategy in mining and mapping security features within large-scale software systems.

Table 5.3: Keyword-Based Search Results per Project

Project	Potential Locations	Identified Features
OpenRefine	171	**44**
Traccar	255	**107**
LibrePlan	22	**35**
OpenMRS	257	**143**
Broadleaf Commerce	289	**218**
Java NATS	153	**119**

RQ1:*How can security features be identified and located within the code of software systems?* In response to the first research question, this thesis adopted an iterative and tool-supported methodology to tackle the challenge of identifying and locating security features in large software systems. Given the widely recognized complexity of this task—due to factors such as feature scattering, lack of standardized documentation, and high false-positive rates in automated tools—the study tested and refined multiple techniques including static analyzers, keyword-based search, and embedded annotations. Through continuous experimentation and adaptation, the methodology achieved comprehensive and precise feature identification. As demonstrated in the results in Section 5.1.1 and Section 5.1.3, this iterative approach was effective in overcoming traditional limitations and enabled systematic exploration and annotation of security features at scale.

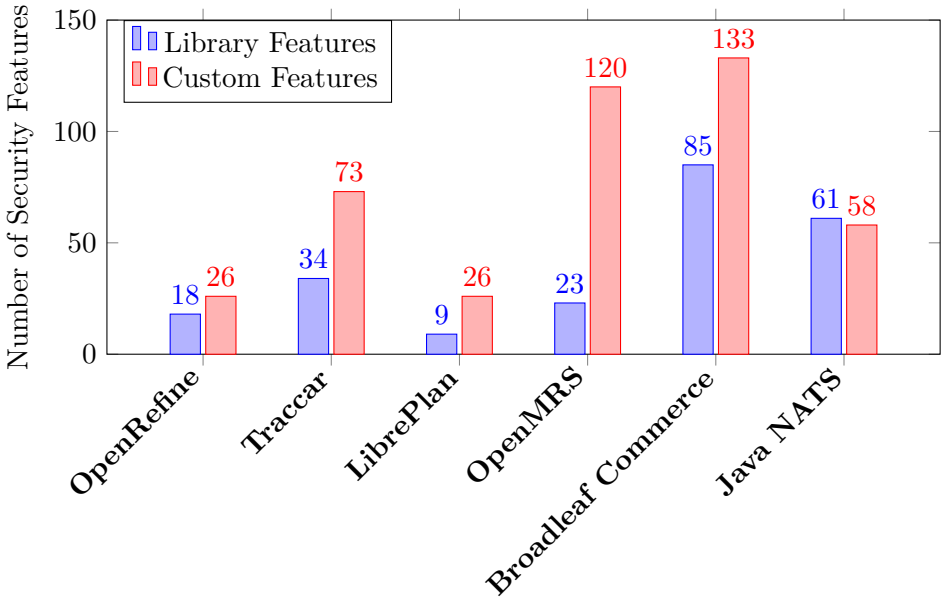
5.2 Categorizing Security Features (RQ2)

After the validation phase confirmed the existence of security features at many of the locations identified by the Keyword-Based Search, each feature was then annotated

using the HAnS annotation format. This annotation process went beyond simply assigning a descriptive name to the feature; it also involved assigning resource-based labels to indicate whether the feature was custom-developed by the project's developers or imported from an external security framework or library. This dual labeling was essential to support both the taxonomy-based classification and the source-based analysis of security features. Once labeled, the validated features were systematically categorized into their appropriate category and subcategory based on the adopted security feature taxonomy, and recorded in the .feature-model file. This structure provided a clear and traceable representation of security feature distribution across projects.

As shown in Table 5.1 and Table 5.4, the number of security features implemented via external libraries versus those custom-developed is presented in detail both collectively on project level and on category and subcategory level. In addition, a breakdown of the functional categories of identified features reveals that `Access Control` accounts for nearly half of all detected security features. This highlights its fundamental role and wide applicability in securing software systems. Additionally, approximately `25%` of the identified features fell under the category of `Cryptography`, indicating a positive trend in the adoption of cryptographic mechanisms in practice. However, their presence alone does not guarantee correct or secure implementation, an issue that will be examined further in Section 5.7. The full statistics of feature types and their categorical distribution are presented in Table 5.4.

Figure 5.1: Comparison of Library vs Custom Features Identified per Project



RQ2: ***What security features exist in a software system?*** In response to the second research question, the study employed the security features taxonomy by Hermann et al. [6] as a foundation for structured feature categorization. Security keywords were mapped to defined categories and subcategories, ensuring that each identified feature could be systematically classified. The taxonomy enabled uniformity in annotation and allowed for consistent analysis across diverse projects. Moreover, the iterative improvement of the keyword database—driven by manual validation feedback—further enhanced the completeness and accuracy of the categorization effort (see Section 2.6 and Section 3.3.3.1).

Table 5.4: Categorizing of Identified Features under the Taxonomy across Projects

Category	**Subcategory**	**OpenRefine** L | C | T	**Traccar** L | C | T	**LibrePlan** L | C | T	**OpenMRS** L | C | T	**Broadleaf Commerce** L | C | T	**Java NATS** L | C | T	**Total**	**CL Security Features**
Access Control	Authentication	1 19 20	3 20 23	3 16 19	2 32 34	29 61 90	0 6 6	38 154 **192**	**28**
	Authorization	0 0 0	0 26 26	1 5 6	0 37 37	11 36 47	0 4 4	12 108 **120**	**11**
Cryptography	Encryption	3 3 6	1 1 2	0 0 0	3 4 7	2 2 4	3 3 6	12 13 **25**	**13**
	Key Management	0 0 0	13 2 15	0 0 0	2 2 4	1 0 1	6 13 19	22 17 **39**	**17**
	Signature	0 0 0	6 4 10	0 0 0	0 0 0	0 0 0	5 2 7	11 6 **17**	**6**
	Hashing	1 1 2	0 8 8	3 0 3	3 3 6	4 1 5	6 4 10	17 17 **34**	**17**
	Steganography	1 0 1	0 0 0	0 0 0	5 5 10	2 2 4	9 18 27	17 25 **42**	**25**
Security Monitoring	Logging	0 0 0	0 0 0	0 0 0	0 0 0	3 0 3	0 0 0	3 0 **3**	**0**
	Automated Response	0 0 0	0 0 0	0 0 0	0 5 5	0 0 0	0 0 0	0 5 **5**	**3**
	History Maintenance	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 **0**	**0**
Secure Data Handling	Data Validation	0 1 1	0 0 0	2 0 2	0 18 18	6 3 9	0 0 0	8 22 **30**	**3**
	Data Sanitization	8 2 10	0 0 0	0 0 0	2 1 3	6 1 7	0 0 0	16 4 **20**	**4**
	Retention Control	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 **0**	**0**
	Secure Storage	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 **0**	**0**
	Trusted Sources	2 0 2	2 0 2	1 2 3	2 3 5	2 2 4	3 1 4	12 8 **20**	**8**
System State Protection	Resource Management	1 0 1	0 0 0	0 0 0	0 0 0	5 1 6	0 0 0	6 1 **7**	**0**
	System State Validation	0 0 0	4 5 9	0 0 0	0 0 0	0 18 18	0 0 0	4 23 **27**	**1**
	Session Management	0 0 0	4 0 4	0 2 2	1 10 11	12 0 12	0 0 0	17 12 **29**	**12**
	State Synchronization	0 0 0	0 0 0	1 0 1	3 0 3	2 5 7	28 0 28	34 5 **39**	**5**
	Miscellaneous	1 0 1	5 3 8	0 0 0	0 0 0	0 0 0	2 6 8	6 11 **17**	**11**

Note: L | C | T = Library Security Features | Custom Security Features | Total
CL Security Features = Custom Security Features based on Library Features in Implementation

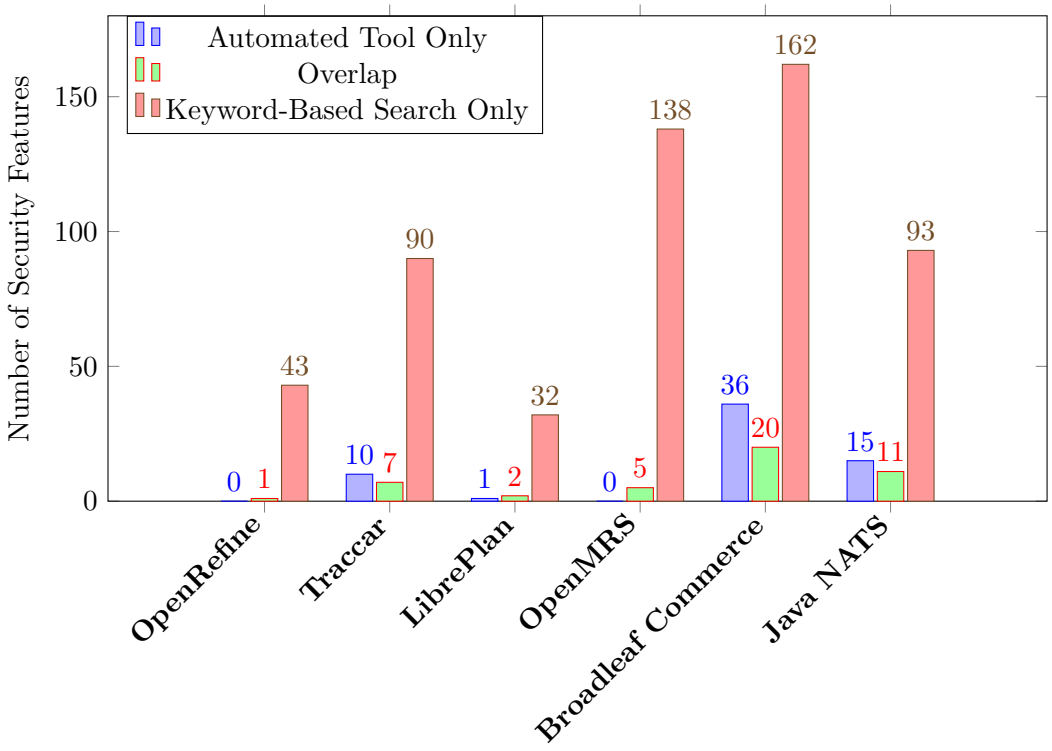
In this study, the term **"Custom Security Features"** would intuitively refer to security features developed from scratch by the developers, without relying on predefined APIs from external libraries or frameworks. However, it is important to note that a subset of these features—though labeled as custom—are architecturally built on top of external library functionalities. These wrapper features encapsulate or extend existing library-based operations while offering customized logic or domain-specific behavior. The number of such features is listed in the final column of Table 5.4 under "CL Security Features." A closer inspection reveals that cryptographic categories, particularly Encryption and Key Management, contain a high number of these library-based custom wrappers, suggesting that developers often rely on well-established primitives while customizing the integration. Similarly,

many Authentication features follow the same pattern, where the developer wraps authentication mechanisms provided by frameworks such as Spring Security. This hybrid construction emphasizes both reliance on established security tooling and the need for customization to meet application-specific requirements (see Section 5.5.1 for further analysis of this pattern).

5.3 Security Library API Detection Tool Results (RQ3)

One of the most significant advantages of using the Automated Features Locating Tool was the elimination of the need for manual validation of the results. Since the tool relies on matching known API calls from established security frameworks and libraries, all identified locations could be considered reliable by design. The only required follow-up was to classify each identified feature under the appropriate category and subcategory of the security feature taxonomy, thereby streamlining the overall annotation and classification process.

Figure 5.2: Comparison of Feature Detection Sources per Project



An interesting observation was the overlap between this tool and the keyword-based search approach. On average, `8` security features per project were commonly identified by both methods, confirming consistency in those specific detections. However, the automated tool uniquely identified an average of `10` security features per project that were not captured by the keyword-based approach. These exclusive findings demonstrate the complementary role that the automated tool can play in expanding feature coverage. The full breakdown of results per project is presented in Figure 5.2.

5.4 Security Library API Detection Tool Evaluation

While the tool performed well in locating relevant security-related components, it is important to note that many of the detected elements were low-level features identified by the automated tool. These typically represent small helper components or individual operations imported from external security libraries and frameworks. In contrast, higher-level features such as `sign` in example in figure 5.3, which encapsulate a complete security functionality, were identified semi-automatically through the keyword-based search. This distinction illustrates the different granularity levels considered by each tool: **the automated tool** excels at uncovering fine-grained, reusable building blocks, while **the keyword-based** search is better suited for capturing broader, semantically complete security features. As such, the tools complement each other effectively, ensuring a more comprehensive identification of security features. An example of this granularity difference is illustrated in Figure 5.3.

From a usability perspective, the tool was straightforward and efficient to operate. However, one limitation was its inability to directly annotate the identified features within the codebase. Instead, results were provided in JSON format, requiring additional integration into the main script for automatic annotation using the HAnS format.

Despite this limitation, the tool demonstrated strong scalability and extensibility. During the study, several security-related libraries and frameworks not originally included in the tool's detection scope were discovered through the keyword-based search. These were then manually added to the tool's internal search database, enhancing its capabilities for future iterations or new projects. The list of additional libraries and frameworks identified during this process is presented in Table 5.5, while those already covered by the tool are listed in Table 4.1. This adaptability makes the tool particularly useful for ongoing or large-scale mining studies, where updating and expanding coverage is crucial to maintaining relevance and accuracy. And also to give a glimpse about the number of security features that were identified from these frameworks and libraries, we can see these numbers per project in Table 5.6.

These numbers show how including more frameworks and libraries to this tool can increase the number of automatically identified features rapidly and in an efficient way. After personally using this tool, I would argue that it would be so much better and beneficial, if it can could annotate the identified features instantly, and if also adding a newly discovered library or a framework that include security features could be possible with just typing the name or the link of it.

```
// &begin[sign]
public byte[] sign(byte[] input) throws GeneralSecurityException, IOException {  Scott Fauerbach *
    Ed25519PrivateKeyParameters privateKey = new Ed25519PrivateKeyParameters(getKeyPair().getPrivate().getEncoded());
    Ed25519Signer signer = new Ed25519Signer(); // &line[Signature_Ed25519Signer_L]
    signer.init( b: true, privateKey); // &line[Signature_init_L]
    signer.update(input,  i: 0, input.length); // &line[Signature_update_L]
    return signer.generateSignature(); // &line[Signature_generateSignature_L]
}
// &end[sign]
```

Figure 5.3: High Level and Low Level Features Example

Table 5.5: List of Frameworks and Libraries Discovered by the Keyword-Based Search

Framework/Library	Used Feature Categories	Reference
Apache Lucene	Data Sanitization	https://lucene.apache.org/
Apache Commons	Steganography Data Sanitization Trusted Sources	https://commons.apache.org/
Java.Net	Steganography	https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html
Java.Util	Hashing Steganography State Synchronization	https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html
Jakarta Servlet	Session Management	https://jakarta.ee/specifications/servlet/6.1/
Jakarta RESTful Web Services	Miscellaneous	https://jakarta.ee/specifications/restful-ws/3.0/apidocs/
OWASP AntiSamy	Data Validation	https://github.com/nahsra/antisamy
GoogleAuth	Authentication	https://github.com/wstrange/GoogleAuth
Jasypt	Authentication Encryption	http://www.jasypt.org/
javax.mail	Authentication	https://javaee.github.io/javamail/
javax.crypto	Encryption Key Management	https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html
javax.servlet.http	Authentication	https://docs.oracle.com/javaee/7/api/javax/servlet/http/package-summary.html
javax.net.ssl	Miscellaneous	https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/package-summary.html
javax.validation	Data Validation	https://docs.oracle.com/javaee/7/api/javax/validation/package-summary.html

Table 5.6: Security Features identified from Frameworks and Libraries in Table 5.5

Project	Identified Security Features per Project
OpenRefine	15
Traccar	15
LibrePlan	9
OpenMRS	18
Broadleaf Commerce	26
Java NATS	38

RQ3: ***What security features are custom implemented by developers, and What are integrated from security frameworks and libraries?*** The third research question was explored by introducing a resource-based classification of features, distinguishing between custom-developed and library-imported implementations. A combination of automated tools and keyword-based methods was used to perform this classification, followed by automatic annotation using the HAnS fomat. As discussed in Section 4.2, many of the features were found to wrap or extend imported functionalities, particularly in cryptographic and authentication domains. Table 5.1 summarizes the split between feature sources, and the results challenge assumptions in prior work by showing that custom-developed features still play a significant role despite the availability of reusable security libraries[35].

5.5 Security Features Distribution Evaluation (RQ4)

To better understand how security features are structurally integrated within software systems, this study evaluates their distribution characteristics using three key metrics: Scattering Degree, Tangling Degree, and Nesting Depth. These metrics were computed for each identified feature across all analyzed projects, and the average values per project are reported in Table 5.7.

The **Scattering Degree** measures the number of distinct files in which a security feature is implemented or used. Higher values indicate that a feature is more dispersed across the system, potentially increasing maintenance complexity and reducing traceability. For example, Session Management shows notably high scattering degrees (averaging `8.06`), especially in `LibrePlan` and `Broadleaf Commerce`, suggesting widespread usage across multiple files.

Tangling Degree represents how interwoven a security feature is with other functionalities within the same file. A higher tangling degree indicates greater complexity and potential difficulty in isolating and maintaining security logic. Notably, features such as Key Management (`2.67`) and Signature (`2.98`) under Cryptography exhibit higher tangling degrees, indicating their close integration with other functionalities.

Nesting Depth refers to the average structural nesting level, such as within control statements, method calls, or class hierarchies, indicating integration complexity. A higher nesting depth can reduce readability and maintainability. Features such as Steganography (`1.85`) and Data Sanitization (`3.18`) display higher average nesting depths, reflecting potentially deeper structural integration within the system.

Table 5.7: Security Features Distribution Aspects per Subcategory / Project

Category	Subcategory	OpenRefine S | T | N	Traccar S | T | N	LibrePlan S | T | N	OpenMRS S | T | N	Broadleaf Commerce S | T | N	Java NATS S | T | N	Avg.	Total	
Access Control	Authentication	1 1.35 1.6	1.43 0.67 1.21	1.17 1.44 1.5	1.65 0.69 1.33	2.5 1.15 1.34	1 0 1	**1.29**	**0.93**	**1.33**
	Authorization	0 0 0	2.38 0.33 1.05	1 0.43 1.14	2.33 0.4 1.06	2 0.6 1.26	1 0 1	**1.74**	**0.35**	**1.14**
Cryptography	Encryption	1 2.67 2.17	1 1.5 1.5	0 0 0	1.14 1 1.43	2 0 1	2 1 1	**1.33**	**1.44**	**1.52**
	Key Management	0 0 0	1.07 5 3.27	0 0 0	1.33 1 1.67	1 0 1	1 2 3	**1.10**	**2.67**	**2.32**
	Signature	0 0 0	1.1 2.6 2.9	0 0 0	0 0 0	0 0 0	3 4.33 1.38	**1.37**	**2.98**	**2.29**
	Hashing	1.5 1 1.5	1.5 3.25 2	1 0 1	4.5 1 1.52	3.5 1 1.6	2.75 2 1.75	**2.46**	**1.20**	**1.47**
	Steganography	1 2 3	0 0 0	0 0 0	1.33 1.11 1.5	2.5 0.5 1.08	1.83 1.33 1.82	**1.53**	**1.39**	**1.85**
Security Monitoring	Logging	0 0 0	0 0 0	0 0 0	0 0 0	1 0 1	0 0 0	**1**	**0**	**1**
	Automated Response	0 0 0	0 0 0	0 0 0	1 0 1	0 0 0	0 0 0	**1**	**0**	**1**
	History Maintenance	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	**0**	**0**	**0**
Secure Data Handling	Data Validation	1 0 1	0 0 0	2 0 1	4.27 0.13 1	6 3 9	0 0 0	**2.65**	**0.63**	**2.60**
	Data Sanitization	1.2 0 1.1	0 0 0	0 0 0	1.67 1 1.44	6 1 7	0 0 0	**2.72**	**0.67**	**3.18**
	Retention Control	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	**0**	**0**	**0**
	Secure Storage	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0	**0**	**0**	**0**
	Trusted Sources	1 1 2	1.5 3.5 3.25	1 2 2.67	1.25 1 1.5	2 1 1.33	2.5 1.5 1.25	**1.54**	**1.67**	**1.83**
System State Protection	Resource Management	1 0 1	0 0 0	0 0 0	0 0 0	1 0 1	0 0 0	**1**	**0**	**1**
	System State Validation	0 0 0	7.38 0.25 1.03	0 0 0	0 0 0	2 1 1	0 0 0	**4.69**	**0.63**	**1.02**
	Session Management	0 0 0	1.75 0.5 1.12	22 0.5 1.02	4.22 0.44 1.02	8.33 2.33 1.57	0 0 0	**8.06**	**0.96**	**1.18**
	State Synchronization	0 0 0	0 0 0	1 0 1	2.33 0 1	1.5 0.5 1.5	1.33 0 1	**1.61**	**0.25**	**1.13**
	Miscellaneous	1 0 1	1.25 1.88 1.5	0 0 0	0 0 0	0 0 0	2 1 1.25	**1.42**	**0.96**	**1.08**

Note: S | T | N = Scattering Degree | Tangling Degree | Avg. Nesting Depth

RQ4:*How are security features distributed across the software architecture, and what patterns can be observed in their distribution?* To address the fourth research question, this thesis conducted an architectural-level analysis based on two distribution metrics: *Scattering Degree* and *Nesting Depth*. These metrics were computed for each feature across the six analyzed repositories using data from HAnS annotations. As presented in Table 5.3, the findings confirmed that security features are often scattered and structurally embedded, reinforcing earlier and recent claims in literature about the cross-cutting and decentralized nature of security logic [6, 35]. These distribution characteristics complicate traceability and justify the need for structured annotations and architectural awareness during development and maintenance.

5.5.1 Insights on Modular Design

During the manual validation and classification of security features, two key observations emerged regarding the modularity and structural organization of security-relevant code within the analyzed projects.

The first and most recurring pattern involved the relationship between custom security features and security libraries. In numerous instances, the core logic of a custom-developed security feature was found to wrap around or extend a standard

functionality provided by an external library or framework. This shows how developers often rely on well-established libraries to fulfill the cryptographic or authentication logic, while building an application-specific security interface or feature layer on top of it. This modular reuse of security primitives aligns with previous research observations, such as those by Acar et al. [21], which argue that developers often integrate third-party security APIs without fully understanding or encapsulating their implications, potentially risking misuse when abstraction is handled incorrectly. A representative example of this wrapping pattern is shown in Figure 5.4, where cryptographic API calls are encapsulated within application-level feature implementations.

```
@Override
        //&begin[decrypt]
public String decrypt(String encryptedMessage) {
    return this.encryptor.decrypt(encryptedMessage); //&line[encryptor_decrypt]
}
      //&end[decrypt]

@Override
        //&begin[encrypt]
public String encrypt(String message) {
    return this.encryptor.encrypt(message); //&line[encryptor_encrypt]
}
        //&end[encrypt]
```

Figure 5.4: Observed Pattern Example

Table 5.8: Observed Pattern Occurrences per Project

Project	#Occurrences per Project
OpenRefine	5
Traccar	25
LibrePlan	3
OpenMRS	16
Broadleaf Commerce	28
Java NATS	29

The second observation concerns the organizational structure of security features within the project codebases. In most repositories analyzed, security-related functionality was scattered across multiple unrelated locations, confirming the natural dispersion and low cohesion of security features noted in prior studies. However, a notable exception was observed in the `Broadleaf Commerce` project, where a clear modular structure was adopted. This project included a dedicated security subdirectory in various parts of the codebase, and most security features identified in the study were implemented in files contained within those directories. This structured modularity not only enhances security traceability but also reflects principles

of security-by-design, promoting improved maintainability and auditability of sensitive features. An example of this directory structure is presented in Figure 5.5. These insights emphasize the importance of architectural decisions and modular design patterns in enhancing the visibility, traceability, and reliability of security features in large-scale software systems.

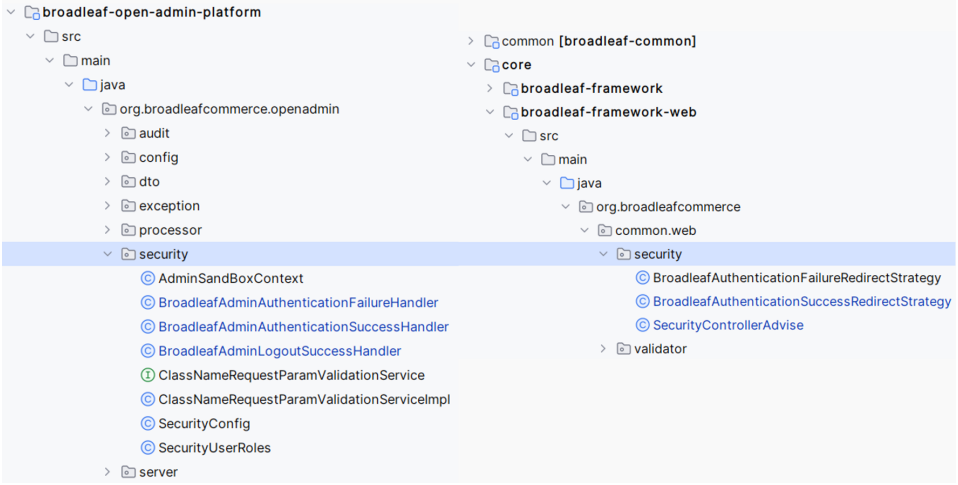


Figure 5.5: Security Modular Structure Example

5.6 Insights on Context of Use

During the semantic validation and classification of security features, it was observed that different categories of features tend to appear in distinct usage contexts within the software systems. Most notably, Authentication features were predominantly utilized in login-related contexts, serving as entry points for user verification across the analyzed projects. In contrast, Cryptographic features exhibited a wider contextual distribution, supporting use cases such as message encryption, signature verification, and secure exchange of cryptographic parameters. This diverse utilization highlights the foundational role cryptography plays in securing various components of a system. Furthermore, Session Management features were primarily employed in web-related functionalities, governing user session behavior, particularly in scenarios involving session creation, expiration, or validation.

5.7 Comparison with Best Practices and Standards (RQ5)

To assess the quality and security robustness of identified features, a comparative evaluation was conducted between implemented security features and established

best practices and OWASP standards. In this section, we show an analysis example of the `loginCustomer` method, a method identified in the `Broadleaf Commerce` project, which represents a core authentication feature within the system. The evaluation was carried out with reference to the OWASP Top 10 (2021), focusing primarily on `A07: Identification and Authentication Failures, A05: Security Misconfiguration, and A09: Security Logging and Monitoring Failures`.

The `loginCustomer` method implements user login functionality using Spring Security components, such as `UserDetailsService, UsernamePasswordAuthenticatio -nToken, AuthenticationManager`, and `SecurityContextHolder`. As shown in Figure 5.6, the method follows the architectural conventions of Spring Security, delegating authentication logic to the framework's built-in components and appropriately setting the security context upon successful login.

```
@Override  2 usages   ± Brian Polster +2 *
        // &begin[loginCustomer]
public Authentication loginCustomer(String username, String clearTextPassword) {
    UserDetails principal = userDetailsService.loadUserByUsername(username); // &line[SessionManagement_loadUserByUsername_L]
    UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthenticationToken( // &line[Authentication_UsernamePasswordAuthenticationToken_L]
            principal, clearTextPassword, principal.getAuthorities() // &line[SessionManagement_getAuthorities_L]
    );
    Authentication authentication = authenticationManager.authenticate(token); // &line[Authentication_authenticate_L]
    SecurityContextHolder.getContext().setAuthentication(authentication); // &line[SessionManagement_getContext_L, SessionManagement_setAuthentication_L]
    customerStateRequestProcessor.process(getWebRequest());
    cartStateRequestProcessor.process(getWebRequest());
    return authentication;
}
// &end[loginCustomer]
```

Figure 5.6: `loginCustomer` Feature

From a best practices perspective, the feature exhibits clear strengths. It leverages Spring Security's abstraction layers for authentication, reducing the likelihood of insecure custom logic. It also ensures that authenticated sessions are correctly registered via the security context. These choices align well with modularity principles and security-by-design patterns. However, the implementation reveals several deviations from OWASP recommendations, as outlined in Table 5.9. The method handles plaintext passwords without any visible in-memory protection, which increases the risk of sensitive data leakage. Additionally, there is no observable mechanism for brute-force protection, such as login attempt throttling or account lockout, and the absence of multi-factor authentication (MFA) significantly reduces the feature's resilience against common attack vectors. Further weaknesses include the lack of structured logging, error handling, and no clear indication of Cross-Site Request Forgery (CSRF) protection or session expiration enforcement.

Table 5.9: Security Concerns and OWASP Violations in the `loginCustomer` Method

Issue	Description	OWASP Reference	Recommendation
Plaintext Password Handling	Password is stored and passed as a `String`, exposing it in memory longer than needed.	**A07:2021** – Identification and Authentication Failures	Use `char[]` instead of `String` and clear it after use. Minimize memory exposure.
No Brute-force Protection	No throttling or account lockout mechanism for repeated failed login attempts.	**A07:2021** – Identification and Authentication Failures	Implement login attempt tracking, rate limiting, and temporary lockouts.
Lack of Logging	No indication that login attempts (success/failure) are being logged for monitoring.	**A09:2021** – Security Logging and Monitoring Failures	Securely log authentication attempts (without passwords) for audit and detection.
Missing Multi-Factor Authentication (MFA)	Only username and password are used for authentication.	**A07:2021** – Identification and Authentication Failures	Integrate MFA such as TOTP, SMS-based OTP, or authenticator apps for additional security.
No Error Handling	No try-catch block is present for handling authentication failures.	**A07:2021** – Identification and Authentication Failures	Handle exceptions securely and provide generic messages like "Invalid credentials."
Unclear CSRF/Session Handling	Session expiration, CSRF protection, and session management aren't visible in this method.	**A05:2021** – Security Misconfiguration	Ensure CSRF tokens are used and sessions are invalidated on logout or timeout.

To further respond to RQ5, a comparative evaluation was also conducted on the other login-related authentication features implemented in the other five analyzed projects.The analysis revealed a wide spectrum of design choices and security practices:

OpenRefine implements its own login logic, managing cookies and credentials manually. While CSRF token checks are present, there is no observable error handling, MFA, or brute-force protection. Sensitive credential data is handled in plaintext strings and persisted in cookies, raising significant concerns regarding memory safety and session misuse.

Traccar demonstrates a more mature design. It includes multi-factor authentication through TOTP, supports LDAP integration, and contains explicit checks for user state and session token validation. However, password memory safety and log-

ging mechanisms are not clearly visible in the implementation.

LibrePlan delegates authentication to Spring Security using UsernamePasswordAuthenticationToken and AuthenticationManager, offering a clean and modular design. However, due to abstraction, auxiliary security features such as brute-force defense and MFA are not assessable at the code level and depend on the broader framework configuration.

OpenMRS performs contextual authentication with custom session notifications. Although the method provides structured exception handling and user session tracking, it lacks MFA and logging of failed login attempts, and password protection mechanisms are not clearly observable.

Java NATS does not employ conventional username/password authentication. Instead, it relies on cryptographic key-based identity and challenge-response verification using Ed25519 keys. This method is robust against traditional login vulnerabilities (e.g., brute-force attacks) but also places the burden of secure key handling and storage on the user.

In summary, while all systems implement authentication features, only **Traccar** and **Java NATS** exhibit stronger adherence to OWASP recommendations. Most implementations—especially those relying on custom or semi-custom code—lack comprehensive protective measures such as MFA, secure credential storage, and brute-force mitigation. This comparison reinforces a key insight from the study: adopting security frameworks alone is not sufficient unless accompanied by conscious integration of holistic security practices around the authentication logic.

RQ5:*How secure are security features implemented, and do they follow best practices and security standards?* To answer the fifth research question, a comparative analysis revealed that while most projects rely on established frameworks, their implementations often lack essential protections such as brute-force defense, MFA, and secure password handling. This confirms that using a security framework alone does not ensure adherence to best practices or known security standards.

6 Discussion

6.1 Implications for Software Security

This study underscores the critical role that security features play in the broader context of software security. By investigating how security features are implemented, distributed, and maintained in real-world software systems, this work contributes to a deeper understanding of their practical nature and sheds light on current limitations in their identification and traceability.

The methodologies proposed and evaluated in this study aim to improve the process of locating and tracking security features in size-variant and complex codebases. In doing so, they address one of the core challenges faced by practitioners: the ability to rapidly detect, analyze, and patch security features in response to vulnerabilities. Since effective remediation is contingent upon accurate identification, it becomes essential to develop time-efficient and resource-conscious techniques that reduce the cognitive and operational overhead for developers. This study emphasizes that one cannot fix what cannot be found—hence, the importance of refined, accurate feature identification tools and processes.

Moreover, the categorization of security features based on a formal taxonomy enables a clearer understanding of the varying levels of criticality and sensitivity associated with different features. Such classification is instrumental in identifying which features directly safeguard sensitive data or enforce core security goals—confidentiality, integrity, and availability—and therefore demand more rigorous scrutiny. The analysis of feature distribution within software architectures also offers key insights into how security features are embedded and whether their placement supports or hinders security-by-design. By studying their structural dispersion, developers and architects can evaluate whether certain distribution patterns correlate with greater robustness or, conversely, introduce maintainability and traceability challenges.

Finally, the resource-based classification—differentiating between custom-developed and library-based security features—provides a lens through which to interpret developer practices and preferences in implementing security. This distinction not only reflects varying levels of security expertise but also highlights the need to enhance the usability of security libraries and frameworks. It further supports ongoing efforts

to raise awareness among developers regarding secure coding practices and the importance of understanding the implications of security feature integration, whether manually or through third-party.

6.2 Implications on Related Work

Interestingly, contrary to assumptions made in prior research—which suggest that developers primarily rely on external libraries for implementing security [35]—our findings show a more balanced or even custom-heavy usage of security features. However, due to the relatively small input set in this study, such observations must be interpreted cautiously and should not be generalized beyond the current sample.

The findings and discovered metrics shown in 5.5 provide quantitative evidence supporting the challenges of managing security features in practice. The observed distribution patterns align with prior research findings, such as those by Hermann et al. [6, 35] and Santos et al. [15], which emphasize that security functionality is often not localized, but rather scattered and deeply embedded across multiple layers of software systems. These findings highlight the importance of traceability tools and structured annotations, such as those employed in this study, for maintaining security-relevant code and understanding its broader architectural impact.

6.3 Contribution Limitations

While this study offers valuable insights into the identification and analysis of security features in software systems, several limitations must be acknowledged. First, the scope of this research was limited to software projects written in Java. Java was selected due to its widespread adoption in the software engineering community and its strong presence in large-scale systems. However, the language-specific nature of the analysis implies that the findings may not directly generalize to software developed in other programming languages. Languages such as `Python, C++`, or `JavaScript` differ in syntax, paradigms, and security practices, which may influence or affects how security features are implemented and distributed.

6.4 Validity Threats

6.4.1 Construct Validity

Construct validity concerns whether the study accurately measures what it intends to measure. In this thesis, the central construct is the notion of "security features" and their traceability within software systems. One threat to construct validity lies in the reliance on security keywords to represent security features. While grounded in a well-defined taxonomy and supported by a curated keyword list, this approach may still overlook security features that are implemented in non-standard or highly abstract ways. Moreover, using a script-based and annotation-driven methodology risks missing context-sensitive implementations that cannot be detected through static analysis alone.

6.4.2 Internal Validity

As I am a student with a background in IT-Security, this may have introduced bias during the manual validation process. Overconfidence in identifying and classifying security features could have led to subjective decisions, particularly in borderline cases or ambiguous code segments. While efforts were made to remain consistent and systematic, this human factor cannot be entirely excluded.

6.4.3 External Validity

External validity addresses the extent to which findings can be generalized beyond the studied cases. The vast size and scale of modern software systems pose a significant challenge in this thesis, as they complicate the identification, analysis, and evaluation of security features. It also makes it hard to estimate the needed volume of input to derive clear and meaningful findings. With millions of lines of code and complex interdependencies, the dispersion of security measures across large systems further complicates feature identification. Additionally, the study is limited to Java-based repositories and a relatively small input set of 6 projects. While these were selected to vary in size, complexity, and domain, the small sample may not fully represent the diversity of software practices, reducing the generalizability of the findings to other programming languages or ecosystems.

6.5 Future Work

6.5.1 Continuous Improvements

One of the primary avenues for future work involves the continued refinement of the tools and datasets used in this study. By analyzing a larger and more diverse set of software systems, the security keyword list can be further expanded and validated, improving its effectiveness in capturing real-world implementations. Likewise, the security library API detection tool should be iteratively extended with newly discovered frameworks and libraries. These updates aim to maximize the success rate of identifying and locating security features while minimizing false positives, ultimately enhancing the robustness and reliability of the mining process.

6.5.2 Integrating ML Models

Another potential direction is the integration of machine learning techniques. If a sufficiently large dataset of annotated software repositories with their identified security features can be compiled, it could serve as training data for machine learning models. These models could then automatically identify and locate both custom-developed and library-imported security features. Furthermore, such models could be trained to classify features according to predefined taxonomies and even evaluate the quality of implementations, offering recommendations or flagging weak security patterns for developer attention.

6.5.3 Using LLM Models

A forward-looking idea involves leveraging large language models (LLMs) trained on a curated dataset of securely implemented security features. Such a model could support developers during development by suggesting complete, secure templates for specific security features, which can be easily adapted to project-specific needs. Additionally, it could offer context-aware recommendations and best practices, helping developers implement robust security features even without deep security expertise. This approach has the potential to democratize access to secure development practices and significantly improve software security outcomes across diverse development environments.

7 Conclusion

This thesis presented a structured investigation into the practical implementation and distribution of security features within real-world Java software systems. By employing a combination of SCA tools, keyword-based searches, and structured annotation methods, security features were systematically identified, categorized, and evaluated, providing a comprehensive overview of their nature within software systems codebases.

Security features were located and identified through a methodology combining security libraries API detection tool and keyword-based search tool, resulting in the automated detection of **230** security features imported from libraries and frameworks and **436** custom-implemented features across six Java open-source projects. Notably, the analysis revealed that a portion of security logic (**144** custom-implemented features identified) extends or wraps existing functionalities from established security libraries, especially in cryptographic (e.g., Encryption, Signature) and authentication domains.

Evaluating the structural integration and robustness of security features revealed distinct distribution and implementation patterns. Metrics such as Scattering Degree, Tangling Degree, and Nesting Depth showed that features like Session Management and Cryptography tend to be widely dispersed and deeply embedded, confirming the cross-cutting nature of security logic that complicates maintenance and validation. A complementary assessment aligned with OWASP standards highlighted significant variance in implementation quality, indicating that relying solely on security frameworks does not inherently guarantee adherence to best practices.

In conclusion, this thesis provides actionable insights and tools that enhance the understanding, traceability, and security robustness of security features implemented within software systems. It highlights the necessity of holistic security practices that complement library usage with comprehensive integration strategies. Future research should focus on refining automated detection methods by expanding the scope of supported security libraries, incorporating machine learning to enhance accuracy and efficiency, and extending the approach to diverse programming environments. Such advances promise further improvements in secure software development practices and more secure software systems.

List of Figures

Bibliography

[1] Gary McGraw. “Software security”. In: *IEEE Security & Privacy* 2.2 (2004), pp. 80–83.

[2] Amir A Khwaja, Muniba Murtaza, and Hafiz F Ahmed. “A security feature framework for programming languages to minimize application layer vulnerabilities”. In: *Security and Privacy* 3.1 (2020), e95.

[3] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. “SNIAFL: Towards a static noninteractive approach to feature location”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15.2 (2006), pp. 195–226.

[4] Sven Apel and Christian Kästner. “An overview of feature-oriented software development.” In: *J. Object Technol.* 8.5 (2009), pp. 49–84.

[5] Julia Rubin and Marsha Chechik. “A survey of feature location techniques”. In: *Domain Engineering: Product Lines, Languages, and Conceptual Models* (2013), pp. 29–58.

[6] K. Hermann, S. Schneider, C. Tony, A. Yardim, S. Peldszus, T. Berger, R. Scandariato, A. Sasse, and A. Naiakshina. “A Study of Functional Security Features and Their Representation in Security Frameworks”. In: (2024).

[7] Katja Tuma, Sven Peldszus, Daniel Strüber, Riccardo Scandariato, and Jan Jürjens. “Checking security compliance between models and code”. In: *Software and systems modeling* 22.1 (2023), pp. 273–296.

[8] S. Peldszus, F. Reiche, K. Hermann, S. Corallo, T. Berger, and R. Heinrich. “Can I Check What I Designed? Mapping Security Design DSLs to Code Analyzers”. In: (2024).

[9] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. “What is a feature? a qualitative study of features in industrial software product lines”. In: *Proceedings of the 19th international conference on software product line*. 2015, pp. 16–25.

[10] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.

[11] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. “Scaling step-wise refinement”. In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 355–371.

[12] Kun Chen, Wei Zhang, Haiyan Zhao, and Hong Mei. "An approach to constructing feature models based on requirements clustering". In: *13th IEEE International Conference on Requirements Engineering (RE'05)*. IEEE. 2005, pp. 31–40.

[13] KYOC KANG, SHOLOMG COHEN, JAMESA HESS, WILLIAME NOVAK, and AS PETERSON. "Feature-Oriented Domain Analysis(FODA) feasibility study(Final Report)". In: (1990).

[14] Matthias Riebisch. "Towards a more precise definition of feature models". In: *Modelling variability for object-oriented product lines* (2003), pp. 64–76.

[15] Joanna CS Santos, Katy Tarrit, and Mehdi Mirakhorli. "A catalog of security architecture weaknesses". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 220–223.

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

[17] John M Blythe, Nissy Sombatruang, and Shane D Johnson. "What security features and crime prevention advice is communicated in consumer IoT device manuals and support pages?" In: *Journal of Cybersecurity* 5.1 (2019), tyz005.

[18] Rémy Druyer, Lionel Torres, Pascal Benoit, Paul-Vincent Bonzom, and Patrick Le-Quéré. "A survey on security features in modern FPGAs". In: *2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE. 2015, pp. 1–8.

[19] Ariessa Davaindran Lingham, Nelson Tang Kwong Kin, Chen Wan Jing, Chong Heng Loong, et al. "Implementation of Security Features in Software Development Phases". In: *arXiv preprint arXiv:2012.13108* (2020).

[20] Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, and Jan-Philipp Steghöfer. "Featracer: Locating features through assisted traceability". In: *IEEE Transactions on Software Engineering* (2023).

[21] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. "Comparing the usability of cryptographic apis". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 154–171.

[22] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. "Usability Smells: An Analysis of {Developers'} Struggle With Crypto Libraries". In: *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. 2019, pp. 245–257.

[23] Irshad Ahmad Mir and SMK Quadri. "Analysis and evaluating security of component-based software development: A security metrics framework". In: *International Journal of Computer Network and Information Security* 4.11 (2012), p. 21.

[24] Jacob Krüger, Thorsten Berger, and Thomas Leich. “Features and how to find them: a survey of manual feature location”. In: *Software Engineering for Variability Intensive Systems*. Auerbach Publications, 2019, pp. 153–172.

[25] Jacob Krüger, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. “Effects of explicit feature traceability on program comprehension”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 338–349.

[26] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. “Feature-oriented software evolution”. In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*. 2013, pp. 1–8.

[27] Dapeng Liu and Shaochun Xu. “A Tool Suite for Java Program Tracing and Feature Location”. In: *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*. IEEE. 2009, pp. 469–474.

[28] Willian DF Mendonça, Silvia R Vergilio, Gabriela K Michelon, Alexander Egyed, and Wesley KG Assunção. “Test2feature: Feature-based test traceability tool for highly configurable software”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume B*. 2022, pp. 62–65.

[29] Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. “A common notation and tool support for embedded feature annotations”. In: *Proceedings of the 24th acm international systems and software product line conference-volume b*. 2020, pp. 5–8.

[30] Johan Martinson, Herman Jansson, Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho-Quang. “HAnS: IDE-based editing support for embedded feature annotations”. In: New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450384704.

[31] Xiang Li, Jinfu Chen, Zhechao Lin, Lin Zhang, Zibin Wang, Minmin Zhou, and Wanggen Xie. “A mining approach to obtain the software vulnerability characteristics”. In: *2017 fifth international conference on advanced cloud and big data (CBD)*. IEEE. 2017, pp. 296–301.

[32] Andreas Mauczka, Christian Schanes, Florian Fankhauser, Mario Bernhart, and Thomas Grechenig. “Mining security changes in FreeBSD”. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE. 2010, pp. 90–93.

[33] Vinod Ganapathy, David King, Trent Jaeger, and Somesh Jha. “Mining security-sensitive operations in legacy code using concept analysis”. In: *29th International Conference on Software Engineering (ICSE’07)*. IEEE. 2007, pp. 458–467.

[34] Stratoflow. *Why is Java so popular?* Accessed: 2025-02-14. 2023. URL: https://stratoflow.com/why-is-java-so-popular/?utm_source=chatgpt.com.

[35] Kevin Hermann, Sven Peldszus, Jan-Philipp Steghöfer, and Thorsten Berger. "An Exploratory Study on the Engineering of Security Features". In: *arXiv preprint arXiv:2501.11546* (2025).