Design and Evaluation of Security Features in RealSpec Real Time Executable Specification Language

Ph.D Thesis

By

Muniba Murtaza

135-FBAS/PHDCS/F15

Supervisor

Dr. Amir Khwaja

Co-Supervisor

Dr. Humaira Ashraf

Department of Computer Science

Faculty of Computing and IT

International Islamic University, Islamabad

August 2024

[illegible]
K

TH-26764

PhD
005.8
MUD

computer security

Programming language (Electronic computers)

Real-time programming

Computer software. Testing

Computer software - Specification languages.

DEPARTMENT OF COMPUTER SCIENCE

INTERNATIONAL ISLAMIC UNIVERSITY ISLAMABAD

FINAL APPROVAL

Dated: 29th Aug 2024

It is certified that we have read the thesis titled "**Design and Evaluation of Security Features in RealSpec Real Time Executable Specification Language**" submitted by Ms.Muniba Murtaza (135-FBAS/PhD (CS)/F15). It is our conclusion that this thesis is of sufficient standards to warrant its acceptance by the International Islamic University, Islamabad for the PhD Degree in Computer Science.

COMMITTEE

EXTERNAL EXAMINER

Dr Hikmat Ullah Khan
Assistant Professor,COMSATS
Institute Wah Cantt

[illegible]

Dr Basit Shahzad
Associate Professor,
NUML University,Islamabad

[illegible]

INTERNAL EXAMINER

Dr Shireen Tahira
Assistant Professor,
Department of Computer Science
FC&IT, IIU, Islamabad

[illegible]

DEAN

Prof.Dr Asmatullah khan
Professor,
Faculty of Computing & Information Technology
IIU, Islamabad

[illegible]

Dean
Faculty of Computing
International Islamic University
Islamabad (IIUI)

CHAIRMAN

Dr Muhammad Nadeem
Assistant Professor,
Department of Computer Science
FC&IT, IIU, Islamabad

[illegible]

SUPERVISOR & CO-SUPERVISOR

Dr Amir Khawaja
Director of Engineering Qualcomm, United State

for [illegible]

Dr Humaira Ashraf
Assistant Professor,
Department of Computer Science
FC&IT, IIU, Islamabad

for [illegible]

Assistant Professor,
Dept.Computer Science, FC&IT
International Islamic University Islamabad

A dissertation submitted
to the Department of
Computer Science,
International Islamic University, Islamabad
as a partial fulfillment of the requirements for the award of the
degree of Doctor of Philosophy in Computer Science.

Declaration

I hereby declare that this thesis, neither as a whole nor as a part thereof has been copied out from any source. It is further declared that no portion of the work presented in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

Muniba Murtaza
135-FBAS/PHDCS/F15

Dedication

To myparents, my husband, mybrothers, and my children (Manha, Hashim, Maryam, Haider, Gul Meena and Issa)

Acknowledgments

First and foremost, I express my gratitude to the *ALMIGHTY ALLAH* for granting me the opportunity to successfully complete my degree.

I would like to extend my heartfelt appreciation to my supervisor, *Dr. Amir Khwaja*, and co-supervisor, *Dr. Humaira Ashraf*, for their invaluable advice, unwavering support, patience, and guidance throughout the thesis writing process. Their constant encouragement played a significant role in my achievement.

I would also like to acknowledge and extend my gratitude to especially *Dr. Asim Munir, Dr. Umara Zahid, Dr. Muhammad Nadeem* and *Dr. Qammar Abbas* whose valuable feedback and reviews refined the research.

I extend my gratitude to the Dean of FCIT, *Dr. Muhammad Asmat Ullah Khan* for his continuous motivation and support.

I am profoundly grateful to my family—my mother, father, brother, husband, and children—for their unwavering support, understanding, and financial assistance throughout this journey. Their presence and patience, especially during times when I couldn't manage certain household responsibilities, were invaluable. I am especially at a loss for words to express my gratitude to my mother, whose encouragement was crucial in my decision to pursue higher education and who stood by me through every challenge. My thanks also go to my father for his constant motivation during moments when my enthusiasm waned. I sincerely appreciate my brother for his essential support in the documentation process and ongoing guidance. Lastly, I am deeply thankful to my husband, who not only provided financial support but also supplied me with a laptop and technical gadgets, along with offering valuable technical guidance throughout this entire process.

I would like to acknowledge the *Higher Education Commission (HEC)* for providing me with funding through the indigenous *5000 Ph.D. batch II* fellowship program. This opportunity was instrumental in enabling me to pursue my academic goals.

Lastly, I extend my sincere appreciation to the International Islamic University, Islamabad, Pakistan, for providing me with a platform to realize my aspirations and accomplish my academic objectives.

Abstract

The thesis investigates security vulnerabilities in web software, which arise from programming languages underlying vulnerabilities and somehow programmers' lack of security knowledge and the use of insecure libraries. It critiques the "penetrate-and-patch" method due to its high costs and the risk of exploitation before patches are available. To tackle these issues, the thesis introduces two frameworks: SEFF, which assesses the security features of programming languages like C++, C#, Java, Ruby, and Python, and SRFS, which evaluates executable specification languages. SEFF finds C# has the highest security coverage (69.4%) while C++ has the lowest (47.22%). SRFS reveals Secure Descartes with the highest coverage (91.75%) and SysML-Sec with the lowest (32.9%). Overall, SEFF and SRFS provide methods for evaluating the security of programming and specification languages, with RealSpec performing best in terms of security coverage. Following the SRFS, security requirements are delineated using the Real-Time Specification Language (RealSpec), which serves as a representative case. RealSpec functions as an executable specification language commonly utilized in the requirement specification stage of the software development life cycle of embedded systems. Its structure closely resembles that of a high-level programming language, reducing the learning curve for programmers and necessitating fewer abstractions. The validation of these specified security requirements is conducted using a custom model checking tool, followed by testing against attack specifications. This approach ensures security measures are implemented from the initial design phase onward. Additionally, RealSpec is compared against other executable specification languages like SysML- Sec, SecureDescartes, S-Promela, and Ponder using SRFS and the result shows that RealSpec gives more security coverage at 90.625%. The findings indicate that RealSpec offers the most comprehensive coverage of security features.

Contents

Table of Tables

Table of Figures

List of Publications

- **Published Articles in Wiley online**

– A. Khwaja, M. Murtaza, and H. F. Ahmed, "A security feature framework for programming languages to minimize application layer vulnerabilities." *Security andPrivacy*, vol. 3, no. 1, p. 95, 2019, doi: 10.1002/spy2.95.

- **Published Paper**

– Murtaza, Muniba. "S-RealSpec: A Security Extension to Detect SQLI attack and Sensitive Data Exposure." Kurdish Studies 12.5 (2024): 757-769.

- **Accepted paper**

– Muniba Murtaza and Humaira Ashraf, "A Security Requirement Framework for Secure Web Application Development", International Journal of Business Information Systems.June 2023. ISSN 1746-0972-1746-0980. https://doi.org/10.1504/IJBIS.2023.10058319

ANSI	American National Standard Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Location Randomization
ASP	Active Server Pages
CA	Certificate Authority
CFG	Control Flow Graph
CFI	Control Flow Integrity
CRED	C Range Error Detector
DynIMA	Dynamic Integrity Management
HTTPS	Hyper Text Transfer Protocol
IETF	Internet Engineering Task Force
IPC	Inter Process Channel
JAAS	Java Authentication and Authorization Service
JAR	Java Archive
JRE	Java Runtime Environment
LINQ	Language Integrated Query
MDS	Model Driven Security
MDSE	Model-Driven Security Engineering
MVC	Model View Controller
NX-bit	non-executable bit
OAuth	Open Authorization
OpenSSL	Open Secure Socket Layer
OWASP	Open Web Application Software Program
PRNG	pseudo-random number generator
RealSpec	Real-Time Executable Specification Language
ROP	Return Oriented Programming
RPC	Remote Procedure Call
RSA	Rivest, Shamir, and Adi's
SDLC	Software Development Life Cycle
SecureSOA	Secure Service Oriented Architecture
SEFF	Security Feature Framework for Programming Language
SFRS	Security Requirement Framework for Specification Language
SHA	Secure Hashing Algorithm
SID	Session Identity
SQLI	Structured Query Language Injection Attack
SSL	Secure Socket Layer
SSO	Single Sign On
SysML-Sec	System Modeling Language-Security
TLS	Transport Layer Security
TOCTOU	Time-Of-Clock to Time-Of-Use attack
UML-Sec	Unified Modeling Language-Security
XSS	Cross-Site Scripting Attack

Chapter 1

INTRODUCTION

Research studies have revealed that vulnerabilities in software systems often stem from issues within the programming languages themselves, such as insecure libraries or a lack of knowledge concerning vulnerabilities present in the underlying programming languages [1], [2], [3]. Vulnerabilities can be seen as weaknesses or gaps in the software system that may appear either unintentionally during the development process or due to mistake by the developers [1]. These vulnerabilities are sometimes a consequence of the need to rapidly release software systems to the market or the absence of expertise in security by the developers, causing them to overlook certain security features [1]. Typically, security has been handled as a non-functional requirement and often considered as an afterthought, leading to the "penetrate-and-patch" approach. In this approach, vulnerabilities are addressed by releasing patches once they are located [4]. The "penetrate-and-patch" approach to addressing vulnerabilities has some drawbacks. One of the primary disadvantages is the potential exploitation of vulnerabilities before they are discovered or before a patch is released [4]. This leaves software systems exposed to potential attacks and can result in notable damages. Additionally, the "penetrate- and-patch" approach can lead to elevated maintenance costs. Continuously discovering vulnerabilities, producing fixes, and delivering them can be time-consuming and resource- intensive, which makes it difficult for organizations to stay ahead of the continuously shifting threat landscape [4]. Another challenge with this approach is the integration of non-functional requirements, such as security, with other functional requirements. Treating security as a posterior can result in difficulties in effectively integrating security measures into the overall software design and architecture. This can lead to unstable and insecure system. Model-Driven Security Engineering (MDSE) [1]-[23] has emerged as a subfield of model- driven engineering to address these challenges. In the early stages of software development, MDSE focuses on determining requirements that are functional as well as non-functional, including security and to specify security requirements in specification languages. MDSE attempts to address issues such as poor software quality and elevated maintenance costs by incorporating security requirements from the beginning [5]. The basic idea behind MDSE is that if a model accurately depicts the desired system, it can be used to generate a bug-free implementation. The model is carefully made, and different methods such as model-based testing, theorem proving, model checking, or model validation can be used to assess and confirm the program code's compliance to the model [6]-[9]. These strategies ensure that the implemented code matches the anticipated behaviors indicated by the model, increasing the system's stability and quality.

1.1. Real Spec

RealSpec is a real-time executable specification language for concurrent systems [24]-[29]. It is identified as a declarative language used for formal specification. RealSpec works at analysis and design phases of software development. The primary choice of RealSpec as a case study is to extend security support for Real Spec. RealSpec is executable specification language and its syntax is similar to high-level programming languages that can minimize learning curve for developers. By using RealSpec, the model becomes executable, providing sufficient information for verification without the need for additional resources. The semantics of RealSpec are represented in a mathematically tractable and unambiguous manner, guaranteeing accurate and reliable analysis. This approach can help in ensuring the accuracy and reliability of the developed software [16].

1.2. Research Motivation

OWASP TOP 10 is a standard document for web application development [16]. The OWASP TOP 10 list is updated every three years because these attacks are still happening today despite many mitigation techniques for preventing them. SQLI [31]-[34], XSS [34]-[37], buffer overflow attack [38]-[40], [44], [48], TOCTOU [45], Sensitive information exposure [47] and broken authentication [46] are among them.

The previous specification languages do not model/specify security features collected from programming languages used to provide countermeasures against specific OWASP TOP 10 attacks [5], [7], [8], [10]-[15]. Many frameworks for evaluating specification language security coverage provide limited security features [16]-[22]. Furthermore, past security specification languages defined security needs that were either domain-specific or extensions to specific specification language with syntax and semantics related to that specification language. To convert the code from the model to a high-level programming language, these security definition methodologies necessitate far too many transformations from one model to another and then transformation to implantation language.

Another challenge was to identify secure programming functionalities from popular programming languages and prevention techniques to identity those functionalities that are inherently provided by programming languages to secure them from above-mentioned OWASP attacks. Combining security features into a taxonomy allows for evaluating the security capabilities of programming languages. Traditional comparison frameworks [4], [49] used to assess programming languages often lack a precise and comprehensive set of security features. This framework was needed to provide implementation phase level security functionalities to be incorporated as a security requirement starting from design phase. This all led to the creation of a research issue.

1.3. Statement of the Problem

The following problem statement is derived by research motivation current comparative research on programming languages for example Java, C++, Ruby, C#, and Python and specification languages for instance UMLSec, SecureUML, AMF, and SecureSOA fail to provide sufficient security features such as output validation, immutability and secure error message to address certain OWASP attacks. Traditional executable specification languages including S-Promela, SysML-Sec, Secure Descartes, and Ponder do not specify security requirements during initial phases of software development.

1.4. Research Questions

Q1. Which security features are present in popular programming languages (C#, C++, Ruby, Python, Java) to prevent OWASP attacks?

Q2. Which abstract security features exist in executable specification languages that can counter specific OWASP attacks?

Q3. How security requirements can be defined during initial phases of software development?
Q4. How security specified in RealSpec can be evaluated?

1.5. Research Aims and Objectives

The primary objective of this research is to build a security requirement framework for determining security coverage by specification language that effectively collects, abstracts, and models security features.

The objectives of this research work are

- To analyze security features in programming languages including Java, C++, Ruby,C#, and

Python and in prevention techniques that provide safety against XSS, SQLI, Sensitive Data Exposure, TOCTOU, and buffer overflow attack and development of a security feature framework to evaluate security feature coverage in programming languages.

- To develop a security requirement framework to evaluate security coverage of specification languages such as UMLSec, SecureUML, AMF, Ponder, SysML-Sec, Secure Descartes and SecureSOA.
- To specify security features such as output validation, immutability, and secure error message using RealSpec.
- To develop a custom tool to transform RealSpec code to C++ code and giving attack specifications to test the resultant code prevention against above-mentioned OWASP attacks for evaluation.

1.6. Research Contribution

In this thesis, we extend the work on RealSpec enhancing it to provide security support. The novel contribution of this research is as follows:

1.6.1. Contribution 1: Identifying Security Features and Designing SEFF

It is suggested that programming languages offers security features that safeguard against OWASP TOP 10 attacks [31]-[44], including SQLI, XSS, race conditions, sensitive information exposure, as well as buffer-overflow attacks. Additionally, SEFF assessed the extent of security feature coverage in five well-known programming languages in academia and offered suggestions for Java.

a) Identifying Security Features and Designing SEFF

The first step in enhancing the security of any system is to identify the relevant security features that need to be incorporated into the design. Understanding security requirements conducting thorough analyses of the system's operational environment and user requirements to understand the security needs. This can include identifying threats, vulnerabilities, and regulatory

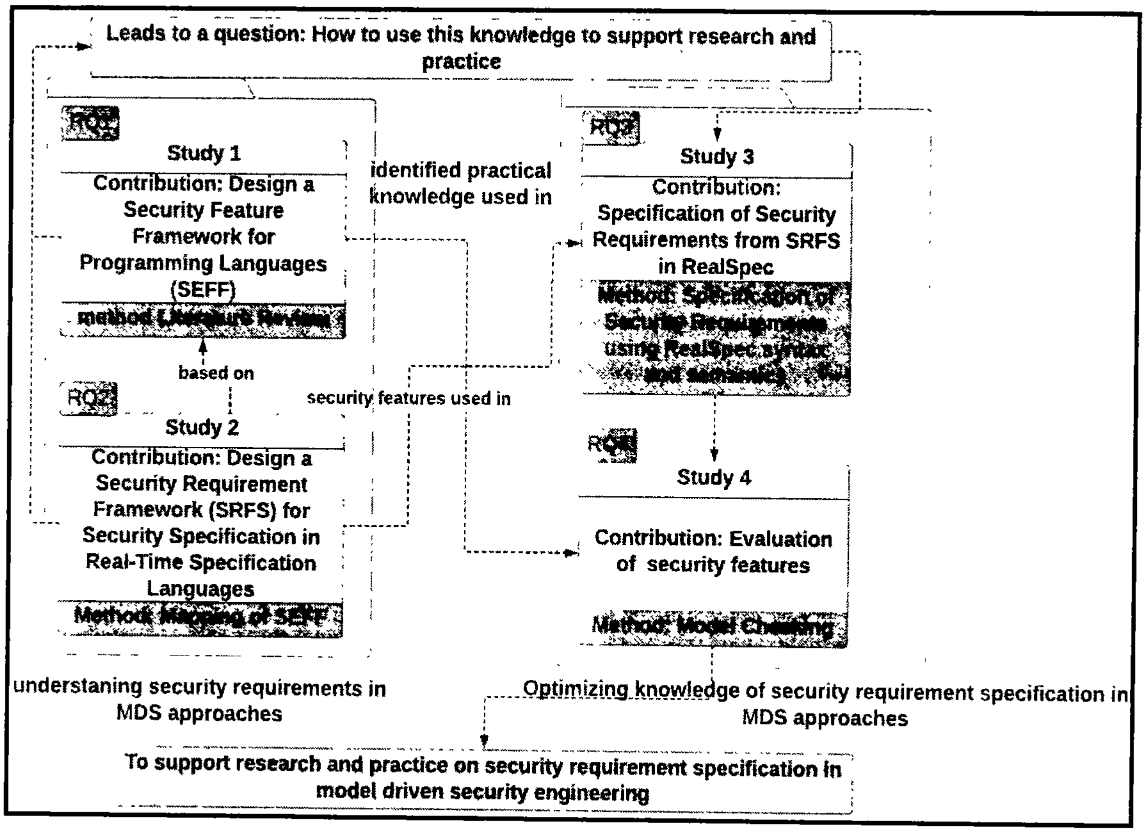

Leads to a question: How to use this knowledge to support research and practice
RQ1
Study 1
Contribution: Design a Security Feature Framework for Programming Languages (SEFF)
method Literature Review
identified practical knowledge used in
based on
RQ2
Study 2
Contribution: Design a Security Requirement Framework (SRFS) for Security Specification in Real-Time Specification Languages
Method: Mapping of SEFF
security features used in
RQ3
Study 3
Contribution: Specification of Security Requirements from SRFS in RealSpec
Method: Specification of Security Requirements using RealSpec syntax and semantics
RQ4
Study 4
Contribution: Evaluation of security features
Method: Model Checking
understaning security requirements in MDS approaches
Optimizing knowledge of security requirement specification in MDS approaches
To support research and practice on security requirement specification in model driven security engineering

Figure 1.1 Detailed Research Diagram

compliance requirements. Engaging with stakeholders (e.g., end-users, system architects, and security experts) to gather insights about the expected security attributes and potential security risks [31][44]. Categorizing security features into distinct groups, such as [50]

- Authentication: Mechanisms for verifying user identities.
- Authorization: Control over what authenticated users can access or modify.
- Confidentiality: Ensuring sensitive data is accessible only to authorized users.
- Integrity: Protecting data from unauthorized modifications.
- Non-repudiation: Ensuring that actions or transactions can be proven to have occurred, preventing denial by involved parties.

Mapping Security Features to System Components Establishing relationships between identified security features and specific system components or processes, facilitating targeted security interventions.

b. Designing the Security-Enhanced Functional Framework (SEFF)

Based on the identified security features, the next step is to design a Security-Enhanced Functional Framework (SEFF) that integrates these features into the system architecture. Architectural Framework Developing a modular architecture that incorporates the identified security features into different layers of the system. This allows for flexible and reusable security solutions that can be adapted to various scenarios. Design Patterns and Best Practices Implementing design patterns that promote security, such as the use of access control lists, cryptographic techniques, and secure communication protocols. Documenting best practices for implementing these security features effectively, ensuring they are robust and maintainable.

Integration with RealSpec ensuring that the SEFF aligns with the principles of Real Spec, allowing for the specification of real-time system behaviors along with security attributes. This could involve extending the Real Spec syntax or semantics to accommodate security features. Verification and Validation establishing methods for verifying that the SEFF meets the specified security requirements and for validating its effectiveness in mitigating identified risks. This may involve the use of formal verification techniques and automated testing tools. Enhanced Security Posture by systematically identifying and integrating security features, the contribution leads to a stronger overall security posture for real-time systems. Guidance for Practitioners the developed SEFF serves as a guide for system architects and developers, providing them with a structured approach to incorporating security into their designs. Facilitating Compliance by mapping security features to regulatory requirements, the contribution assists organizations in achieving compliance with industry standards and regulations.

1.6.2. Contribution 2: A Security Requirement Framework for Secure Web Application Development (SRFS)

The Security Requirement Framework for Secure Web Application Development (SRFS) represents a significant advancement in the field of web application security. By providing a structured approach to identifying, categorizing, and integrating security requirements into the development process, SRFS enhances the overall security posture of web applications, reduces vulnerabilities, and fosters a culture of security awareness among developers. This contribution ultimately leads to the development of more secure and resilient web applications that can withstand the evolving landscape of cyber threats.

The framework emphasizes the importance of gathering security requirements alongside functional requirements. Engaging with stakeholders, including business owners, developers, security experts, and end-users, to identify security needs based on real-world scenarios. Employing threat modeling techniques (e.g., STRIDE, PASTA) to systematically identify potential threats and vulnerabilities specific to the web application. SRFS categorizes security requirements into different domains, such as:

a. **Authentication**: Requirements related to user identity verification mechanisms (e.g., passwords, multi-factor authentication).
b. **Authorization**: Access control measures governing user permissions and roles.
c. **Data Protection**: Requirements for data confidentiality and integrity (e.g., encryption of sensitive data, secure data storage).
d. **Secure Communication**: Guidelines for implementing secure communication protocols (e.g., HTTPS, secure WebSocket's).
e. **Input Validation**: Requirements for validating and sanitizing user inputs to prevent injection attacks.

SRFS is designed to integrate seamlessly with popular development methodologies, such as Agile and DevSecOps, promoting security as a shared responsibility across teams. Incorporating security-focused sprints within Agile development cycles to address identified security requirements. Implementing automated security testing tools that integrate with Code Integration/Code Development (CI/CD) pipelines to ensure ongoing compliance with security requirements.

SRFS includes a repository of best practices and guidelines for secure coding, configuration management, and deployment strategies, tailored to web application development. Providing developers with established security design patterns (e.g., the use of secure session management, the principle of least privilege) to facilitate secure application design.

Developing a framework for verifying and validating that security requirements have been met. Encouraging regular penetration testing to identify vulnerabilities before deployment. Utilizing static and dynamic analysis tools to detect security flaws in the code base. Establishing training programs focused on secure coding practices and the importance of security in web application development. This is critical for building a security-aware culture within development teams.

Contribution to Secure Web Application Development

Here are the following contributions to secure web development.

a. **Proactive Security Measures**:

SRFS shifts the focus from reactive to proactive security, encouraging developers to consider security at the earliest stages of application design and development.

b. **Holistic Approach**:

By covering all phases of the development lifecycle, from requirement gathering to deployment, SRFS ensures that security is not an afterthought but an integral part of the development process.

c. **Standardization of Security Practices**:

The framework provides a standardized approach to defining and implementing security requirements, helping organizations achieve consistency in their security practices across various projects.

d. **Facilitation of Compliance**:

SRFS assists organizations in meeting regulatory compliance requirements by ensuring that security best practices are followed throughout the development lifecycle.

1.6.3. Contribution 3: Security features Specification in early phases of software development using RealSpec Executable Specification Language

The contribution of specifying security features in the early phases of software development using Real Spec is a significant advancement in securing software applications. By formalizing and integrating security requirements into the development lifecycle, this approach not only enhances security but also streamlines the development process across various programming languages. This proactive methodology leads to more robust and secure software systems capable of addressing the complex security challenges of modern applications.

Utilizing Real Spec to formally specify security features before the coding phase begins. This allows developers to understand and address security requirements early in the lifecycle, ensuring they are integrated into the software design from the outset. RealSpec's capability to incorporate temporal logic aids in defining time-sensitive security properties, crucial for real-time systems where security conditions may change over time.

Real Spec allows for the creation of executable specifications that can be tested and validated against actual system behavior. This means that security requirements can be directly linked to executable code, ensuring that they are not merely theoretical but practically applicable. The use of RealSpec promotes model-driven development practices, allowing developers to generate code and security features from high-level specifications automatically. Real Spec can be integrated with various programming languages to enhance the development of secure applications. Some of the related programming languages include:

- **C/C++:**
  Often used for system-level programming and real-time applications, where performance is critical. Security features defined in RealSpec can be translated into C/C++ code, ensuring low-level access control and memory safety.
- **Java:**

  Commonly used for enterprise applications. Real Spec specifications can help define security features like authentication and authorization, which can then be implemented using Java security libraries.
- **C#:**
  Used in .NET applications, where Real Spec can specify security requirements that can be implemented using built-in .NET security features, such as role-based access control.
- **Python:**

  Widely used for web development and scripting. Security features specified in RealSpec can guide the development of secure web applications using frameworks like Django or Flask, with an emphasis on secure coding practices.

- **JavaScript**:

  Critical for front-end web development. By specifying security features in Real Spec, developers can implement secure client-side scripting practices to mitigate common vulnerabilities like XSS (Cross-Site Scripting).

- **Ruby:**

  Commonly used in web development with frameworks like Ruby on Rails. Security specifications can help ensure the application follows best practices for data protection and user authentication.

- **Go:** Increasingly popular for micro services and cloud applications. Real Spec can help define security features for distributed systems, ensuring proper authentication and secure communication.

Security features specified in Real Spec can be interfaced with security mechanisms in various programming languages. For example, specifying authentication workflows in RealSpec can lead to implementations in any supported programming language, ensuring consistency in security practices. By providing a flexible framework for security feature specification, RealSpec can be adapted to different programming environments, facilitating a wide adoption across various platforms and languages. Real Spec's executable specifications can be subjected to automated testing, ensuring that the implemented security features behave as expected and meet the defined specifications. Utilizing model checking techniques to verify that the security features specified in RealSpec conform to desired security properties before actual coding takes place. Offering training resources and workshops to developers on how to effectively use Real Spec for security feature specification, ensuring that they understand the importance of security in early development phases.

1.6.4. Contribution 4: Development of an evaluation tool that takes the RealSpec program.

The development of an evaluation tool specifically designed for RealSpec programs represents a significant advancement in the field of software engineering, particularly for real-time systems. RealSpec, as an executable specification language, allows for the formalization of system behavior and security features. The evaluation tool aims to analyze and assess these RealSpec programs, ensuring they meet defined security and performance criteria. By integrating this tool into the software development lifecycle, developers can proactively identify and address potential vulnerabilities and inconsistencies in their specifications before the implementation phase.

At the heart of the evaluation tool is a modular architecture, which consists of several key components, including a parser, analyzer, and verifier. The parser interprets the RealSpec code and generates an abstract syntax tree (AST) that serves as the basis for further analysis. The analyzer performs semantic checks to ensure that the program adheres to the syntax and semantics of the Real Spec language. This semantic analysis is crucial for detecting logical errors and potential security vulnerabilities early in the development process. The verifier component plays a pivotal role by validating that the specified security properties align with the program's behavior, ensuring that the implemented security features are effective and reliable.

The evaluation tool supports both static and dynamic analysis methodologies. Static analysis allows for the examination of the Real Spec code without executing it, identifying potential security flaws, such as improper data handling and access control issues. This can involve data flow and control flow analyses, which help trace how data moves through the system and how control structures operate, respectively. On the other hand, dynamic analysis involves running simulations of Real Spec programs in a controlled environment to observe their runtime behavior. This includes generating test cases based on specified requirements to ensure comprehensive coverage and validating the security features under different scenarios.

A noteworthy aspect of the evaluation tool is its capability for verification of security properties through model checking. By exploring the state space of RealSpec programs, the tool ensures that all possible states conform to the desired security requirements. It employs temporal logic to verify time-dependent conditions, ensuring that security measures remain effective throughout the program's execution. This rigorous verification process not only enhances the confidence in the system's security posture but also aids in identifying any lapses in compliance with specified requirements.

To enhance user experience, the evaluation tool features an intuitive user interface that allows developers to easily input RealSpec programs, configure analysis settings, and visualize results.

This includes an integrated code editor with syntax highlighting for RealSpec, a dashboard that presents analysis outcomes, and graphical tools for visualizing data and control flow. Furthermore, the tool's integration with popular Integrated Development Environments (IDEs) enables real-time feedback during the coding process, helping developers identify issues as they arise and facilitating smoother debugging and correction.

The comprehensive nature of this evaluation tool, coupled with its robust capabilities for both analysis and verification, significantly contributes to the assurance of software quality in real-time applications. By allowing early detection of vulnerabilities and providing a structured approach to evaluating security features, the tool helps developers adopt best practices in secure coding and system design. Ultimately, this contribution not only enhances the security and reliability of software applications developed using RealSpec but also lays the groundwork for future research and innovations in automated verification techniques and formal methods in software development.

1.7. Thesis Overview

The initial phase involves identifying security features that mitigate vulnerabilities associated with SQL Injection (SQLI), Cross-Site Scripting (XSS), Sensitive Data Exposure, Time-of-Check to Time-of-Use (TOCTOU) issues, and buffer overflow attacks. Subsequently, a Security Feature Framework (SEFF) is proposed, as discussed in Chapter 2. This framework is then abstracted into specification languages (SRFS), as elaborated in Chapter 3. The abstraction of SRFS from SEFF provides a clear mapping between abstract specification-level security features and their corresponding security measures in various programming languages. This clarity is anticipated to bridge the security feature gaps between specification and implementation, thereby strengthening countermeasures against these threats. Following this, selected security features from SRFS are meticulously designed using RealSpec, which is described in detail in Chapter 4. Chapter 5 presents illustrative examples of these security features. An evaluation of the newly integrated security features within RealSpec is provided in Chapter 6. Finally, Chapter 7 concludes the thesis and outlines potential future directions for research.

Chapter 2

BACKGROUND

In today's digital landscape, security has become a critical concern, particularly for software systems that operate in real-time environments. As applications increasingly manage sensitive data and perform crucial functions, they are also becoming prime targets for various cyber threats. The need for robust security measures that can be integrated into the software development lifecycle has led to the exploration of new methodologies and frameworks that facilitate the early identification, specification, and validation of security features.

2.1. RealSpec: An Overview

RealSpec is a formal specification language designed for modeling and specifying real-time systems. It allows developers to create executable specifications that can directly represent system behavior. The advantage of using RealSpec lies in its ability to bridge the gap between high-level specifications and executable code, enabling the validation of system requirements before actual implementation begins. This characteristic makes Real Spec particularly suited for applications where timing constraints and security features are paramount.

Given the increasing complexity of software systems, particularly those involving real-time processing, ensuring the security of these applications requires a methodical approach. Traditional software development practices often fall short in addressing security vulnerabilities early in the design process, leading to gaps between specification and implementation. By leveraging RealSpec, developers can specify security requirements alongside functional requirements, allowing for a more integrated approach to system design.

2.2. Security Challenges in Real-Time Systems

Real-time systems face unique security challenges due to their inherent characteristics. These systems often operate under strict timing constraints, where delayed responses can lead to significant consequences, such as system failures or safety hazards. Moreover, the dynamic nature of real-time environments makes them susceptible to various types of attacks, including:

i. **SQL Injection (SQLI)**: Attackers manipulate database queries through invalidated user inputs, potentially gaining unauthorized access to sensitive data.

ii. **Cross-Site Scripting (XSS):** Malicious scripts are injected into web applications, compromising user data and session integrity.

iii. **Sensitive Data Exposure**: Insufficient protection of sensitive information can lead to data breaches, exposing user credentials and confidential data.

iv. **Time-of-Check to Time-of-Use (TOCTOU)**: This vulnerability occurs when the state of a resource changes between the time it is checked and the time it is used, allowing attackers to exploit the race condition.

v. **Buffer Overflow Attacks**: Exploiting vulnerabilities in memory management can allow attackers to execute arbitrary code, compromising system integrity.

These vulnerabilities highlight the need for a structured approach to specifying security features in real-time systems, ensuring that they are adequately addressed throughout the software development lifecycle.

2.3. The Role of Security Feature Framework (SEFF)

To effectively address these challenges, the development of a Security Feature Framework (SEFF) is essential. SEFF provides a structured approach to identifying and defining security features that are specifically tailored to counteract the identified vulnerabilities. By establishing a set of common security features and their specifications, SEFF aims to create a comprehensive security model that can be applied across various real-time applications. The framework focuses on ensuring that security features are not only well-defined but also seamlessly integrated into the development process. This integration is crucial for minimizing security gaps that often arise when security considerations are introduced too late in the development cycle.

2.4. Abstracting SEFF into Specification Languages (SRFS)

The abstraction of SEFF into Specification Languages (SRFS) is another critical aspect of this background. By mapping abstract specification-level security features to their counterparts in programming languages, SRFS facilitates a clearer understanding of how these security features can be implemented in practice. This mapping process is designed to eliminate discrepancies between the security features specified during the design phase and their actual implementation in code. The development of SRFS enables security requirements to be articulated in a languageagnostic manner, allowing developers to adopt best practices for secure coding across different programming environments. This flexibility is essential for ensuring that security measures are uniformly applied, regardless of the specific technologies being utilized.

2.5. The Significance of Evaluation

Finally, the evaluation of security features within RealSpec is a crucial step in validating their effectiveness. By systematically assessing the specified security features against real-world scenarios and attack vectors, developers can ensure that the implemented measures are robust and capable of mitigating identified risks. This evaluation process not only aids in the identification of potential weaknesses but also provides valuable insights that can guide future enhancements to security frameworks and methodologies.

2.6. Selection Criteria of the Attacks in RealSpec

The selection criteria for attacks in the context of RealSpec revolve around identifying vulnerabilities that are highly relevant to contemporary web applications and real-time systems. This criterion emphasizes the importance of focusing on attack types that are frequently exploited in current environments, ensuring that the analysis effectively addresses actual risks faced by organizations today. By concentrating on prevalent threats such as SQL Injection (SQLI), Cross-Site Scripting (XSS), and Buffer Overflow attacks, the study aims to provide a comprehensive understanding of the security landscape. This targeted approach facilitates the development of effective security features that can be specified within RealSpec, ultimately bridging the gap between theoretical specifications and practical implementations [6]. Moreover, prioritizing attacks that directly impact the integrity, confidentiality, and availability of real-time systems ensures that the security measures implemented are both practical and aligned with the operational realities of modern software development.

2.6.1. Relevance to Modern Applications

Criteria: The selected attack types are of significant relevance to contemporary web application environments and technologies, as supported by various studies [31][36][46].

Rationale: By focusing on attacks that are prevalent in modern applications, this analysis ensures that it addresses real-world risks that organizations encounter. For example, vulnerabilities such as Injection attacks and Broken Authentication are among the most frequently exploited threats in today's web applications.

The relevance of attack types to modern applications is a crucial consideration in the context of cybersecurity. As web technologies evolve, the tactics and methodologies employed by malicious actors also adapt, making it essential to focus on vulnerabilities that pose a tangible threat to current systems [46][47][48]. Selecting attack types that are commonly exploited ensures that security analyses remain grounded in practical realities, enabling organizations to effectively combat the most pressing risks.

Injection attacks, such as SQL Injection (SQLI), are particularly notable for their widespread occurrence. These attacks exploit weaknesses in application input validation, allowing attackers to inject malicious queries into databases, which can lead to unauthorized access, data manipulation, or even complete system compromise. Given the prevalence of databases in modern applications, the risk associated with SQLI is significant, necessitating robust countermeasures that can be specified and validated within frameworks like RealSpec.

Similarly, Broken Authentication vulnerabilities represent a critical threat to web applications. These weaknesses arise when authentication mechanisms are improperly implemented, allowing attackers to bypass security measures and gain unauthorized access to sensitive user data. For instance, weak password policies, failure to implement account lockout mechanisms, and inadequate session management can all contribute to this vulnerability. By emphasizing attacks like Broken Authentication, the analysis not only addresses a widely recognized security concern but also encourages developers to adopt best practices in designing secure authentication systems.

By concentrating on attack types that are directly relevant to modern applications, the analysis aims to create a more effective and targeted approach to security. This focus helps ensure that the proposed security features and frameworks not only meet theoretical requirements but also withstand real-world threats. Ultimately, this relevance fosters a proactive security posture, equipping organizations with the tools and knowledge needed to safeguard their applications against the evolving landscape of cyber threats.

2.6.2. Impact on Security Posture

The impact of selected attack types on an organization's security posture is a critical consideration in the realm of cybersecurity. An organization's security posture refers to its overall security strength and the measures it employs to protect its assets, including networks, systems, and data. Understanding the potential impact of specific attacks is essential for effectively prioritizing security efforts and allocating resources to mitigate risks.

When prevalent attack types, such as SQL Injection, Cross-Site Scripting (XSS), and Broken Authentication, are effectively integrated into security analyses and frameworks, organizations can significantly enhance their security posture. These attacks often exploit vulnerabilities that, if left unaddressed, can lead to severe consequences, including data breaches, financial losses, reputational damage, and regulatory penalties. For instance, a successful SQL Injection attack can compromise entire databases, leading to unauthorized access to sensitive information, which can have catastrophic effects on user trust and organizational integrity.

By proactively identifying and addressing these attack vectors, organizations can implement robust security features and protocols that strengthen their defenses. This includes the adoption of secure coding practices, regular security assessments, and the implementation of intrusion detection systems. Furthermore, a strong security posture is characterized by a culture of security awareness among employees, where individuals are educated about potential threats and best practices for safeguarding information.

Incorporating the insights gained from analyzing the impact of specific attacks allows organizations to make informed decisions about their security strategies. This, in turn, facilitates the development of comprehensive security frameworks that not only meet regulatory requirements but also align with the unique risk profiles of the organization. Ultimately, a robust security posture, informed by the relevance and impact of current attack types, enables organizations to defend against evolving cyber threats and maintain the confidentiality, integrity, and availability of their critical assets.

2.6.3. Criteria and Rationale for Attack Selection in Real Spec

Criteria: The selected attacks possess a significant capacity to undermine the security status of an application, as indicated in various studies [36][37].

Rationale: Focusing on attacks that can result in severe consequence such as data leaks, unauthorized access, or complete system compromise helps to identify and prioritize areas of highest risk within the application. For instance, vulnerabilities like Cross-Site Scripting (XSS) and Sensitive Data Exposure can lead to catastrophic outcomes if not adequately addressed. The criteria for selecting attack types based on their potential to weaken an application's security status underscores the importance of understanding the ramifications of various vulnerabilities. In the context of cybersecurity, not all vulnerabilities carry the same weight; some can lead to minor inconveniences, while others can result in devastating consequences for an organization.

i. **Cross-Site Scripting (XSS)** is a prime example of an attack that can have significant implications. XSS attacks allow attackers to inject malicious scripts into web pages viewed by other users. This can lead to various malicious outcomes, including session hijacking, defacement of web content, or the theft of sensitive information such as login credentials. The ability of XSS to target users directly can create a cascading effect; once one user is compromised, the attacker can leverage that access to infiltrate deeper into the system, potentially impacting multiple users and organizational resources. This is especially alarming in environments where sensitive data is handled, as the repercussions can include severe data breaches and loss of customer trust.

ii. **Sensitive Data Exposure** is another critical attack vector that demonstrates considerable ability to weaken an application's security posture. When sensitive data, such as personal information, financial records, or health data, is inadequately protected, it can be exposed to unauthorized access. This can occur through various means, including insufficient encryption, weak access controls, or poor data handling practices. The consequences of such exposure are far-reaching, leading not only to legal and regulatory penalties but also to long-lasting damage to an organization's reputation. Once sensitive data is leaked, the trust that customers and stakeholders place in the organization can be irreparably harmed, resulting in financial losses and the potential for extensive litigation.

By concentrating on these high-impact attack types, organizations can direct their efforts toward areas that pose the greatest risk. This targeted approach allows for a more efficient allocation of resources, ensuring that security measures are implemented where they will have the most significant effect. For example, investing in robust input validation and output encoding mechanisms can mitigate the risks associated with XSS, while employing strong encryption and access controls can protect against sensitive data exposure. Moreover, recognizing the severe consequences associated with these vulnerabilities fosters a culture of security awareness within the organization. It emphasizes the need for proactive measures, such as regular security assessments, penetration testing, and employee training, to equip teams with the knowledge and skills to identify and address potential security threats before they can be exploited.

2.6.4. Ease of Exploitation

Refers to the relative simplicity with which an attacker can successfully carry out a specific attack against a target application. This criterion is crucial in the selection of attack types for analysis, as it directly influences the potential risk posed to an organization's security posture. Attacks that are easy to exploit often require minimal technical skill, limited resources, or no specialized knowledge, making them attractive options for a wide range of malicious actors, from amateur hackers to more sophisticated adversaries.

For instance, vulnerabilities such as Cross-Site Scripting (XSS) and SQL Injection (SQLI) are notorious for their ease of exploitation. XSS attacks can be carried out by simply inserting malicious scripts into user input fields or URLs, often without the need for advanced programming skills. This simplicity allows attackers to target a broad spectrum of web applications, resulting in a widespread impact on user data and system integrity. Similarly, SQL Injection attacks can be executed by injecting crafted SQL statements into input fields, enabling attackers to manipulate databases and extract sensitive information with minimal effort.

The accessibility of these attack vectors raises significant concerns for organizations, as it increases the likelihood of successful breaches. A vulnerability that can be easily exploited poses a higher risk, as it may be targeted by a large number of attackers, leading to potential data breaches, financial losses, and reputational damage. Therefore, assessing the ease of exploitation is vital for organizations to prioritize their security measures effectively. By focusing on vulnerabilities that are not only impactful but also straightforward to exploit, organizations can implement targeted defenses and mitigation strategies to enhance their overall security posture.

Criteria: The selected attack types exhibit a range of complexities and ease of exploitation, as indicated in various studies [33][38].

Rationale: Including a diverse array of attacks, categorized by their ease of exploitation and complexity, provides a comprehensive understanding of the security landscape. For example, while certain vulnerabilities, such as SQL Injection, can be easily executed with minimal technical expertise, others, like Security Misconfiguration, may require advanced skills and a deeper understanding of the system's architecture.

The complexity and ease of exploitation of attack types play a significant role in determining the threat they pose to organizations. This criterion focuses on understanding that not all vulnerabilities are created equal; some are straightforward to exploit while others may demand a high level of technical proficiency or an in-depth knowledge of system configurations.

i. **Ease of Exploitation**: Some attacks are characterized by their straightforward execution, allowing even novice attackers to execute them with little effort. For instance, SQL Injection attacks involve injecting malicious SQL queries into input fields, which can be done with basic knowledge of SQL syntax. The ubiquity of web applications that fail to properly sanitize user input makes this vulnerability particularly attractive to attackers, as it enables them to manipulate databases easily and extract sensitive information.

ii. **Complexity of Exploitation**: In contrast, other vulnerabilities, such as Security Misconfiguration, may require a more sophisticated approach. Security Misconfiguration refers to scenarios where the security settings of an application, server, or database are improperly configured, leaving the system vulnerable to exploitation. Identifying and exploiting these misconfigurations often necessitates an understanding of the underlying technologies, access to the system's configuration files, and the ability to recognize deviations from best practices. Attackers may need to employ more advanced techniques, such as social engineering or reconnaissance, to gather the necessary information before they can exploit these vulnerabilities.

The rationale for incorporating both simple and complex attack types into security analyses is to provide a well-rounded perspective on the potential risks that organizations face. This diversity ensures that security strategies address vulnerabilities across the spectrum of exploitability. By understanding that some attacks require minimal effort while others demand advanced skills, organizations can better prioritize their security measures, focusing on the most critical areas of vulnerability. For example, while addressing easily exploitable attacks like SQL Injection is essential, organizations must also be vigilant about complex vulnerabilities like Security

Misconfiguration that may go unnoticed but can result in significant security breaches if left unchecked. This balanced approach allows organizations to strengthen their overall security posture by implementing layered defenses that account for both simple and complex attack vectors.

2.6.5. Complexity and Impact on Concurrency

Complexity and its impact on concurrency refer to how the intricate nature of certain attacks can influence the performance and behavior of systems, particularly in concurrent environments where multiple processes or threads operate simultaneously. In the realm of cybersecurity, understanding the interplay between complexity and concurrency is essential for accurately assessing potential vulnerabilities and their consequences on application behavior.

The complexity of an attack can often dictates its method of execution, which in turn can affect how a system handles concurrent processes. For instance, attacks that exploit timing issues, such as Race Conditions, can become particularly problematic in concurrent environments. Race Conditions occur when multiple processes access shared resources without proper synchronization, leading to unpredictable outcomes. An attacker can exploit this complexity by manipulating the timing of events, potentially causing system failures, data corruption, or unauthorized access to resources.

Similarly, complex attacks, such as Denial of Service (DoS) or Distributed Denial of Service (DDoS) attacks, can have severe implications for concurrency. These attacks flood a system with excessive requests, overwhelming its ability to process legitimate transactions. In a concurrent environment, this can lead to significant performance degradation, resulting in service outages or degraded user experiences. The complexity of implementing effective mitigation strategies in the face of such attacks further underscores the importance of understanding how these vulnerabilities interact with concurrent processing.

Furthermore, the impact of complex attacks on concurrency can extend beyond immediate performance issues. Security measures intended to protect against such attacks can inadvertently affect the system's ability to handle concurrent operations effectively. For example, implementing strict access controls or rate limiting to prevent abuse may introduce latency or bottlenecks in system performance, particularly in high-traffic environments.

Criteria: The selected attacks are critical in systems that manage concurrent processes or threads. They can result in unpredictable behavior and lead to significant security issues, such as privilege escalation or data corruption, as indicated in various studies [38][44].

Rationale: Exploiting Time-of-Check to Time-of-Use (TOCTOU) vulnerabilities allows attackers to manipulate the timing of operations, potentially leading to unauthorized access or modifications. This makes the understanding of race conditions crucial in systems where concurrency is a key factor [45][47]. Given that this thesis aims to enhance security within Real Spec, a specification.

The Time-of-Check to Time-of-Use (TOCTOU) vulnerability occurs when there is a time gap between the checking of a condition (such as permission validation) and the subsequent use of that condition in a system. This timing issue can be exploited, particularly in concurrent systems where multiple processes or threads may access shared resources simultaneously. When an attacker is able to introduce a delay or manipulate the timing of events, they can potentially alter the state of the system, leading to unauthorized actions.

i. In systems that handle concurrent processes, the potential for TOCTOU vulnerabilities is significantly heightened. For example, consider a scenario where a process checks if a user has permission to access a particular resource. If there is a delay between this check and the actual usage of that resource, an attacker could exploit this window by changing the underlying conditions—such as the permissions of the resource—between the two operations. This could lead to privilege escalation, where the attacker gains access to resources or capabilities that they should not have. Another significant concern is data corruption. If multiple threads are modifying shared data simultaneously without proper synchronization, an attacker may exploit TOCTOU vulnerabilities to manipulate the data in transit. For example, an attacker could change a file's contents after a program checks its integrity but before it commits any operations that rely on that integrity check. The result can be severe, leading to inconsistencies, data loss, or system instability.

ii. Given that RealSpec is designed for specifying systems that operate concurrently, the inclusion of TOCTOU attacks in the security framework is essential. By addressing TOCTOU vulnerabilities, the thesis aims to create robust security features that protect against these complex attack vectors. This involves implementing strict access controls, ensuring proper synchronization between processes, and establishing comprehensive validation checks that are consistently enforced throughout the execution of concurrent operations.

Furthermore, by examining TOCTOU vulnerabilities within the context of Real Spec, this thesis emphasizes the importance of security at the specification level. It highlights that addressing vulnerabilities early in the software development lifecycle can significantly reduce the risk of exploitation in the final system.

2.6.6. Relevance to Legacy Systems and Languages

Relevance to Legacy Systems and Languages refers to the significance of understanding how older systems and programming languages may be more vulnerable to specific attacks, particularly in the context of cybersecurity. Legacy systems, which often include outdated software and hardware, can exhibit weaknesses that are not only inherent to their architecture but also arise from the languages and frameworks used in their development.

Legacy systems typically lack the modern security features and updates found in contemporary software. As a result, they can be particularly susceptible to various vulnerabilities, such as ***buffer overflows, injection attacks,*** *and* ***improper authentication mechanisms***. The continued reliance on older programming languages, which may have less robust security practices, further exacerbates this problem. For instance, languages that allow direct memory access or provide limited error handling can create opportunities for exploitation that modern languages have largely mitigated through better design and security controls.

The relevance of these vulnerabilities in legacy systems extends beyond mere technical issues; they can have profound implications for organizational security. Many organizations still operate critical functions on legacy systems due to their importance in business processes or the high cost of migration. However, maintaining these systems without addressing their security vulnerabilities can expose organizations to significant risks, including data breaches, regulatory penalties, and damage to their reputation.

Furthermore, the integration of legacy systems with newer technologies can introduce additional complexity, creating scenarios where vulnerabilities in older components can be exploited through modern interfaces. This highlights the need for comprehensive security assessments that consider the entire architecture, including any legacy components, to ensure that vulnerabilities are adequately addressed

Criteria: Buffer overflows are especially pertinent in systems programmed in languages that do not offer automatic bounds checking, such as C and C++. Many legacy systems and specific high-performance applications continue to utilize these languages, making them vulnerable to such attacks [27][42][43][48].

Rationale: These vulnerabilities are commonly found in major platforms and older programs, where modern security measures may not have been fully adopted. A thorough understanding of buffer overflows is crucial for improving system security and protecting these environments.

i. **Buffer Overflow Vulnerabilities**: A buffer overflow occurs when a program writes more data to a buffer than it can hold, leading to adjacent memory being overwritten. This can create vulnerabilities that attackers can exploit to execute arbitrary code, crash the system, or manipulate data. The lack of automatic bounds checking in programming languages like C and C++ allows developers to write code that does not prevent these kinds of errors. As a result, many legacy systems developed years ago without modern security practices remain at risk.

ii. **Relevance in Legacy Systems**: Legacy systems are often crucial for organizations, handling significant transactions or sensitive data. However, these systems frequently rely on outdated programming languages, which do not incorporate the advanced security features present in newer languages. For example, languages like Python or Java provide built-in bounds checking, which mitigates the risk of buffer overflows. Conversely, C and C++ require developers to implement their own checks, leading to potential oversights and vulnerabilities. Many legacy systems have not undergone extensive updates or security audits, allowing these vulnerabilities to persist. Attackers can exploit buffer overflow vulnerabilities to gain unauthorized access or execute malicious payloads. This is particularly concerning for high-performance applications where performance optimizations may lead to even less rigorous coding practices, further increasing vulnerability.

2.7. Inclusive and Exclusive Criteria

In the context of research, project management, or security assessments, **inclusive and exclusive criteria** refer to the guidelines used to determine what is considered relevant or applicable within a particular scope or framework. These criteria help in identifying the parameters for analysis, decision-making, and evaluation. Here's a breakdown of the two concepts:

2.7.1. Inclusive Criteria

Inclusive criteria specify the conditions or attributes that must be present for an item, entity, or aspect to be included in a particular analysis or evaluation. The purpose of inclusive criteria is to ensure that relevant elements are recognized and incorporated into the study or evaluation process. This helps to create a comprehensive understanding of the subject matter. Inclusive criteria often have a wider scope, allowing for the inclusion of various elements that may not be immediately apparent. They focus on positive attributes or characteristics that align with the goals of the study or project. Inclusive criteria may vary based on the context or specific objectives of the analysis.

In the context of evaluating security features for a software application, inclusive criteria may include:

i. The presence of user authentication mechanisms.

ii. Data encryption protocols being implemented.

iii. Input validation measures in place to prevent injection attacks.

2.7.2. Exclusive Criteria

Exclusive criteria define the conditions or characteristics that must be absent for an item, entity, or aspect to be excluded from analysis or evaluation. The purpose of exclusive criteria is to narrow down the focus of the study or evaluation, filtering out elements that are deemed irrelevant or not aligned with the objectives of the research. Exclusive criteria typically have a narrower scope, concentrating on specific elements that do not meet the defined conditions. They focus on attributes that indicate the absence of necessary characteristics or features. Exclusive criteria help clarify what will not be considered, ensuring that the evaluation remains precise and focused.

In the context of assessing vulnerabilities in a web application, exclusive criteria may include:

i. The absence of secure coding practices, such as input sanitization.

ii. Lack of security patches for known vulnerabilities.

iii. No implementation of security headers, such as Content Security Policy

2.8. Inclusive and Exclusive Criteria for Systematic Literature Review (SLR)

In the context of a Systematic Literature Review (SLR), inclusive and exclusive criteria are essential to ensure that the review process remains focused, relevant, and of high quality. These criteria establish the boundaries for what research studies or papers are considered for inclusion or exclusion, helping to ensure that the findings and conclusions drawn are based on the most appropriate and reliable sources.

2.8.1. Inclusive Criteria for SLR

Inclusive criteria are used to identify which studies or research papers should be considered part of the review. These criteria are based on the relevance of the research to the SLR's objectives and its ability to contribute meaningfully to answering the research questions. For instance, studies included must directly address the core focus of the review, such as specific research questions or themes related to the field of study. Additionally, inclusive criteria may specify that only studies from a particular time frame are considered, ensuring that the review reflects the most recent advancements in the field. It also often includes limitations on the type of research accepted, such as focusing on peer-reviewed journal articles or empirical studies over opinion pieces or gray literature. Language restrictions, for example, limiting studies to those published in English, are also common to make the review process more feasible. Furthermore, studies using specific research methodologies, like qualitative, quantitative, or mixed methods approaches, may be prioritized if they align with the study's goals. The purpose of inclusive criteria is to provide a broad yet focused selection of studies that offer a comprehensive overview of the current state of knowledge in the field. This helps ensure the SLR covers all relevant aspects of the topic being investigated, allowing for a thorough exploration of the literature and a balanced synthesis of findings.

2.8.2. Exclusive Criteria for SLR

Exclusive criteria, on the other hand, are used to filter out studies that are irrelevant, outdated, or of poor quality. These criteria define what research should be excluded from the review to maintain focus and integrity. Studies that do not directly relate to the research questions, fall outside the designated time range, or fail to meet specific methodological standards (e.g., non-peer-reviewed sources or studies without clear research designs) are removed from consideration. For example, outdated studies that do not reflect current technologies or trends may be excluded unless they provide foundational insights. Studies with significant methodological flaws, such as lacking empirical data or using weak research designs, are also typically filtered out to ensure the review is based on high-quality, robust research. Additionally, duplicate studies are removed to prevent redundancy, and accessibility issues, such as studies that are not available in full text, can be grounds for exclusion. Exclusive criteria help to sharpen the focus of the review by filtering out studies that do not meet the required standards or do not align with the research's primary objectives. This ensures the review process remains efficient and targeted, and that the resulting analysis and synthesis are based on the most relevant, high-quality literature.

2.9. Security Weaknesses in Programming Languages

One of the main contributors to OWASP attacks, as identified in sources [31]-[44], is the inherent vulnerabilities present in the libraries of widely used programming languages. These vulnerabilities are frequently exploited by attackers using various methods. For instance, Cross-Site Scripting (XSS) attacks have been documented in Ruby versions 6.1.7.3 and 7.0.4.3, as well as Python 3.2 when used with Python [4]. Similarly, race conditions and SQL Injection (SQLi) vulnerabilities have been observed in Java Native Interface (JNI03) [4][43]. Additionally, buffer overflow remains a persistent issue in C/C++ programming, especially when using unsafe functions like strcpy().

While many programming languages come equipped with built-in security features, such as strong typing (a critical security feature that helps by enforcing type constraints on variables), these protections often address only a subset of potential vulnerabilities. Strong typing includes mechanisms like ensuring a variable is correctly typed and providing access permissions based on its class, which can help mitigate certain threats [8]. Languages such as Python, Ruby, C#, and Java are considered strongly typed languages and offer some level of security against common exploits [29]. However, even with these protections, many languages only safeguard against a limited range of attacks, leaving other vulnerabilities unaddressed.

Table 2.1 attack selection criteria

Threat sources	CWE 2003/SANS TOP 25 [1]	OWASP 2013[2]	OWASP 2017[3]	CIS Top 18[4]	Mitre Att&ck[5]
Buffer Overflow attacks	Buffer Overflow attacks Ranked on 1st Out of Bound Read Ranked 5th Use after free Ranked 7th	Using Known Vulnerable Components Ranked A9	Using Known Vulnerable Components Ranked A9	CIS Control 4 Secure Configuration of Enterprise Assets	Privilege escalation
XSS	XSS Ranked 2nd	Ranked A3	Ranked A7	-	Lateral Moveme nt Taint Shared Content
SQLI	SQLI Ranked 3rd Improper Input Validation Ranked 4th	Ranked A1	Ranked A1	-	Command and Injection content Injection
Broken Authenticatio n and Session Management	Improper Authentication Ranked 14	Ranked A2	Ranked A2	-	Credentia l Access Steal Web Session Cookie
Race Condition	Ranked 21	Selected due to RealSpec concurrent nature	Selected due to RealSpec concurrent nature	Selected due to RealSpec concurrent nature	Selected due to RealSpec concurrent nature
Sensitive Data Exposure	Missing Authorization Ranked 16	Ranked A6 Missing Function level Access Control A7	Ranked A3	Data Protection CIS Control 3	Privilege escalation

[1] https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html
[2] https://css.csail.mit.edu/6.858/2014/readings/owasp-top-10.pdf
[3] https://owasp.org/www-project-top-ten/2017/Top_10
[4] https://www.cisecurity.org/controls/cis-controls-list
[5] What is the MITRE ATT&CK Framework? | IBM

A key reason for many of these security exploits is the lack of awareness among software developers regarding the inherent flaws in the programming languages they use. Despite the well-known history of memory-related vulnerabilities in C/C++, these languages continue to be widely used, contributing to the persistence of memory-based attacks like buffer overflows and return-oriented programming [1].

One of the most critical issues leading to these vulnerabilities is the improper handling of not sanitized input. User input, especially in web applications, should never be trusted outright and should always be considered potentially dangerous. Input data must be validated for its type and range to prevent exploitation. Failing to properly sanitize and validate input can lead to a host of serious security issues, including SQL Injection (SQLi) [31-34], Cross-Site Scripting (XSS) [34-36], buffer overruns, and return-oriented programming attacks [38-42, 44]. Moreover, weak input validation is a major cause of concurrency threats [37-45], authentication failures [46], and sensitive data exposure [43-47].

The remainder of this section delves deeper into specific attacks that capitalize on these programming language weaknesses. Additionally, Table 1 provides an overview of several application-level threats along with protective strategies from the existing literature aimed at mitigating these security issues.

2.9.1. SQL Injection

SQL injection (SQLI) attacks occur when an attacker manipulates input fields within a web application to insert malicious SQL queries, ultimately gaining unauthorized access to a database [31]-[34]. By crafting specific inputs, the attacker can bypass authentication, retrieve sensitive data, or even manipulate the database. Table 2.1 attack selection criteria.

Figure 2.1 illustrates the structure of a typical SQL injection attack. This thesis focuses primarily on two types of SQL injection attacks: tautology-based and error-based. However, with further adjustments, other forms of SQLI can also be addressed. A tautology-based SQLI is shown in Figure 1, where the attacker injects SQL code that results in a condition that is always true, allowing access to the database. In contrast, error-based SQLI relies on exploiting error messages returned by the database when invalid SQL queries are executed, providing valuable information to the attacker. One common approach for SQL injection is to use a predefined SQL query template, which the system expects the user to follow [36]. This involves crafting input that will be executed by the database without proper validation.

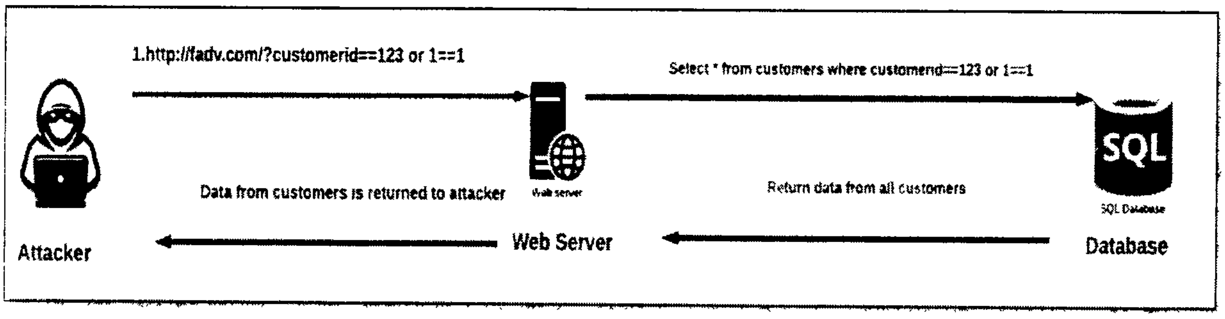

1.http://fadv.com/?customerid==123 or 1==1
Select * from customers where customerid==123 or 1==1
Data from customers is returned to attacker
Web server
Return data from all customers
SQL
SQL Database
Attacker
Web Server
Database

Figure 2.1 Structured Query Injection attack

Several techniques have been developed to detect and prevent SQL injection attacks, including:

1. **SQL Check** [31]: A tool designed to inspect SQL queries for potential vulnerabilities. It compares user input against query templates to detect anomalies.
2. **CANDID** [32]: This method generates templates dynamically by observing the legitimate structure of queries and preventing injections by enforcing proper input validation.
3. **Classification methodology by Pham and Subburaj** [33]: This approach offers a structured classification of SQL injection types, providing a method to systematically detect and prevent them.
4. **DIAVA** [31]: A multi-level regular expression-based method that scans SQL queries for attack patterns, identifying and neutralizing SQL injection attempts.

These techniques provide robust defenses against common SQLI attacks, ensuring that web applications handle user input securely and minimize the risk of database exploitation.

2.9.2. Buffer Overrun Attack

A buffer overrun (also known as buffer overflow) is a memory-based attack in which an attacker overwrites a buffer's boundary and redirects the execution flow to a target address chosen by the attacker [38], [48]. For an attacker to exploit this vulnerability, they typically need to know the starting addresses of critical memory regions like the stack, heap, or system libraries. Once a buffer overflow occurs, it can lead to severe consequences such as system crashes, unauthorized privilege escalation, or corruption of the program's execution state.

To mitigate buffer overflow attacks, several techniques have been developed. One hardware-based solution is the non-executable stack (NX-stack) [38], [48], which marks specific areas of memory as non-executable. This prevents the attacker from executing arbitrary code injected into memory regions like the stack. Another protection method is Stack Smashing Protection (SSP), often implemented using stack canaries, which is also referred to as Stack Guard [48]. A stack canary is a random value inserted just before the return address in the stack by a compiler tool such as Stack Guard. If an overflow occurs, the canary value will be altered, and the program can detect this change, preventing execution of malicious code.

Another important mitigation technique is Address Space Layout Randomization (ASLR), which randomizes the starting locations of key memory regions such as the stack, heap, and dynamically loaded libraries each time the program is executed. This makes it significantly harder for attackers to predict where to inject their malicious payload [27], [48]. By altering the memory layout at load or link time, ASLR helps protect against buffer overflow exploits.

Modern microprocessors also incorporate hardware-based buffer overrun protection using the NX-bit (Non-Executable bit), which is a feature of the Memory Management Unit (MMU). This bit ensures that specific memory regions, like the heap, stack, and shared libraries, are marked as non-executable. This means that even if an attacker manages to inject code into these areas, the system will raise an exception when any execution attempt is made in those regions [48]. With the NX-bit, memory regions can be designated as either writable or executable, but not at the same time. This separation helps further secure applications against buffer overflow attacks.

However, despite these defense mechanisms ASLR, NX-bit, and Stack Guard more advanced types of attacks have emerged, such as Return-Oriented Programming (ROP) and Return-to-libc. ROP is a sophisticated technique that bypasses these protections by reusing existing code in the program's memory. Instead of injecting new malicious code, ROP takes advantage of small fragments of legitimate code, known as "gadgets." These gadgets typically end with a return (ret) instruction and perform simple operations, such as loading a value into a register. By chaining together, a series of gadgets, an attacker can modify the program's control flow and perform malicious actions without injecting new code into the system [38], [40].

In a return-to-libc attack, the attacker exploits a buffer overflow vulnerability to redirect the program's execution to a standard function in the C library (libc), such as system (), allowing them to execute shellcode or commands with elevated privileges [38], [40], [42].A more sophisticated defense against ROP attacks is Dynamic Integrity Management (Dyn I MA), which monitors the integrity of program execution at runtime to detect and prevent ROP-based exploits. DynIMA checks the integrity of the code's control flow and ensures that it hasn't been manipulated by an attacker [38].

OpenSSL, a widely used encryption library for web applications, is employed by major companies such as Amazon, Yahoo, Google, Facebook, LinkedIn, Netflix, Healthcare.gov, and Dropbox [2]. Despite its broad adoption, OpenSSL has been subject to critical vulnerabilities, one of the most notorious being Heartbleed [39]. Heartbleed is a severe security flaw in OpenSSL, specifically stemming from a buffer overflow vulnerability. Buffer overflow vulnerabilities are often exploited in various control flow hijacking attacks, allowing attackers to manipulate a program's execution.

The Heartbleed vulnerability, at its core, is a coding defect within OpenSSL, particularly affecting a Transport Layer Security (TLS) plugin, which compromises secure data transmissions. This flaw occurs due to improper memory management in OpenSSL. Since OpenSSL's memory handling is built on top of the standard C/C++ libc library's memory management functions, conventional security mechanisms like Address Space Layout Randomization (ASLR) are ineffective in protecting against Heartbleed. ASLR typically helps prevent buffer overflow attacks by randomizing memory locations, but in the case of Heartbleed, the vulnerability bypasses ASLR due to its deep integration with OpenSSL's memory structure. Consequently, the buffer overflow in Heartbleed exposes sensitive data, such as encryption keys and personal information, making it a critical issue in modern encryption practices.

2.9.3. Cross-Site Scripting Attack (XSS)

Web applications block scripts running as part of different domains from obtaining cookies that have been created for these domains. However, XSS can get through this simple safeguard. The malicious script might be placed on a trusted website and then run by a different client of the same server. An attacker can see the authentication cookie content since the malicious code executes on the system of the victim. The session-specific user credentials, including the session, are stored in the authentication cookie value. Using the information stored in authentication cookies, an attacker can simply hijack the session of the user [35]-[37].

By retrieving the content of the authentication cookies, the attacker can hijack the victim's session, effectively taking over their account without needing to know their login credentials. This enables the attacker to impersonate the victim and perform unauthorized actions under their identity, leading to serious security breaches [35]-[37]. The most prevalent threat on the OWASP ranking is XSS. To prohibit client-side scripts against accessing cookies, web servers can utilize the secure as well as Http Only options [34]. As a mitigation strategy, Session Shield [34] enforces Http only and secure tags on authentication cookies. This approach, however, cannot appropriately distinguish authentication cookies [34]. The Ruby and C# need to be examined for cross-site scripting vulnerabilities [4]. According to a study [34], 50% of ASP websites that employ sessions make use of the Http only flag, which requires putting the Http only and secure flags on authentication cookies. However, because authentication cookies cannot correctly be detected in order to configure the flags, this practice is not widely employed [34].

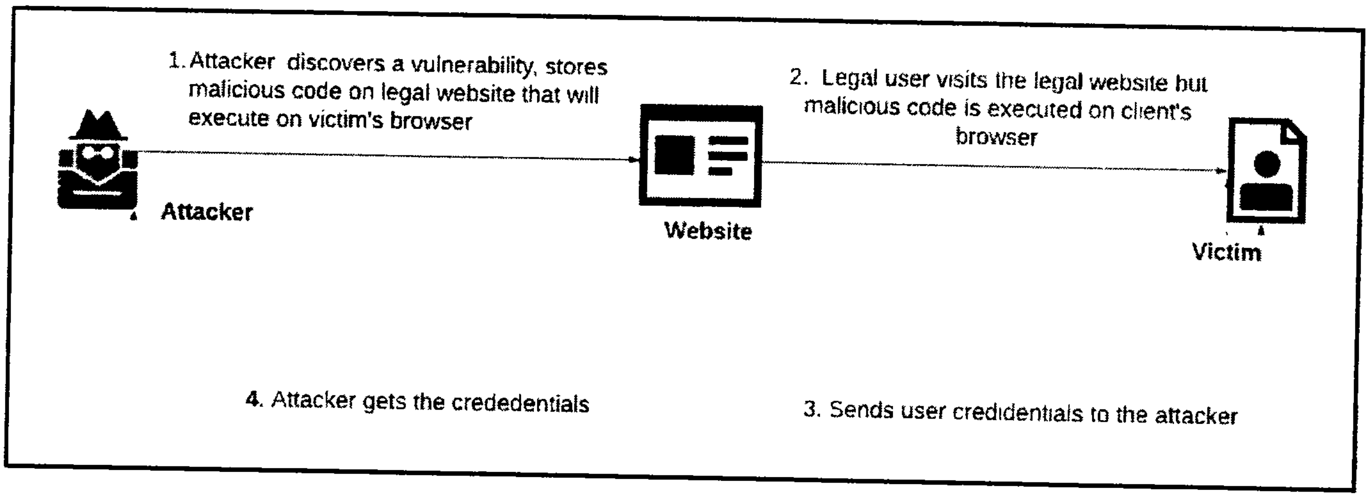

1. Attacker discovers a vulnerability, stores malicious code on legal website that will execute on victim's browser
2. Legal user visits the legal website but malicious code is executed on client's browser
Attacker
Website
Victim
4. Attacker gets the crededentials
3. Sends user credidentials to the attacker

Figure 2.2 Cross-site scripting attack

2.9.4. Time-of-Check to Time-of-Use Bug

TOCTOU (Time-of-Check to Time-of-Use) is a problem that occurs when synchronization is not properly programmed [37] [45]. The Time-of-Check to Time-of-Use (TOCTOU) bug is a class of race condition vulnerability that arises in concurrent computing environments, where multiple processes or threads can access shared resources simultaneously. In a TOCTOU bug, the system checks a condition (such as the validity or state of a file or resource) at one point in time ("time

Table 2.2 Security threats and mitigation techniques

Application Security Threat	Possible Mitigations	Limitations
SQL Injection	SQL Queries constructed from templates or using parameterized query to avoid SQL Injection	It requires significant restructuring at every querypoint, because the coder has to define the Basic structure of the SQL query.
	SQLCheck [34] verify that all input connected with aSQL query adheres to a correct syntactic structure produced through a parse tree, employ context- free grammar in addition to compiler parsing methods.	It gives false positives in certain cases [48].
	Partitioning program symbols in code and non-code values, where non-code values are "closed values" that have been thoroughly evaluated and do not require dynamic adjustments, while open values need dynamic assessment at run-time [48]	Difficulty to define the code, treating benign inputs as tainted and false inputs as trustworthy [48].
Buffer Overrun Attack	No execute bit also known as NX-bit[44][38]	It cannot prevent other forms of buffer over flow attacks
	Stack Smashing Protection (SSP) / StackGuard has stack canaries [44][38]	It cannot prevent other forms of buffer over flow attacks.
	Address Space Location Randomizer (ASLR)[44][38]	It cannot prevent other forms of buffer overflow attacks
ROP or return-to-libc threat	Dynamic Integrity Management (DynIMA) [38]	• Performance of the system can be affected significantly while monitoring for taints [38]. • Must identify meaningful values to go along with the present integrity measurement [38]. • The process image is altered by code rewriting, which may impede the execution and assessment of certified programs [38]
XSS	SessionShield [34] uses the HttpOnly flag and secure tags	It is difficult to correctly Detect authentication cookies [34].
TOCTOU	Taint tracing	Unsafe unless variable checking and use are made atomic [45].
Sensitive Data Exposure	TaintEraser [47]	Users must specifically identify sensitive information ahead of time [47].
Broken Authentication	Single Sign-on (SSO) [46]	• Allows usage to all assets when a client gets authorized, hence enhanced security, for example smart cards as well as one-time credential, is needed. • It may cause third-party websites inaccessible by utilizing social networking services within the firewall of the organizations.
	HTTPS [46]	Exploiting as well as breaching into any certificate issuer can endanger HTTPS.
	Encryption/hashing [46]	• Encryption of information is costly, Demands IT administration, and can lead to issues with current applications.

of check"), but by the time the system actually acts on that condition ("time of use"), the state of the resource may have changed, potentially allowing malicious exploitation. This gap between the check and the subsequent use introduces a window of opportunity for an attacker to intervene and manipulate the resource.

For instance, in a file access scenario, the system might verify that a user has the necessary permissions to access a file (time of check). However, if the attacker can quickly replace or modify the file before the system actually opens it (time of use), they can exploit the gap to gain unauthorized access or perform malicious actions, such as escalating privileges or corrupting data. TOCTOU vulnerabilities are particularly concerning in systems that handle sensitive operations like authentication, file management, or database transactions, where timing is critical [45].

TOCTOU bugs are especially prevalent in concurrent systems where multiple processes are executed simultaneously, as they increase the likelihood of timing discrepancies. Attackers can exploit these timing differences to intercept processes, replace files, or modify system states, leading to privilege escalation, data corruption, or unauthorized access.

Mitigating TOCTOU vulnerabilities requires ensuring that the check and the use of a resource happen atomically, meaning they are executed as a single, uninterruptible action. Techniques like locking mechanisms (to ensure exclusive access to resources during critical operations), using secure APIs that check and use resources in a single step, and minimizing the time window between the check and the use are common strategies to protect against these types of race conditions.

2.9.5. Sensitive Data Exposure

Sensitive data exposure occurs when confidential and private user information, such as location data, photos, login credentials, or documents stored on a website or mobile device, is unintentionally or deliberately exposed. This issue is one of the top ten vulnerabilities identified by OWASP, as poorly designed applications can inadvertently leak sensitive information [43]. For instance, Tom-Skype [47], a text editor, creates temporary copies of data, which can unintentionally expose sensitive information.

A notable prevention method is Taint Eraser [47], which aims to prevent the disclosure of private data by tracking sensitive information through dynamic taint analysis at the application level. Taint Eraser monitors and overwrites tainted data information deemed sensitive with arbitrary bytes before it is sent to the network or written to the local file system. It keeps a hidden list of kernel-level tainted elements in user space to track open files. However, a key limitation of this approach is that the user must manually identify confidential data at the beginning of the process, limiting its effectiveness in fully automated systems.

2.9.6. Broken Authentication and Session Management

Broken Authentication and Session Management occurs when an attacker impersonates a legitimate user by exploiting weaknesses in the authentication process [46]. These vulnerabilities can arise from both technological and human errors in authentication modules. Remembering complex, random passwords is a common challenge in systems that rely on traditional username and password authentication methods. Single Sign-On (SSO) solutions can simplify this process by reducing the need to memorize multiple passwords across various applications [46].

Maintaining secure user sessions and effectively managing user credentials are also crucial for protecting authentication data [46]. One effective solution is to replace custom authentication mechanisms with pre-built modules that have been thoroughly tested by large user communities. This offers a more reliable and secure alternative. Additionally, employing HTTPS and encrypting or hashing login information are essential practices for securing web applications against authentication failures [46].

Table 2.1 highlights various preventative measures for OWASP attacks, such as compiler-based approaches, operating system-level enhancements, hardware fixes, taint tracking, and binary rewriting or instrumentation techniques. Figure 2.4 presents potential mitigations across different tiers of a computer system, demonstrating how these strategies can minimize attack vectors. However, despite these preventive measures, vulnerabilities persist because attackers constantly find new ways to exploit weaknesses, especially in legacy code that lacks modern security mechanisms.

Effectively preventing such attacks requires identifying the specific features of programming languages that contribute to vulnerabilities. Furthermore, keeping developers informed about language-specific risks and secure coding practices plays a critical role in mitigating these security concerns. The next section will focus on essential security features in programming languages that can help combat various vulnerabilities.

2.10. Literature Review on Comparative Studies

There are comparative studies to compare programming languages including Garcia et al. [165] evaluation research compared C++, Java, Lisp, and Perl to check the level of support each language offer for generic. Aldrawiesh et al [163] evaluate programming languages like Java, C++, ANSI C++, as well as C# for creating Web services and distributed systems. Al-Qahtani et al. [49] compared popular programming languages for selecting an appropriate programming language when selecting multi-paradigm language.

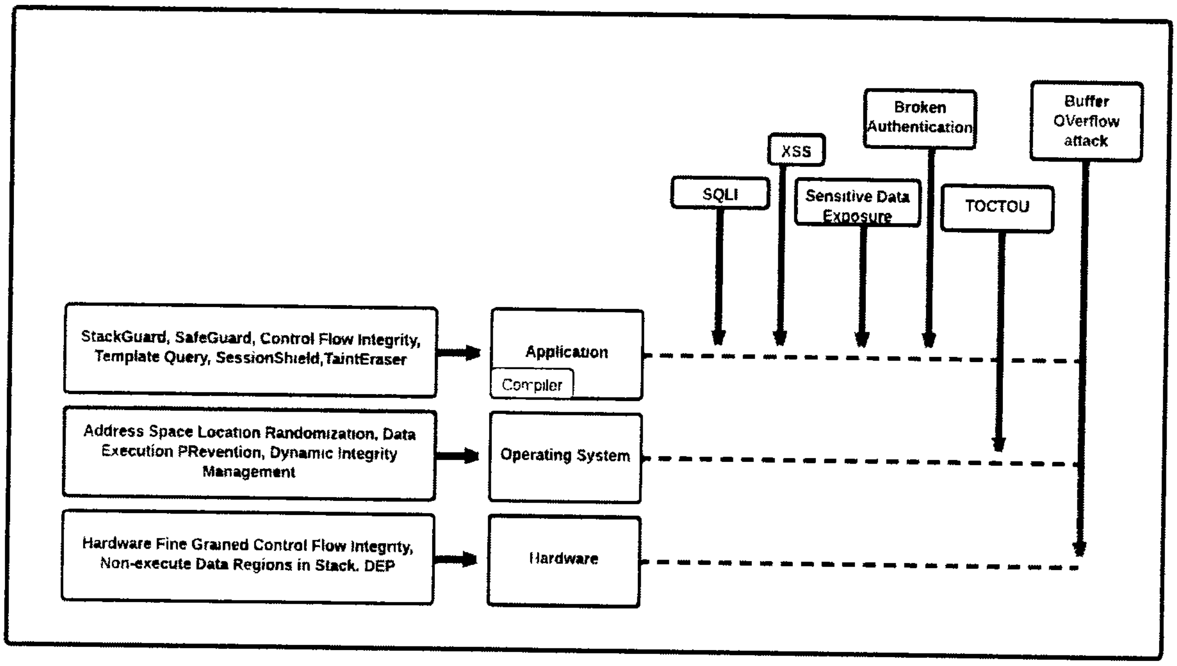

Broken Authentication
Buffer OVerflow attack
XSS
SQLI
Sensitive Data Exposure
TOCTOU
StackGuard, SafeGuard, Control Flow Integrity, Template Query, SessionShield,TaintEraser
Application
Compiler
Address Space Location Randomization, Data Execution PRevention, Dynamic Integrity Management
Operating System
Hardware Fine Grained Control Flow Integrity, Non-execute Data Regions in Stack. DEP
Hardware

Figure 2.3 mitigation techniques on computer system layers from literature review

2.11. Literature Review on Specification Languages

In both industry and academia, numerous specification languages have been developed to incorporate security requirements into software development processes. Some notable specification languages include: UMLSec [7] extends UML (Unified Modeling Language) profiles to incorporate security concerns and provides modeling techniques for specifying and analyzing security properties. Hayati, et al. [5] utilizes graphical models along with constraints expressed using Object Constraint Language (OCL) to capture and enforce security requirements. UWESecurity [6] employs a custom-made grammar to specify security requirements and integrates them into UML models. SysML-Sec [10] aims on security specify in the context of

Systems Modeling Language (SysML) and provides mechanisms for specifying security requirements in system-level models. Secure Descartes [11] is a declarative specification language is specifically designed for defining web access control policies. Ponder [3] [12] utilizes a custom-made grammar and formal methods to capture and enforce security policies in software systems. It is non-semantic based object-oriented. Declarative specification language models access control and authorization for distributed systems and networks. S- Promela [4] is an executable specification language that specifies access control mechanisms in embedded systems. AMF [5] is a formal specification language that specifies access control for authorization systems. Kong, et al. [15] proposes a technique for secure software development that integrates security requirements such as confidentiality, integrity, availability, and security threats into UML models. SysML-Sec [10], a real-time executable specification language, specifies confidentiality, integrity, and authentication for real-time embedded systems. Another model-driven security framework proposed by Deveci and Caglayan [23] specifies authentication, authorization, availability, and fair exchange. These approaches utilize various techniques, such as extending UML, using graphical models with constraints, utilizing formal specification languages, or developing custom-made grammars, to specify security requirements into the software development process. They choose different security requirements, including access control in distributed systems, input validation in web applications, and security specification for embedded systems. However, a quick evaluation of these specification languages shows that only a portion of security features are specified, lacking a toolchain for complete security requirements.

There are numerous comparison frameworks to evaluate the security coverage of specification languages such as Kasal et al. [16] proposed an evaluation framework that compares the specification languages with the following dimensions: paradigm, artifacts, formality, distribution, granularity, executability, verification, tool support, applicability, and security mechanism. There are no details on the dimensions of the security mechanism given bythe framework itself. Villarroel et al. [17] compare eleven methodologies incorporating security in their development phase. However, security requirements are not defined for comparison. The work proposed by Khan and Zulkernain [18] is comprehensive in comparing SSDLC, SSRE, secure software designs, and secure software guidelines. However, the security features proposed do not mitigate the afore-mentioned attacks. Karapati et al. [19] framework is a collection of orthogonal dimensions from previous studies. Moreover, the security requirements are not detailed and lack empirical validation. Lucio et al. [20] presented a comparison framework to evaluate the practical specification languages. Their comparison framework is based on Khwaja and Urban [9] [7] and Kasal et al. [16] frameworks. The dimensions of their framework include application domain, security concerns, specify approach, separation of concerns, model transformation, verification, validation, tool support, and traceability. The only aspect the framework offers is a list of security specifications written in a particular specification language that can be used for comparison. To analyze state-of-the-art specification languages. Nguyen et al. [22] recommended a systematic literature review (SLR). Security issues, model-to-model and model-to-text translation tools are some of the assessment criteria. Security concerns include confidentiality,

Table2.3 Literature Review on Programming Languages Comparative Studies

Programming Languages Comparative Studies	Type	Methodology	Criteria for comparison	Limitations
Croft. R. [157]	Empirical study	Compares 15 programmers' security discussion on Stack over flow and GitHub. Derived Taxonomy of 18 security challenges 6 categories	• Type safety, • type checking, • memory management	Security features are not precise
Al-Qahtani et al. [49]	Evaluation study to select an appropriate language for multi paradigm language	Compares programming languages for multi-paradigm so that best decisions can be made for programming language suitability.	secure programming practices is one criterion among other criteria • web applications development, • web services design and composition, • objectoriented-based abstraction, • reflection, aspect-orientation, • functional programming • declarative programming,	Security programming features offered by programming languages is discussed but it did not mention a single criterion for each programming language used for comparison. And also, the focus of the study is to select best language for mutli-paradigm
K. Aldrawiesh[162]	Evaluation of Programming Languages for Building Remote Systems in the Web Environment	offer an overview of the languages used for programming such as Java, C++, ANSI C++, and C# for creating Web services and distributed systems.	The evaluation criteria include • high integrity, • decentralized system, • simplicity • usability, • concurrency, • platform, • maintenance, and • dependability.	Security features criteria is not discussed in detail
R. Garcia[163]	Evaluation research	the author compared C++, Java, Lisp, and Perl	Reusability, mobility, dependability, readability, efficacy, existence of compilers and tools, familiarity, and expressiveness are the criteria for evaluation.	Security features criteria are not discussed in detail

Table 2.4 Literature Review on Existing Security Specification Languages

Author and reference	Type	Domain	Methodology	Security Features Coverage	Reason for selection
Secure Descartes [11]	declarative specification language	Embedded systems	Specifies security features using secure Descartes syntax	access control policy for web applications	- Executable specification language - Embedded system specifications
S-Promela [13]	executable specification language	embedded systems	Specifies security features using security pragmas such as integrity pragma, authentication pragma	Security features covered are - access control	- Executable specification language - Embedded system specifications
SysML-Sec [10]	the real-time executable specification language	Real-time embedded systems.	Specifies security features using SysML syntax	Security features covered are - Confidentiality, - Integrity - Authentication	- Executable specification language - Security features specifications for concurrent systems
Deveci, E., & Caglaya [22]	Model Driven Security framework	Information Systems	Defines security requirements, models using UML profile extension, OCL, and, Promela and model checking for validity	Security features covered are - Authentication - Authorization - Secrecy - Validation Security features covered are - Integrity - Audit - Fair exchange	Security features such as immutability is missing.
Damianou . N. [12]	Non-semantic-based object-oriented declarative specification	Distributed system and networks.	Specifies access control using Ponder syntax	Security features covered are - Access control and - authorization	- Declarative specification - Security specification
Al-Mekhlal and A Ali Khwaja [164]	Synthesis	Big data modeling characteristics	Develop a framework for characteristics related to big data modeling.	Security is not major concern in this paper	a framework that offers dimensions for big data modeling
Hayati. P et al. [5]	Extension of UML profile, graphical models	Web application	Specifies input validation using activity diagram	Input Validation	Security feature modeling

Table 2.4 Literature Review on Existing Security Specification Languages

Auth or and refer ence	Type	Domain	Methodology	Security Features Coverage	Reason for selection
Busch [6]	Extension of UML profile, graphical models	Web application	Specifies session management and authorization using UML diagram	Security features specified are Session timeout , access control, authorization	Security feature modeling Web application
Jürjens [7]	Extension of UML profile, graphical models	Distributed Systems	Specifies security tags specification and extension of UML profile	Security features specified are - Refinement, - Secrecy, - Secure Information Flow, - Integrity, - Authenticity, Freshness [27]	Security features are not defined in detail
Secure UML[8]	Extension of UML profile, graphical models	Distributed Systems	Specifies access control and ownership using UML diagram and OCL constraints	Security features specified are - Access control	It specifies only access control
Hu and Ahn[14]	formal specification language	Authorization system.	Specifies security features using RCL2000 specification and the transformation to alloy specification language	Security features specified are - access control	Model to code transformations. Model checking Security features specification

authenticity, authorization, availability, and integrity. However, the framework does not model significant security concerns such as input validation. Another SLR conducted by Van den Berghe et al. [21] compares specification languages based on security dimensions such as confidentiality, auditability, privacy, access control, availability, cryptography, and integrity. However, the security dimensions mentioned are comprehensive, but still, there are some of the security features not covered, such as input validation, type safety, and others. A quick review of comparison frameworks to evaluate overall security requirements coverage by various specification languages [16]-[23] show a lack of comprehensive security dimensions for comparison frameworks. The identified security features are not based on the security features collected from programming languages security functionalities that avert the above- mention attacks. We previously created a security feature framework (SEFF) to analyze security features in programming languages in our earlier work [50]. SEFF offers a precise set of programming language security features for preventing certain OWASP attacks. A similar methodology, but at a higher level, is required to evaluate specification languages for security feature coverage.

Table 2.5 Literature Review on Comparative Studies Evaluating Specification Languages

Comparison Frameworks	**type**	**methodology**	**Security feature coverage**	**Limitations**
Villarroel, et al.[17]	Comparison Framework	Compared 11 methodologies incorporating security in development phase	security dimensions for the database are compared access control	Security features are not precise for comparison
Kasal et al. [16]	Comparison Framework	Compares MDS usingdimensions • paradigm, • artifacts, • formality, • distribution, • granularity, • executability, • verification, • tool support, • applicability, • security mechanism.	Securitymechanism access control Security Protocol Intrusion Detection Mechanism	Security features for comparison are only those supported by specification languages used for comparison
Khan & Zulkernine [18]	SLR	evaluate SDLC, SSRE, secure software designs, and secure software guidelines	securitydimensions are taken from Jurjens (2002) • access control • Confidentiality • Integrity • Constraints	Security features are not precise
Karpati et al. [19]	Comparison Framework	Compares MDS using a collection of orthogonal dimensions from Previous studies	• Confidentiality • Integrity • Availability	Security features are not precise
Lucio [20]	Comparison Framework	Compares MDS using • application domain, • security concerns, • specification approach, • separation of concerns, model transformation, • verification, • validation, • tool support, • traceability	to identify specification languages' security coverage using securitydimensions as • Authorization, • confidentiality, • integrity, • availability	Security requirements are not precise

Table 2.5 Literature Review on Comparative Studies Evaluating Specification Languages

Comparison Frameworks	type	methodology	Security feature coverage	Limitations
Van den Berghe [22]	SLR	Security features are compared in specification languages	Security Features compared are • confidentiality • auditability • privacy • access control • availability • cryptography Integrity	Security needs are not precise such as to mitigate certain OWASP attack features such as immutability, output validation, secure error message is not specified.
Nguyen [21]	SLR	security concerns, model-to-model and model-to-code transformation tools.	Security Features compared are • Confidentiality • Authenticity • Authorization • Availability integrity	Security needs are not precise

2.6. Identification of Security Features from Literature

This section has identified security features from the existing comparison frameworks where security dimensions are given. Table 2.5 has laid the basis of selection of security features for this thesis. However, there are many sub-features or main features for these security features and these features are not sufficient and comprehensive to mitigate OWASP TOP 10 attacks. Some organization consider access control in functional requirement [14]. Security is difficult to quantify as there is no common agreed metrics for this purpose according to [41]. In [41], multi dimension verification aspects are taken and security is taken as extra functional requirement. The key targeted security services commonly represented as extra-functional aspects for verification are confidentiality, integrity, and authentication. Verification of security feature is highly dependent upon the types of the attacks and attacker model.

2.7. Security Framework

A security framework outlines the policies and procedures necessary for implementing and maintaining security controls. It clarifies the processes used to safeguard an organization against cybersecurity risks. These frameworks assist IT security professionals and teams in ensuring compliance and protecting their organization from cyber threats. Figure 2.4 show the proposed framework to detect, prevent, recover from mentioned cyber-attacks.

2.8. Security Features in Programming Languages

This section highlights important programming language features that can be leveraged for minimizing the above-mention attacks. These security features together make a Security Feature Framework to evaluate programming languages for security coverage. The process of creation of these features was first the features that are mitigating attacks are collected from mitigation techniques and the guidelines provided by languages manual. Then those features were given a name-based on their nature of similarity for example error message control prevents SQLI as verbose error message gives extra information to the unintended user. However, error message control is placed under error handling and log file protection due to their similar nature with logging. In SEFF, each sub-feature is placed under main feature because each sub-feature is relevant to the main feature and help provide the main feature functionality.

2.8.1. Error Handling and Logging Protection

Exceptions or errors are considered unusual incidents since they show that the internal states of the system have been damaged, and either the application program must be restored before the system can continue functioning normally, or an appropriate notification must be presented to the user prior to the system returns to normal operation [51] [52]. Handling Exceptions in programming languages eliminates data loss, which could jeopardize application security. Security sub-features of error handling include log file protection, log Information level, as well as error message control. Since these sub-features are related to logging and error handling that is the reason they are selected to be categorized under error handling and log file protection.

2.8.1.1. Log File Protection

The management of errors is frequently combined with the logging method. Logging is a tool for troubleshooting and diagnosis [51]. Failure of input validation, access control, authentication, system events, backend TLS connection, cryptography, along with improper session token usage as [54]-[56]. Keeping logs in simple text is a vulnerability that can provide intruders valuable details that can be used in OWASP attacks. The log must not contain any executable code [52].

2.8.1.2. Log Information Level

Logs capture data pertaining to runtime of the application that can be utilized for debugging and

Table 2.6 Security Features identification from literature review

Schemes	Security Features												
	Access control	**Intrusion Detection**	**Security Protocol**	**Confidentiality**	**Integrity**	**Availability**	**Authorization**	**Authentication**	**Privacy**	**Auditability**	**Cryptography**	**Secure Informatio**	**Sssion Timeout**
Villarro el et al. [17]	Yes	No	No	No	No	No	No	No	No	No	No	No	No
Kasal et al. [16]	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No
Khan & Zulkernine[18]	Yes	No	No	Yes	Yes	No	No	No	No	No	No	No	No
Karpati et al. [19]	No	No	No	Yes	Yes	Yes	No	No	No	No	No	No	No
Nguyen [21]	No	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
Van den Berghe[22]	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Jürjens[7]	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Secure Descartes [11]	Yes	No	No	No	No	No	No	No	No	No	No	No	No
Lucio [20]	No	No	No	Yes	Yes	Yes	Yes	No	No	No	No	No	No
Secure Descartes [11]	Yes	No	No	No	No	No	No	No	No	No	No	No	No
S-Promela [13]	Yes	No	No	No	No	No	No	No	No	No	No	No	No

diagnosis later on. A lot of logging takes up space, yet not enough logging makes troubleshooting difficult. However, correct logging procedures are not always followed. The log level functionality enables programmers to customize the amount of information recorded. Apache contains the logging library log4j [56] to regulate logging, and a log statement can use logging levels such as trace, debug, info, warn, error, along with fatal. Trace signifies the most verbosity, while fatal indicates the least amount of verbosity. Given the section in which logging comments are written, developers must identify the appropriate logging level [56]. The level of logging in the exception section becomes more verbose, and in the handler section it becomes less verbose, according to [56]. The logging module in Python is set to "warning" by default [57]. Log4cpp [55] is a logging library for C++, similar to Python. Log4r, a Ruby logging package that allows the developer to choose the severity of the data to be recorded [54] [55] [59]. Log4Net is a C# logging utility that use "debug" as the default logging level and allows the developer to modify it [52].

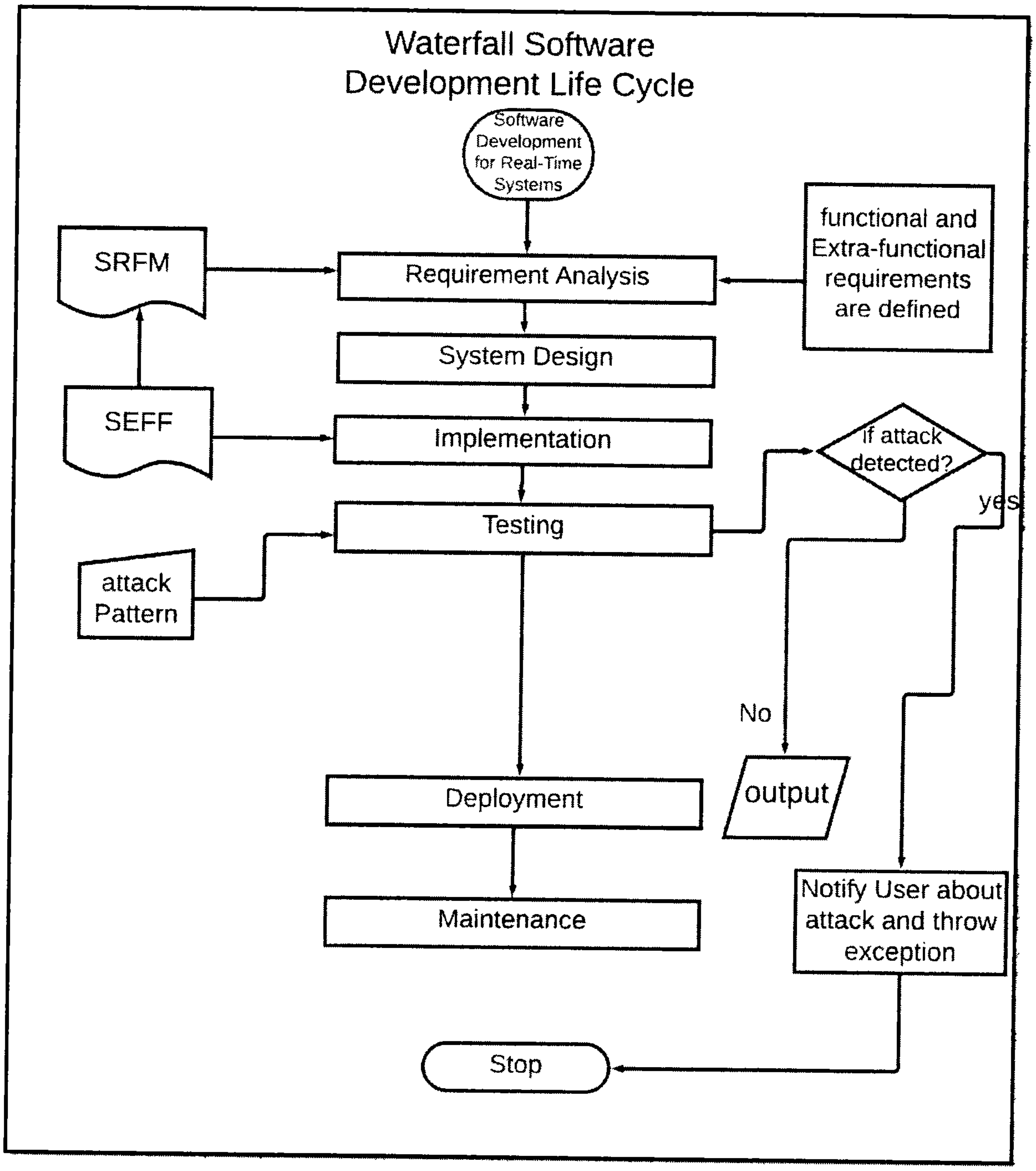

Waterfall Software Development Life Cycle
Software Development for Real-Time Systems
SRFM
Requirement Analysis
functional and Extra-functional requirements are defined
System Design
SEFF
Implementation
If attack detected?
yes
Testing
attack Pattern
No
Deployment
output
Maintenance
Notify User about attack and throw exception
Stop

Figure 2.4 Security Framework

2.8.1.3. Error Message Control

Attackers can take advantage of error message details, especially facts about incorrect usernames along with credentials [60], or the Stacktrace attribute, which reveals the sequence of calling functions [60].

2.8.2. Input Validation

Input-related vulnerabilities caused by flawed database queries, unverified user inputs, input buffer limit overflow, as well as a lack of output validation can all be avoided with effective data validation. This security feature includes sub-features that consist of database query security, user input security, input buffer size, along with encoded output. Since, these sub-features are related to inputs that is the reason these cares categorized under input validation.

2.8.2.1. Database Query Security

Almost all software programs require databases to store organizational facts, rules, configuration settings, and authorizations. The corporate logic is typically performed in the database layer. Database code problems can lead to severe weaknesses that can be leveraged by SQLI or simply other input validation exploits [52]. Although programmers can build stored functions, this does not ensure SQLI defense [1]. Almost all software programs require databases to store organizational facts, rules, configuration settings, and authorizations. The corporate logic is typically performed in the database layer. Earlier versions of the .Net framework might be more susceptible to SQLI, but newer versions of the .Net framework prevent the SQLI attack by data annotation library embedded in ASP.NetModel View Controller (MVC). The data annotation library is made up of regular expressions for data inspection [61]. C# support language-level query abstraction including queries over relational database and LINQ has removes possibility of SQLI [166]. A developer may want to employ SQL queries explicitly in code in some instances. Parameter collection in the .Net framework allows developers to impose type and length verification [63]. SQL Injection is difficult since the SQL server perceives input as a literal value instead of executable code [63]. Java, C++, Ruby, and Python all support the structuring approach that avoids SQLI. Developers can securely add borders to their query strings utilizing the templating capability, which is then filled by an application programming interface (API). This method is used by both Python's MySQLdb component as well as Java's java.sql component [61]. For constructing a SQL query, Ruby has SQLI helper functions and a prepared method [49]. Ruby has a taint tracking support [166]. In contrast, template strings remain strings and can be modified by combining unsafe invalidating input [61]. SQLI is possible in Java if input validation is inadequate [4]. As well as private data are all stored in log files [52]. Ruby logs each request submitted to a website by default, that can provide a significant security concern [53] [54] since the logs may contain passwords. Ruby provides a filtering function for deleting passwords and other sensitive data, but it requires the intervention of a coder [54]. Logs are kept in simple text in the Java, Python, and C#

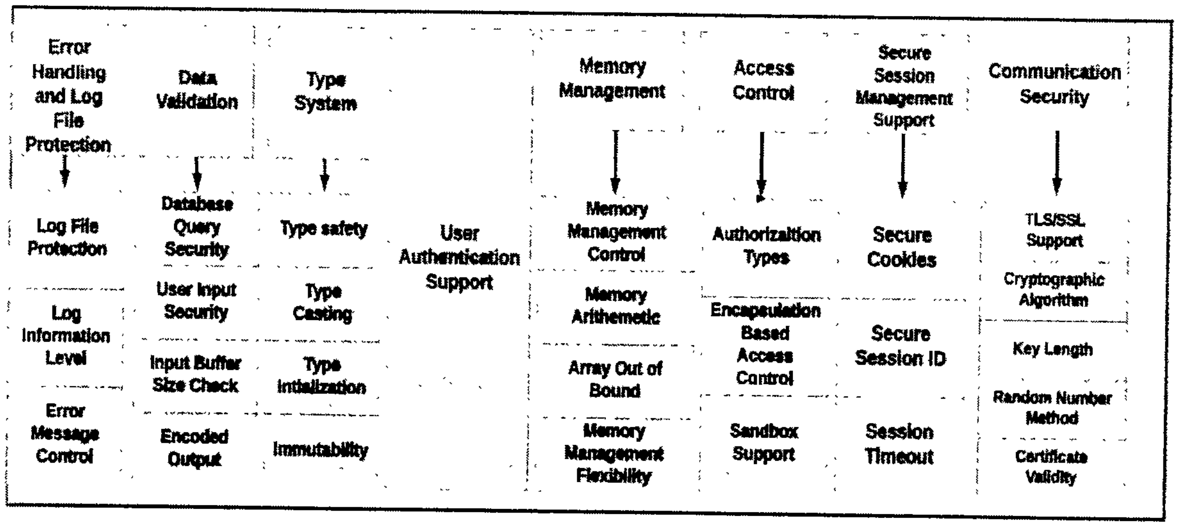


Figure 2.5 Proposed SEFF

2.8.2.2. User Input Security

User input safety deals with the validation of input submitted to an application. Most programs depend on user input [49] that can be dubious. SQLI, XSS, along with buffer overflow exploits are all a result of poor input checks [1]. Checking for illegal input is inadequate since adversaries can circumvent it. Programming languages should have ways for defining and limiting acceptable input data. To validate numbers and strings, regular expressions should be employed. Entering huge numbers, for instance, could end in a negative value if the application is not thoroughly validated, providing a security risk. Java contains a package termed as regular expressions for input validation. You can use this package to handle input using an advanced pattern-matching strategy [64]. C++, on the other hand, includes a standard library known as regex which allows regular expressions. However, because to the complexities of the regex library [65], handling input in C++ is problematic. For regular expressions, C# has a regex class. The programmer has to check the user input before generating the query. Otherwise, the query string cannot be verified once that been created [43]. Ruby supports taint monitoring but has several input validation problems [49]. Python solves input validation by creating three Web server modules: SafeString, SafeServer, along with SafeSql [43]. Standard string input is accepted by the SafeString class. SafeString class objects maintain track of the elements used to create unsafe strings. SafeServer bridges the developer's and server's code by altering its built-in method to receive and post user input in SafeString. SafeSql is a database communication tool for SQL. Python modules are language plugins, not built-in features, which can be avoided by changing private variables of objects or supplying some default settings [43]. Python treats user-defined strings in the same way that it treats built-in strings [43]. Ruby includes a method for escaping vulnerable SQL characters by default. This default technique, however, continues to support SQLI attacks. Ruby includes the ActiveRecord::Sanitize method[66] to sanitize potentially dangerous input strings. In computer languages, each input item must be checked against the permissible input buffer size, allowable symbols, along with input data types. [4], [63].

2.8.2.3. Input Buffer Size Check

The most prevalent source of buffer overflow attacks is failing to check the size of the buffer while copying data to buffers. Java offers bytecode checks for input buffer limit validation. C# has an in-built input buffer limit verification technique; for instance, arrays in C# start at zero index, and C# checks the input size to the maximum capacity of the buffer [65]. There is absence of boundary verification in C/C++, and vulnerable methods like gets as well as strcpy facilitate buffer overrun attacks [67]. Boundary validation is also supported at runtime in Ruby as well as Python [68], [69]. Array bound checking is provided by Java, that minimizes the bulk of buffer overflow vulnerabilities. On the other hand, Native Java programs can circumvent memory protection [4], [70], [71].

2.8.2.4. Encoded Output

Output encoding is a critical component for preventing XSS attacks. If the work includes any user input displayed to the Web browser, the output must be encoded [72] .Net has a technique for converting a string to an HTML-encoded value [72]. The Python programming language has functions for ASCII encoding [73].

2.8.3. Memory Management

Intruders may exploit memory management of an underlying language; therefore, it must be resilient. Memory-related attacks may occur as a result of improper memory management. Memory management sub-features may include memory management control, memory address arithmetic, array out-of-bounds check, as well as memory management flexibility. Since, these sub-features are related to memory management and that is the reason these are categorized under memory management feature. Moreover, Array out of bound is also a sub-feature as it can cause buffer overflow attack if the memory is not properly released or arrays are accessed past the bounds and that is the reason it is categorized under memory management.

2.8.3.1. Memory Management Control

Memory management is in responsible for memory assignment and cleanup. Object creation is not a problem, but object disposal is. C++ supports manual memory management. Garbage collection and automatic memory management are both provided by Ruby and Python [73]. Some problems related to manual memory management are removed by automated memory management, like dangling pointers along with double-freeing bugs [68] [74] [75]. Despite the fact that Java provides garbage collector, memory loss is common. If an exception mechanism fails to delete a registered object from the queue, the garbage collector has no way to clear its memory, causing servlet memory loss [51]. C# provides garbage collection through a generational technique. The garbage collection approach assumes that handling a portion of the heap is cheaper than handling the entire heap [75]. The garbage collector's timetable, on the other hand, is unpredictable, making it unsuitable for real-time applications.

2.8.3.2. Memory Address Arithmetic

Pointers in C/C++ enable unauthorized possession of memory segments. Any adversary with access to the pointer data can utilize arithmetic to jump to the exact location of the fraudulent payload triggering ROP. Pointers are no longer utilized in Java; hence pointer arithmetic is forbidden [65] [76]. In C#, pointer arithmetic is permitted, but only in the unsafe block. Pointers, on the other hand, are still used for connectivity as well as performance-critical hotspots [74].

2.8.3.3. Array Out of Bound Check

The size of an array must be verified to ensure that a buffer overrun does not occur. Because C/C++ lacks array bound checks, they are vulnerable to buffer overflow and ROP exploits. [65] [67] Byte code verification in Java verifies array boundaries. In C#, arrays begin with zero indexes, thus accessing an array beyond its boundaries raises the Index Out Of Range error [65]. Python and Ruby both have runtime checks for array out of bounds [49].

2.8.3.4. Memory Management Flexibility

Performance-oriented developers typically utilize custom memory allocation. C++ allows for customized memory allocation through the use of new along with remove overloaded operators. The C language includes malloc and free for custom memory allocation. Regions are used by several custom allocators to offer performance. Regions are currently gaining attention as a possible substitute to garbage collection. Although regions provide tremendous efficiency, they need a developer keeping all memory associated with a region until the final item of the region is deleted [49]. Unlike Java and C#, C++ utilizes both the stack and the heap for pointers; all instances are references and created on the heap. In Java, Python, Ruby, plus C#, garbage collector deallocates references [64] [77] [78]. Python offers developers to customize memory acquisition and deletion [74]. Ruby additionally offers this sub-feature by manually deleting files or network variables to ensure that the block is linked to the begin block [75].

2.8.4. Access Control

Limiting access privileges and purpose of activities can assist in avoiding application-level flaws that include TOCTOU, buffer overrun, along with sensitive information disclosure attacks. Authorization types, encapsulation-based access control, even sandbox support may be combined as sub-features. Since these sub-features are related to access control and provide somehow access to the objects based on some privileges that is the reason these are categorized under access control.

2.8.4.1. Authorization Types

The type of procedures that certain individuals can perform is determined by authorization rules. Access control refers to the policy or capability that allows, denies, as well as limits access to a system. Every individual trying to get entry to a system must initially be authorized so that particular rights can be assigned [12] [43] [79]. A developer should remember the concept of minimal privilege while assigning responsibilities within a database application. An attacker cannot access the remaining software in this manner. File access, network access, user interface access, and platform-specific access are all examples of Java language permissions [80]. In Java, the least privileges can be achieved statically through policy files and dynamically using the Java security access controller method [71]. Java annotations are an alternative method of restricting permission in Java [63]. C++ and C# support restricted access to system assets [81] [82]. Ruby allows for controlled access to system resources; nevertheless, this is reliant on how the programmer implements it [80]. Python has no way for controlling resource access because the rexec() module was removed in Python 2.3 [59]. Python includes a method for controlling file access [73].

2.8.4.2. Encapsulation-Based Access Control

Encapsulation-based access control governs the accessible scope of an item that might be public, protected, or private. A programming language defines a resource as a private data and access to these private data is provided through public methods. In Ruby, Java, C++, as well as C#, classes are utilized for achieving procedural encapsulation [84]. Python prohibits private, protected, as well as public visibility scopes. By default, all objects are public, but they can be kept private by adding an initial underscore prior to the variable name, like _var [84]. A different method to describe an asset is through type-based encapsulation. The untrusted code is given a resource pointer; however, the pointer type is abstracted, making direct access to the resource impossible. The code is able to use the asset by triggering one of the methods given in the abstract type signature [67]. Polymorphism is a characteristic of the Python, C++, C#, along with Java [64] [77] [78] [84].

2.8.4.3. Sandbox Support

It is a security method that evaluates unverified or untrusted code for viruses or malicious code without exposing the whole system in danger [85] [86]. Sandboxing grants privileges to program code at the language level [80]. The Java sandbox runs unsafe code with the lowest level of privilege feasible, and if the sandboxed code tries to enter security-sensitive program, the Java Runtime Environment (JRE) returns an exception. Java [80] supports file, socket, user interface, as well as platform-specific access controls. C++, C#, Ruby, and Python, include a sandbox for testing untrusted code. The disadvantage of this sub-feature is that it can abuse the user's privileges. It has the capacity to run any program inside the controlled code runtime environment [85].

2.8.5. Type System

A programming language with strong type support may resist buffer overrun, ROP, and TOCTOU exploits. Sub-features are type safety, type casting, and type initialization, along with immutability. These sub-features are related to type system and every programming languages provide these however, immutability is choosen to be the type due to functional programming languages provide this type system and imperative programming languages provide weaker immutability [88] [90].

2.8.5.1. Type Safety

A data type is a programming resource limit that describes the set of valid values that conform to the type [87] [65]. Strong typing reduces the possibility of buffer overruns and ROP exploits [87]. Assume Employee is the base class, from which the subclasses Manager, Clerk, as well as Salesman descends. Understanding the type of identification is crucial in this case in order for employees to be permitted access permissions [12]. Programming languages are categorized as statically or dynamically typed [87]. Java and C# are examples of statically typed safe, meaning that data type verification occurs at the time of compilation. Pointer mathematics is only permitted in unsafe blocks in C#. C++ is a computer language that is unsafe and statically typed, and it enables pointer arithmetic. Python and Ruby are object-oriented programming languages with duck typing (dynamic typing). The type of the object is checked during runtime. Duck typing is additionally referred to as "soft typing" [87].

2.8.5.2. Type Casting

Particular conversions among types, including double to Boolean, are either automatically or explicitly allowed. Implicit type casting or automatic is a potentially dangerous casting approach. It is not supported by Python, Java, or C#; however, it is supported by Ruby [65],[87]. C/C++ are likewise weakly typed and permit implicit type casting [65]. Implicit casting relieves programmers of the need to write redundant type conversion code. The same functionality, however, can result in data loss and security risks [87].

2.8.5.3. Type Initialization

It means that a variable should be initialized after it is declared. According to the language description. Variables in C# are immediately set to zero and references to NULL, however in C++, a variable created without a start value contains garbage. The system behavior is undefined if the language does not provide any default initialization. To safeguard against buffer overflow, set the variables to their original values and pad the over-read value with zero.

2.8.5.4. Immutability

The item is then referred to be an unchanging or unchangeable object. In an application program, its content cannot be changed [88]. Mutable objects, on the other hand, may alter after or while a method or constructor call is running. Immutable objects in concurrent programming languages ensure thread safety [80] [89]. The final keyword in Java can be utilized to make a class immutable. If the mutable input is stored in a field of an object, a malicious user may exploit a race condition in the outer class. In Java, for example, a TOCTOU inconsistency can be exploited if a mutable input contains one value through a Security Manager inspection and a different value when the input is later used. In C++, the const keyword declares immutable fields. A different reference to the same object fails to offer a guarantee for an object declared with const [88]. In Java, the final keyword offers reference immutability that is less powerful for the reason that a reference cannot be transferred to refer to another object. The value of the mentioned item, on the other hand, may vary [89]. In C#, immutability is provided by read-only keywords. Declare read-only variables in the constructor or as part of object setup. A read-only reference prevents an item from being altered through a single reference however not necessarily by all of its references [89]. C# also has a const feature, which differs from C++. In C#, const must be statically defined; alternatively, it causes a compile-time error [88]. Ruby uses freeze method to make an object immutable, and there is no way to modify it [69], whereas Python provides both immutable and mutable objects for standard as well as user-defined types [90]. Strong or weak immutability can be provided by programming languages. If an object has single immutable field and the rest are changeable, it is regarded weakly immutable; if an object contains all immutable fields, it is regarded as strongly immutable. Immutability is lower in imperative programming languages that include Java, C++, Ruby, and C# than in functional programming languages [88] [90].

2.8.6. User Authentication Support

User authentication refers to the process of verifying that an individual or organization is who they pretend to be [52] [79] [91]. It can take several forms, including username/password, biometric authorization, SMS single-use passwords, and security frameworks such as Single Sign-on (SSO) as well as Open Authorization (OAuth) [63]. Almost all secure applications use a common authentication approach that involves a user ID and a secret code. Extra credentials or multiple-step authentication is necessary in security-critical applications including online banking. CAPTCHA technology has the potential to be used to stop spam along with automated OWASP attacks. Java Authentication and Authorization Service (JAAS) component is included in the Java platform version 1.4. JAAS takes administrator and user credentials. JAAS also supports form authentication, which uses the HTTP POST method to deliver the login credential sand password for verification [92], [93]. Ruby includes a plug-in for Restful-Authentication [63].C# uses predefined authentication modules and allows for the registration of custom modules [94]. Python and C++ authentication components are available [95], [96]. The primary goal of an attacker is to compromise authentication mechanism. Any defect in the design architecture may end in authentication bypass, granting application access.

2.8.7. Session Management Support

Support for this feature could aid in the fight against XSS as well as broken authentication vulnerabilities. Sub-features that involve secure session ID, secure cookies, along with session timeout could be included because these sub-features help in secure session management.

2.8.7.1. Secure Session ID

It handles the authenticated session of the user after successfully logging into a Web application. In a Web application, a user can ask for multiple Web pages, while the web itself might get several requests from unidentified users. A legal individual provides an authentication token or a session id to a Web application to initiate a session. SIDs are used to maintain the status of a stateless HTTP protocol that connects to the internet. SID is an alphanumeric code generated by a website and sent to its user via cookies. ALL modern web languages like PHP, ASP, and JSP, facilitate session management [34] [66]. JSESSIONID (J2EE) and ASP.NET_SESSIONID(ASP.NET) are language-specific terms for the session ID. SID is the primary exploitation vector since precise SID acquisition and replay enables an intruder to instantly log into a vulnerable website. The most common method of getting SID is through an XSS attempt [34]. SID must be16 bytes (128 bits) long to prevent against brute force attacks [97]. In Ruby, SID is a 32-byte long MD5 hashing value [63]. The default name of the framework must be changed by the developer to something more generic. Cryptographic hashing algorithms including SHA1 (160 bits) should be used to build strong SIDs [53]. C++ also provides the default session name for SID [98].

2.8.7.2. Session Cookies

To protect against session attacks, processes including tagging cookies with a secure flag are in place. The secure cookie attribute stops this cookie from being attached to HTTP requests, so minimizing eavesdropping. Yet another protection is by marking the authentication cookie with the HttpOnly attribute, which prohibits scripts written on the client side from accessing it and therefore preventing XSS attacks. [97]. The default status of HttpOnly cookies in Java, Python, C++, and also C# is false, and it must be explicitly changed to true [59] [100]-[103]. The Httponly flag is enabled by default in current Ruby versions [100].

2.8.7.3. Session Timeout

Session handling code must give a session timeout. Failure to set the session timeout leads to XSS attack since SID can be reused [53]. A cookie can be used multiple times before it expires. If no expiration date is selected, the cookies will be deleted when the Internet Explorer is closed[53]. The typical session timeout in Java and C# is 30 minutes [53] [59] [99].

2.8.8. Communication Security

It may protect sensitive data from being exposed. Sub-features include SSL/TLS version support, cryptographic algorithms, key length, random number method, along with certificate validity because these sub-features are relevant to communication security and these are put together because these aid in providing data integrity.

2.8.8.1. SSL/TLS Version Support

Transport layer protocols include Secure Socket Layer (SSL) and Transport Layer Security (TLS). The industry standard for secure client-server communication is presently SSL [3]. The TLS protocol is the SSL standard created by the Internet Engineering Task Force (IETF). It is SSL [39] [91]'s replacement. A web page that starts with "HTTPS://" in its Uniform Resource Locator (URL) has SSL/TLS encryption. There are five versions of the SSL/TLS protocol that are regularly used: SSLv2, SSLv3, TLSv1, TLSv1.1, along with TLSv1.2. The latest and safe version is TLSv1.2. Both SSLv2 along with SSLv3 are insecure. Installing the most recent software version is typically preferred. Sometimes, the server only supports TLSv1.2. The server must also handle older versions of SSL/TLS in order to prevent blocking clients who use them [41]. The developer must develop code in C to manually block SSLv2 and SSLv3, and to support higher versions [41]. .Net supports SSL v 2.0 and 3.0 in backward compatibility, as well as TLS v 1.0, 1.1, 1.2, and 1.3 [105]. There is no default version supported by .Net. TLSv1.2 is supported as a default version in JDK8 [106].

Table 2.7 Security Features and sub-features mitigating cyber-attacks

Feature	Sub-Feature	Attacks Mitigated
Error Handling and Logging Protection	Error Handling and Logging Protection	Sensitive Data Exposure
	Log Information Level	Sensitive Data Exposure
	Error Message Control	SQLI,
Input Validation	Database Query Security	SQLI
	User Input Security	XSS Attack, SQLI
	Input Buffer Size Check	Buffer Overflow Attack, SQLI
	Encoded Output	XSS
Memory Management	Memory Management Control	Buffer Overflow Attack, ROP Attack
	Memory Address Arithmetic	Buffer Overflow Attack, ROP Attack
	Array Out of Bound Check	Buffer Overflow Attack
	Memory Management Flexibility	Buffer Overflow Attack, ROP Attack
Access Control	Authorization Types	TOCTOU Attack
	Encapsulation-Based Access Control	Sensitive Data Exposure
	Sandbox Support	Buffer Overflow Attack, Sensitive Data Exposure
Type System	Type Safety	Buffer Overflow Attack, ROP Attack
	Type Casting	Buffer Overflow Attack
	Type Initialization	Buffer Overflow Attack
	Immutability	TOCTOU Attack
User Authentication Support		XSS
Web Session Management Support	Secure Session ID	XSS Attack, Broken Authentication
	Secure Cookies	XSS Attack
	Session Timeout	XSS Attack
Communication Security	SSL/TLS Backward Compatibility	Sensitive Data Exposure,
	Cryptographic Algorithms	Sensitive Data Exposure
	Key Length	Sensitive Data Exposure
	Random Number Method	Sensitive Data Exposure
	Certificate Validity	Sensitive Data Exposure

2.8.8.2. Cryptographic Algorithm

Data exchanged between two different entities is continually vulnerable to disclosure. Cryptographic techniques play an important role in keeping sensitive information from being leaked. Asymmetric and symmetric algorithms are the two types of cryptographic algorithms. The X.509Certificate class in C# contains methods for supporting X509Certificates [107]. CryptoAPI or DPAPI are common C++ encryption APIs for Win32, and OpenSSL or CSS are popular Linux cryptography APIs [81]. The Java Cryptography Extension (JCE) extends Java with a number of encryption algorithms [81] [92]. The Ruby enables asymmetric as well as symmetric encryption through the OpenSSL library [4]. Cryptographic modules in Python include cryptographic.io, M2crypto, PyCrypto, PyNaCl, along with Keyczar, however they all suffer from security problems such as certificate trustworthiness, secure key preservation, and simple text key storage [108].

2.8.8.3. Key Length

Symmetrical cryptography techniques necessitate the use of the same key by both the sending and receiving parties. In contrast, asymmetric cryptography algorithms require the usage of both public and private keys. The key length of a symmetric encryption scheme must be at least 128 bits in order for it to be difficult to predict by brute force. Rivest, Shamir, and Adi's (RSA) 512-bit key size was once thought to be safe, however it is now easily guessable by brute force. The following goal is 1024 bits. As a result, the asymmetric key should be at least 2048 |bits long [41]. The key sizes in JCE are big [92]. Long key lengths are also supported by Ruby, C#, along with C++ for both asymmetric as well as symmetric encryption [81] [107]. The Python 1024-bit key size is weak for the RSA algorithm in M2Crypto as well as PyCrypto [108].

2.8.8.4. Random Number Method

High entropy (randomness) achieved by a pseudo-random number generator (PRNG) is a crucial aspect of the cryptographic technique. A PRNG is necessary to produce random, difficult to estimate keys, C++ provides the CryptoAPI. A suggested PRNG for CryptoAPI is RNGCryptoServiceProvider [81]. The PRNG in Java must be SecureRandom [81]. The SecureRandom.hex module [41] is used by Ruby to produce a random integer. Python contains a random class that produces arbitrary numbers; however, it lacks entropy. Although the os.random component produces genuine random numbers, it is hard to use [109].

2.8.8.5. Certificate Validity

The SSL certificate concept secures key exchange in asymmetric encryption. An SSL certificate is an open key for the host machine that has been verified by the Certificate Authority's (CA) secret key. The server transmits an SSL certificate to the client, but the user must use public keys to validate it. There are several clients and Internet Explorer to choose from. Because any CA can sign a certificate for any computer, special software detects whether a certificate of the device is unusual. For instance, Firefox has a Cert Patrol component [41]. Certified certificates will not disclose critical information. Programmers frequently omit SSL implementation and prefer to depend on standard SSL libraries for example OpenSSL, JSSE, GnuTLS, as well as CryptoAPI [3].

The SSL/TLS language libraries OpenSSL, cURL, JSSE, GnuTLS, as well as CryptoAPI contain a check for verifying the organization that issued the certificate authority. To validate the chain of trust, the server gives a certificate authority identity as well as the higher level up to the root certificate authority. Each certificate is signed by the CA directly above it. The client must check the certificate of the server for deadline, and the CA in the "basic constraint" field has the CA bit set. After confirming the chain of trust, the user must certify the server's identity. Upon creating a collection of server identities, the user compares full DNS name with each of the server names using the string comparison function.

Certificate revocation is required for appropriate certificate verification. OpenSSL supports certificate revocation; however, a certificate revocation list (CRL) must be supplied by the user. The JSSE validates its own CRL. SSL of Python does not support checking CRLs. Several X.509 extensions include security-critical information including name restrictions, key usage, and certificate policy. A list of CA names that the sub-CA can validate is included in the name restrictions. The vital usage is the key that permits the CA to sign certificates. OpenSSL validates the name constraint erroneously. There is no mechanism in cURL for setting the certificate strategy. The JSSE library cannot verify the hostname and has a problem in certificate validity [3]. Certain Python modules fail to verify the hostname as well; therefore, the application must verify hostname [105]. OpenSSL offers chain-of-trust validation but not hostname validation [3]. In Ruby OpenSSL [66], verification of certificates is disabled by default. Almost all of Python cryptography APIs fail to perform effective certificate validation [108]. Table 2.5 highlights the security features and their sub-features, as well as the attacks that these features/sub-features prevent. The features and sub-features are listed in Columns 1 and 2, respectively. Column 3 lists threats that may be prevented by the security features/sub-features.

2.9. Summary

This chapter had defined various OWASP attacks brought on by coding errors. To lessen the security load on the developer and security programming deficiencies in order safeguard against various security threats, specific security features have been defined. A SEFF is suggested in the study. Five well-known programming languages in academia were evaluated using the SEFF. C++ had the least security feature coverage that is 47.22%, according to the recommended feature coverage percent. The Java security feature loopholes assessment were used as a case study, and a few suggested solutions from the literature were used to close the loopholes.

Chapter 3

PROPOSED FRAMEWORK

3.1. Proposed Framework for Real Specification (REALSPEC)

The Proposed Framework for Real Specification (REALSPEC) aims to provide a structured approach to evaluating and ensuring the security features of programming languages. This framework emphasizes the importance of robust security measures in software development, particularly given the increasing sophistication of cyber threats. Below are key aspects of the REALSPEC framework:

The REALSPEC framework outlines specific criteria that programming languages should meet to ensure adequate security support. Ensuring that user inputs are properly checked to prevent attacks such as SQL Injection (SQLI) and Cross-Site Scripting (XSS). Implementing reliable user authentication methods, including multi-factor authentication and secure password storage. Proper handling of user sessions to prevent session hijacking and replay attacks. Minimizing information leakage through careful management of error messages and logs. The framework integrates the Security Features Framework (SEFF) to provide a benchmark for assessing the security capabilities of programming languages. High-level security attributes that programming languages should support, such as type safety, memory management, and encryption. Specific aspects within the primary features that further define the security capabilities. For example, under type safety, sub-features may include type casting and immutability.

The REALSPEC framework serves as a guide for both software developers and language designers. Developers can use it to assess the security posture of their chosen programming language and identify areas that require improvement. Language designers can reference the framework to ensure that newly developed languages include necessary security features from the outset. The framework is designed to be dynamic, allowing for updates and enhancements as new security threats emerge. By incorporating feedback from real-world applications and security incidents, REALSPEC remains relevant and effective in addressing the evolving landscape of cybersecurity.

To facilitate practical implementation, the REALSPEC framework includes: Examples of successful implementation of the framework in various programming languages.

Recommendations for tools and resources that can help developers assess and improve security features in their applications. Ultimately, the implementation of the REALSPEC framework aims to strengthen the overall security of software applications, reducing vulnerabilities and protecting against potential attacks. By providing clear guidelines and benchmarks, the framework contributes to a culture of security awareness in software development.

3.2. Evaluation of Programming Languages Using the SEFF Framework

The Security Features Framework (SEFF) is a systematic approach designed to evaluate the security features of programming languages. With the rapid evolution of technology and the increasing frequency of cyber threats, assessing the security capabilities of programming languages has become critical for software development. This note delves into the evaluation process of programming languages using the SEFF framework, exploring its significance, methodology, and implications for developers and organizations.

The SEFF framework serves as a benchmark for evaluating the security features provided by programming languages. The framework allows developers and organizations to identify the security capabilities and limitations of various programming languages. By pinpointing potential vulnerabilities, the SEFF framework facilitates the enhancement of an application's security posture. Language designers can leverage SEFF to ensure that their programming languages incorporate essential security features from the outset.

The SEFF framework is structured into several key components. High-level security attributes that are essential for programming languages.

i. **Type Safety**: Ensures that variables are used in a manner consistent with their declared types, reducing risks such as buffer overflows and type-related vulnerabilities.

ii. **Memory Management**: Addresses how memory is allocated, used, and deallocated, helping to prevent memory leaks and buffer overflow attacks.

iii. **Input Validation**: Validates user inputs to protect against common vulnerabilities like SQL Injection and Cross-Site Scripting (XSS).

These are specific aspects that further define the primary features. For example:

Under Type Safety, sub-features may include type casting and immutability. Under Input

Table 3.1 Proposed Security Feature Framework

Security Feature	Sub-Feature	Possible Values
Error Handling and Logging Protection	Log File Protection	- Fully Supported: Encrypted; - Partial Support: partially encrypted - No Support; Non-encrypted
	Log Information Level	- Fully Supported; The language sets appropriate log message level/type control by default; - Partial Supported, the language provides the library, but the developer has to choose the appropriate level; - No Support, the language does not permit to set log message level/type control
	Error Message Control	- Fully Supported; The language less verbose error message level by default; - Partial Support; The language provides library but depends upon the developer to choose appropriate level; - Not Supported;
Input Validation	Database Query Security	- Fully Supported: safe SQLQuery; - Partially Supported: safe SQLQuery; - No Support: No safety
	User Input Security	- Fully Supported: The language itself fully keep tract of user inputs; - Partial Supported: The language provides all possible user input validation but depends on the developer knowledge; - No Support: The language does not provide any data validation support
	Input Buffer Size Check	- Fully supported: input buffer size check is provided by the language; - Partially Supported: checks for input buffer size are written by the developer - No Support: The language does not provide checks for input buffer size neither the developer applied this check
	Encoded Output	- Fully Supported: the language encodes the output by default; - Partially Supported; - Not Supported;
Memory Management	Memory Management Control	- Fully Supported: garbage collection; - Partially Supported: programmer manually write code; - No Support;
	Memory Address Arithmetic	- Fully Support: Allow memory calculations; - Partially allow this though language library - Not Supported: Does not allow memory arithmetic;
	Array Out of Bound Check	- Fully check for array bounds; - Partially check for array bounds - Does not check for array bounds
	Memory Management Flexibility	- Fully Supported: gives complete memory management versatility; - Partially Support: - No Support;
Access control	Authorization Types	- Fully Supported: minimal access control privilege; - Partial Supported: developer has to write code; - No Support;
	Encapsulation-based Access Control	- Fully Supported: Strong encapsulation; - Partially Supported: Weak encapsulation; - Not Supported: No encapsulation
	Sandbox Support	- Fully Supported: Least access are given to unverified code; - Partially Supported: User access are provided to unverified code, - No Support: System access are provided to unvalidated code
Type System	Type Safety	- Fully Supported: Static type safe - Partially Support: Dynamically typed (duck typed); - No Support: Static type unsafe
	Type Casting	- Fully Supported: Explicit - Partially Supported: Implicit, - Not Supported; No type casting permitted

Security Feature	Sub-Feature	Possible Values
Type System	Type Initialization	- Fully Supported: if the language provides initial values for variables - Partial Supported: if the language does not initialize objects and developers has to set it; Not supported: if the language set garbage value for the variables
	Immutability	- Fully Supported: Strong immutability; - Weakly Supported: Weak immutability - Partial Supported: read-only, mutex, const or final variables; No Supported: no immutability variables
User Authentication Support		- Fully Supported: language provides all types of authentication features; - Partial Support: Language gives some of the user authentication methods; Not Supported;
Secure Session Management Support	Secure Session ID	- Fully Supported: cookies that have authentication data are stored by arbitrary name; - Partial Supported: developer has to change the name of authentication cookie; - Language-specific name for the authentication cookies;
	Secure cookies	- Fully supported: HttpOnly flag default status? True, - Partially supported: HttpOnly flag default status? False, but can be turned on; Not supported: HttpOnly flag cannot be altered
	Session timeout	- Fully Supported: Default timeout of session set by the language, - Partially Supported; Not Supported; No developer must set the expiry time
Communication security	SSL/TLS backward compatibility	- Fully Supported - Partially Supported; - Not Supported
	Cryptographic algorithms	- Fully Supported: there is a library of standard algorithms and custom algorithms are permitted - Partially Supported: some algorithms are supported; - Not Supported: the language does not have any strong algorithms
	Key length	- Fully Supported: Secure key length is supported; - Partially Supported: secure key length is supported; - Not Supported: Secure key length is not supported
	Randomnumber method	- Fully Supported: gives strong entropy; - Partially Supported: Supports strong entropy but hard to use it; - Not Supported: Does not help - strong entropy
	Certificate validity	- Fully Supported: Comprehensive checks are supplied; - Partially Supported: some checks are given; - Not Supported: certificate verification is given

Validation, sub-features may encompass database query security, user input security, and output encoding. Value Assessment: Each primary and sub-feature is assessed against a set of predefined values or criteria, providing a clear understanding of the programming language's security capabilities.

3.3. Security Features Gap Analysis for the Java Programming Language

As an example, this section examines the security feature problems in the Java programming language through the security feature evaluation results in Table 3.3. Although Java is regarded as a secure programming language, it nevertheless has several security flaws, as seen in Tables 3.3. This section discusses the shortcomings in the security aspects of the Java programming language, as well as any potential remedies to fix these loopholes.

3.3.1. Log File Protection

Gap Analysis: Because log files include vital event data, they might be a major target for an attacker. Storing log files in plain text can lead to the disclosure of sensitive information [52]. Table 4 indicates that log messages in Java are stored in plain text.

Review of Possible Solutions: Log file encryption is a great way to protect confidential data. Intruders will be unable to access or change the log files without a secret key for encryption. Logfiles should be protected prior to being written to storage. Java logging procedures must include the ability to secure log files. It is possible for the program to generate decryption keys for logfiles. One approach includes first processing the data that needs to be stored in a log file to determine its level of sensitivity according to some established standards, and then protecting only those parts of the information that meet the criteria while keeping the remainder of the logfile unencrypted [110]. The data must subsequently be decrypted to be able to be examined by the log file viewers [110]. Using JAR programmable characteristics, Sundareswaran et al. [111] provide an object-centered approach to cloud data security. Their logging method is protected by regulations and user data to ensure that obtaining user data starts authentication along with automated logging within the JARs [111]. As a cheap option to maintaining log records, Ray et al. [112] advise using the cloud, highlighting the need for the log service must be able to store data in an ordered manner, offer effective data retrieval, and additionally to safe log management. Ray et al. [112] emphasized correctness, tamper resistance, verifiability, secrecy, along with privacy as essential characteristics of a secure logging service. They emphasized using cryptographic techniques for all aspects of log management, such as log collecting, transportation, storage, as well as retrieval. On the other hand, if cryptography is carried out on a computer that is available to intruders, they might be able to recover the data [113]. Schneier and Kelsey [114] suggested using a reliable machine or a network of unfriendly machines for collaborating on the encryption keys. As a consequence, the log files can be protected by the unauthorized system without being able to retrieve them.

3.3.2. Input Buffer Size Checks

Gap Analysis: Java allows native code, involving C code, to be run for performance benefits like using cryptography packages or running system calls that are not permitted in pure Java programs. The Java native code interface was designed with inadequate consideration for security. As indicated earlier in Section 2.3 and Table 3.3 Java native code fails to verify buffer length, which could end up in a buffer overrun attack [70]. Java programs are susceptible to SQLI attacks if user inputs are entered without verification [5].

Review of Possible Solutions: Another solution is to isolate Java as well as native code execution into separate processes that interact through remote process calls (RPC) as well as inter-process channels (IPC) [115] [116]. The Java process may function with complete Java application safety rights using this strategy, the native code executes with restricted privileges. However, as NativeGuard [116] illustrated, this strategy could lead to in process management and information duplication costs. Another viable approach is sandboxing with Java security-manager inspections into system call access for sandboxed programs [115] [116]. The suggested way of running native code is the same as executing untrusted programs in a sandbox, however with appropriate access constraints. Alternatively, the sandbox may be vulnerable to runtime privileged elevation attacks by illegal programs [70] [117]. This approach limits native code inside a secure environment. However, it severely limits the execution of certain instructions and prevents various types of code from running in the sandbox, such as Just-in-Time (JIT) compilers as well as stack unwinders [70]. To deal with the buffer overrun issue in Java native programs, David et al. [70] suggested CHERI JNI, a hardware-oriented Java Native Interface (JNI). It enabled to offer a security approach to native programming, such as Java. The technique offers safe immediate access to the Java Virtual Machine's (JVM) buffers.

There are additionally a few more generic, language-independent solutions provided. Cowan et al.[44] introduced the StackGuard compiler extension approach, which is transparent to regular program function but defends against buffer overflow attacks by preventing changes to the return address while a method continues to be running. In order jump to the attack payload, attackers commonly overflow the buffer of the stack and alter the return address. StackGuard aims to find modifications to the return address prior to the procedure returns and prevents writes to the return address. Ruwase and Lam [118] presented the "C Range Error Detector" (CRED), a dynamic buffer overrun detection system that replaces all out-of-bounds pointer data with the location of a special out-of-bounds (OOB) object that preserves the original pointer data and referent object information. Prior to getting dereferenced, pointers obtained from addresses are bounds verified, and they can safeguard against all buffer overruns. In [119], it is given a control-flow integrity (CFI) prevention approach that demands program execution to follow a predefined control-flow graph (CFG) architecture. CFGs can be pre-defined using source code examination, binary assessment, and runtime profiling. Static validation, binary-code instrumentation, as well as run-time checks are used in the CFI. In real time, these runtime inspections check and preserve the Control stream within a given CFG. This method can detect any unusual control flow alterations, even a return address modification triggered by a buffer overflow attack.

3.3.3. Encapsulation-Based Access Control

Gap Analysis: The keyword private allows for encapsulation-based access restriction. In Java, however, private objects can be made public by carrying out a public function. As a consequence, encapsulation access control in Java is inefficient and potentially harmful [120]. A "reliable" Java class having private variables and procedures can be typecast to a "hacker" class, which can duplicate the layout of these attributes and methods and retrieve these "private" objects during runtime.

Review of Possible Solutions: One feasible method is to use object ownership concepts to correctly conceal internal, stateful elements within an external owner object [120] [121]. Ownership demands that an item possesses its representation, and its internal state has no duplicates [120] [121]. Nesting and encapsulation of representation objects within other objects is required. Each type in the program is marked with its own object in that method, and only the owner object is entitled to the enclosed object, assuring that other individuals are unable to use the owned objects [122]. Object ownership can be accomplished through implementing code practices with checkers to guarantee conformance, or by including language level support by significantly modifying a programming language with the option to add ownership parameterization to the grammar and explicitly defining inside the type system of programming language [123]. Potanin et al. extended the ownership idea to "generic ownership" by combining object ownership and universal by keeping both type and ownership information in a single parameter space so non-this calls on owned objects are forbidden and owners are preserved as component of the type [124]. Object pointers can be augmented with transitive access restrictions to reduce the effects of aliasing whereas permitting entire referential object sharing [125].

3.3.4. Type System

Gap Analysis: TOCTOU attacks may be possible due to lack of immutability of Java [92]. Methods may also provide a reference to genuine internal structures, including arrays, which can be mutable, permitting malicious code to change system states [125].

Review of Possible Solutions: Strong immutability benefits from easy state management, thread safety, safe and efficient resource sharing, and increased security [47] [71]. The concept of ownership types can be used to limit object modifications to object contexts where an object can only edit objects it owns, including aliases to owned objects [89] [120]. Ownership contributes to the threaded safety of changeable resources by forcing locks to be placed in the correct order depending on ownership structures. For the prevention of data races and deadlocks in Java, a type system based on ownership types with related protections is proposed [121]. The protection mechanism is a variable type that can show that an item can be accessed by many threads and can refer to the lock that is used to protect the object referred by this variable [121].To prohibit altering items accessed via immutable objects, transitive immutability is required. To force transitive immutability, immutability annotations are proposed [89]. All class fields must be transitively immutable if a class constructor delivers a firm annotated result [89]. With Java generics and annotations, Zibin et al. [120] presented a Java extension that incorporates a type system for defining and enforcing reference and object immutability. Pechtchanski and Sarkar [126] presented an annotation-based approach for increased immutability expressiveness and code optimization evaluation. It does not, however, address concurrency concerns or guarantee that shared data stays consistent [128]. This feature should be statically verified by the language system, and these immutability properties should be included in the static type system[130]. [129] Haack et al. outline numerous rules for achieving immutability, the majority of which should be statically enforced. Enforcing these constraints should aid in the prevention of race problems in multi-threaded Java and limit exposure to changing states of internal structures and objects [129]. A Java language addition has been proposed. It has an immutable attribute that may be used with classes that can be constructed as immutable objects, as well as constraining objects and methods that its immutable objects can utilize [129].

3.3.5. Communication Encryption

Gap Analysis: If the algorithm property in the SSL client is an empty string or NULL, the JSSE library method SSLSocketFactory can bypass hostname verification [3]. As a result, rather than being caused by a flaw in the Java language, this flaw is produced by a flaw in the SSL modules, and it is transferred to the software application through these modules. Another likely explanation for this flaw is that intermediary levels of the application program stack block the validation of certificates unknowingly, or that programmers deactivate certificate verification for testing reasons [3] [122].

Review of Possible Solutions: By default, JSSE avoids hostname verification, leaving it to the other program. Programmers that are unaware of these tiny aspects may skip this phase as well. The recommended solution is to change the SSL module so that the SSL library handles hostname identification [3]. a few of prevention strategies identified in [3] include programmers performing black-box and hostile checking for unusual SSL certificates, license verification, which must be turned on afterwards if turned off by the programmer for evaluation reasons, and at all times clearly setting the options for secure SSL connections rather than relying on the default settings. It was demonstrated how to use the Flash player plugin to give socket functionality not naturally available in newer browsers in order to simulate an incomplete SSL handshake to capture fraudulent certificates [133].

Table 3.2 Evaluation of Programming Languages through SEFF

Features	Sub-Features	Languages to Evaluate				
		Java	**Python**	**C++**	**C#**	**Ruby**
Error Handling and Logging	**Log File Protection**	Not encrypted [109]	Not encrypted [85]	Not encrypted [46].	Not encrypted [56]	Not encrypted [48][63]
	Log Information Level	Partially Supported [56]	Partially Supported [85]	Partially Supported [46]	Partially Supported [56]	Partially Supported [56]
	Error Message Control	Partially Supported [60]	Partially Supported [135]	Partially Supported [137]	Partially Supported [77]	Partially Supported [104]
Data Validation	**Database Query Security**	Partially Protected SQLQuery [34][43]	Partiallyprotected SQLQuery [43]	Fully protected SQLQuery [138]	Fully protected SQLQuery due to MVC [63]	Fully protected SQLQuery [142]
	User Input Security	Partially Supported [64]	Not Supported [69]	Not Supported [49][65]	Partially Supported [65]	Fully Supported [68]
	Input Buffer Size Check	Partially provided checks for input buffer size [70]	Fully provided input buffer size checks [69]	Language does not provide checks for input buffer size [67]	Fully provided input buffer size checks [65]	Fully provided input buffer size checks [68]
	Encoded Output	Partially provided by the language [64]	Partially offered by the language [69]	Partially provided by the language [138]	Partially provided by the language [72][77]	Partially provided by the language [68]
Memory Management	**Memory Management Control**	Fully Supported [49][64]	Fully Supported [75][136]	Partially Supported [78]	Fully Supported [74]	Fully Supported [71][49]
	Memory Address Arithmetic	Fully Supported [65][67]	Fully Supported [98][136]	Not Supported [49]	Partially Supported [74]	Fully Supported [49]
	Array Out of Bound Check	Partiallyprovided [65]	Fully Supported [98]	Not Supported [49]	Fully Supported [65]	Fully Supported [49]
	Memory Management Flexibility	Fully Supported [49]	Fully Supported [136]	Not Supported [49]	Fully Supported [70]	Fully Supported [75]
Access Control	**Authorization Types**	Fully Supported [80[[83]	Partial Supported [76]	Partial Supported [81]	Fully Supported 49][74]	Partial Supported [120]
	Encapsulation-based Access Control	Weak encapsulation [64] [64]	Weak encapsulation [116][77]	Weak encapsulation [78][120]	Weak encapsulation [77][120]	Weak encapsulation. [120]
	Sandbox Support	Fully Supported [80][83]	Fully Supported [85]	Fully Supported [49]	Fully Supported [85]	Fully Supported [85]
Type System	**Type Safety**	Fully Supported Static typed safe [49][87]	Dynamic typed [87]	Static typed unsafe [87]	Strongly typed (Static typed safe) offers dynamic type by *dynamic* term[87]	Dynamic typed [87]
	Type Casting	Explicit [49][87]	Explicit [87]	Implicit [87]	Explicit [63]	Explicit [63]
	Type Initialization	Fully Supported [49]	Fully Supported [87]	Not Supported [49]	Fully Supported [63]	Fully Supported [63]
	Immutability	WeaklySupported [49][109]	Weakly Supported [90]	Weakly Supported [90][139]	Weakly Supported [88][90]	Weakly Supported [90][143]
U ser		Fully Supported [90][96]	Fully Supported [95]	Fully Supported t [96]	Fully Supported [94]	Fully Supported [63]
Web Session Managem ent	**Secure Session ID**	Partially Supported [34]	Partially Supported [59]	Partially Supported [3]	Partially Supported [34]	Fully Supported [63]
	Secure Cookies	Partially Supported, *HttpOnly* attribute status? False [103]	Partially supported, *HttpOnly* attribute status? False [59]	Partially supported, *HttpOnly* attribute status? False [140]	Partially supported, *HttpOnly* attribute status? False [102]	Fullysupported, *HttpOnly* attribute status? True [100]

Features	Sub-Features	Languages to Evaluate				
		Java	**Python**	**C++**	**C#**	**Ruby**
	Session Timeout	Fully Supported [53]	Fully Supported 59]	Fully Supported [3]	Fully Supported [53]	Fully Supported [63]
Communication Security	**SSL/TLS backward compatibility**	Fully supported [107]	Fully Supported [108]	Partially supported [41]	Fully Supported [105]	Partially supported [66]
	CryptographicAlgorithms	Fully Supported, there is apackage of tested algorithms, as well as custom algorithms are permitted [53][42]	Partially Supported, some algorithms are safe whereas others might not be secure [108]	Fully Supported, there is apackage of tested algorithms, as well as custom algorithms are permitted [141]	Fully Supported, there is a package of tested algorithms, as well as custom algorithms are permitted [53][107]	Fully Supported, there is a package of tested algorithms,as well as custom algorithmsare permitted [66]
	Key Length	Secure key size supported [92]	Partial Key size supported [108]	Secure key size supported [141]	Secure key size supported [107]	Securekeysize supported [66]
	Random NumberMethod	Offers strong entropy [53]	Offers strong entropy but is hard to use [109]	Offers strong entropy [53]	Offers strong entropy [53]	Offers strong entropy [63]
	Certificate Validity	Partial verification is provided [3]	No certificate verification is offered [3][108]	Partial verification is offered [3]	Partial verification is offered [141]	Partial verification is offered[66]

According to Fahl et al. [134], relying on engineers to build safer programming processes or simplify SSL components is not a viable approach. They advocated for a significant shift in the SSL module approach, in which nearly all of SSL utilization should be provided by the operating system itself as utilities that may be included in applications by setting rather than development. This would allow the platform to provide customizable SSL service alternatives to programmers avoiding the need to work around security limits at the level of the software [134].

Table 3.3 Percentage of Security Features Coverage by Programming Language

Programming Language	Security Feature Coverage Percentage
Java	63 88%
C#	69.4%
Ruby	65.74%
Python	52 75%
C++	47.22%

3.4. Abstraction of Security Features to Security Requirements

Traditional specification languages comparison framework as discussed in Table 2.4 to evaluate specification languages do not specify security requirements in detail. In our previous work [50], we defined a SEFF to evaluate security features in programming languages. SEFF provides a comprehensive set of security features for programming languages. A similar framework is needed to evaluate specification languages for security feature coverage, albeit at an abstract level. The impact of a language of programming choice on the security of software created in that language is what we wish to gauge. If there is such a factor, software developers or their managers could consider it when deciding which programming system to utilize for a certain task. This knowledge might facilitate risk mitigation and better resource allocation [166]. We have several grounds for thinking that a programming language's features could affect how secure applications created with that language are [167]. The investigation has demonstrated that type systems, for instance, can statically discover (and hence prevent, by stopping the compilation of specific sorts of defects. Generally speaking, static typing can reveal defects that might be vulnerabilities not discovered until they were exploited in a dynamically typed language. Additionally, standard frameworks of one language may be more accessible than another, making them less error-prone [168].

Developers may be able to recognize risky situations and get away from them with the aid of an up- to-date exception resolution framework [169]. The differences between languages used for programming go far beyond the scripts themselves, though. Every language has a unique community; these communities frequently have different beliefs and values. Hence, we want to see if selecting a language has a measurable impact on the security of the entire application. If so, it might be helpful to know if any particular class of weakness is better handled by one language than another [1]. If it is the case, writers might concentrate their efforts on the classes in that their programming style does not provide adequate support and lessen their concern for the ones for that data that indicate their dialect is strong [2].

This section proposes a security requirement framework for specification languages (SRFS) to evaluate the security capability of specification languages. A specification language with comprehensive security requirement coverage can help formulate correct, complete, and consistent security requirements early in development and influence better software design and implementation [24]. Using the SEFF framework for programming languages as a baseline provides a comprehensive set of security features that are abstracted to the specification language framework and helps remove gaps from the transformation of abstracted specification features into some platform-specific programming language [50]. Hence, there can be a direct correlation established between the two frameworks.

To define an abstract modelling-level features framework, each feature and sub-feature in **Table 2.4** from chapter 2 is evaluated with justification to determine if it can or should be considered at the modelling level. **Table 3.5** has four columns; column 1 and column 2 are the security features and sub-features from **Table 2.4.** Column 3 identifies whether a feature can be abstracted or not. Column 4 justifies column 3. Column 4 maps a security feature to a security requirement. The framework is also shown in **Figure 3.1.**

3.4.1. Error Handling and Log File Protection

Error Handling and Log file protection has following sub-features that are abstracted based on literature review. Sub-features log file protection, log message control and error message control can be abstracted based on following reasons.

3.4.1.1. Log File Protection

Logging is for record-keeping and accountability [145]. Specify audit requirements not only abides by the organization's rules but also ensures that the intended software system will follow auditing rules demanded by law-making organizations [147]. Moreover, specifying Secure Auditing is vital to check that all the objects in the system are honest [145]. It can be specified as: Graphical notation using class diagrams [150] or state charts [145]. Audit constraints as [148] or custom-made grammar [147] [149].

3.4.1.2. Log Message Control

Log message control is to store information in logs based on the severity level, such as information, debug, trace, warning, and error [57]. To abstract logs, message control is essential, as studies have shown that developers do not log with appropriate severity levels [57]. It can be specified as: Log message control using a graphical notation, such as using the log method of logger class [150]. Constraints limiting information to be stored in the logs [149].

3.4.1.3. Error Message Control

Error message control means controlling the verbosity of an error message. Some error message details can guide the attacker to exploit the possible values for the wrong entry. Error message control specify can help identify and specify these controls as constraints, like the log message control feature. It can be specified as: a low verbose error message constraint for a specific role and a verbose error message for a developer or authorized user.

3.4.2. Data Validation

Data validation feature has sub-features such as database query security, user input security, input buffer size security, encode output. Each of the sub-feature is abstracted based on the reasons defined in each feature sub-section.

3.4.2.1. Database Query Security

Database query security is essential because tainted user input in the query can lead to SQLI. Database query security specification can prevent SQLI from properly constructing database queries using specification language construct for the query, along with rules for correct query formation [150]. It can be specified as: Prevention of XSS and SQLI tags [150]. Constraints defining deny lists features of SQLI, XSS attacks, or safelists of acceptable inputs.

abides by the organization's rules but also ensures that the intended software system will follow auditing rules demanded by law-making organizations [147]. Moreover, specifying Secure Auditing is vital to check that all the objects in the system are honest [145]. It can be specified as: Graphical notation using class diagrams [150] or state charts [145]. Audit constraints as [148] or custom-made grammar [147] [149].

3.4.1.2. Log Message Control

Log message control is to store information in logs based on the severity level, such as information, debug, trace, warning, and error [57]. To abstract logs, message control is essential, as studies have shown that developers do not log with appropriate severity levels [57]. It can be specified as: Log message control using a graphical notation, such as using the log method of logger class [150]. Constraints limiting information to be stored in the logs [149].

3.4.1.3. Error Message Control

Error message control means controlling the verbosity of an error message. Some error message details can guide the attacker to exploit the possible values for the wrong entry. Error message control specify can help identify and specify these controls as constraints, like the log message control feature. It can be specified as: a low verbose error message constraint for a specific role and a verbose error message for a developer or authorized user.

3.4.2. Data Validation

Data validation feature has sub-features such as database query security, user input security, input buffer size security, encode output. Each of the sub-feature is abstracted based on the reasons defined in each feature sub-section.

3.4.2.1. Database Query Security

Database query security is essential because tainted user input in the query can lead to SQLI. Database query security specification can prevent SQLI from properly constructing database queries using specification language construct for the query, along with rules for correct query formation [150]. It can be specified as: Prevention of XSS and SQLI tags [150]. Constraints defining deny lists features of SQLI, XSS attacks, or safelists of acceptable inputs.

3.4.2.2. User Input Security

Specify input validation ensures the system operates on correct and meaningful input [5]. It can be specified as safelist and deny list constraints for user input [5], [152].

3.4.2.3. Input Buffer Size Check

Checking buffer boundary limits before taking input from the user is a must. It can be specified as attack prevention mechanism as [153] Constraint to check bounds and then throw an exception.

3.4.2.4. Encode Output

Encoding output prevents vulnerabilities when the invalidated input flows towards the output, such as stored XSS. In stored XSS, the illegitimate user stores maliciously crafted inputs in a legal website which executes for all other users of that Website. Encoding output will prevent invalidated inputs, resulting in vulnerable outputs achieved on the client machine. It can be specified as similar to input validation [5]. Secure information flow [146], such as defining outputs on particular inputs. Attack steps of XSS in graphical notation and model solutions as steps to prevent XSS [34]-[37]. Blocklist and allow list constraints.

3.4.3. Memory Management Control

Memory management control has sub-features such as memory address arithmetic, array out of bound check, memory management flexibility. Some of the sub-features cannot be abstracted as they are high-level features such as memory address arithmetic and memory management flexibility.

3.4.3.1. Memory Management

Specify memory management as thread-safe system resource constructs with specific methods to manipulate these resources [25]. Release of resources as needed using predefined specification language methods or keywords [30]. Secure information flow [146].

3.4.3.2. Array Out of Bound Check

Checking of buffer boundary before copying data into the buffer can prevent vulnerabilities. Specify input buffers as array resources [25]. It can be specified as constraints to ensure array boundary limitations, and further takes these requirements to other phases.

3.4.3.3. User Authentication Support

The least privileged access to the system resources should be given to legal users. Hence, user authentication support must model at the requirement specification level. It can be specified as constraints validating username and password. Attack specification and prevention mechanism in the form of constraints [154].

3.4.5. Access Control

Access control has sub-features such as authorization types, encapsulation-based access control, sandbox support. Here too some of the sub-features cannot be abstracted such as cannot be specified at a high level because the sandbox is a low-level security environmental feature. It is an execution environment to test untrusted code for malicious activity.

3.4.5.1. Authorization Types

It is essential to specify authorization types because the confidentiality of information requires controlled and least privileged access given to system resources [144]. It can be specified as authorization types using graphical notation for each role in the organization [6] [12]. Specify Permissions for each role using specification language constraints [149]. Role-based access control specification [5] [7] [14] [144].

3.4.5.2. Encapsulation- Based Access Control

Encapsulation-based access control states that access to private variables must be through public methods of that class. Specify encapsulation-based access control is essential to ensure controlled access to objects and resources. It can be specified as access level scopes by specification language constructs, such as mutex, immutable, private, and so [145]. Constraints similar to the case of authorization types. Graphically by system resources, objects, and permissions as in SecureUML [144].

3.4.6. Type Safety

Specification language supporting type declaration serves as assertions about the meaning of the variable [25]. It can be specified as appropriate types of variables for a specification language [25].

3.4.6.1. Type Casting

Unsafe type casting or implicit type casting can cause a loss of useful information. It is a low-level programming security feature that requires built-in functions for every data type and its corresponding data type, such as integer to float. Implementation of this feature in a programming language and specification is impossible.

3.4.6.2. Type Initialization

Specification of type initialization is vital to resolve the fact that the values are undefined [25]. This security feature is specified by giving default values to the variables [150].

3.4.6.3. Immutability

Immutability prevents race conditions in a multithread application a role assignment before performing permission is a must. Specify immutability by an immutable variable for multi- process. Changes to the shared variable must be made atomic using mutex or semaphore to model thread safety as in RealSpec specification Language [25].

3.4.7. Secure Session Management

Secure Session management has sub-features secure session id, secure cookies, session timeout are abstracted based on the following reasons defined.

3.4.7.1. Secure Session Id

Session ID names depict language details [34]. It can be specified as Session ID tag [6]. Preventive solutions in graphical notation [154].

3.4.7.2. Secure Cookies

The secure or HTTP-only flag should protect the authentication cookies holding session information [34]. Secure information flow states that critical information should not flow to a less secure level [6]. It can be specified as constraints similar to secure session id and XSS attack and prevention [154]. HTTPS tag instead of HTTP.

3.4.7.3. Session Timeout

Client-server communication is done through sessions to ensure a secure connection. One of the requirements is to log out after a specified session idle time [154]. It can be specified as. Session timeout constraint [6].

3.4.8. Communication Security

The sub-features of communication security are TLS/SSL version support, cryptography algorithm, key length, random number generator and certificated validation are abstracted based on the following reason defined such as key length and random number generator are low-level features.

3.4.8.1. TLS/SSL version Support

For secure communication use of an SSL/TLS secure version is essential. Specify as Secure Configuration as KAoS goal-based language [150]. Denylist and safelist of software version as a protocol [155].

3.4.8.2. Cryptographical Algorithm

To ensure developers use well-known well- vetted cryptographic algorithms. Specify as Secure Configuration [150] Confidentiality [155].

3.4.8.3. Certificate Validation

Data origin authentication refers to the security of information at the message's origin. Using X.509 certificates is one solution. X.509 is a public key infrastructure (PKI) standard, and a website belongs to a recognized domain. [155]. Specify as entity authentication [155].

3.5. Evaluation of Specification Languages using Proposed Framework

Section 2 gives an overview of SRFM. Table 3.6 is used in this section to evaluate state-of-the-art specification languages to cover security requirements. These specification languages capture and validate security requirements early in software development. The reason for selecting them is either security requirement specification due to the application domain or the executable nature of specification language.

Table 3.4 Security Feature Abstraction

Security Feature	Security Sub-feature	Abstractedfor Specification?	Security Requirement abstract name
Error Handling and Logging Protection	Log File Protection	Yes	Secure Auditing
	Log Message Control	Yes	Secure Auditing
	Error Message Control	Yes	Secure Error Message
DataValidation	Database Query Security	Yes	Input Validation
	User Input Security	Yes	Input Validation
	Input Buffer Size Check	Yes,	Buffer Limit access prohibition
	Encode Output	Yes	Output Validation
Memory Management	Memory Management Control	Yes	Memory Safety
	Memory Address Arithmetic	No	Nil
	Array Outof Bound Check	Yes	Buffer Boundary Limit Access Prohibition
	Memory Manageme nt Flexibility	No	Nil
User Authentication Support		Yes	Authentication
Access Control	Authorization Types	Yes	Authorization
	Encapsulation-Based Access Control	Yes	Ownership
	SandboxSupport	No	Nil
Type System	Type Safety	Yes	Type safety
	Type Casting	No	Nil
	Type Initialization	Yes	Type Initialization
	Immutability	Yes	Immutability
Secure Session Management	Secure Session ID	Yes	Authorization
	Secure Cookies	Yes	Secure information flow
	Session Timeout	Yes	time-based accesscontrol
Communication Security	SSL/TLS versionsupport	Yes	Secure Configuration
	Cryptographic Algorithms	Yes	Confidentiality
	Key Length	No	nil
	RandomNumber Method	No	Nil
	Certificate Validation	Yes	Integrity

3.5.1. SysML-sec

SysML-sec is a specification language for concurrent systems. Security requirements, along with other functional requirements. Blocks define tasks and nodes; the "allocate" relationship defines allocation. It specifies requirements using SysML blocks and a state machine, and pi-calculus. The solution is given as pragma, for example, confidentiality, Integrity, and authentication. SysML-sec is a goal-based technique. It uses encrypt() to define cryptographic algorithms. The transformation of specifications is done in TTool and Proverif for verification. Moreover, it notes a 40% delay when describing security properties in SysML-sec and safety properties.

3.5.2. Secure Descartes

Secure Descartes [11] is an extension to Descartes executable specification language. It provides a policy framework for web applications by extending Descartes language constructs to specify policy entities and policy rules to be used in the policy application. A secure policy framework has policy entities, policy rules, policy applications, a policy knowledge database, and a policy 2manager. The basic building blocks of a security policy framework are entities or components that include: subjects, objects, constraints, and actions. Policy entities define policy rules. Policy Knowledge Database (PKD) stores policy priority, policy conflicts, and policy history. The policy manager resolves conflicts in the policies by assigning priorities to the policies [11]. Secure Descartes handles security concerns provided by the SANS institute [11].

3.5.3. UMLsec

UMLsec [7][146] was the first to introduce security notations into software specifications and designs in 2001 by extending the standard UML profile, and it also provides a baseline for comparison with other notations [17]. UMLsec uses UML diagrams, stereotypes, tags, and constraints in specification security requirements such as user authentication support, input validation, access control, database query security, type system, and partial support of log message control. In addition, UMLsec represents secure communication through stereotypes and tags [155]. UMLsec models secure auditing using a state chart diagram, providing log entry accountability [12]. Certificate Validity is demonstrated by a client and server model with a previously shared key from the certificate authority for mutual authentication [155]. However, it requires transforming the model into a .xml file and then into platform-specific language [7].Even though UMLsec provides comprehensive security feature coverage, there are security features not covered by UMLsec, such as output prohibition, Immutability, and secure error message.

3.5.4. Input Validation Using UML

The approach offered by Hayati et al. [5] extends the UML profile to express input validation using OCL constraints as well as UML activity, use case, sequence, as well as class diagrams. This approach validated each input against five attributes. The right type, format, size, character set, along with reasonableness of user input are all checked. For example, the suitable character set is any combination of [0-9], the appropriate input age type is an integer, the appropriate length is two or three characters, and so on. A valid format is two or three integers without spaces or dashes, and valid reasonableness is greater than zero.

3.5.5. AMF

Assurance Management Framework (AMF) [7] models authorization types and access control for authorization systems. It is a framework with four-tiered process for software development. First, business policies, actors, as well as authorization needs are defined. The requirements are then mapped to RCL2000 constraints and transformed to Alloy formal specification language. Finally, specification of requirements through UML using OCL for constraint specification. Furthermore, Alloy Analyzer verifies Alloy policies and conformance checks the resulting implementation. The drawback of this scheme is that it requires too many transformations from one model to another.

3.5.6. S-Promela

S-Promela [13] is a security extension to Promela executable specification language for embedded systems. The proposed work models role-based access control depending upon the ideas of processes, channels, constraints, events, as well as actions. It has five concepts and three rules. Concepts include subject, object, action, constraints, and event, whereas rules are authorization, obligation, and prohibition. The entities such as subject and object interact with each other using communication means known as a security policy (SP). It uses four half-duplex channels to show the flow direction between the entities. Each channel is accessible for either insertion or extraction: It inserts data using the write mode and extracts data using the read mode. It absorbs all refused requests through the Out channel. Promela is a specification language that is type-safe [13]. As a result, S-Promela was born. S-Promela semantics employs an operational model [13].

3.5.7. Ponder

Ponder [145] is an object-oriented and declarative specification language for business security specification. Authorization, obligation, refrain policy, information filtering, object policies, as well as delegation policies are all used in ponder policies [145]. However, there is no explicit semantics for ponder [12]. Ponder can help with identity, auditing, and access control [12]. Ponder is a specification language that is type-safe [12].

3.5.8. SecureSOA

Vahid and Ramin [156] proposed a security framework for Service-Oriented Software Architecture (SOA). It specifies security features such as Authentication, Confidentiality, and Access Control [22] at the early design phase in Alloy. The Alloy has a set of atoms and their respective restrictions. Signatures define atoms, and facts describe restrictions. Atoms or signatures are the objects and relationships. A model must satisfy facts (logical statements) or constraints. Signatures define access control entities such as roles, trusted roles, participants, permission, resource type, action type, authorization constraint, and collaboration session. The alloy meta-model defines access control specifications. The security analysis is performed by defining some properties and applying assertions to prove them in the Alloy analyzer [156]. Alloy analyzer solves constraints, which is why it is suitable for performing detailed security analysis on the protocol level and not used to model whole software applications [15]. Alloy is reasonable to model transport layer security. Table 3.2 evaluates the above specification languages by SFRS for security feature coverage. Based on the result of Table 3.2, analysis of language feature/sub-feature coverage, an approximate estimate of security percentage coverage for above-mentioned specification languages is generated. Ponder [12] offers the highest security coverage percentage whereas Secure Descartes [11] provides the second highest coverage among the evaluated specification languages proving to be the safest executable specification language for secure web application development.

3.6. Mathematical Model

The mathematical model is designed to evaluate the security feature coverage of various specification languages by calculating the Security Percentage Coverage (SCP). The model assesses how well a particular specification language supports both partially and fully supported security features, relative to the total number of sub-security features. Here's a detailed breakdown of the formula and its components:

- **SCP**: **Security Percentage Coverage**. This is the overall percentage that represents how well a specification language covers the required security features.
- **TNPS**: **Total Number of Partially Supported Features**. These are features that the specification language supports but not completely or optimally. A partial solution to a security requirement may exist, but it lacks full implementation.
- **TNFS**: **Total Number of Fully Supported Features**. These are the security features that the specification language completely supports with no gaps or compromises.
- **TNSSF**: **Total Number of Sub-Security Features**. This represents the total count of all the specific security features that need to be evaluated (for example, authentication, confidentiality, access control, etc.).

The mathematical model used to evaluate above-mentioned specification languages is as follows:

$$\therefore \text{SCP} = \left[\frac{(TNPS * 0.5) + TNFS}{(TNSSF)} \right] * 100\% \qquad (1)$$

Table 3.5 Performance Measures

Features Supported	Values	Symbol
Not Supported	0	NS
Partially Supported	0.5	PS
Fully Supported	1	FS

3.7. Analysis of Specification Languages using SRFS

The previous section identified several security features gaps in the evaluated modeling languages. This section provides a detailed analysis of the gaps along with some recommendations and related research for addressing potential coverage for each gap. According to Table 3.2, it can be devised that the security features can be abstracted either using graphical notation such as extending UML profile, structured text using some constraint language, modeling language constructs such as graph transformation, or by modeling attack steps using misuse case and defining its countermeasures using some notation.

Error Handling and Logging Protection

Gap Analysis

Evaluation from Table 3.6 shows that secure auditing is fully covered by UMLSec, partially covered by Secure Descartes as it provides support for maintenance, monitoring and analysis of logs yet no encryption for log statements is provided, this sub-feature is not offered by AMF, Hayati et al., SysML-Sec, S-Promela. Secure error message is not covered by any specification language.

Possible Recommendations

Log file protection by secure logging is a mandatory mechanism which can be specified by storing log messages in encrypted form and decrypted when privileged users need to access logfiles. The log file message is controlled by modeling language constraints specifying the information to be stored in the logs such as date, time, log type, log info, and log condition. Hochreiner et al. [11] uses graphical notation of UMLsec to show auditing by calling log method of the logger class to store database query in log file. Error message control can be specified by creating attacker as a role and model his actions step by step using misuse case notations and then also modeling preventive actions. Log files can also be specified by UML profile extension approach proposed by Deveci and Caglayan [22]. Their approach defines stereotype asset with its associated tag, showing read or write protection during communication, and the level of importance of the asset. The approach also provides design level stereotypes and associated tags used to embed security properties into analysis and design phase. The model is then exported to Extensible Markup Language (XML) Metadata Interchange (XMI) which is later verified using SPIN model checking [22].

Hoisl and Strembeck [146] extended UML for the specification of audit rules, audit events and relevant actions. The proposed scheme consists of event-based modeling of audit properties where the software can produce and consume events. The approach first extends metaclasses ofUML metamodel to specify audit requirements such as audit rules, audit events and relevant actions, taking care of the consistency of the semantics of extended metaclasses with existing metaclasses. Security audit pack-age is then defined, which extends the UML metamodel [146]. Graphical notation of modeling elements with context free grammar using Backus Naur Form (BNF) of event-based auditing are first defined with OCL constraints. In addition, the modeling of audit system from different perspective is shown using activity diagram, state chart diagram, sequence diagram, and textual based modeling [146]. However, this approach models event- based auditing, which can be extended to model secure auditing. Another approach known as process-centric modeling language based on Secure Business Process Modeling Language (SecBPML) is proposed in [158]. In this approach, set of security rules are specified using security annotations and textual structure that are given in the form of River template for River Definition Language (RDL) [158] used to implement business logic layer. Furthermore, whenever the system changes, a revised version of new artifacts is introduced and then checked for compliance. The proposed approach specifies only auditing and accountability [158]. However, this approach can be modified to include encryption and decryption while modeling secure logging. The verbosity of log messages can be controlled by specifying attributes for logging. Error message control can be modelled by controlling and customizing error messages and explicitly checking any verbosity in error messages or limiting verbosity level in error messages.

Input Validation

Gap Analysis

Specification of input validation is fully supported by UMLsec [7] and Hayat et al. [5] and it is not supported by the other evaluated specification languages. Bound access prohibition is only fully supported by Secure Descartes [10] because it handles boundary defense and Hayati et al.[5] supported this using OCL constraints for length verification. Encode output is not supported by any of the evaluated modeling languages.

Possible Recommendations

Hayati et al. scheme [5] can be extended to provide validation of suspected variable that copies unvalidated input and then use that variable in the code and then to the output. Activity diagram along with OCL constraints can be used to model suspected variable copying unvalidated inputs. Enhancement of [5] can also be performed to track an unvalidated input that moved towards output, such as file or network socket, where the input must be escaped or encoded before being exposed to file or network socket. Encode output sub-feature can be graphically modelled using activity diagram to validate the output as the case of input validation in [5] along with encoded output. Kong et al. [15] proposed an approach that can be extended to model input validation threats such as buffer overflow [38], SQLI [31], XSS [34], and its countermeasures where graph transformations may be performed for the verification of security threats [15].

Memory Management

Gap Analysis

Buffer boundary limit access prohibition and memory safety is only fully offered by Secure Descartes [10].

Possible Recommendations

RealSpec [29] uses resource construct to model abstract data structures such as arrays, queue, and stack as well as various hardware resources such as memory. These resources are internally modelled using the list data type. Since the internal representation is using a list data type, trying to access data structure elements beyond their size will result in 'nil' value and, hence, will not expose memory leaks. Furthermore, the resources such as queue and stack are protected against out of bounds by using special semaphores. All resources in RealSpec are thread safe using semaphores and mutex. Users can define new resources by using the resource construct. Memory and other data structure resources may be allocated and released by calling resource methods or defining keywords for these controls [6].

Type System

Gap Analysis

Type safety is not supported by SecureSOA and SysML-Sec and all other evaluated modeling languages support this sub-feature. Type initialization is only fully supported by Secure Descartes, S-Promela and Ponder. Immutability is not supported by any of the evaluated modeling language.

Possible Recommendations

UMLpac [152] security package can be defined for race condition and its mitigation. Immutability can be specified using mutex and semaphore resource to provide atomicity in a multithread environment thereby preventing race condition.

Access Control and User Authentication

These features are fully specified by all the evaluated specification languages.

Possible Recommendations

Ownership can also prevent race condition as unnecessary aliasing to a variable is prohibited and only legal owners can access their objects. This concept can be defined using graphical approach used by SecureUML [8] by defining resources of the system as well as roles, permissions and constraints for the authorization to access those resources. Secure guards can also be used to define secure resources and to check permissions for resources [7, 150]. However, since the model is only used to define resources and their access but not the order of access, there is no assurance of synchronization necessary to avoid race conditions.

Immutability can be modelled by adding synchronization to the event-based scheme proposed by Basin et al. [156] that models RBAC and ownership, extending SecureUML. KAoS [144] is a policy specification language and is used to specify obligation policies, refrain policies, and conflict resolution. Policies in KAoS are represented in structured text using semantic Web language known as OWL (Ontology Web Language). KAoS policy ontology defines actors and policies for these actors. KAoS policy constraints that allow or refrain from performing an action are called positive and negative authorizations. Positive and negative obligations are constraints required to represent some action for event or state triggering. RBAC policy is built from the above four constraints. The priority of RBAC policy can be used to model synchronization, thereby avoiding race conditions.

Secure Session Management

Gap Analysis

Secure information flow and time-based access control are fully specified by secure Descartes and ponder whereas S-Promela fully offers time-based access control while partially offers support for secure information flow.

Table 3.6 SFRS used to Evaluate Security Capability of Specification Languages

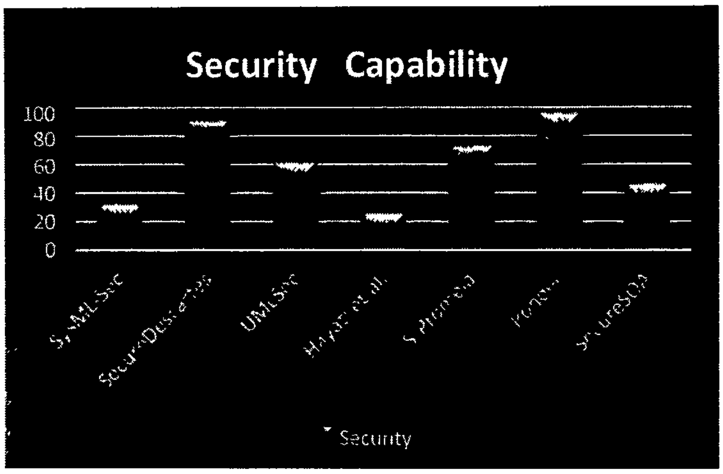
Security Feature	Sub-features	SysML-sec[10]	Secure Descartes [11][158]	UMLsec [7][146][155]	AMF [14]	Hayati et al.[5]	S-Promela [13]	SecureSOA [156]	Ponder [12][145]
Error handling and LogFile Protection	Secure Auditing	Not Supported	Partially Supported#14 maintenance, monitoring, and analysis of logs	Fully Supported by protection tags for logs [146] <<encrypt>> tags	Not Supported [22]	Not Supported [5]	Not Supported [13]	Fully Supported cryptography [22]	Partially Supported Auditing from [145]
	Secure Error	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported
Data Validation	Input Validation	Not Supported	Not Supported	Fully Supported Database class and tags to protect queries [150] and separate input validation class	Not Supported [22]	F S Using activity diagram and OCL constraints [5] Does not validate input when copied to another variable	Not Supported	Not Supported	Not Supported
	Bound Access Prohibition	Not Supported	Fully Supported#13 Boundary Defense	Not Supported	Not Supported [22]	Fully Supported Using OCL constraints [5]	Partially Supported [13]	Not Supported	Fully Supported[11]
	Output Validation	Not Supported	Not Supported	Not Supported	Not Supported [22]	Not Supported [5]	Not Supported	Not Supported	Not Supported
Memory Management	Memory Safety	Not Supported	Fully Supported#15 Controlled Access	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported	Not Supported
		Fully Supported [11]	Fully Supported [11]	Fully Supported [11]	Not Supported [14]	Not Supported[5]	Partially Supported [11]	Not Supported	Fully Supported [11]
Access Control	Ownership	Fully Supported Authentication Pragma	Fully Supported#15 Controlled Access	Fully Supported Using guarded access [150]	Fully Supported By OCL Constraints , formal specification of RBAC model [14]	Not Supported[5]	Fully Supported [13]	Fully Supported [11] Authentication [22]	Fully Supported Identification [145]

Security Feature	Sub-features	SysML-sec[10]	Secure Descartes [11][158]	UMLsec [7][146][155]	AMF [14]	Hayati et al.[5]	S-Promela [13]	SecureSOA [156]	Ponder [12][145]
	Authorization	Fully Supported Authentication Pragma	Fully Supported#15 Controlled Access	Fully Supported Using guarded access [150]	Fully Supported By OCL constraints, formal specification of RBAC model [14]	Not Supported [5]	Fully Supported [13]	Fully Supported [22] Access control specified in alloy formal specification language	Fully Supported Identification [145]
	Buffer Boundary Limit Control	Not Supported	Fully Supported#13 Boundary Defense	Not Supported	Not Supported[22]	Not Supported [5]	Partially Supported [13]	Not Supported	Fully Supported [11]
User Authentication Support		Fully Supported Authentication Pragma	Fully Supported#15 Controlled Access	Not Supported	Fully Supported By OCL constraints, formal Specification of RBAC model [14]	Not Supported [5]	Fully Supported [13]	Fully Supported [22] Access control specified in alloy formal Specification language	Fully Supported Identification [145]
Type System	Type Safety	Not Supported	Fully Supported[11] #17 Data prevention loss	Fully Supported using some data and their types in the class diagram [150	Fully Supported Using constraints [14]	Fully Supported [5] Using variable and their types in the class diagram [9]	Fully Supported [13]	Not Supported	Fully Supported [11]
	Type Initialization	Not Supported	FS [11] #17 Data prevention loss	Not Supported	Not Supported [22]	Not Supported [5]	Fully Supported [13]	Not Supported	Fully Supported [11]
	Imutability	Not Supported	Not Supported [11] #17 Data prevention loss	Not Supported	Not Supported [22]	Not Supported [5]	Not Supported [13]	Not Supported	Not Supported[11]
Secure Session Management	Secure Information Flow	Not Supported	FS#6 Application Software Security	Not Supported	Not Supported [22]	Not Supported [5]	P.S. #6 Application Software Security [13]	Not Supported	Fully Supported#6 Application Software Security[11]

Security Feature	Sub-features	SysML-sec [10]	Secure Descartes [11][158]	UMLsec [7][146][155]	AMF [14]	Hayati et al. [5]	S-Promela [13]	SecureSOA [156]	Ponder [12][145]
	Time-based Access Control	Not Supported	Fully Supported #16` Controlled access [11]	Not Supported	Not Supported [22]	Not Supported [5]	Fully Supported #16` Controlled access [13]	Not Supported	Fully Supported #16` Controlled access [13]
Secure Communication	Confidentiality	Fully Supported Confidentiality Pragma	Partially Supported [11] #17 Data Loss Prevention	Fully Supported[7]	Not Supported [22]	Not Supported [5]	Not Supported [13] #17 Data loss Prevention	Fully Supported [22]	Not Supported [11] # 17 Data loss Prevention
	Secure Configuration	Not Supported	Fully Supported #10 Secure configuration s for network devices such as firewalls, routers, and switches [42]	Not Supported	Not Supported [22]	Not Supported [5]	Partially Supported FS #10 Secure configurations for network devices such as firewalls, routers, and switches [11]	Not Supported	Fully Supported #10 Secure configuration for network devices such as firewalls, routers, and switches [11]
	Integrity	Fully Supported Integrity Pragma	Fully Supported#16 ` Controlled access [11]	Fully Supported Using sequence diagrams [144] [155]	Not Supported [22]	Not Supported [5]	Fully Supported #16` Controlled access [11]	Fully Supported [22] Integrity specification	Fully Supported #16` Controlled access [11]

Table 3.7. Evaluation Results

Specification Languages	SysML-Sec	Secure Descartes	UMLSec	Hayati et al. scheme	AMF	S-Promela	Ponder	SecureSOA
Domain	Real-time/embedded systems	Web applications	Distributed System	Distributed System/Web Applications	Authorization System	Embedded System	Business Security	Service Oriented Architecture
Type	Executable Specification Language	Executable specification language	Graphical modeling language	UMLprofile extension + Formal Specification Language	Graphical modeling language + formal specification language	Executable Specification Language	A declarative, object-oriented language	Formal specification language
Security Features Coverage	32.911%	91.75%	61.76%	26.449%	35.29%	73.52%	97.05%	47 06%



Possible Recommendations

Web sessions are often combined with authentication. The duration of an active session is an important attribute of session security [6]. Secure information flow and time-based access control can be specified using modeling language constraints similar to session timeout in UWESecurity [6]. Busch [6] modelled secure session management through tags, and stereotypes such as tag{session data} with <<session>> stereotype. It allows the modeler to specify session class variables to be modelled graphically [6]. Session information flow can be specified as secure information flow or no-down flow [7]. For example, not exposing high privilege to low privilege using stereotypes and tag.

Kong et al. [15] suggested modeling broken authentication and session management threat by state chart and class diagram with security threat verification performed via graph transformation. Kong et al.'s [15] approach may be extended to include session ID modeling as well.

Communication Security

Gap Analysis

Confidentiality and integrity are fully supported by SysML-Sec using confidentiality and integrity pragma. Secure Descartes fully supports secure configuration and integrity while partially supporting confidentiality. UMLSec and SecureSOA fully offers support for confidentiality and integrity. S-Promela partially supports secure configuration whereas fully offers integrity yet no support for confidentiality.

Possible Recommendations

This feature may be fully modelled by extending UWESecurity [6] to provide addition tags and enumerated values which can list vulnerable and safe versions. These lists may be used later to verify a secure SSL/TLS connection. Another possible way to specify secure configuration is through modeling language constructs, such as configuration settings package [152], which can provide a blacklist of vulnerable versions and whitelist of safe versions.

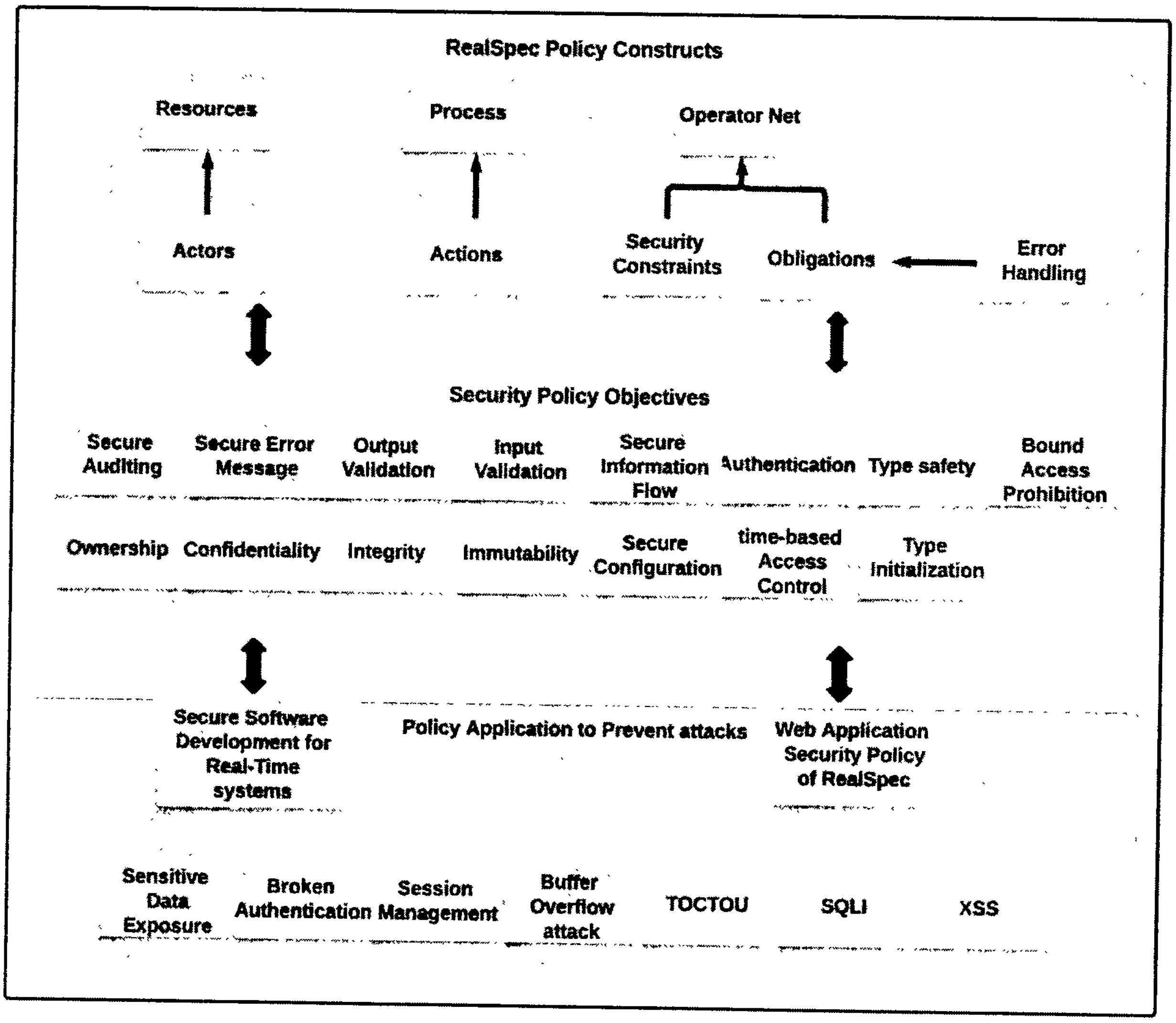


Figure 3.1 Proposed Solution

3.8. Proposed Solution

Table 3.6 identifies that all evaluated specification languages do not support secure error message, output validation, and immutability. Therefore, a solution is proposed to construct a security policy [11]-[13] in RealSpec to extend security support for it. RealSpec has been chosen because it is a type-safe language and it is an executable specification language where specifications are mathematically tractable [24]-[29]. RealSpec processes are used to describe actions. In Figure 3.1, it is shown that the security requirements from proposed SRFS is specified in RealSpec using the language constructs such as user-defined resources, processes, and operator nets. Here, RealSpec resources are used to specify actors, processes are used to define actions whereas operator nets are used to define security constraints and obligations. This security policy can be used to specify web application security policy as well as secure software development. The proposed solution is going to specify secure error message, output validation and immutability along with other security requirements and system functional requirements.

3.9. Summary

Chapter 3 introduced several key frameworks aimed at enhancing security in software development. Section 3.1 presented a Security Feature Framework (SEFF) tailored for programming languages, evaluating five popular languages. Section 3.2 identified gaps in Java using SEFF and proposed potential solutions. Section 3.3 introduced the Security Requirement Framework for Specification Languages (SRFS), adapting SEFF's implementation-level features to the specification level to evaluate specification languages for security coverage in web applications. In Section 3.4, SRFS evaluated various security specification languages, revealing minimal security requirement coverage across most languages, with Secure Descartes achieving the highest coverage at 91.75%. This gap between specification and implementation was identified as a critical factor in current web application security vulnerabilities. Section 3.5 analyzed gaps in specification languages and suggests remedies, while Section 3.6 introduced a mathematical model to quantify security coverage percentages. Section 3.8 proposed integrating security features identified by SRFS into RealSpec, and Section 3.9 provided a comprehensive summary of the entire chapter's findings and contributions.

Chapter 4

DESIGN

RealSpec Overview

RealSpec [24]-[29] is a specification language for embedded systems. RealSpec statements are declarative and also executable. RealSpec is built on the concept of a functional dataflow. RealSpec specifications, unlike typical imperative programming languages, include equations which includes sequence and filters rather than methods for changing memory locations.Because the statements in RealSpec act as theorems based on which further claims can be deduced, it is referred to as a definitional language. These claims define the behavior of the system in consideration.

4.1.1. RealSpec Data and Iterative Statements Specification

The RealSpec statement claims x to be an endlessly distinct set of data series <1, 2, 3, 4, 5,...> at time indicator <t0, t1, t2, t3, t4,...>:

```
a
 where {
  a = 1 fby a+1;
 }
```

Where clause is a statement in the aforementioned RealSpec statements, and it is also used to specify a group of instances utilized in the statement. RealSpec variables may be referred to as operator nets. Abstract iteration over sequences is offered by the Boolean operator fby (followed by). The first variable to fby represents the initial value, whereas the subsequent parameters represent ordered future values.

4.1.2. RealSpec Objects Specification

Complex types of data and objects in RealSpec are represented using customized algebras. RealSpec instance parameters may referred to as operator nets or functions. These instance variables define the object's state, or the inner workings of an object can be viewed and modified by the object's operator nets. A data object's instance variables must be fully functional. RealSpec streams that advance in data object streams are likely going to be followed by a n symbol of stream.

4.1.3. System, Processes, and Threads Specification

The RealSpec code starts from a system construct which offers perspective for the entire specification. System resources, statically declared procedures, processes along with thread building sequence, and global system-level methods are all defined in system structure.

```
system sys {
    resources { ... }
    processes { ... }
    functions { ... }
}
```

A process construct [24]-[29] defines a process object. The keyword process is used in process construct, accompanied by the process name then the process body surrounded in curly brackets. A primitive data variable declaration, multiple passive or active objects, or an array of process functions may be found in the body of a process specifications. A process's functions are declarative assertions which describe the process's operator nets. Every process method or operator net executes continually and simultaneously. A process includes one execution thread by default. Yet, a process can include numerous threads as it chooses. For example, when the running thread is th0, as determined by the attribute pid, a has the value a+1; otherwise, a gets the value a*2. As a consequence, the result for thread th0 will be 1, 2, 3, 4,...>, since the result of thread th1 will be <1, 2, 4, 8,...>:

```
process process1() threads th0, th1 {
  a
   where {
     a = 1 fby if pid == 0 then a+1
        else a*2;
   }
}
```

4.1.4. Resource Specification

RealSpec provides an overview of system assets using objects for data [24]-[29]. The pre-configured resources in RealSpec comprise (a) hardware resources like signal and analogue IO and (b) abstract data structure resources like arrays, semaphores, mutex, queue, along with stack. The resource objects that have already been set up are multi-thread secure; which means that they may be accessed by several threads at once using internal semaphores and a mutual exclusion strategy. The resource statement construct template listed below can be used by users to define new unique resource objects:

```
resource <name>(<arguments>) {
    <resource data>
    <resource methods>
}
```

Prior to use, resource instances must first be created after being declared. As an illustration, a system sys defines two resource objects: a four-element queue data structure and an signal input resource for obtaining a switch condition:

```
system sysname {
        resources {
          signalin switch1;
          queue    qu1(4);
        }
    }
```

4.2. Input Validation and SQL Injection Prevention:

The process Input Validation () checks the input provided by users for potential SQL injection attempts. If the input contains malicious patterns, such as "1==1 – ana OR", it will be detected as a SQL injection attempt, and the system will return an error message like "Invalid input: SQL injection tautology detected". However, when a legitimate query such as select ename from employees where id==001 is executed, it passes validation, and the message "No SQLI detected" is shown. This safeguards the system from injection attacks by filtering out malicious inputs.

Input validation is a critical security measure in Real Spec, where incoming data is verified to ensure it adheres to expected formats and values. Real Spec includes built-in functions known as prefix operators to validate different types of data, helping to protect against common vulnerabilities such as SQL injection, buffer overflows, and data corruption. These prefix operators include functions like is number to check if the input is numeric, is word to validate if the input is a valid word, is null to verify null values, is string to ensure the input is a string, length to measure the input's length, and is of to check if the input has reached the end of a file or data stream. These basic functions help block invalid or malicious inputs, such as using is number to prevent non-numeric values in fields expecting numbers.

However, Real Spec can benefit from additional input validation functions that are currently missing. These would enhance its ability to detect more complex types of malicious inputs. For example, is email could validate email addresses by checking for proper formatting, is date could ensure correct date formats, is phone could verify the validity of phone numbers, and is pattern could enable regular expression-based validation for more flexible and precise input checks. These new functions would provide further protection against various types of malformed or potentially harmful data.

4.2.1. Specification of Error Handling and Log File Protection in RealSpec

RealSpec defines Log File as a resource as a shown in Table 4.1. The first of the three inputs for the logfile are encryptionStatus, which is followed by pol and severityLevel. For instance, to ensure thread safety in a concurrent environment, pol is a policy specified for mutex resources, severityLevel of the log is used to select the level of information to be stored in log files, and encryptionStatus is required to figure out whether the log file is encrypted. Currently, two policies are supported: the default policy, which uses first come, first served (FCFS), and the priority system, that uses the thread's priority. ERROR, WARNING, TRACE, DEBUG, INFO,ALL, and OFF are only some of the severity levels of logs that the log4Net and Log4j programming language libraries provide. Internally, a logfile is described as a list resource. Moreover, there is an index which is used to hold a curser point where a file can be read from and in case of write mode the index is set to 0 that means index is set at starting position of the file. Algorithm1 and Algorithm 2 show secure auditing write and read function respectively.

Table 4.1 Log File Signature

Signature	**Logfile (bool encryptedStatus, int pol, int severityLevel)**
System variables	

Private variables	**list** ldisk=[];
	bool status= encryptedStatus;
	int index; **mutex**
	file(pol); **list** qlist;
	generic input; **generic**
	buffer;

User variables	

User functions	**int** open (**int** mode);
	generic operator << (**generic** input, **generic** p);
	generic operator>> (**generic** buffer);
	bool isEncrpted();
	bool fileSize();
	generic loglevel (**int** severity);

Algorithm 1 Secure Auditing Write function

Require: ldisk is log file, ldisk is a mutex file to check mutual exclusion while write operation to ldisk, encryption status checks encrypted text or plain text, index variable to check current location to read a file, eof to show end of file

Ensure: mutual exclusion while write operation, encrypt the log statement if the log is stored in encrypted form.

```
Start
 Initialize ldisk to logfile
   While index !=eof do
       For each ldisk open in write mode do
          Lock ldisk
        If ldisk is encrypted
          Write ldisk in encrypted form
```

```
      Else
        Write ldisk
      End if
    Unlock ldisk
End While
```

In Appendix 1, Read(>>) and write (<<) operators are overloaded for the logfile. The open function accepts the mode, such as read or write mode, and performs the function based on the defined mode. Whena logfile is opened, the open function `mode` parameter is passed to the open function to check if the file is opened for read or write. `Mode` variable takes constant values such as *WRITE_ONLY* and *READ*. Function **open(int mode, int severityLevel)** checks if the mode is *WRITE_ONLY,* then the logfile contents are wiped, and the `index` pointer is set to 0. The severityLevel shows which logging level should be applied as logging everything can exhaust system resources, and logging too less can complicate debugging. Thus, the recommended level

Algorithm 2 Secure Auditing Read function

Require: ldisk is log file, encryption status checks encrypted text or plain text, index variable to check current location to read a file while read operation to ldisk, eof to show end of file.
Ensure: decrypt the log statement if the log is stored in encrypted form,

```
Start
 Initialize ldisk to logfile
   While index !=eof do
     For each ldisk open in read mode do
       If ldisk is encrypted
         Decrypt ldisk
         read ldisk
       Else
         Read ldisk
       End if
 End While
```

of logging is information. The multithread-safe logfile write() and read(>>) methods are provided by employing the mutex resource. The logfile is overloaded for both << and >>. When a thread locks the logfile while writing to the disc, the write () operator writes a string to the logfile. The logfile ldisk is also unlocked soon after the thread finishes writing to it. The operator first finds whether the logfile, ldisk, is encrypted. Subsequently it encrypts the input string using keysize. Lastly, it upgrades the logfile, ldisk, by converting the input variable into a list using the list operator [%%] and appending to the end of the ldisk using the append operator. <>. Lastly, the `index` pointer is updated by 1. The highlighted area shows the immutability feature preventing race conditions, thereby preventing TOCTOU. Function *CheckSeverity(int initLevel)* checks the severity of a message to be stored in a log file. The variable *initLevel* is assigned with an allowed number of characters to be stored in a log file. This specifies the log severity level.

4.2.2. Specification of Secure Error Message

Table 4.2 Specification of Secure Error Message

Signature	SecureErrorMessage ()
System variables	
Private variables	**List** Policy1
User variables	Strıng role **int** username **int** actions **generic** object
User functions	**Bool** isAuthorızedUser() **generic** CheckMessageVerbosity()

Algorithm 3 Secure Error Message

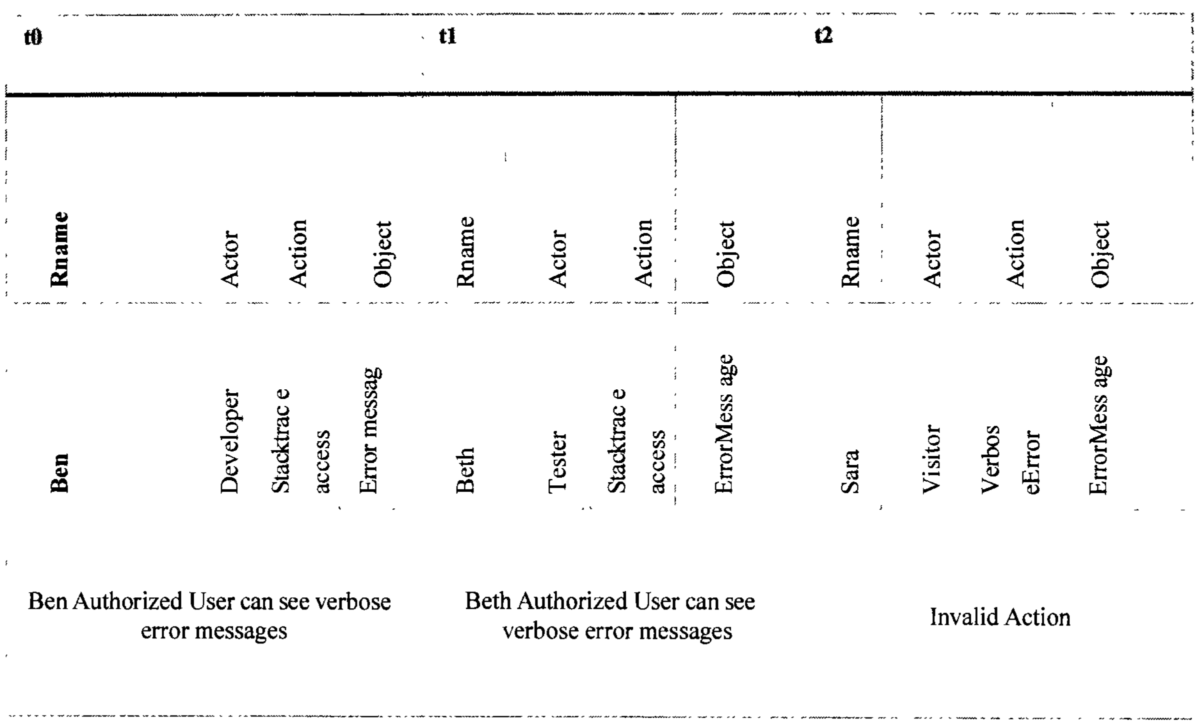
Require: List policyto store organization policy, r is role, level is number of words to control message wordiness, m is the error message, E is the total number of epochs.

Ensure: each role is assigned with a specific message wordiness level to control message verbosity

```
Start
Initialize policyto OrganizationPolicy
  For each epoch e= 1 to E
        If r is authorized user for level on m
           Show level number of words from m to r
        Else
           Show error message
        End if
  End for
```

Testing

In Appendix 2, a list of roles is given with their names, roles, permission and resource. If the user is developer or tester then he has full access to view the stacktrace of error message. Otherwise if the user is a visitor he cannot view stacktrace. The following output is given on different intevals of time from t0,t1,t2 etc.



4.2.3. Specification of Data Validation in RealSpec

RealSpec defines data validation as a resource as a shown in Table 4.3. There are two user variables and two processes one input validation and one is output validation. Process *InputValidation()* has one operator net *dataSanitization()* that detects SQLI pattern such as 1==1 in the input and if found it throws error message. *Pattern()*, *encode*() and *decode()* user functions are added as a library function in RealSpec. In *OutputValidation()* process the output is taken from the user or the data receive from server is check against the patterns <, >, (,), script, alert, if the pattern is found then that data is encoded so that the client machine is prevented from any executable code. The equivalent decode function is also provided as a user function and added to RealSpec. *EncodeData(string s)* takes the ith value of the string s using ith(i,s) and store it in val variable encode that value and then again store that encoded value on the ith place and move to next value using iterative statement *i= 1 fby i+1*. In Figure 4.1, user gives an input the web application checks for the input and finds out if there is some pattern that matches with SQLI attack pattern then if the pattern is found then the web browser throws an exception otherwise the request is processed. Similarly, In Figure 4.2, the user sends request to web browser the web browser checks if there is XSS attack pattern then if there is pattern then the request is not processed and also the user is notified and then the output received from the user is encoded so to avoid any executable code.

Table 4.3 Specification of Input Validation

Signature	DataValidation ()
System variables	
Private variables	
User variables	String input; String output
User functions	**generic** dataSanitization(**generic** query) **generic** EncodeData(string DataFromServer) **generic** Decode(**string** s)

Algorithm 4 Input Validation

Require: I is user input, p is the pattern to find in the input, E is the total number of epochs, q is a query
Ensure: regrex find P in I

```
Start
Initialize I with user input q
  For each Epoch e=1 to E do
      For each where clause do
          Find P in user input
            If found then attack print attack message
            Else print output
            End if
      End for
  End for
```

Algorithm 5 Output Validation

Require: O is user output, p is the pattern to find in the output, E is the total number of epochs,
Ensure: regrex find p in O

```
Start
Initialize O with user output
  For each Epoch e=1 to E do
      For each user input do
          Find P in user Output
            If found then attack encode output
            Else print output
            End if
      End for
      End for
```

4.3. Output Sanitization and XSS Prevention:

The process Output Sanitization () addresses XSS attacks. If the server sends data containing malicious script elements, such as "<script> alert () </script> + document. Cookie", it can trigger an XSS attack. The function Encode Data (data from server) prevents this by converting the executable code into a safe format, ensuring that sensitive data like cookies are not exposed to malicious actors. This step ensures that any user-facing output is sanitized to prevent code execution on the client side.

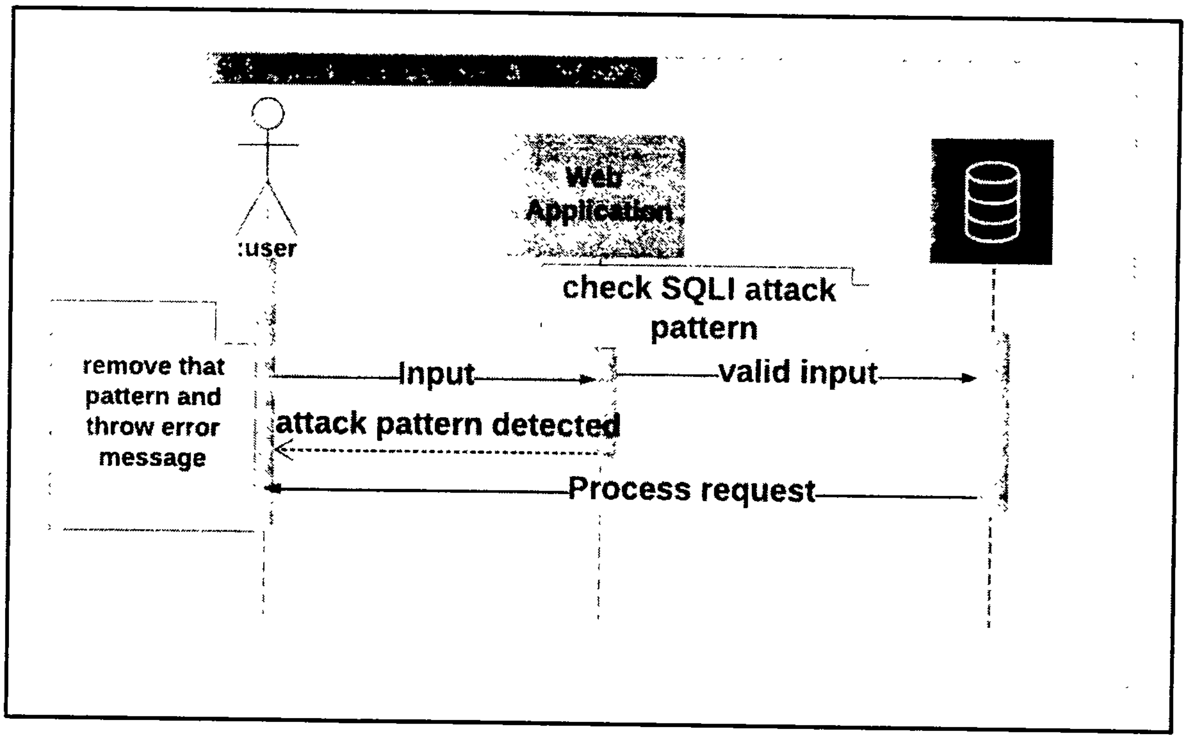


Figure4.1 Specification of Input Validation

In Appendix 3, the process *InputValidation(),* if input is given by illegitimate user as "1==1-- ana OR" thenthe output will be displayed as *Invalid input: SQL injection tautology detected* if select ename from employees where id==001 is given then no SQLI detected is shown. In the process *OutputSanitization(),* if dataFromServer is "<script> alert()</script> + document.cookie" then it can launch an XSS attack. *EncodeData(string dataFromServer)* can convert executable code to prevent the user from sensitive data exposure thereby preventing XSS.

4.4. Bound Access Prohibition and Buffer Overflows:

This step focuses on preventing buffer overflow vulnerabilities. In Real Spec, both input and output buffers are represented as lists, and when an attempt is made to access elements beyond the buffer's bounds, the system returns nil. This behavior is outlined in Table 4.4 and where Bound Check List serves as the resource for managing buffer boundaries.

4.4.1. Bound Access Prohibition

The input buffer or output buffer can be represented by a list in RealSpec and If accessing passes the bounds it will result in nil. This is shown in table 4.4. *BoundCheckList* is a list resource. The system *BufferBoundCheck* has one private variable and one user variable input. It has two operator nets *BoundCheck()* and *addToList()*. The *addToList()* adds items to the list as soon as *asa* the buffer reaches to nil. The operator *asa* will evaluate the right-hand side first. This means the buffer bound is check first for its limit and then items are added to the list. This operator <> appends the input to the *BufferCheckList* by first converting input to the list item using *[% %]* and if the list reaches its limit then exception *BufferOutOfBoundCheckException()*is thrown. This whole function iterates for n number of times using i=*1 fby i+1* in which *i* is initialized with 1 and then *fby* stands for followed by gives the subsequent values of *i*.

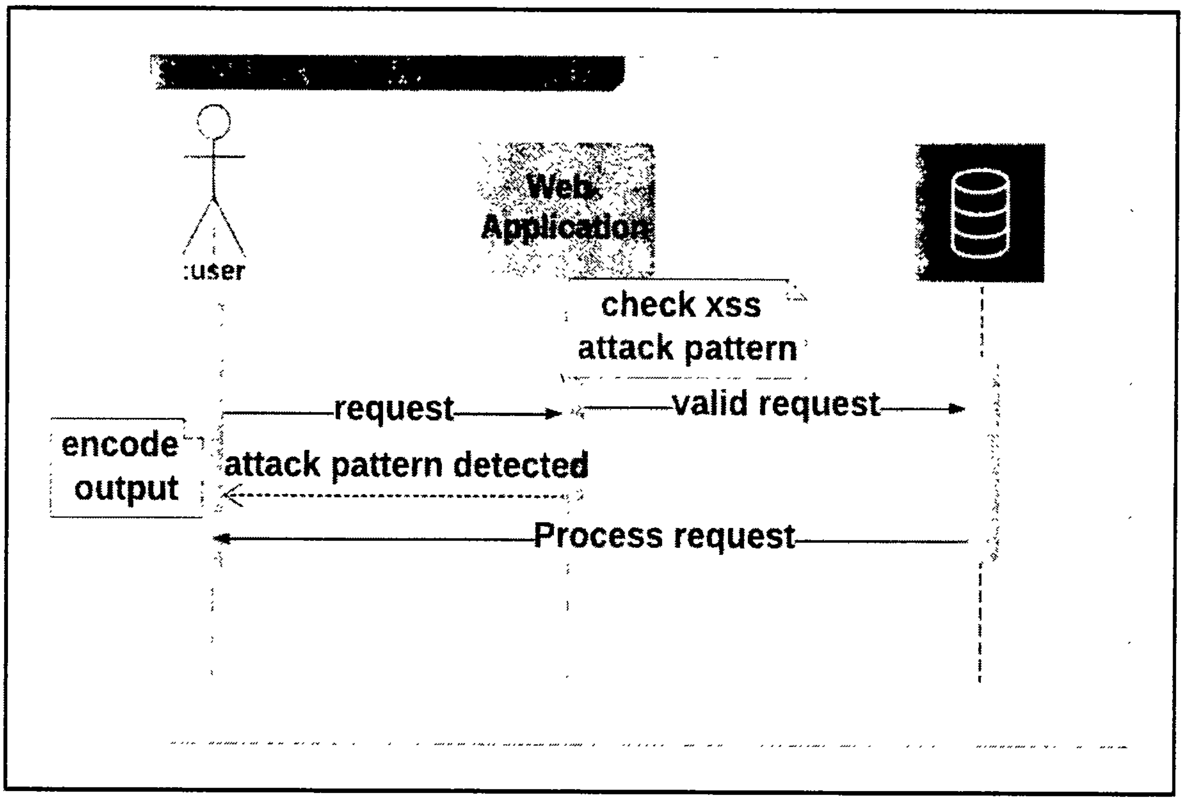


Figure 4.2 Specification of Output Validation

Table 4.4 specification of bound access prohibition

Signature	BufferBoundCheck()
System variables	
Private variables	**list** BoundCheckList
User variables	String input
User functions	**generic** BoundCheck() **generic** addToList()

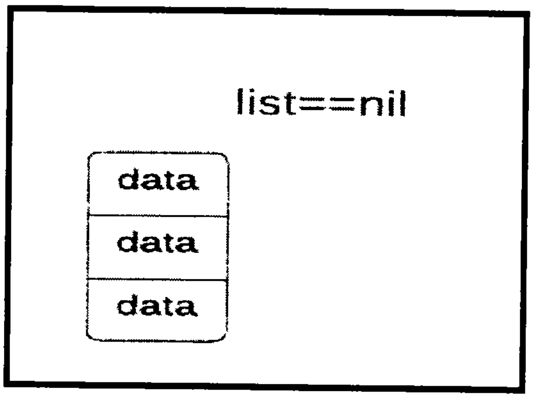


Figure 4.3 Specification Buffer Bound Access

Similar to Java programming language where writing past the length of the buffer throws an exception preventing buffer overflow attack. *BoundCheckList* has three data elements and *addToList*() will always check if the buffer size has reached maximum by checking for nil otherwiswe continue to append data in the list.

Testing

	t0	t1	t2
d0	2	4	Nil
d2	3	6	Nil
d3	4	8	Nil

4.4.2. User Functions

RealSpec has some of the built-in input validation functions or prefix operators such as <prefix

operator>:: = isnumber ¦isword¦ isnull¦isstring¦length¦iseof. However, some prefix operators or validation methods are still missing that can improve the input validation in RealSpec. Data validation has sub-features as input validation, bound access prohibition, output validation RealSpec has some built-in prefix operator as defined below. Some of the functions are identified which are not present in RealSpec, and the addition of these functions in RealSpeccan enhance its security.

Table 4.5 User Functions

Functions	**Uses**
Pattern ([a-z, special characters, A-Z]	RealSpec has • ==, !=, • *Length,* • *Isstring, Isnumber*
Strncpy(s1, s2,n)	When the s2 string is copied to another one, s1 and s2 must be validated and then copied to s1 N is an integer that tells several characters to be copied to prevent buffer overflow
Encode String	Malicious code can contain JavaScript and executable code. So, the input must be checked against (,), <,>, & special characters and replaced with ASCII value. When a client visits this web page, the output will be encoded and not in the form of executable code.
Range (int min, int max)	To check the data is within the valid range.

4.4.3. String copy method

Strncpy (s1, s2, n) mathematically, the S2 string should be checked for string type and length, and then the S1 length is checked to determine whether it is equal to or greater than the length of the n. Only then is the copy of S1 is allowed. S11, S12,....S1N and S21, S22,.....S2N are two string constants streams n is the number of characters to be copied to prevent buffer overflow. For instance, the strcpy (S1, S2) function in C has a flaw and leads to a buffer overflow. As shown in RealSpec specification below

```
S11=S21
S12=S22
.....
 S1N=S2N
```

Here, the Realspec code is using *CopyString()* operater net to check the length of S1 to be greater than S2, if this condition is true then another operator net *checkString* is called that checks the datatype of S2 to be string and then it copies S2 to S1. If S2 is not a string type then an exception is thrown which tells the user that the input is not valid to be copied.

String copy in RealSpec

```
String s1,s2;
bool strncpy(s1, s2, 2)= CopyString
        where{
        CopyString = if length S1>=length S2 then checkString else throw
         InvalidDataException()
            where{
            CheckString=  if isstring  S2 then  S1=S2 else
           throw InvalidDataException()
            }
Exception InvalidDataException() = PrintError
        where{
        printError= "Invalid input"
        }
```

4.4.3. Specification of Type System

Real Spec is a strongly typed system that ensures proper handling and safety of data types. This means that each variable or resource in Real Spec must adhere strictly to a predefined type, enhancing reliability and minimizing errors. It supports both type of initialization and immutability, ensuring that once a type is set or an immutable object is created, it cannot be modified. However, type casting, or converting one data type to another, is not built into the core specification of Real Spec but rather is left as an implementation-specific feature, allowing for some flexibility based on use case needs. Real Spec emphasizes thread safety and immutability, which are crucial for preventing issues in multi-threaded environments. By using synchronization primitives like mutex (mutual exclusion) and semaphore, Real Spec ensures that only one thread can access shared resources at a time, avoiding data corruption and race conditions.

The system provides a range of built-in types, including:

- **Primitive types**: int, long, Boolean, string, float, reals
- **Special types**: time and generic (a flexible type for broader data handling)

However, the date type is currently missing in Real Spec. To address this, the function Now () is specified to return the current date and time. The date type is defined as a resource that takes three parameters: day, month, and year. The specification for type safety related to the date type is detailed in Table 4.6 and Appendix 5. This specification includes several operator nets:

- **Check Date ()**: Verifies whether the date input is valid.

- **Validate Date ()**: Ensures the date corresponds with the correct number of days in the month and checks for leap years.
- **Format ()**: Takes a generic date and formats it into a standard **dd/mm/yyyy** format.

Additionally, the Now(d, m, y) function returns the current date using the day, month, and year parameters. VerifyDays() checks the validity of the number of days in a given month, accounting for leap years, to ensure accuracy.

RealSpec also supports immutability, which ensures that certain data, once set, cannot be altered. This immutability is particularly useful in multi-threaded environments to avoid unintended modifications. The handling of immutable resources is illustrated in Appendix 1, where an example of a user resource named logfile is provided. In this example, a file can be opened either in read or write mode. For writing operations, the operator << is used to direct data to the file.

To guarantee thread safety during file writing, Real Spec employs synchronization mechanisms such as write (), file. Lock(p), and file. Unlock (), ensuring mutual exclusion when accessing the file in a critical section. These operations prevent race conditions, ensuring that processes are executed in a specific order based on their priority and mutual exclusion protocols. This also prevents TOCTOU (Time of Check to Time of Use) vulnerabilities, where the state of a resource might change between the time it is checked and the time it is used.

Through the use of mutex, semaphores, and immutability, Real Spec ensures that operations within critical sections are performed in a controlled and synchronized manner, preventing conflicts and preserving the integrity of the data.

year. *Format(generic date)* will return the date in dd/mm/yyyy format.

In write mode, operator << is used, and by write() asa file.lock(p) asa file unlock() ensures mutual exclusion in a critical section (see [24]-[29]) thereby preventing race conditions and also order of the statements and synchronization of process using process priority and mutual exclusion prevents TOCTOU.

Table 4.6 Specification of Type Safety

Signature	date(**int** d, **generic** m, **int** y)
System variables	
Private variables	
User variables	
User functions	**generic** CheckDate() **generic** VerifyDate(**int** d, **int** m, **int** y) **generic** Format(**generic** date) **generic** now(**int** d, **generic** m, **int** y)

4.4.4. Specification of User Authentication

In figure 4.4, if the username and password are correct then the user is authenticated. Error message showing either username or password invalid prevents information loss by making an attacker wonder if the username or password is incorrect preventing sensitive information loss. Table 4.7 outlines system user variables and user functions. There is one process *Authentication()* that has one user function that validates username and password with the saved credentials if matches then *ValidUserMsg()* is printed.

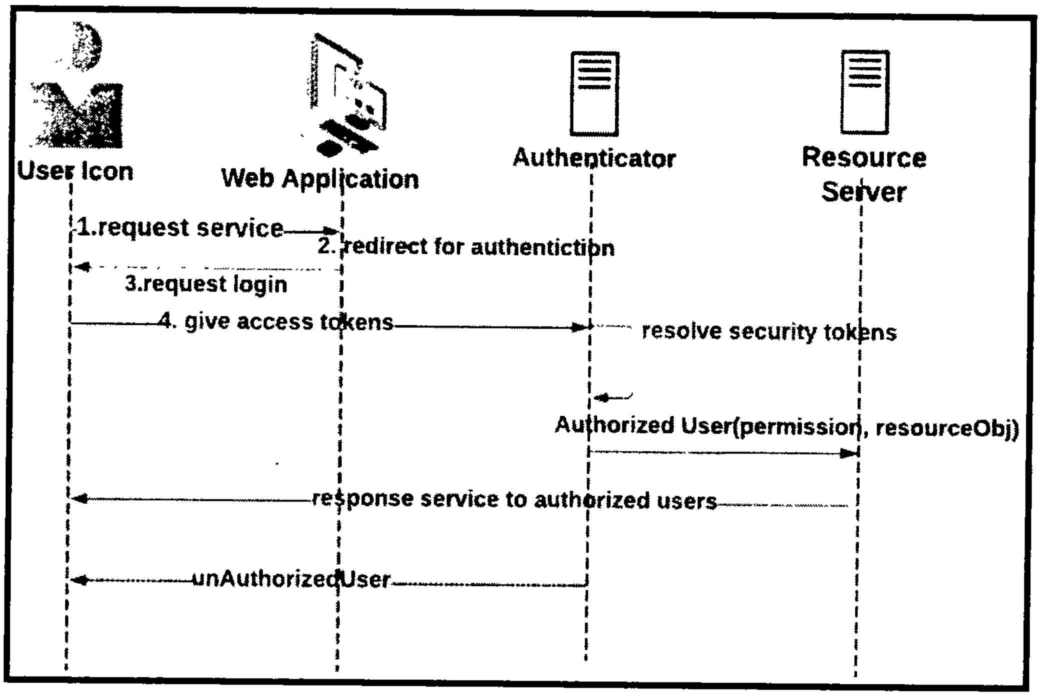


Figure 4.4 Single Sign On Authentication

Table 4.7 Specification of Authentication

Signature	UserAuth ()
System vanables	
Private variables	
User variables	stnng name, string pass
User functions	**bool** isAuthenticated(role r)

4.4.6. Specification of Memory Management

User-defined resources are created in RealSpec. However, release of resources is not shown. In

Table 4.8 the specification of memory safety is given. Here, there is only one user variable and this resource has two operator nets *allocate()* and *deallocate()*. A resource int obj is created after performing some tasks the resource is released if the object is equal to eof using deallocate operator net. The operator eof is added as a keyword in RealSpec.

Table 4.8 Specification of Memory Safety

Signature	memorySafety ()
System variables	
Private variables	
User variables	**int** obj
User functions	Allocate() deallocate()

Testing

In the specification given in Appendix 7, a global resource tl is declared and after some manipulation, the resource is released to prevent memory exploits. A special symbol eof checks the end of data forthe timer and then nil is assigned to release it. The output of the tl resource given on different time intervals such as t0 to t5 and at t5 the resource is released and assigned to eof/nil to prevent any memory leaks.

t0	tl	t2	t3	t4	t5
12:00:2	12:01:2	12:02:2	12:03:02	12:04:02	eof/nil

In Table 4.9, specification of secure information flow is shown. Here, there are two user variables string username and timer t, and process session manager is responsible for creating a session using operator net *createSession()* creates a user session it has variables to store current time and session expiration time. Process *AuthenticationCheck()* checks the authentication using operator net *Authenticate()* to check if the user session is still available by checking *currentTime<=expirationTime (see Appendix 8).*

In secure information flow specification, a user is assigned with a commID after the user is an authorized user and can perform desired operation if the session timeout is not over by @ expirationtime. After the desired operation is performed by the user on the resource the session is terminated. This prevents a broken authentication and session management attack, as well as establishing a timeout for an idle online session.

4.4.7. Specification of Communication Security

Communication Security is modeled here by specifying the blacklisted version and whitelisted version of SLS/TLS. Certificate validity can be replaced by user authentication. In the following code snippet, TLS/SLS security is taken as a resource and it has a default version set as TLSv1.2.If the default version is SSLv3 or SSlv2 then *ExceptionVulnerableVersion()* is thrown. Using secure versions can prevent sensitive information exposure. Note: the RealSpec Specification is a formal specification language and it is at design and analysis phase, implementation to many functions are not provided as it is a specification language and the implementation is provided in implementation phase using a specific programming language.

Table 4.9 Specification of Secure Information Flow

Signature	SecureInformationFlow ()
System variables	
Private variables	
User variables	String username **Timer** t1
User functions	**generic** CreateSession()

Table 4.10 Specification of Communication Security in RealSpec

Signature	CommunicationSecurity ()
System variables	
Private variables	
User variables	**generic** certificate
User functions	**generic** Integrity() **generic** Confidentiality()

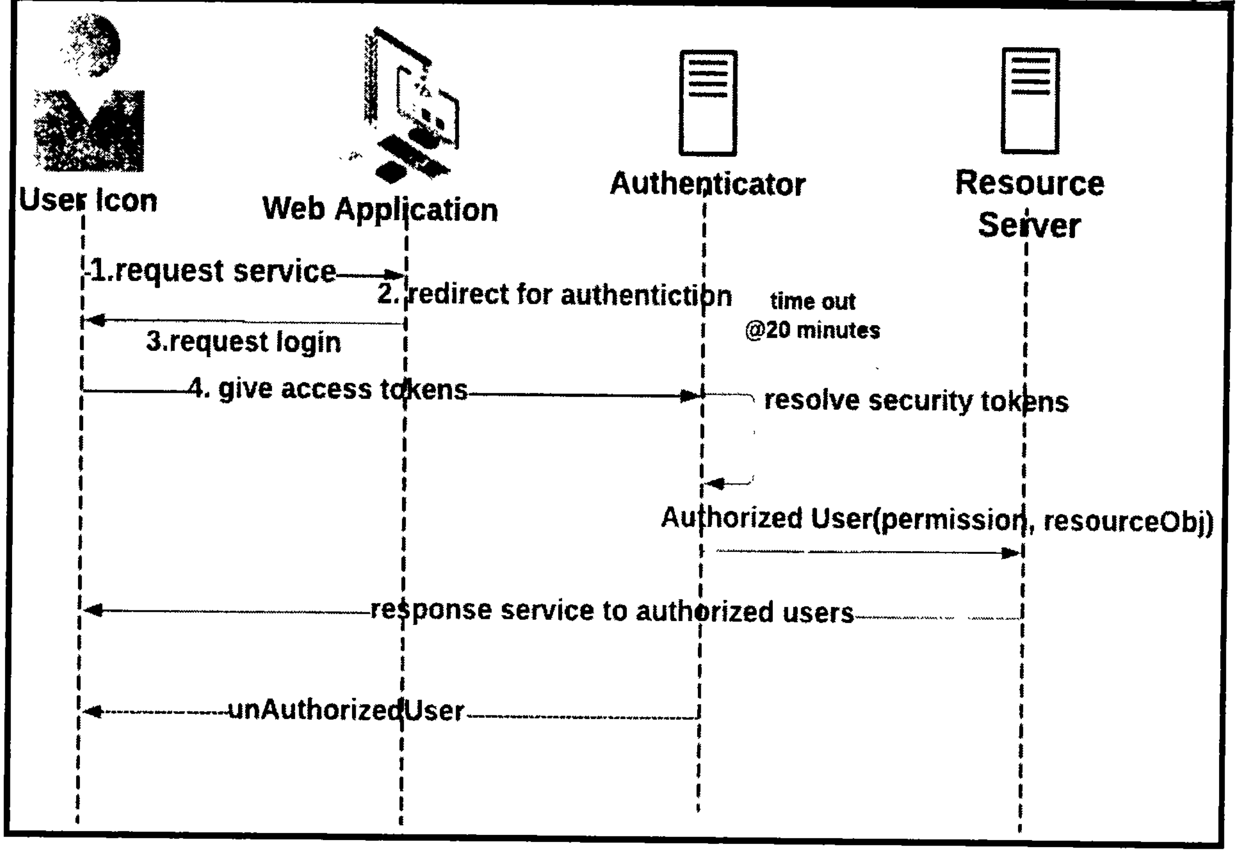
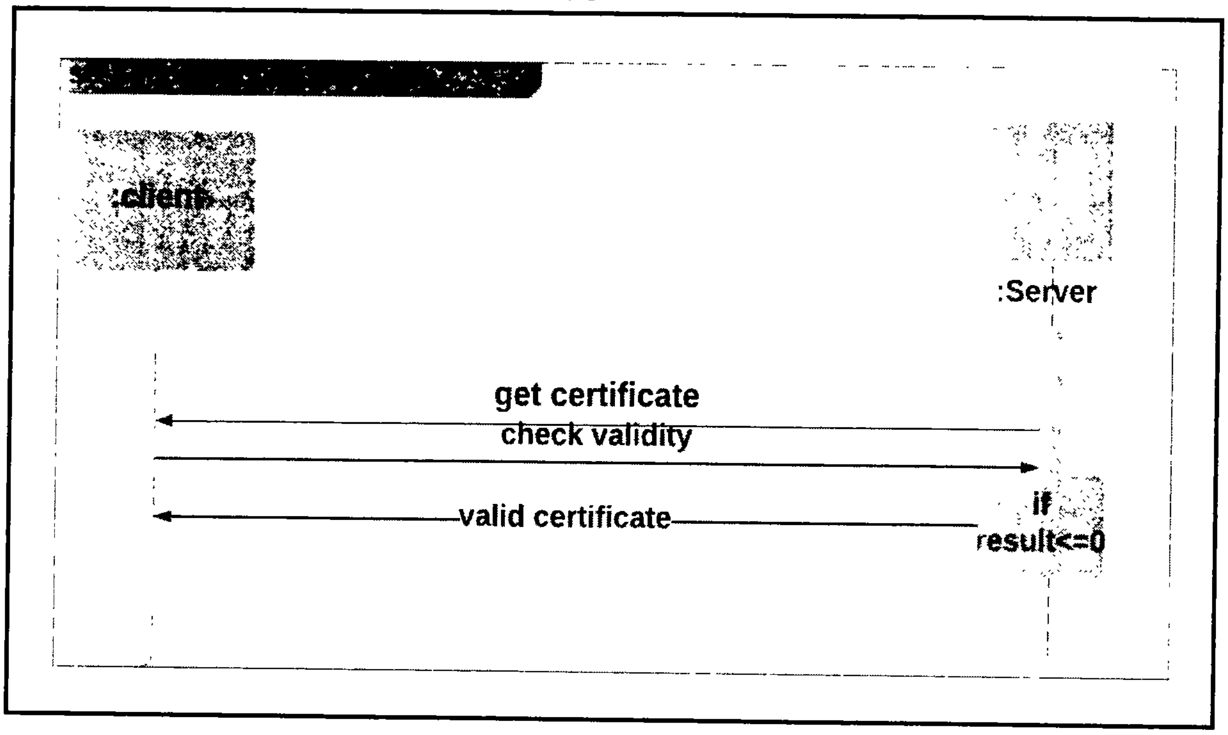


Figure 4.5 Session Management and session timeout

Figure 4.6 Secure Configuration



4.5. Comparative Analysis of RealSpec vs Model Checking techniques

This section evaluates the RealSpec approach in comparison to various model checking techniques. It provides an overview of major model checking methods and contrasts their general methodologies with the executable model approach of RealSpec.

S-Promela: S-PROMELA (a Secure Process Meta Language) is used for modeling, specifying, and validating Security Policies (SP), drawing inspiration from software engineering practices. The approach is based on Executable Specification Policy (ESP) and reachability graph concepts, introduces S-Promela as a new executable language for SP specification. This language is structured around channels, processes, and operations like read and write using Labelled Transition Systems semantics. Validation proceeds through three key steps: detecting inconsistencies in SP rules, proving completeness using reachability graphs, and verifying the preservation of safety and liveness properties.

SecureSOA is secure specification language for service-oriented architecture. It employees Alloy[156] to specify authentication, confidentiality and integrity. Alloy [156] is a declarative specification language designed for expressing complex structural constraints and behavior in software systems. Drawing upon first-order logic principles with influences from the Z language and syntax resemblant of the Object Constraint Language (OCL), Alloy excels in creating concise micro-models that undergo automated correctness verification. At the heart of Alloy is its analyzer, inspired by model checkers, which functions as a "model finder" powered by a Boolean satisfiability (SAT) solver. This tool generates instances of model invariants, simulates model operations, and verifies user-specified properties. Notably, Alloy supports incremental model analysis, allowing developers to refine and validate models step-by-step, providing immediate feedback. To maintain computational feasibility, Alloy's analyzer imposes constraints on model complexity by limiting the scope to a finite number of defined objects, ensuring the decidability of the model finding process while potentially constraining the generality of its results.

Model Checking Methods vs. RealSpec Execution Model: Model checking may be a complementary approach to the RealSpec executable approach. Model checking may provide the following advantages over RealSpec:

Verification Scope – Model checking methods verify the correctness of the entire specification by exhaustively searching the system's state space. In contrast, RealSpec's execution model restricts validation to the current context of inputs and user interactions, providing a narrower scope for verifying correctness based on specific scenarios and behaviors.

RealSpec may provide the following benefits over model checking methods:

- **Usability** – Model checking typically involves substantial preparatory work before verification can commence, including translating the specification into a formalism compatible with a modeling tool and formulating system properties using logical formalisms. In contrast, RealSpec's execution model is more straightforward and resembles programming languages, making it easier to directly simulate and observe system behaviors without the upfront formalism conversion required by model checking methods.
- **Completeness** – Model checking faces challenges regarding the completeness of system properties specification. While it verifies if a model satisfies a specified set of properties, it cannot confirm whether these properties encompass all requirements the system should meet [170]. Similarly, the RealSpec execution model may encounter similar issues where it's difficult to ascertain if the specification covers all system properties comprehensively. However, RealSpec's advantage lies in its ability to execute and observe the specification in a defined context of inputs and expected outputs, which can help highlight any missing properties during testing and simulation.
- **Debugging** – While model checking methods offer automated verification, in practice, human intervention is often necessary to interpret verification results and debug errors, especially when a negative result occurs and an error trace needs to be analyzed. Similarly, RealSpec's execution model also requires human analysis in case of failures. However, debugging formal system models with an exhaustive state space and property axioms expressed in another language adds considerable complexity compared to debugging RealSpec specifications. RealSpec's debugging process is simpler because it is based on a straightforward functional input/output model, making it more intuitive and less intricate than dealing with the complexities of formal model checking methodologies.
- **Applicability to Software** – Model checking has proven effective in verifying hardware systems and communication protocols but faces challenges with software due to its less structured nature compared to hardware [170]. Software systems often involve dynamic and complex interactions that are harder to formalize and verify exhaustively using model checking techniques. In contrast, RealSpec is specifically designed as an executable specification language tailored for modeling software systems. It closely resembles programming languages,

making it more suitable for capturing and validating the behaviors and interactions typical of software applications. Therefore, RealSpec does not encounter the same difficulties with software verification that model checking may face, leveraging its approach to simulate and validate software behaviors directly.

- **Complexity Control** – Model checking faces the significant challenge of state space explosion in systems with numerous interacting components or complex data structures, leading to an impractically large number of potential global states to explore. Despite advancements in methods like symbolic model checking and state space reduction techniques, these solutions often introduce additional complexity to the already intricate modeling and verification processes. In contrast, the RealSpec execution model sidesteps the state space explosion problem altogether by focusing on the direct execution and simulation of specified system behaviors within a defined input-output framework. This approach simplifies verification efforts by avoiding the exhaustive exploration of all possible system states, making it a more straightforward and efficient method for validating software systems compared to traditional model checking approaches.

4.6. Evaluation of RealSpec Using Security Features Framework

The evaluation of Real Spec using the established Security Features Framework, as outlined in Table 4.11, provides a comprehensive assessment of the system's security capabilities. This framework serves as a benchmark against which the various security features and sub-features of Real Spec can be systematically analyzed.

In Table 4.11, the first column lists the specific security features or sub-features relevant to Real Spec, while the second column details their respective implementations or characteristics within the system. The third column offers an analysis or evaluation of how effectively each feature addresses security concerns, focusing on the robustness and reliability of the mechanisms in place.

The evaluation process considers several key aspects, including:

Input Validation: The framework assesses the built-in input validation functions available in Real Spec, such as prefix operators like is number, is string, and others. The effectiveness of these functions in preventing invalid inputs and potential security vulnerabilities, like SQL injection, is evaluated is a critical consideration.

Table 4.11 Evaluation of RealSpec using SEFF

Security Feature	Security Sub-feature	Supported by RealSpec?	How?
Log File Protection	Log File Protection	Yes	Secure Auditing is supported by RealSpec using **lock** and **unlock** to read and write file Providing mutual exclusion and thread safety.
	Log Message Control	Yes	Log Message control is supported by using severity level of the log such as **debug, info, warning, information,** and **error** message
	Error Message Control	Yes	Error message is specified by defining a **role** as a **resource,** its permission on error message verbosity level.'
Data Validation	Database Query Security	Yes	By specifying **database** as a **resource** and passing input to the database but that input is checked against XSS and SQLI attack pattern.
	User Input Security	Yes	Providing **checks** when the input is taken from the user.
	Input Buffer Size Check	Yes,	It is specified using **operator net** to **check bounds** and then throw an exception in case the access is past the bounds of the limit
	Encode Output	Yes	This feature is specified by using **operator net** that checks if **XSS attack** pattern input is detected in the output then the output is encoded before being displayed to the user.
Memory Management	Memory Management Control	Yes	This feature is specified by defining a resource in RealSpec and then released after it is no longer needed it uses eod or nil to release the resource.
	Memory Address Arithmetic	No	It cannot be specified at the specification level because it is a low-level security feature implemented in a programming language using functions or operations on pointers if the underlying programming language supports this.
	Array Out of Bound Check	Yes	Same as input buffer bound size check
	Memory Management Flexibility	No	Same reason as Memory Address Arithmetic
User Authentication		Yes	This is specified by defining a role, its permission on the resources. Then giving **username** and **password** and an **OTP** to perform user authentication.
Access Control	Authorization Types	Yes	This is specified by defining **roles** as **resource** and the defining organization policy. RealSpec employees **RBAC.**
	Encapsulation-Based Access Control	Yes	This constraint is defined by defining **System variables** that are accessible to only using user-functions/operator nets.
	Sandbox Support	No	It cannot be specified at a high level because the sandbox is a low-level security environmental feature. It is an execution environment to test untrusted code for malicious activity.
Type System	Type Safety	Yes	It is specified by defining **date** type for RealSpec.
	Type Casting	No	Unsafe type casting or implicit type casting can cause a loss of useful information It is a low-level programming security feature that requires built-in functions for every data type and its corresponding data type, such as integer to float. Implementation of this feature in a programming language and specification is impossible.
	Type Initialization	Yes	This feature is specified by giving **default values** to the variables.
	Immutability	Yes	This feature is specified by defining **lock** and **unlock** feature same as log file protection.

Security Feature	Security Sub-feature	Supported by Real Spec?	How?
Secure Session Management	Secure Session ID	Yes	This feature is specified by assigning **sessID** for each role when it is authorized.
	Session Timeout	Yes	This security feature is defined by using a **@timeout** constraint for each sessionID as 30 seconds
	Secure Cookies	Yes	This is partially specified as it is a flag which is to be set for HTTPOnly
Communication Security	SSL/TLS version support	Yes	This feature is partially specified as it is a network level feature which has some implementation requirement to be implemented in design and implementation phase. It is defined as an operator net where older versions of SSL/TLS checked and an exception is thrown if the client is using an older version of SSL/TLS.
	Cryptographic Algorithms	Yes	This feature is specified by providing encryption and decryption on a file if it contains sensitive information.
	Key Length	No	It is a low-level security feature
	Random Number Method	No	It is a low-level security feature
	Certificate Validation	Yes	This feature is partially specified as the path of the CA is required. Here it specified in RealSpec as an operator net that checks for certificate authority and certificate expiry and throws an exception if the certificate is expired or the certificate authority is not valid.

Memory Management: The evaluation examines how Real Spec manages memory through user-defined resources, focusing on the proper allocation and deallocation of memory to prevent leaks and exploits. The presence of mechanisms for resource release is scrutinized to ensure effective memory safety.

Session Management: The framework evaluates the secure session management features, including user authentication and session timeout mechanisms. The effectiveness of these features in preventing broken authentication and ensuring that sessions are terminated appropriately

Communication Security: This aspect focuses on the use of secure protocols, such as TLS and SSL, and the specification of blacklisted and whitelisted versions. The evaluation assesses whether the implementation safeguards against known vulnerabilities and protects sensitive data during transmission.

User Authentication: The evaluation includes the processes for verifying user identities and assigning unique identifiers (comm IDs) for authorized operations. The effectiveness of these measures in preventing unauthorized access is critically assessed.

Overall, the evaluation of Real Spec using the Security Features Framework offers a structured analysis of its security features, identifying strengths and potential areas for improvement. By systematically reviewing each feature against established security standards, this evaluation helps to ensure that Real Spec is robust against common vulnerabilities and threats, ultimately enhancing its overall security posture.

4.7. Summary

In Chapter 4, introduction of RealSpec is given such as its syntax and semantics. In Section 4.1, input validation and SQLI prevention is given, moreover, security features such as log file protection, input validation, are specified in RealSpec using its syntax and semantics. This section also provided a testing of each feature which show correctness of the proposed scheme. Section 4.2 discussed about output validation and type safety, user authentication, memory management, secure session management, and communication security. In Section 4.4, bound access prohibition and buffer overflows prevention is showed. Section 4.4 was the comparative analysis of the proposed method with model checking. Section 4.5 gave a comparative analysis of RealSpec vs model checking methods. Section 4.6 evaluated RealSpec using SEFF. Section 4.7 summarized the chapter.

Chapter 5

Case Study

Web application security is a critical concern in today's web-based environment, primarily due to the increasing number of cyber threats and vulnerabilities that target online systems. Among these threats are broken authentication and session management, SQL injection attacks (SQLIA), cross-site scripting (XSS), race conditions, and buffer overruns, which are some of the key vulnerabilities identified by the Open Web Application Security Project (OWASP). These vulnerabilities can lead to significant security breaches, compromising sensitive data and undermining user trust.

To effectively address security issues—considered a non-functional requirement—it's essential to incorporate security measures throughout the various phases of the web application development lifecycle. By doing so, developers can improve the overall quality of web applications and enhance their stability against security threats. This proactive approach helps in identifying and mitigating risks early in the development process, ensuring that security is not an afterthought but an integral part of application design and implementation.

Access control is a vital component of web application security, allowing information system authorities to grant appropriate access to various system resources. There are several types of access control mechanisms, including Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Task-Based Access Control (TBAC). However, Role-Based Access Control (RBAC) has emerged as a particularly popular framework. In RBAC, roles, permissions, and resources are the key entities. Users are assigned specific roles within an organization, and each role is granted access to system resources based on its permissions. This model streamlines access management by aligning user privileges with organizational roles, thereby minimizing the risk of unauthorized access.

Real Spec, as a formal specification language, outlines various banking obligations and policies relevant to security in financial applications. It defines user resources and their interactions, ensuring that security measures align with regulatory requirements and industry best practices. By specifying these elements, Real Spec aims to provide a structured approach to security, Facilitating the development of secure web applications that can withstand prevalent threats while maintaining user trust and data integrity.

5.1. Specification of Security Features in Real Spec: A Case Study

This section presents a case study of a banking system to illustrate the specification of security features within the Real Spec framework. The case study is derived from sources identified in [14] and serves as a practical example for detailing various security measures, including access control, data validation, secure error messaging, immutability, secure auditing, bound access prohibition, integrity, confidentiality, and secure information flow.In this context, the proposed work aligns with the Role-Based Access Control (RBAC) standard developed by the American National Standards Institute (ANSI), as referenced in [160]. The RBAC model provides a structured way to manage user permissions and roles within the banking system, ensuring that only authorized individuals have access to sensitive information and critical functionalities.

5.1.1. Access Control:

The specification emphasizes the importance of robust access control mechanisms, allowing for the assignment of user roles that dictate access levels to various system resources. By implementing RBAC, the banking system can effectively manage user permissions based on their roles, thereby minimizing the risk of unauthorized access.

5.1.2. Data Validation:

The case study highlights the necessity of rigorous data validation processes to prevent common vulnerabilities such as SQL injection and cross-site scripting. Real Spec facilitates the specification of input validation methods that ensure only legitimate data is processed, thereby safeguarding the integrity of the banking system.

5.1.3. Secure Error Messaging:

Secure error messages are crucial for maintaining confidentiality and preventing information leakage. The specification in Real Spec details how to implement error handling that does not expose sensitive system information, thus enhancing overall security.

5.1.4. Immutability:

Immutability is emphasized in the specification to ensure that once data is created, it cannot be altered, thereby preventing unauthorized changes. This characteristic is vital for maintaining the integrity of financial transactions and user information within the banking system.

5.1.5. Secure Auditing:

The case study also discusses the implementation of secure auditing features that log user actions and system events. This audit trail is essential for tracking access and changes, facilitating compliance with regulatory standards, and enabling forensic analysis in the event of a security breach.

5.1.6. Bound Access Prohibition:

The specification addresses bound access prohibition to prevent buffer overflows and unauthorized memory access, ensuring that user actions remain within defined limits.

5.1.7. Integrity and Confidentiality:

Maintaining the integrity and confidentiality of data is paramount in a banking system. The specification details measures to protect sensitive information from unauthorized access or modification, ensuring that customer data remains secure.

5.1.8. Secure Information Flow:

Finally, the specification outlines how secure information flow is maintained throughout the system, ensuring that data is transmitted securely and is only accessible by authorized users. Overall, this case study demonstrates how Real Spec can be used to specify and implement a comprehensive security framework within a banking system. By addressing these critical security features, the system is better equipped to withstand potential threats and vulnerabilities, ultimately enhancing its resilience and reliability in a digital environment.

5.2. Banking System Case Study

In this banking system case study, various roles such as tellers, customer service representatives, loan officers, accounting managers, internal auditors, and branch managers are identified as key participants in the operational structure. These roles interact with different system resources or objects, including deposit accounts, loan accounts, ledger accounts, ledger posting rules, and general ledger reports. User authentication is facilitated through query submissions, ensuring that only verified individuals can access specific functionalities.The case study outlines the permissions, also referred to as actions or business tasks that each actor may perform on system resources:

a) Create, delete, and modify customer deposit accounts.

b) Set up a ledger report.

c) Establish, remove, and change customer loan accounts.

d) Modify or verify the ledger posting rules.

Each actor in the banking system has designated responsibilities:

a) Tellers are responsible for entering or updating customer deposit details.

b) Customer Service Representatives can set up or alter deposit accounts.

c) Loan Officers have the authority to establish loan accounts.

d) Accounting Managers may alter the ledger reports.

e) Internal Auditors can set up ledger posting rules.

f) Branch Managers possess the capability to perform all tasks inherited from other roles, reflecting a high level of authority and responsibility.

To maintain security and ensure proper operational integrity, several authorization rules are implemented in the banking system, focusing on separation of duties and the principle of least privilege. These rules include:

Rule 1: Certain duties, such as those of a customer service representative and an account manager, cannot be assigned to the same user, preventing potential conflicts of interest.

Rule 2: Users cannot activate certain bank roles, such as customer service representatives and loan officers, within the same session, which helps to prevent unauthorized actions.

Rule 3: Certain bank actors are restricted from assuming multiple bank roles simultaneously, ensuring that no individual holds excessive power within the system.

Rule 4: A bank actor may only play role A if they have been assigned a different role B, establishing clear role definitions and boundaries.

Rule 5: A banking actor may conduct a banking task only if they are also permitted to perform another banking role, reinforcing the principle of least privilege.

Rule 6: The number of users that a banking position can accommodate should be limited, helping to prevent any single point of failure or overloading of authority.

Overall, this case study illustrates the careful design of access control within the banking system, ensuring that security measures are effectively implemented to protect sensitive information and maintain operational integrity. By clearly defining roles, responsibilities, and associated permissions, the banking system can mitigate risks associated with unauthorized access and potentially fraudulent activities.

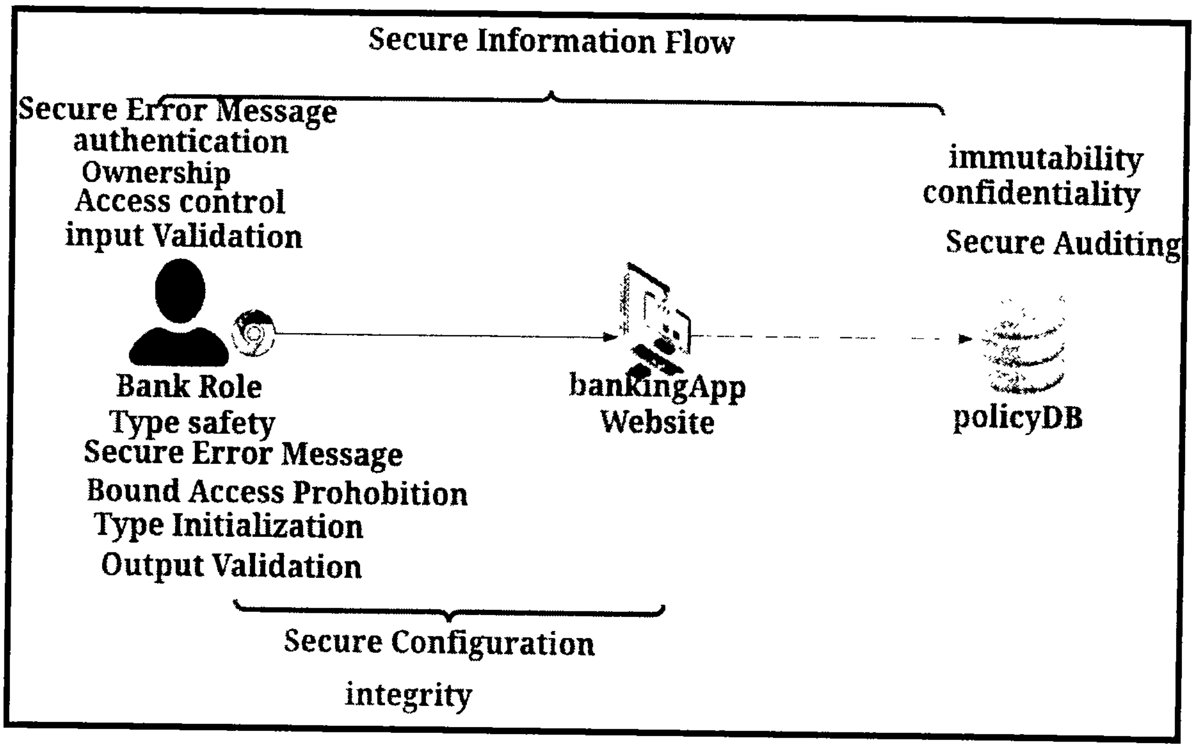


Figure 5.1 Security Features in a Web Application System

Table 5.1 Banking system access control roles and policies

RoleID	BankRoles/Subject	Bank Permissions/Permission	BankResources/Object
1	CustomerServiceRep	createDepositAccount	DepositAccount
2	CustomerServiceRep	modifyDepositAccount	DepositAccount
3	Teller	inputDepositAccount	DepositAccount
4	Teller	modifyDepositAccount	DepositAccount
5	LoanOfficer	createLoanAccount	LoanAccount
6	Accountant	CreateLedgerAccount	LedgerAccount
7	AccountingManager	ModifyLedgerReport	General Ledger Report
8	InternalAuditor	CreateLedgerPositingRule	LedgerPostingRule
9	BranchManager	VerifyLedgerReport	LedgerPostingRule
10	Client	modifyDepositAccount	DepositAccount

Table 5.2 RealSpec constructs for the banking system

RealSpec Processes and Resources	Type	Description
PolicyDatabase	Global Resource	Specifies database object
DepositAccount, LoanAccount, LedgerAccount, GeneralLedgerReport, LedgerPostingRule,	Global Resources	Specifies account type
PolicySetupProcess	Process	Defines policy, stores in database
AuthorizationCheckProcess	Process	Validates users and permits resources
Role	Local Resource	Defines roles
Policy	Local Resource	Defines policy
dataValidation	Global Resource	Validates user input and user output
LogFile	Global Resource	Logs user data
Authentication	Global Resource	Authenticates a user
SecureCommunication	Global Resource	Provides integrity and confidentiality
SecureInformationFlow	Global Resource	checks bound limit prohibition

5.3. Specification of Banking System Application in RealSpec

In the Real Spec specification for a banking system application, the foundation is established with a system construct that outlines the overall structure and framework for defining the banking system. This structure is composed of declarations for global resources and processes integral to the application's functionality.

The global resources identified within this case study include Policy Database, Deposit Account, Loan Account, Ledger Account, General Ledger Report, and Ledger Posting Rules. These resources serve as essential components for managing various aspects of the banking operations and ensuring compliance with organizational policies.

Within the banking system application, there are two synchronous processes: Policy Setup Process and Authorization Check Process. The execution of these processes follows a specific order, with Policy Setup Process initiating first, setting the stage for subsequent activities, and then followed by Authorization Check Process, which plays a critical role in validating user permissions and access.

The Policy Setup Process is responsible for defining roles, establishing policies, and storing these organizational policies in the Policy Database. This process comprises two user functions: set Policy Definition and add User Roles.

1. **Set Policy** Definition takes two integer arguments, namely operation and bank Role, which are utilized to check and store the permissions and roles associated with various banking

activities. The function ensures that the relevant policies are stored within the Policy Database resource using the set Policy function.

2. **The implementation of set Policy** Definition incorporates a case statement to manage the different roles and permissions efficiently. As illustrated in Table 22, there are two specific permissions—create and update—assigned to the CUSTOMERSERVICEREP bank role. To effectively demonstrate the relationship between roles and permissions, nested case statements are employed, highlighting multiple permissions available for both CUSTOMERSERVICEREP and TELLER roles. It is important to note that the default case is optional, allowing for flexibility in handling undefined cases.

The second user function, add User Roles, plays a crucial role in assigning user roles to the Policy Database through the db Policy Storage instance. This function ensures that all users are appropriately categorized according to their roles, reinforcing the security and operational integrity of the banking system. Overall, the specification of the banking system application in Real Spec provides a comprehensive framework for establishing a secure and efficient operational environment. By clearly defining global resources, processes, and user functions, the banking system can effectively manage permissions, enforce policies, and maintain the necessary safeguards against unauthorized access, thereby enhancing its resilience against potential security threats.

Listing 13: Banking System Application

```
System Banking System Web Application{
    Resources {
        Policy Database db.
        Policy Storage ;
         Deposit Account deposit Account;
        Loan Account loan Account;
        Ledger Account ledger Account;
        GeneralLedgerReport
        generalLedgerReport; LedgerPostingRule
        ledgerPostingRule;
        }
Processes {
    Policy Setup Process,
    Authorization Check Process;
    }
}
```

Local Resources: Role and Policy

In the context of the banking system application as specified in Real Spec, local resources Role and Policy are defined to manage user roles and access permissions effectively. These resources encompass specific variables that facilitate the tracking and control of user access within the banking system.

The Role resource consists of four key variables:

i. **Username**: This variable stores the name of the user associated with a specific role.

ii. **Role Name**: This variable represents the primary bank role assigned to the user.

iii. **Role2**: This variable signifies a prerequisite role that may be required for certain functions within the banking system.

iv. **num**: This variable is used to store the number of clients that a particular role can attend, thereby helping to manage workload and access efficiently.

Similarly, the Policy resource is defined with three critical variables:

i. **Name**: This variable captures the name of the user role.

ii. **permission**: This variable specifies the access permissions granted to the user based on their role.

iii. **obj**: This variable identifies the banking object to which access permissions apply, ensuring that users can only interact with authorized resources.

Global Resource: Policy Database

The global resource policy database is essential for managing and storing role and policy information within the banking system. It is structured to contain two main lists:

i. **User List**: This list is utilized to store the various roles assigned to users within the organization.

ii. **Policy List**: This list contains the policies that govern access and operations within the banking system.

iii. The Policy Database resource is equipped with four user functions that facilitate the management of policies and roles:

iv. **set Policy ()**: This function is responsible for adding new policies to the policy List. It first checks for the existence of a policy using the check Policy Exist ()

function, which traverses the policy List to determine if the policy is already present.

v. **Check Policy Exist ()**: This function scans through the policy List to verify whether a specified policy exists. It returns a Boolean indicating the presence or absence of the policy.

vi. **Add User Role ()**: This function is used to add roles to the user List, ensuring that all user roles are properly documented and tracked.

vii. **Check Role Exist ()**: This function checks if a specific user role is present within the user List, enabling the system to validate user roles effectively.

Functionality of set Policy ()

When the set Policy () function is invoked, it first utilizes the check Policy Exist () function to ascertain whether the desired policy is already recorded in the policy List. If the policy is not found, indicated by policy List == nil, the policy List is then updated. This update process involves converting the data elements of the new policy into a suitable format for storage, ensuring that the policy is accurately recorded in the system.

This structured approach to defining local resources and managing global resources like Policy Database establishes a robust framework for access control within the banking system. It enhances security by clearly delineating user roles, permissions, and policies, thereby minimizing the risks associated with unauthorized access and ensuring compliance with organizational standards.

In the Real Spec specification for the banking system, policies are defined using the structure Policy (name, permission, resource Obj), which outlines essential components, including name as the unique identifier for the policy, permission that specifies the access rights granted by the policy, and resource Obj identifying the specific banking resource to which the policy applies. When a new policy is created, it is appended to the policy List using a combination of the list conversion operator and the append operator. The process begins with the list conversion operator which converts the policy data into a format compatible for storage within the policy List. Following this, the append operator (<>) is utilized to add the newly formatted policy to the end of the policy List, ensuring its integration into the existing collection of policies within the system.

Before a new policy can be added to the policy List, the system performs a check to confirm whether the policy already exists, thereby preventing duplication. If the check Policy Exist () function determines that a policy with the same name is already present, a message is printed to indicate this, notifying the user that the attempted addition was unnecessary. If the policy does not exist, it is converted and appended to the policy List. This structured approach to policy management enhances the security framework of the banking system by ensuring that policies are defined clearly, stored systematically, and checked for duplication. This, in turn, supports effective access control and compliance with regulatory standards.

Check Policy Exist() operator net

```
operator net to check policy in the database
Bool checkPolicyExist (int bankRoleId, int permission, generic resourceObj) =findPolicyExist () asa L==nil
where {
L is current policyList;
findPolicyExist          ()     =if     (temp.     roleName==bankRoleId          &&
                         temp. permission==permission && temp.obj== resourceObj then true else false
where {
if L!=nil then temp = hd(L) fby tl(L) else throw outOfBoundAccess();
}
}
```

The check Policy Exist () function in Real Spec is designed to verify the existence of a specific policy within the banking system's policy list. It takes three arguments: two integer variables, bank Role and permission, and one generic variable, resource. The function begins by initializing a local list variable L with a frozen version of the policy List, ensuring that the values of L remain constant during the computation. Additionally, a variable named temp is initialized with hd(L), which refers to the head element of the list L.

As the function executes, the temp variable is updated at each iteration using the (followed by) operator, which allows the function to traverse the list. Specifically, temp takes the value of the tail of L, effectively removing the first element of the previous list from consideration. This process continues until the list L becomes nil, at which point the function returns either true or false based on whether the specified policy was found. If the function successfully identifies a match for the given bank Role and permission, it returns true; otherwise, it continues iterating until the end of the list, ultimately returning false if no match is found. This systematic approach to checking policy existence enhances the overall integrity and security of the banking system's policy management process.

Add User Role () in a user List

```
// operator net to set roles in database
addUserRole         (string   userName,    int   roleName,    int   PRRole,    int   num)   =
                    if checkRoleExist(userName, roleName, PRRole) then PrintMsg else CheckRules()
where {
printMsg= "Role Already Present in DB";
checkRules()=          if    HasSOD(roleName,PRRole)||hasDSOCD(roleName,PRRole)
                       then conflictMsg else addRole()
Where {
conflictMsg = "cannot add roles as conflicting Role";
addRole () = userList <> [%role (userName, roleName, PRRole, num) %];
}
}
```

The add User Role() function plays a crucial role in managing user roles within the banking system by ensuring that roles are appropriately assigned and comply with security protocols. Initially, it checks if the specified role is already associated with a particular user by utilizing the check Role Exist () function. Furthermore, the function addresses the separation of duties (SoD) concern and the dynamic separation of duties (DSoD) rules. The SoD constraint ensures that a bank actor cannot be assigned two conflicting roles, which helps prevent potential fraud or misuse of authority. In contrast, the DSoD constraint verifies that a bank actor cannot assume two conflicting roles within the same session. Both constraints are outlined in Table 23. However, a banking actor is permitted to perform each role in separate sessions. If the checks for both SoD and Dos'd are satisfied, the role is added to the user's profile in the database using the add Role () user function. This function operates similarly to the set Policy () function, effectively integrating the new role into the existing user roles (see Appendix 10).

Following the role management, the banking system includes an access control process known as Authorization Check Process (), which verifies whether a user is authorized to access specific elements within the banking system. This process relies on the policy database established during the earlier Policy Setup Process, which outlines valid users and their respective access rights.

In the Banking System Check Process, user inputs such as name, role Id, permission, and obj are received to assess access permissions. A timing constraint is specified to ensure that the authorization checks are executed within a defined timeframe. The Authorized User() user function is employed to determine if the user's role grants them access to the specified banking elements. This check must be performed within a duration of 1 minute; if it exceeds this timeframe, an exception is thrown, and a message is displayed stating, "login failed - user is not allowed to operate on the required object!" This structured approach to user role management and access control not only enhances the security of the banking system but also ensures compliance with regulatory standards and best practices in secure application design.

Execution of the Real Spec Banking System Specification Table 5.4 outlines various bank roles assigned to usernames within the Real Spec banking system. It provides examples of input variables utilized in the Banking System Check Process (), including name, ruled, permission, and obj. The execution begins with the Sanitize Input () function, which checks for SQL Injection Attacks (SQLIA) by calling the input Validation method of the data Validation resource accessed via its object, input. If the input passes this validation and is free of SQLIA, the next step is the execution of the Authorized User () function.

Figure 5.2 illustrates a sequence diagram of the banking system application, detailing the flow of input from the user for authentication and access to system resources. The user-provided input undergoes validation for SQLIA patterns. If the user is deemed valid, the authorization system checks the existence of the role and associated policies. If the user is not authorized, a concise error message is displayed, indicating that they are not permitted access.

Once authenticated, a session is created, and a log file is generated to save user information, including an encrypted session ID. If the role and policy are confirmed to allow access to the 6

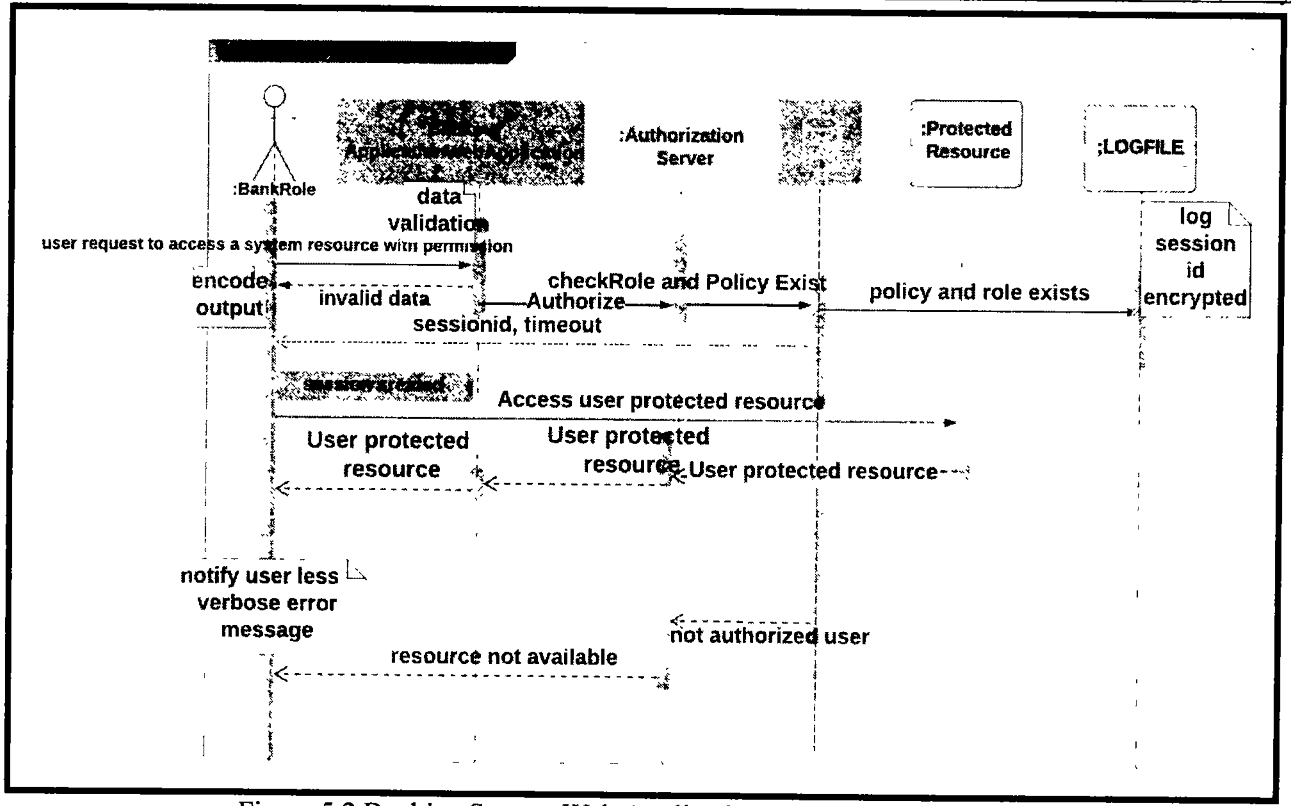


Figure 5.2 Banking System Web Application Sequence Diagram

Table 5.3 bank roles

Rules	bankRole	bankRole2
SoD	CUSTOMERSERREP	ACCOUNTMANAGER
SoD	CUSTOMERSERVREP	INTERNALAUDITOR
SoD	LOANOFFICER	ACCOUNTMANAGER
SoD	LOANOFFICER	INTERNALAUDITOR
SoD	ACCOUNTMANAGER	INTERNALAUDITOR
SoD	TELLER	ACCOUNTANT
SoD	TELLER	LOANOFFICER
SoD	ACCOUNTANT	LOANOFFICER
SoD	ACCOUNTMANAGER	INTERNALAUDITOR
DSoD	CUSTOMERSERVICEREP	LOANOFFICER

Table 5.4 sample userList data

	Username	bankRole	bankRole2
time	d0	d1	d2
t0	Hashim	CUSTOMERSERVREP	TELLER
t1	Amina	TELLER	CUSTOMERSERREP
t2	Saad	LOANOFFICER	ACCOUNTANT
t3	Manha	ACCOUNTANT	INTERNALAUDITOR
t4	Beth	ACCOUNTMANAGER	ACCOUNTANT
t5	Waqar	INTERNALAUDITOR	ACCOUNTANT
t6	Adil	BRANCHMANAGER	CUSTOMERSERREP

Table 5.5 sample input streams for policyList

	roleId	permission	Obj
time	d0	d1	d2
t0	CUSTOMERSERVREP	CREATE	DepositAccount
t1	CUSTOMERSERVREP	MODIFY	DepositAccount
t2	TELLER	DELETE	DepositAccount
t3	TELLER	MODIFY	DepositAccount

Table 5.6 Input streams in AuthorizationCheckProcess()

	Name	roleId	permission	Obj
time	d0	d1	d2	d3
t0	Hashim	CUSTOMERSERVREP	CREATE	DepositAccount
t1	Amina	TELLER	DELETE	DepositAccount
t2	Amina	TELLER	MODIFY	DepositAccount

At t1, inputs to the *isAuthorizedUser()*function are as follows: name is Amina, permission is DELETE, and obj is DepositAccount. However, this set of values will result in the specification throwing an exception and displaying "Amina is not allowed to perform DELETE on the DepositAccount". The reason is Amina is not permitted to perform a delete operation on the deposit account (see Table 5.4, Table 5.5 and Table 5.6).

5.4. Limitation of Case Study

The case study employee role-based access control. The limitation with this type of access control is that for each organization the database and access control policy have to updated according to its own rules causing maintenance overhead.

5.5. Analysis of the Findings

The case study discussed above undergoes analysis and design phases where security features, including those specified in Table 2.5 along with other functional requirements of the system, are detailed. Table 2.5 is revised by adding an additional column to verify the specification of security features and identify the attacks these specifications can mitigate as shown in Table 5.7. By

Table 5.7 Security Specification in RealSpec and attacks mitigated

Feature	Sub-Feature	Security Requirements Mapped in the case study	Attacks Mitigated
Error Handling andLogging Protection	Error Handling and Logging Protection	This feature is specified using logfile resource	Sensitive Data Exposure
	Log Information Level	This feature is specified using log file severity level	Sensitive Data Exposure
	Error Message Control	This feature is specified by verbosity control of error message using roles and privileges for error messages	SQLI,
Input Validation	Database Query Security	This feature is specified using database resource and validating user input against XSS and SQLI attack pattern.	SQLI
	User Input Security		XSS Attack,SQLI
	Input Buffer Size Check	This feature is specified by verifying buffer limit and an exception is thrown in case buffer limit has been crossed.	Buffer Overflow Attack, SQLI
	Encoded Output	This feature is specified by encoding output if there is any XSS attack pattern found in the data before it is being displayed to user	XSS
Memory Management	Memory Management Control	This is specified by specifying resource and then release it when it is out of scope	Buffer Overflow Attack, ROP Attack
	Memory Address Arithmetic	-	Buffer Overflow Attack, ROP Attack
	Array Out of Bound Check	Same as Input Buffer Size Check	Buffer Overflow Attack
	Memory Management Flexibility	-	Buffer Overflow Attack, ROP Attack
Access Control	Authorization Types	It is defined by specifying RBAC for banking application	TOCTOU Attack
	Encapsulation-Based Access Control	It is specified by defining private variables.	Sensitive Data Exposure
	Sandbox Support	-	Buffer Overflow Attack, Sensitive Data Exposure
Type System	Type Safety	It is specified by defining date type for RealSpec	Buffer Overflow Attack, ROP Attack
	Type Casting	-	Buffer Overflow Attack
	Type Initialization	It is specified by giving initial values for variables	Buffer Overflow Attack
	Immutability	It is specified using lock and unlock on logfile to preserve mutual exclusion	TOCTOU Attack
User Authentication Support		It is specified by username and password and OTP	XSS
Web Session Management Support	Secure Session ID	It is specified by defining SessID for the session established when the user is authenticated	XSS Attack, Broken Authentication
	Secure Cookies	It is partially specified by defining XSS attack pattern on the input	XSS Attack
	Session Timeout	It is specified using @timeout constraint when the session remains idle for 20 s	XSS Attack
Communication Security	SSL/TLS Backward Compatibility	It is specified by attack pattern on the input validation	Sensitive Data Exposure,
	Cryptographic Algorithms	It is specified by encryption and decryption of the log file.	Sensitive Data Exposure
	Key Length	-	Sensitive Data Exposure
	Random Number Method	-	Sensitive Data Exposure
	Certificate Validity	It is specified by integrity	Sensitive Data Exposure

incorporating these security features during the analysis and design phases, the goal is to proactively address vulnerabilities such as Sensitive Data Exposure, SQL Injection (SQLI), XSS, Buffer Overflow attacks, Return-Oriented Programming (ROP), and broken authentication early in the Software Development Life Cycle (SDLC).

5.6. Comparison with similar techniques

Table 5.8 provides a comparison of existing executable specification languages with the proposed method across various dimensions, including domain, type, and security requirements. This comparison highlights gaps identified in Table 3.6 of Chapter 3. Notably, existing specification languages have not adequately addressed critical security requirements such as secure error messaging, output validation, and immutability. The proposed method explicitly incorporates these specific security features to effectively mitigate potential threats, including Time-of-Check to Time-of-Use (TOCTOU) attacks, exposure of sensitive information, and Cross-Site Scripting (XSS) attacks. By integrating these elements, the proposed specification language aims to enhance the overall security posture of systems developed using it, thereby providing a more robust framework for secure application development.

Table 5.8 Comparison of Specification languages showing security feature coverage

Specification Languages	SysML-sec [10]	Secure Descartes [11]	S-Promela [13]	SecureSOA [156]	Ponder [12]	Proposed method
Domain	Real-time/embedded systems	Web applications	Embedded System	Business Security	Service Oriented Architecture	Real Time systems
Type	Executable Specification Language	Executable specification language	Executable Specification Language	Declarative, object-oriented language	Formal specification language	Executable specification language
Secure Error Message	No	No	No	No	No	Yes
Output validation	No	No	No	No	No	Yes
Immutability	No	No	No	No	No	Yes

5.7. Comparison of Proposed Frameworks with Existing Frameworks

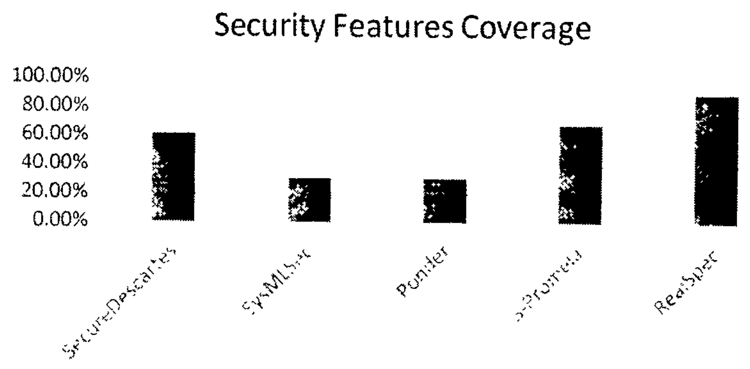
The proposed Security Framework for Real Spec (SFRS) has been compared with various existing security specification frameworks to evaluate the extent of security coverage they provide. The frameworks included in this comparison are those developed by Kasal et al. [16], Villarroel et al. [17], Khan and Zulkarnaen [18], Karamat et al. [19], Lusio et al. [20], Van den Berghe et al. [22], Nguyen et al. [21], Al-Mekhela and Khwaja [165], and Laborde et al. [161]. As illustrated in Table 5.9, the SFRS stands out by incorporating a comprehensive set of security features.

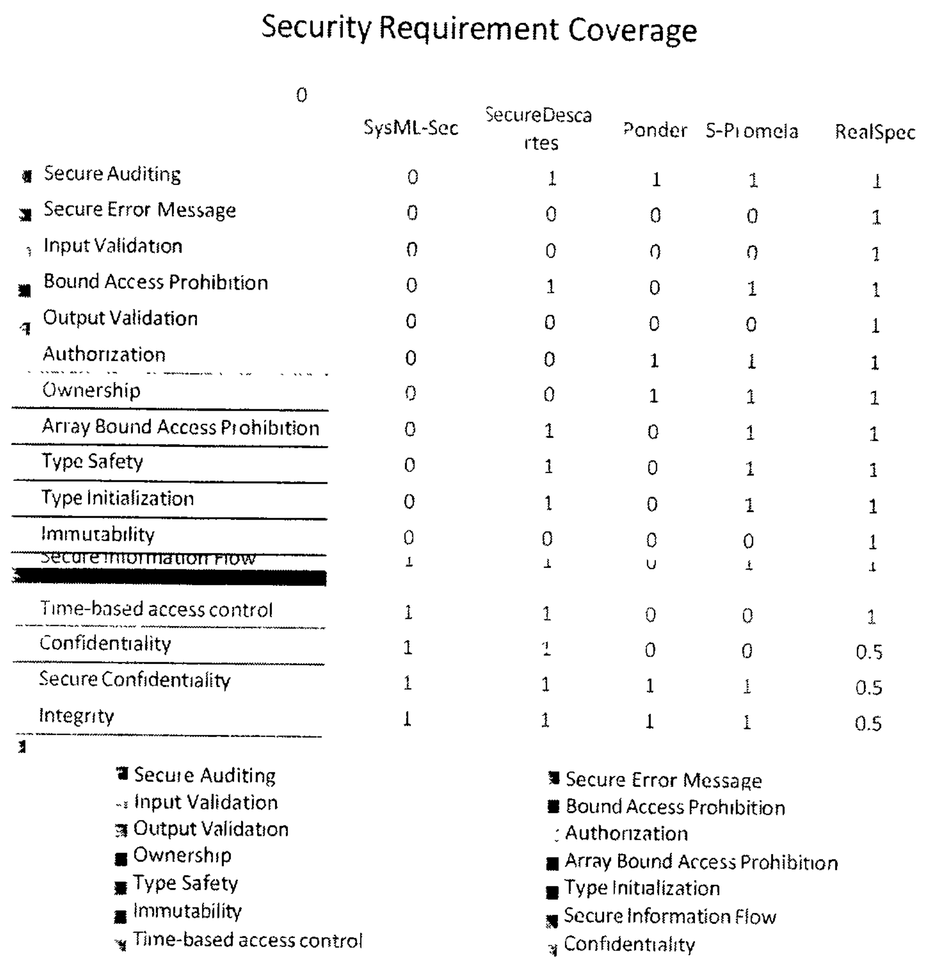
Table 5.9 Comparison of Existing Frameworks and SFRS with SEFF

	Error Handlingand Log File Control		Data Validation			Access Control		Type System			Secure Session Management		Secure Configuration		
	Secure Auditing	Secure Error Message	Input Validation	Bound Access Prohibition	Output Validation	Authorization	Ownership	Type Safety	Type Initialization	Immutability	Secure Information Flow	Time-based AccessControl	Confidentiality	Secure Configuration	Integrity
Laborde etal [162]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Yes
M. Al-Mekhlal and A. Ali Khwaja [165]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Villarroelet al. [17]	-	-	-	-	-	Yes	Yes	-	-	-	-	-	-	-	-
Kasal et al.[16]	-	-	-	-	-	Yes	Yes	-	-	-	-	-	-	-	-
Khan and Zulkernain[18]	-	-	-	-	-	Yes	Yes	-	-	-	-	-	Yes	-	-
Karapati etal. [19]	-	-	-	-	-	-	-	-	-	-	-	-	Yes	-	Yes
Lusio et al.[20]	-	-	-	-	-	-	-	-	-	-	Yes	Yes	Yes	-	-
Nguyen etal [21]	-	-	-	-	-	Yes	Yes	-	-	-	Yes	Yes	Yes	-	Yes
Van den Berghe et al. [22]	Yes	-	Yes	-	-	Yes	Yes	-	-	-	Yes	Yes	Yes	-	Yes
proposed framework	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

RealSpec [24]-[30]	S-Promela [13]	Ponder [12]	Secure Descartes [11]	SysML-Sec [10]		
Yes	Yes	Yes	Yes	No	**Secure Auditing**	**Error Handling andLog File Control**
Yes	- No	No -	- No	No -	**Secure Error Message**	
Yes	- No	No -	- No	No	**Input Validation**	**Data Validation**
Yes	Yes	No -	Yes	No	**Bound Access Prohibition**	
Yes	- No	No -	- No	No	**Output Validation**	
Yes	Yes	Yes	- No	No	**Authorization**	**Access Control**
Yes	Yes	Yes	No -	No	**Ownership**	
Yes	Yes	No	Yes	No	**Array Bound AccessProhibition**	**Mem ory Manage**
Yes	Yes	No -	Yes	No	**Type Safety**	**Type System**
Yes	Yes	No -	Yes	No	**Type Initialization**	
Yes	- No	No -	- No	No	**Immutability**	
Yes	Yes	- No	Yes	Yes	**Secure Information**	**Secure Session Managem ent**
Yes	Yes	- No	Yes	Yes	**Time-based AccessControl**	
Partially	- No	No -	Yes	Yes	**Confidentiality**	**Secure Configurati on**
Partially	Yes	Yes	Yes	Yes	**Secure Configuration**	
Partially	Yes	Yes	Yes	Yes	**Integrity**	
90.625%	68.75%	31.25%	62.5%	31.25%	**Percentage**	

Table 5.10 Comparison of Executable Specification Languages using SRFM




Security Requirement Coverage

0

	SysML-Sec	SecureDescartes	Ponder	S-Promela	RealSpec
Secure Auditing	0	1	1	1	1
Secure Error Message	0	0	0	0	1
Input Validation	0	0	0	0	1
Bound Access Prohibition	0	1	0	1	1
Output Validation	0	0	0	0	1
Authorization	0	0	1	1	1
Ownership	0	0	1	1	1
Array Bound Access Prohibition	0	1	0	1	1
Type Safety	0	1	0	1	1
Type Initialization	0	1	0	1	1
Immutability	0	0	0	0	1
Secure Information Flow	1	1	0	1	1
Time-based access control	1	1	0	0	1
Confidentiality	1	1	0	0	0.5
Secure Confidentiality	1	1	1	1	0.5
Integrity	1	1	1	1	0.5

■ Secure Auditing
■ Secure Error Message
■ Input Validation
■ Bound Access Prohibition
■ Output Validation
■ Authorization
■ Ownership
■ Array Bound Access Prohibition
■ Type Safety
■ Type Initialization
■ Immutability
■ Secure Information Flow
■ Time-based access control
■ Confidentiality

Further analysis in Table 5.10 reveals the percentage of security features covered by each framework. SysML-Sec covers 31.25% of the necessary security features, while Secure Descartes achieves 62.5% coverage, and S-Promela follows closely with 68.25%. Ponder, on the other hand, covers only 31.25%. In contrast, RealSpec excels significantly, boasting a remarkable 90.625% coverage of security features.

These results indicate that the RealSpec framework provides the highest level of security feature coverage among the compared frameworks, suggesting its effectiveness and reliability in ensuring security in system specifications. This positions RealSpec as a leading choice for practitioners and researchers seeking robust security measures in their applications.

5.8. Summary

This chapter provided a detailed case study to specify functional and non-functional requirement in RealSpec. Security features specified in chapter 4 have been applied to the case study. Security features communication security, input validation, access control, user-authentication, secure session management, log file protection was tested in the inputs given in Table 5.7. In Section 5.1, banking system website case study is stated. Section 5.2 executed some inputs on t0,t1,t2. Each role has its own permission policy defined. Section 5.3 is the limitation of the case study, Section 5.4 was the analytical discussion of the findings. Section 5.5 compared RealSpec with similar techniques and also quantified the performance in terms of security levels. Table 5.11 compared RealSpec with similar techniques and this percentage was calculated using similar formulae defined in Chapter 3 Section 3.6. these percentages indicate promising results, suggesting that RealSpec achieves the highest coverage of security features at 90.625% compared to the SRFS framework. Section 5.7 summarized the chapter.

Chapter 6

Evaluation of Security Features

The evaluation of the specifications outlined in Chapters 4 and 5 is carried out using a prototype compiler developed for Real Spec, which consists of three main components: the lexical analyzer, parser, and code generator. The lexical analyzer breaks down the input specifications into manageable tokens by scanning the source code for defined patterns, such as keywords and operators, which simplifies the parsing stage. The parser then takes these tokens and organizes them into a syntax tree, ensuring that the specifications adhere to the grammatical rules of the Real Spec language. This step is crucial for detecting any structural errors before moving on to code generation. Finally, the code generator translates the parsed structure into executable code, enabling practical testing and validation of the defined security features and functionalities within Real Spec. The entire compiler is implemented in C#, utilizing various packages, including System for fundamental classes and data types, System. Collections. Generic for type-safe collections, System.IO for input and output operations, and System. Text for string manipulation and encoding. By integrating these components and packages, the Real Spec compiler effectively evaluates the specifications, ensuring their syntactic correctness and executable functionality, which is essential for validating the proposed security measures and enhancing the overall quality of the Real Spec framework.

6.1. Compilation Process

The compilation process of the Real Spec compiler, as depicted in Figure 6.1, begins with the specification written in Real Spec, which is first passed to the lexical analyzer. The primary role of the lexical analyzer is to tokenize the input statements, breaking them down into individual components such as keywords, operators, and identifiers that can be easily processed. Once tokenization is complete, the parser takes these tokens along with the defined language grammar to ensure they conform to the syntactical rules of Real Spec. This validation step is crucial for identifying any structural errors before proceeding further.

After parsing, the valid tokens are mapped to corresponding C++ code, which acts as an intermediary representation of the Real Spec specifications. This C++ code is then compiled using a C++ compiler, allowing it to be executed within a standard programming environment. To evaluate the effectiveness of the specifications, the generated C++ code is subjected to a series of test cases designed to simulate potential attack patterns. If an attack pattern is detected during this testing phase, an exception is thrown, signaling a security issue. Conversely, if no threats are identified, the code produces a normal output, indicating successful execution. The detailed implementation of this compilation process, including the compiler's code, is provided in GitHub, further illustrating the mechanics of how Real Spec specifications are transformed into executable code while ensuring security measures are in place.

6.1.1. Lexical Analyzer

The lexical analyzer serves as the initial component in the compilation process, responsible for processing the input specifications written in Real Spec. It takes the entire input code and breaks it down into manageable pieces known as tokens. This tokenization is achieved using regular expressions, which match specific patterns within the input. Each pattern corresponds to different elements of the code, such as keywords, identifiers, operators, and symbols. When a pattern is matched, the lexical analyzer performs an associated action, which may include categorizing the token or storing it in a data structure for further processing. By converting the raw input into tokens, the lexical analyzer simplifies the code and prepares it for the next stages of compilation.

6.1.2. Parser

Following the lexical analysis, the parser takes the generated tokens and constructs an Abstract Syntax Tree (AST). The AST is a hierarchical representation of the input code that reflects its logical structure and relationships among the various components. During this stage, the parser checks the tokens against the grammar rules defined for Real Spec, as outlined in the reference provided in [29]. This grammar validation ensures that the input code adheres to the syntactical requirements of the Real Spec language. If the code violates any grammar rules, the parser will flag errors, providing essential feedback for debugging. The AST serves as a foundation for further processing in the compilation pipeline, allowing for more complex operations and analyses.

6.1.3. Code Generator

The final component of the compilation process is the code generator, which takes the validated AST and transforms it into executable code. In this phase, each process and resource defined in the Real Spec specification is converted into a corresponding class in the target programming language (C++ in this case). These classes encapsulate all predefined and user-defined types, along with methods that represent the operators and actions specified in the Real Spec. The operator net in the Real Spec is mapped to method names, complete with appropriate parameter lists. This transformation from high-level specifications to executable classes is critical for enabling the practical execution of the Real Spec code, ensuring that all defined functionalities and security measures are implemented correctly in the resulting C++ application.

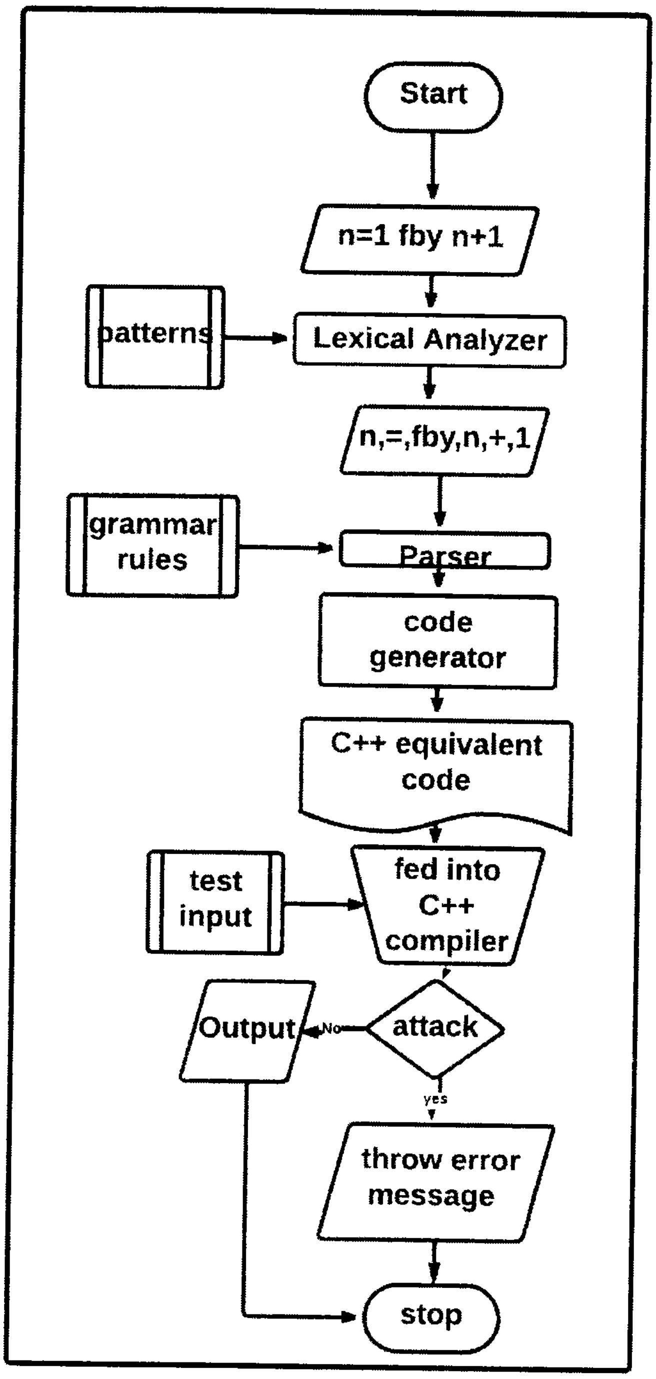


Figure 6.1 Compilation Process

6.2. Model-to-Code Transformation

Model-to-code transformation is a critical step in the Software Development Life Cycle (SDLC), particularly when transitioning from the analysis and design phases to implementation. In the context of Real Spec, the specifications, which are primarily focused on defining security features and system behaviors, need to be converted into an executable programming language for practical application. This chapter outlines the transformation process where the Real Spec specifications are systematically converted into C++ code.

The Real Spec compiler facilitates this transformation by taking the specifications outlined in Chapters 4 and 5 and generating equivalent C++ code. The generated code reflects the structural and behavioral definitions laid out in the Real Spec specifications. For instance, the generated C++ code starts with necessary header files such as <iostream>, <stream>, <string>, and <mutex>, which provide essential functionalities for input/output operations, file handling, string manipulation, and multithreading, respectively.

By utilizing these header files, the C++ equivalent code not only adheres to standard practices but also ensures that all features specified in Real Spec can be effectively implemented in C++. The transformed code is then ready for manual execution in a C++ compiler, allowing developers to test and validate the functionalities defined in the Real Spec specification. This model-to-code transformation is pivotal in bridging the gap between high-level design specifications and practical implementation, ultimately enabling the development of robust and secure software systems. The code for compiler is uploaded in the Github PhDThesisCode/Appendix at main · MunibaMurtaza-phd/PhDThesisCode (github.com).

6.3. Summary

Section 6.1 outlined the compilation process designed to validate RealSpec specifications comprehensively. This process introduced a custom compiler that operated in two key phases. Initially, the RealSpec code was processed through a lexical analyzer to tokenize inputs, followed by parsing using pattern matching techniques to generate equivalent C++ code. Subsequently, the translated C++ code underwent compilation using a standard C++ compiler and was subjected to rigorous testing against predefined attack patterns. Section 6.2, outlined the process of model to code transformation. Section 6.3, summarizes the chapter.

Chapter 7

Future Work

7.1. Contributions

This thesis set four goals and also made following contributions such as

7.1.1. Identification of Security Features which can provide

- **Authentication:** Mechanisms for verifying user identities.
- **Authorization:** Control over what authenticated users can access or modify.
- **Confidentiality:** Ensuring sensitive data is accessible only to authorized users.
- **Integrity:** Protecting data from unauthorized modifications.
- **Non-repudiation:** Ensuring that actions or transactions can be proven to have occurred, preventing denial by involved parties.

This thesis successfully identified security features which can provide above goals. The above goal not only provide security features but also laid a guidance for practitioners. The developed SEFF serves as a guide for system architects and developers, providing them with a structured approach to incorporating security into their designs.

7.1.2. Designing a Security Requirement Framework for Secure Web Application Development (SRFS)

A framework has been developed for verifying and validating that security requirements have been met. Implementing automated security testing tools that integrate with Code Integration/Code Development pipelines to ensure ongoing compliance with security requirements. SRFS included a repository of best practices and guidelines for secure coding, configuration management, and deployment strategies, tailored to web application development. It provided the developers with established security design patterns (e.g., the use of secure session management, the principle of least privilege) to facilitate secure application design.

7.1.3. Specification of Security features in early phases of software development using RealSpec Executable Specification Language.

Specification of security features in the early phases of software development using Real Spec is a significant advancement in securing software applications. By formalizing and integrating security requirements into the development lifecycle, this approach not only enhances security but also streamlines the development process across various programming languages. This proactive methodology leaded to more robust and secure software systems capable of addressing the complex security challenges of modern applications.

7.1.4. Development of an evaluation tool that takes the RealSpec program.

An evaluation tool solely designed for RealSpec programs gives a significant advancement in the field of software engineering, particularly for real-time systems. RealSpec, as an executable specification language, aids the formalization of system behavior and security features. The goal of the evaluation tool is to analyze and assess these RealSpec programs. It also ensured that they meet defined security and performance criteria. By integrating this tool into the software development lifecycle, developers can proactively identify and address potential vulnerabilities and inconsistencies in their specifications before the implementation phase. This thesis answered all of the above goals as shown in Figure 7.1.

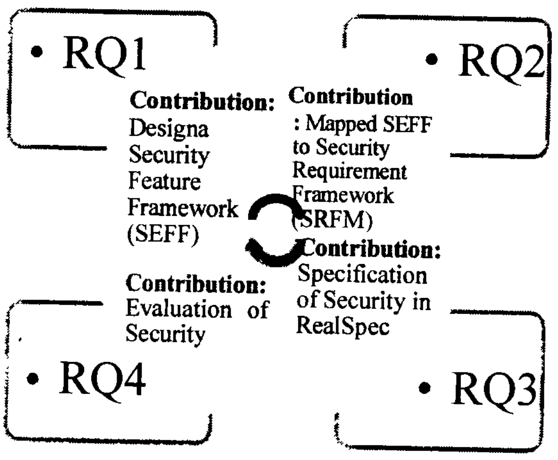

• RQ1
Contribution: Designa Security Feature Framework (SEFF)
• RQ2
Contribution : Mapped SEFF to Security Requirement Framework (SRFM)
• RQ3
Contribution: Specification of Security in RealSpec
• RQ4
Contribution: Evaluation of Security

Figure 7.1 Contributions

7.2. Future Work

The design and evaluation of security features in Real Spec, a real-time executable specification language, has laid a strong foundation for ensuring that systems built using Real Spec maintain a high level of security. However, there are several avenues for future work to extend and improve the security mechanisms within the language. One important area is the enhancement of Real Spec's type system to include support for fine-grained access control and security policies. This would allow developers to define security rules at the type of level, enabling Real

Spec to enforce confidentiality and integrity constraints on data flows and system interactions. In addition, future work should explore the integration of formal verification tools that can automatically check Real Spec programs for security vulnerabilities such as buffer overflows, injection attacks, and race conditions. This would help in verifying security properties of real-time systems more rigorously.

Another promising direction involves the incorporation of cryptographic protocols into Real Spec to protect communication channels between distributed components in real-time systems. Future iterations could integrate a cryptographic framework to ensure secure communication with strong encryption, authentication, and integrity guarantees. Additionally, runtime monitoring tools could be developed for Real Spec that dynamically checks for security violations, allowing systems to detect and respond to malicious activities in real time. Secure code generation is another potential area of development. As Real Spec is an executable specification language, future work should focus on generating secure, optimized, and verified code that not only meets performance constraints but also adheres to strict security requirements.

Furthermore, future research could examine how RealSpec can be adapted to emerging paradigms like quantum computing and blockchain-based systems. These technologies introduce new security challenges and ensuring that Real Spec can handle the cryptographic requirements and potential vulnerabilities in these contexts would make it more versatile and future-proof. Finally, Real Spec's security features could benefit from community-driven extensions, where developers can contribute new security modules and patches, creating a more adaptive and collaborative environment for maintaining the security of the language. In sum, the future work for Real Spec should focus on enhancing its security model through formal verification, cryptography, runtime monitoring, and secure code generation, while preparing it for integration with future technologies. This work focused on only two types of SQLI attacks such as tautology and error-based and only DOM based XSS attack. Future work is to focus on other types of SQLI, XSS attacks and other OWASP attacks. Review their mitigation techniques and identify security features that prevent these attacks and update the SEFF and SFRS and then specify those features in RealSpec.

The proposed framework SEFF can be improved by a literature review on new programming languages security features and security features to mitigate attacks other than OWASP attacks can be added to the current framework. Next, the enhanced SEFF is abstracted to generate an enhanced version of SFRS. SEFF can be used to evaluate other programming languages. In future, the security requirements from updated SRFS can be specified in RealSpec or other executable specification languages. RealSpec, as an executable specification language rooted in the Lucid dataflow programming language, ensures correctness through operational models. To Introduce a mathematical proof system for RealSpec would enhance user confidence in the validity of specifications, enabling the application of theorem provers and proof checkers to verify consistency and completeness. This formal proof system would bolster the reliability and rigor of RealSpec specifications, aligning them more closely with formal verification standards. Enhancement of compiler with updated features in RealSpec can be done.

REFERENCES

1. A. Ruef and C. Rohlf, "Programming Language Theoretic Security in the Real World: AMir or the Future?" *Advances in Information Security*, vol. 2, no. 3, pp. 307-321, 2015, d 10.1007/978-3-319-14039-1_15.
2. M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "T Devil is in the Constants: Bypassing Defenses in Browser JIT Engines." *Proceedings 20 Network and Distributed System Security Symposium*, vol. 3, no. 2, pp. 211-225, 2015, d 10.14722/ndss.2015.23209.
3. M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The m dangerous code in the world." *Proceedings of the 2012 ACM conference on Computer a communications security*, vol. 2, no. 1, pp. 38-49, 2012, doi: 10.1145/2382196.2382204.
4. S. Turner, "Security vulnerabilities of the top ten programming languages: C, Java, C objective-C, C#, PHP, visual basic, Python, Perl, and ruby." Journal of Technology Resear vol. 5, pp.1-17, 2014, doi: 10.3403/30383974.
5. P. Hayati, N. Jafari, S. M. Rezaei, S. Sarenche, and V. Potdar, "Modeling Input Validatio UML." *19th Australian Conference on Software Engineering (aswec 2008)*, vol. 4, no.2, 663-672, 2008, doi: 10.1109/aswec.2008.4483260.

6 M. Busch, "Integration of Security Aspects in Web Engineering," master's thesis, Inst.fur Informatik, Ludwig-Maximilians-Univ., 2011

7. J. Jürjens, "UMLsec: Extending UML for Secure Systems Development." *«UML» 2002 — The Unified Modeling Language*, vol. 3, no. 5, pp. 412-425, 2002, doi: 10.1007/3-540-45800-x_32.
8. T. Lodderstedt, D. Basin, and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security." *«UML» 2002 — The Unified Modeling Language*, pp. 426-441, 2002, doi: 10.1007/3-540-45800-x_33.
9. A. A. Khwaja and J. E. Urban, "A Synthesis of Evaluation Criteria for Software Specifications and Specification Techniques." *International Journal of Software Engineering and Knowledge Engineering*, vol. 12, no. 5, pp. 581-599, 2002, doi: 10.1142/s0218194002001062.
10. Y. Roudier, L. Apvrille, "SysML-Sec - A Model Driven Approach for Designing Safe and Secure Systems." *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, pp. 655-664, 2015, doi: 10.5220/0005402006550664.
11. V. N. Inukollu and J. E. Urban, "Secure Descartes: A Security Extension to Descartes Specification Language." *International Journal of Software Engineering & Applications*, vol. 11, no. 5, pp. 1-11, 2020, doi: 10.5121/ijsea.2020.11501.

12. N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language." *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pp. 18-38, 2001, doi: 10.1007/3-540- 44569-2_2.

13. R. Abbassi and S. G. El Fatmi, "S-Promela: An executable specification security policies language." *2009 First International Conference on Communications and Networking*, pp.1-8, Nov. 2009, doi: 10.1109/comnet.2009.5373568.

14. H. Hu and G. Ahn, "Constructing Authorization Systems Using Assurance Management Framework." *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 4, pp. 396-405, 2010, doi: 10.1109/tsmcc.2010.2047856.

15. J. Kong, D. Xu, and X. Zeng, "UML-Based Modeling and Analysis Of Security Threats." *International Journal of Software Engineering and Knowledge Engineering*, vol. 20, no. 6, pp. 875-897, 2010, doi: 10.1142/s0218194010004980.

16. K. Kasal, J. Heurix, and T. Neubauer, "Model-Driven Development Meets Security: An Evaluation of Current Approaches." *2011 44th Hawaii International Conference on System Sciences*, pp. 1-9, Jan. 2011, doi: 10.1109/hicss.2011.310.

17. R. Villarroel, E. Fernández-Medina, and M. Piattini, "Secure information systems development – a survey and comparison." *Computers & Security*, vol. 24, no. 4, pp. 308-321, 2005, doi: 10.1016/j.cose.2004.09.011.

18. M. U. A. Khan and M. Zulkernine, "A Survey on Requirements and Design Methods for Secure Software Development," School of Computing, Queen's University, Canada, pp. 1–22, 2009, doi: 10.1109/ssd.2009.11.

19. P. Karpati, G. Sindre, and A. L. Opdahl, "Characterizing and Analyzing Security Requirements Modelling Initiatives." *2011 Sixth International Conference on Availability, Reliability and Security*, pp. 710-715, 2011, doi: 10.1109/ares.2011.113.

20. L. Lúcio, "Advances in Model-Driven Security." *Advances in Computers*, vol. 93, pp. 103-152, 2014, doi: 10.1016/b978-0-12-800162-2.00003-8.

21. P. H. Nguyen, M. Kramer, J. Klein, and Y. L. Traon, "An extensive systematic review on the Model-Driven Development of secure systems." *Information and SoftwareTechnology*, vol. 68, pp. 62-81, 2015, doi: 10.1016/j.infsof.2015.08.006.

22. A. Van den Berghe, R. Scandariato, K. Yskout, and W. Joosen, "Design notations for secure software: a systematic literature review." *Software & Systems Modeling*, vol. 16, no. 3, pp. 809-831, 2015, doi: 10.1007/s10270-015-0486-9.

23. E. Deveci and M. U. Caglayan, "Model driven security framework for software design and verification." *Security and Communication Networks*, vol. 8, no. 16, pp. 2768-2792, 2015, doi: 10.1002/sec.1200.

24. A. A. Khwaja and J. E. Urban, "RealSpec: An Executable Specification Language for Modeling Control Systems." *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, vol. 4, no. 1, pp. 219-227, 2009, doi: 10.1109/isorc.2009.36.

25. A. A. Khwaja and J. E. Urban, "RealSpec: An Executable Specification Language for Prototyping Concurrent Systems." *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pp. 3-9, 2008, doi: 10.1109/rsp.2008.9.

26. A. A. Khwaja and J. E. Urban, "RealSpec: An Executable Specification Language for Prototyping Concurrent Systems." *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pp. 97-102, 2008, doi: 10.1109/rsp.2008.9.

27. A. A. Khwaja and J. E. Urban, "Timing Precedence and Resource Constraints in the RealSpec Real-Time Specification Language". *Proceedings of the 2008 IASTED International Conference on Software Engineering and Applications*, November 16–18, 2008.

28. A. A. Khwaja and J. E. Urban, "Preciseness for predictability with the RealSpec real-time executable specification language." *2010 IEEE Aerospace Conference*, pp. 1-9, 2010, doi: 10.1109/aero.2010.5446788.

29. A.A. Khwaja, Doctoral thesis, *"RealSpec: An executable real-time specification language."* Arizona StateUniversity, 2009.

30. A. A. Khwaja, "Modeling Big Data Analytics with a Real-Time Executable Specification Language." *Handbook of Research on Trends and Future Directions in Big Data and Web Intelligence*, pp. 289-312, 2015, doi: 10.4018/978-1-4666-8505-5.ch014.

31. H. Gu, "DIAVA: A Traffic-Based Framework for Detection of SQL Injection Attacks and Vulnerability Analysis of Leaked Data." *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 188-202, 2020, doi: 10.1109/tr.2019.2925415.

32. P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID." *ACM Transactions on Information and System Security*, vol. 13, no. 2, pp. 1-39, 2010, doi: 10.1145/1698750.1698754.

33. B. A. Pham and V. H. Subburaj, "An experimental setup for detecting sqli attacks using machine learning algorithms", *Journal of The Colloquium for Information Systems Security Education*, vol. 8, 2020.

34. N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. "SessionShield: Lightweight Protection against Session Hijacking." *Engineering Secure Software and Systems Third International Symposium*, pp. 87-100, 2011, doi: 10.1007/978-3-642-19125-1_7.

35. S. Goswami, N. Hoque, D. K. Bhattacharyya and J. Kalita, "An unsupervised method for detection of XSS attack". *International Journal of Network Security*, vol. 19, no. 5, pp. 761-775, 2017.

36. S. Shalini and S. Usha, "Prevention of cross-site scripting attacks (xss) on web applications in the client side," IJCSI International Journal of Computer Science Issues, vol. 8, no. 4, 2011.

37. S. Rathore, P. K. Sharma, and J. H. Park, "XSSClassifier: An Efficient XSS Attack Detection Approach Based on Machine Learning Classifier on SNSs." *Journal of Information Processing Systems*, vol. 13, no. 4, 2017, doi: 10.3745/jips.03.0079.

38. P. Wang, K. Lu, G. Li, and X. Zhou, "DFTracker: detecting double-fetch bugs by multi-taint parallel tracking." *Frontiers of Computer Science*, vol. 13, no. 2, pp. 247-263, 2019,

doi: 10.1007/s11704-016-6383-8.

39. S. Gujrathi, "Heartbleed bug: An openssl heartbeat vulnerability," *International Journal Computer Sciences and Engineering*, vol. 2, no. 5, pp. 61–64, 2014.

40. L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation." *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1-6, Jun. 2014, doi: 10.1109/dac.2014.6881460.

41. X., Lai, A. Balakrishnan, T. Lange, M. Jenihhin, T. Ghasempouri, J. Raik, and D. Alexandrescu. "Understanding multidimensional verification: Where functional meets non-functional." *Microprocessors and microsystems*, 71, p.102867, 2019.

42. M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines." *Proceedings 2015 Network and Distributed System Security Symposium*, pp. 3-17, Feb. 2015, doi: 10.14722/ndss.2015.23209.

43. C. B. Hamilton, "Security in Programming Languages," 15 December 2015. [Online]. Available: http://www.cs.tufts.edu/comp/116/archive/fall2015/chamilton.pdf . [Accessed July, 16, 2023]

44. S. Sondarva, D. P. Sharma, and P. D. Dholariya, "Prevention to Sensitive Information Disclosure via OSINT." *International Journal of Scientific Research in Science, Engineering and Technology*, pp. 109-114, 2021, doi: 10.32628/ijsrset218317.

45. J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In HotPar, pages 15–15, 2012.

46. D. Huluka and O. Popov, "Root cause analysis of session management and broken authentication vulnerabilities." *World Congress on Internet Security (WorldCIS-2012)*, pp. 82-86, Jun. 2012.

47. D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser." *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142-154, 2011, doi: 10.1145/1945023.1945039.

48. D. Ray and J. Ligatti, "Defining Injection Attacks.", *ACM-SIGACT Symposium on Principles of Programming Languages*, 2014, doi: 10.1007/978-3-319-13257-0_26.

49. S. Al-Qahtani, P. Pietrzynski, L. Guzman, R. Arif, and A. Tevoedjre, "Comparing selected criteria of programming languages java, php, c++, perl, haskell, aspectj, ruby, cobol, bash scripts and scheme revision 1.0-a team cplgroup comp6411-s10 term report. ." *arXiv preprint arXiv:1008.3434.*, 2010.

50. A. A. Khwaja, M. Murtaza, and H. F. Ahmed, "A security feature framework for programming languages to minimize application layer vulnerabilities." *Security and Privacy*, vol. 3, no. 1, pp. 95-125, 2019, doi: 10.1002/spy2.95.

51. E. A. Barbosa, A. Garcia, and S. D. J. Barbosa, "Categorizing Faults in Exception Handling: A Study of Open-Source Projects." *2014 Brazilian Symposium on Software Engineering*, vol. 5, no. 6, pp. 11-20, 2014, doi: 10.1109/sbes.2014.19.

52. D. Stuttard and M. Pinto, *The Web Application Hacker*. John Wiley & Sons, 2011.

53. "OWASP Secure Coding Practices - Quick Reference Guide." https:// owasp.org/ www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf (accessed: Aug. 05, 2023).

54. "Addison-Wesley Professional Ruby Series." https:// www.amazon.com/ Addison-Wesley-Professional-Ruby-Series-20-book-series/ dp/ B08BTXWBHN (accessed: Aug. 05, 2023).

55. "Python Language Tutorial => Introduction to Python Logging." https:// riptutorial.com/ python/ example/ 14214/ introduction-to-python-logging (accessed: Aug. 05, 2023).

56. B. Chen, J. Song, P. Xu, X. Hu, and Z. M. (Jack) Jiang, "An automated approach to estimating code coverage measures via execution logs." *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 305-316, 2018, doi: 10.1145/3238147.3238214.

57. H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement? (Journal-first abstract)." *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 468-490, Mar. 2018, doi: 10.1109/saner.2018.8330234.

58. "Logging in Python – Real Python." https:// realpython.com/ python-logging/ (accessed: Aug. 05, 2023).

59. "Django Tutorial Part 8: User authentication and permissions - Learn .." https:// developer.mozilla.org/ en-US/ docs/ Learn/Server-side/ Django/ Authentication (accessed: Aug. 05, 2023).

60. M. Press and M. Howard, *Writing Secure Code*. Irwin/McGraw-Hill, 2002.

61. "System.ComponentModel.DataAnnotations Namespace | Microsoft .." https:// learn.microsoft.com/ en-us/ dotnet/ api/ system. componentmodel. dataannotations? view=net-7.0 (accessed: Aug. 05, 2023).

62. W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL injection attacks and countermeasures." *Proceeding IEEE International Symposium Secure Software Engineering*, vol. 1, pp. 13-15, 2006.

63. "Ruby on Rails - OWASP Cheat Sheet Series." https:// cheatsheetseries.owasp.org/ cheatsheets/ Ruby_on_Rails_Cheat_Sheet.html (accessed: Aug. 05, 2023).

64. H. Schildt, *Java: The Complete Reference, Twelfth Edition*. McGraw Hill Professional, 2021.

65. R. Naim, M. Nizam, S. Hanamasagar, J. Noureddine, and M. Miladinova. "Comparative Studies of 10 Programming Languages within 10 Diverse Criteria-a Team 10 COMP6411-S10 Term Report." 2010. arXiv preprint arXiv:1008.3561.

66. "Securing Rails Applications — Ruby on Rails Guides." https:// guides.rubyonrails.org/ security.html (accessed: Aug. 05, 2023).

67. X. Leroy, "Computer Security from a Programming Language and Static Analysis Perspective." *Programming Languages and Systems*, vol. 2618, pp. 1-9, 2003, doi: 10.1007/3-540-36575-3_1.

68. H. Collingbourne and C. Takemura, *The Book of Ruby*. No Starch Press, 2011.

69. R. L. Halterman. "Learning to Program with Python." https:// prognoztech.com/

resources/ content/Learn-to-Program-with-Python.pdf (accessed: Aug. 05, 2023).

70. D. Chisnall, "CHERI JNI," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 569-583, 2017, doi: 10.1145/3093337.3037725.

71. "Secure Coding Guidelines for Java SE." https:// www.oracle.com/ java/ technologies/ javase/ seccodeguide.html (accessed: Aug. 05, 2023).

72. "Server.HTMLEncode Method | Microsoft Learn." https://learn.microsoft.com/en-us/previous-versions/iis/6.0-sdk/ms525347(v=vs.90) (accessed: Aug. 05, 2023).

73. "Python Tutorial." https://bugs.python.org/file47781/Tutorial_EDIT.pdf (accessed: Aug. 05, 2023).

74. J. Albahar and J. Albahari. "C# 4.0 in a Nutshell: The Definitive Reference," O'Reilly Media. Inc., 2010.

75. "PHP vs. Python vs. Ruby {The web scripting language shootout." https:// www.semanticscholar.org/ paper/ PHP-vs.- Python- vs.- Ruby- %7B- The- web-scripting-Purer/1a6de8aae72d187da194c39dbf584025d1922849 (accessed: Aug. 05, 2023).

76. S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, doi: 10.1109/isca.2012.6237017.

77. K. Watson, J. Hammer, D. Reid, M. Skinner, D. Kemper, and C. Nagel, *Beginning visual C# 2012 programming*. John Wiley & Sons, 2012.

78. R. Lafore, *Object-Oriented Programming in C++*. Pearson Education, 1997.

79. W. Stallings and L. Brown, *Computer Security: Principles and Practice PDF ebook, Global Edition*. Pearson Higher Ed, 2015.

80. A. Chaudhuri and J. S. Foster, "Symbolic security analysis of ruby-on-rails web applications," *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 585-595, 2010, doi: 10.1145/1866307.1866373.

81. "chmod | Microsoft Learn." https:// learn.microsoft.com/ en-us/ cpp/ c-runtime-library/reference/chmod?view=msvc-170 (accessed: Aug. 06, 2023).

82. "File.GetAccessControl Method (System.IO) | Microsoft Learn." https:// learn.microsoft.com/ en-us/ dotnet/ api/ system.io.file.getaccesscontrol? view=netframework-4.8.1 (accessed: Aug. 06, 2023).

83. B. Cannon and B. Wohlstadter, "Controlling access to resources within the python interpreter," *Proceedings of the Second EECE*, vol. 512, pp. 1-8, 2010.

84. D. Kuhlman, *A python book: Beginning python, advanced python, and python exercises*. 2009.

85. "Fundamental Practices for Secure Software Development." https://safecode.org/publication/SAFECode_Dev_Practices0211.pdf (accessed: Aug. 06, 2023).

86. "Detecting Malware and Sandbox Evasion Techniques | SANS Institute."

https://www.sans.org/white-papers/36667/ (accessed: Aug. 06, 2023).

87. "Laurence Tratt: Dynamically Typed Languages." https:// tratt.net/ laurie/ research/ pubs/ html/ tratt dynamically_ typed_ languages/ (accessed: Aug. 06, 2023).

88. J. Elford. "Generic Immutability and Nullity Types for an imperative object-oriented programming language with flexible initialization." 2012. http:// www.doc.ic.ac.uk/teaching/distinguished-projects/2012/j.elford.pdf.

89. "Exploring language support for immutability | Proceedings of the." https:// dl.acm.org/ doi/ 10.1145/ 2884781.2884798 (accessed: Aug. 06, 2023).

90. Y. Motara. "Functional programming and security." arXiv preprint (arXiv:1201.5728). 2012. vailable at https://arxiv.org/pdf/1201.5728.pdf.

91. A. N. Kataria, D. M. Adhyaru, A. K. Sharma, and T. H. Zaveri, "A survey of automated biometric authentication techniques." *2013 Nirma University International Conference on Engineering (NUiCONE)*, pp. 1-6, Nov. 2013, doi: 10.1109/nuicone.2013.6780190.

92. B. Rubin, "Crypto Basics." *Information Security*, pp. 17-49, 2011, doi: 10.1002/9781118027974.ch2. [Online]. Available: https:// www.ibm.com /developerworks /java /.

93. "Java Authentication and Authorization Service (JAAS) Reference." https:// docs.oracle.com/ javase/ 8/ docs/technotes/ guides/ security/ jaas/ JAASRefGuide.html (accessed: Aug. 06, 2023).

94. "AuthenticationService Class (System.Web.ApplicationServices .." https:// learn.microsoft.com/ en-us/ dotnet/ api/ system. web. applicationservices. authenticationservice?view=netframework-4.8.1 (accessed: Aug. 06, 2023).

95. R. V. Chandra and V. B. S., *Python requests essentials*. . Birmingham, UK: Packet Publishing, 2015.

96. A. Petrov, C. Schumann, and S. Gysin, "User Authentication for Role-Based Access Control." *In Proceedings of ICALEPCS*, Oct. 2007.

97. S. Calzavara, G. Tolomei, M. Bugliesi, and S. Orlando, "Quite a mess in my cookie jar!" *Proceedings of the 23rd international conference on World wide web*, pp. 189-200, Apr. 2014, doi: 10.1145/2566486.2568047.

98. L. M. and K. V., "Teaching Algorithmization and Programming using Python Language." *Information Technologies in Education*, vol. 20, pp. 013-023, 2014, doi: 10.14308/ite000493.

99. "C++ Web Programming | Tutorialspoint." https:// www.tutorialspoint.com/ cplusplus/cpp_ web_ programming.htm (accessed: Aug. 05, 2023).

100. Y. Zhou, and D. Evans, "*Why aren't HTTP-only cookies more widely deployed.*" Proceedings of 4th Web, 2, 2010.

101. "Class Poco::Net::HTTPCookie." https:// docs.pocoproject.org/ current/ Poco .Net .HTTPCookie .html (accessed: Aug. 05, 2023).

102. "HttpCookie.HttpOnly Property (System.Web) | Microsoft Learn." https:// learn.microsoft.com/ en-us/ dotnet/ api/ system. web. httpcookie. httponly? view= netframework -4.8.1 (accessed: Aug. 05, 2023).

103. W. D.,Kou, L. Mirlas, & Y.C. Zhao. *U.S. Patent No. 7,216,236*. Washington, DC: U.S. Patent and Trademark Office, 2007.

104. "Ruby | Exception handling - GeeksforGeeks." https:// www.geeksforgeeks.org/ ruby-exception-handling/ (accessed: Aug. 05, 2023).

105. "SslProtocols Enum (System.Security.Authentication) | Microsoft Learn." https:// learn.microsoft.com/ en-us/ dotnet/ api/ system. security. authentication. sslprotocols? view=net-7.0 (accessed: Aug. 05, 2023).

106. "JDK 8 will use TLS 1.2 as default." https://blogs.oracle.com/java/post/jdk-8-will-use-tls-12-as-default (accessed: Aug. 05, 2023).

107. "RijndaelManaged Class (System.Security.Cryptography) | Microsoft .." https://learn.microsoft.com/en-us/ dotnet/ api/ system. security. cryptography. rijndaelmanaged? view=net-7.0 (accessed: Aug. 05, 2023).

108. Y. Acar. "Comparing the Usability of Cryptographic APIs." *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, doi: 10.1109/sp.2017.52.

109. A. Sweigart. "Hacking Secret Ciphers with Python." https://inventwithpython.com/hackingciphers.pdf (accessed: Aug. 05, 2023).

110. B. A. Curtis, *U.S. Patent No. 7,475,260*. Washington, DC: U.S. Patent and Trademark Office., 2009.

111. S. Sundareswaran, A. Squicciarini, and D. Lin, "Ensuring Distributed Accountability for Data Sharing in the Cloud." *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 556-568, 2012, doi: 10.1109/tdsc.2012.26.

112. I. Ray, K. Belyaev, M. Strizhov, D. Mulamba, and M. Rajaram, "Secure Logging as a Service—Delegating Log Management to the Cloud." *IEEE Systems Journal*, vol. 7, no. 2, pp. 323-334, 2013, doi: 10.1109/jsyst.2012.2221958.

113. "Locking Down Log Files: Enhancing Network Security by Protecting Log Files." *Issues In Information Systems*, vol. 7, no. 2, pp. 43-47, 2006, doi: 10.48009/2_iis_2006_43-47.

114 B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics." *ACM Transactions on Information and System Security*, vol. 2, no. 2, pp. 159-176, 1999, doi: 10.1145/317087.317089.

115. J. Siefers, G. Tan, and G. Morrisett, "Robusta." *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 201-211, 2010, doi: 10.1145/1866307.1866331.

116. M. Sun and G. Tan, "NativeGuard." *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pp. 165-176, 2014, doi: 10.1145/2627393.2627396.

117. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android." *Information Security Conference*, pp. 346-360, 2011, doi: 10.1007/978- 3-642-18178-8_30.

118. O. Ruwase and M. S. Lam, "A Practical Dynamic Buffer Overflow Detector." *Network and Distributed System Security Symposium*, vol.2004, pp. 15-169, Feb. 2004.

119. M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti, "Control-flow integrity principles,

implementations, and applications." *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 1-40, 2009, doi: 10.1145/1609956.1609960.

120. Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst, "Ownership and immutability in generic Java." *ACM SIGPLAN Notices*, vol. 45, no. 10, pp. 598-617, 2010, doi: 10.1145/1932682.1869509.

121. C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming." *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA* , pp. 211-230, 2002, doi: 10.1145/582419.582440.

122. N. Krishnaswami and J. Aldrich, "Permission-based ownership." *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 96-106, 2005, doi: 10.1145/1064978.1065023.

123. A. Potanin, J. Noble, D. Clarke, and R. Biddle, "Generic ownership for generic Java." *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 311-324, 2006, doi: 10.1145/1167515.1167500.

124. G. Kniesel and D. Theisen, "JAC?Access right based encapsulation for Java." *Software: Practice and Experience*, vol. 31, no. 6, pp. 555-576, 2001, doi: 10.1002/spe.372.

125. S. Weber, M. Coblenz, B. Myers, J. Aldrich, and J. Sunshine, "Empirical Studies on the Security and Usability Impact of Immutability." *2017 IEEE Cybersecurity Development (SecDev)*, vol. 3, no. 1, pp. 50-63, 2017, doi: 10.1109/secdev.2017.21.

126. I. Pechtchanski and V. Sarkar, "Immutability specification and its applications." *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, vol. 6, no. 2, pp. 202-211, 2002, doi: 10.1145/583810.583833.

127. A. Skyrme, N. Rodriguez, and R. Ierusalimschy, "A survey of support for structured communication in concurrency control models." *Journal of Parallel and Distributed Computing*, vol. 74, no. 4, pp. 2266-2285, 2014, doi: 10.1016/j.jpdc.2013.11.005.

128. C. Haack, E. Poll, and A. Schubert, "Immutable Objects in Java | Baeldung." https://www.baeldung.com/java-immutable-object (accessed: Aug. 05, 2023).

129. C. Haack, E. Poll, J. Schäfer, and A. Schubert, "Immutable Objects for a Java-Like Language." *Programming Languages and Systems*, vol. 2, no. 4, pp. 347-362, 2007, doi: 10.1007/978-3-540-71316-6_24.

130. A. Mettler, D. Wagner, and T. Close, "Joe-E - Wikipedia." https://en.wikipedia.org/wiki/Joe-E (accessed: Aug. 05, 2023).

131. S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops." *Proceedings of the 38th International Conference on Software Engineering*, vol. 3, no. 1, pp. 935-946, 2016, doi: 10.1145/2884781.2884790.

132. C. Reis, S. Gribble, T. Kohno, and N. Weaver, "Detecting In-Flight Page Changes with Web Tripwires." vol. 8. [Online]. Available: https:// www.usenix.org / legacy /events /nsdi08 /tech/ full_papers /reis /reis.pdf.

133. L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson, "Analyzing Forged SSL Certificates in the Wild." *2014 IEEE Symposium on Security and Privacy*, 2014, doi: 10.1109/sp.2014.13.

134. S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an amplified world." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS,* vol. 8, no. 5, pp. 49-60, 2013, doi: 10.1145/2508859.2516655.

135. "Errors and Debugging | Python Data Science Handbook." https://jakevdp.github.io/PythonDataScienceHandbook/01.06-errors-and-debugging.html (accessed: Aug. 05, 2023).

136. "GDC 2002: Game Scripting in Python." https://www.gamedeveloper.com/programming/gdc-2002-game-scripting-in-python (accessed: Aug. 05, 2023).

137. "C++ standard exceptions." https://cplusplus.com/doc/tutorial/exceptions/ (accessed: Aug. 05, 2023).

138. "UTF8-CPP: UTF-8 with C++ in a Portable Way." https://utfcpp.sourceforge.net/ (accessed: Aug. 05, 2023).

139. "Ownership and Immutability in Generic Java." https://palez.github.io/papers/ownership-immutability-oopsla2010.pdf (accessed: Aug. 05, 2023).

140. "POCO C++ Libraries Release Notes." https:// docs. pocoproject. org/current/99100-ReleaseNotes. html (accessed: Aug. 05, 2023).

141. M. Welschenbach. *Cryptography in C and C++*. Apress, 2001.

142. "Ruby/ DBI Database Access Tutorials point." https:// www.tutorialspoint.com /ruby / ruby_ database_ access. htm(accessed: Aug. 05, 2023).

143. G. Noack and M. S. Y. Welsch. "TIFI+: A Type Checker for Object Immutability with Flexible Initialization." 2010. https://www.cs.ru.nl/E.Poll/papers/javimu09.pdf.

144. A. D. Brucker, J. Doser, and B. Wolff, "A Model Transformation Semantics and Analysis Methodology for SecureUML." *Model Driven Engineering Languages and Systems*, pp. 306-320, 2006, doi: 10.1007/11880240_22.

145. S. Naqvi, A. E. Arenas, and P. Massonet, "Deriving Policies from Grid Security Requirements Model." *Achievements in European Research on Grid Systems*, pp. 151-163, doi: 10.1007/978-0-387-72812-4_12.

146. J. Jürjens, "Modelling Audit Security for Smart-Card Payment Schemes with UML- Sec." *IFIP Advances in Information and Communication Technology*, vol. 3, no. 4, pp. 93-107, 2001, doi: 10.1007/0-306-46998-7_7.

147. B. Hoisl and M. Strembeck, "A UML Extension for the Model-Driven Specification of Audit Rules." *Proceedings of the 2nd International Workshop on Information Systems Security Engineering (WISSE)*, vol. 2, no. 3, pp. 16-30,2012, doi: 10.1007/978-3-642-31069-0_2.

148. M. Memon, M. Hafner, and R. Breu, "SECTISSIMO: A Platform-Independent Framework for Security Services." *In MODSEC@ MoDELS.*, Sep. 2008.

149. E. Fernández-Medina, J. Trujillo, R. Villarroel, and M. Piattini, "Access control and audit model for the multidimensional modeling of data warehouses." *Decision Support Systems*.

vol. 42, no. 3, pp. 1270-1289, 2006, doi: 10.1016/j.dss.2005.10.008.

150. C. Hochreiner, Z. Ma, P. Kieseberg, S. Schrittwieser, and E. Weippl, "Using Model Driven Security Approaches in Web Application Development." Information and Communication Technology, vol. 5, no. 3, pp. 419-431, 2014, doi: 10.1007/978-3-642-55032-4_42.

151. M. Johnson, "Kaos Semantic Policy and Domain Services." Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 2, no. 3, pp. 119-138, 2004, doi: 10.1007/0-387-23344-x_6.

152. A. Rutle, A. Rossini, Y. Lamo, and U. Wolter, "A formal approach to the specification and transformation of constraints in MDE." The Journal of Logic and Algebraic Programming, vol. 81, no. 4, pp. 422-457, 2012, doi: 10.1016/j.jlap.2012.03.006.

153. M. Peterson, J. Bowles, and C. Eastman, "UMLpac: An Approach for Integrating Security into UML Class Design." Proceedings of the IEEE SoutheastCon 2006, pp. 267-272, doi: 10.1109/second.2006.1629362.

154. T. Sommestad, M. Ekstedt, and H. Holm, "The Cyber Security Modeling Language: A Tool for Assessing the Vulnerability of Enterprise System Architectures." IEEE Systems Journal, vol. 7, no. 3, pp. 363-373, 2013, doi: 10.1109/jsyst.2012.2221853.

155. J. Jürjens and P. Shabalin, "Tools for Secure Systems Development with UML: Security Analysis with ATPs." Fundamental Approaches to Software Engineering, pp. 305-309, 2005, doi: 10.1007/978-3-540-31984-9_23.

156. V. Rafe and R. Hosseinpouri, "A security framework for developing service-oriented software architectures." Security and Communication Networks, vol. 8, no. 17, pp. 2957-2972, 2015, doi: 10.1002/sec.1222.

157. D. Basin, M. Clavel, and M. Egea, "A decade of model-driven security." Proceedings of the 16th ACM symposium on Access control models and technologies, 2011, doi: 10.1145/1998441.1998443.

158. R. Croft, Y. Xie, M. Zahedi, M. A. Babar, and C. Treude, "An empirical study of developers' discussions about security challenges of different programming languages." Empirical Software Engineering, vol. 27, no. 1, pp. 1-52, 2021, doi: 10.1007/s10664-021-10054-w

159. M. Salnitri, A. D. Brucker, and P. Giorgini, "From Secure Business Process Models to Secure Artifact-Centric Specifications." Enterprise, Business-Process and Information Systems Modeling, vol. 2, no. 3, pp. 246-262, 2015, doi: 10.1007/978-3-319-19237-6_16.

160. "American National Standards Institute (ANSI)." Van Nostrand, 2005.

161. R. Laborde, S. T. Bulusu, A. S. Wazan, A. Oglaza, and A. Benzekri, "A Methodological Approach to Evaluate Security Requirements Engineering Methodologies: Application to the IREHDO2 Project Context." Journal of Cybersecurity and Privacy, vol. 1, no. 3, pp. 422-452, 2021, doi: 10.3390/jcp1030022

162. Y. Zuo, "Big data and big risk: a four-factor framework for big data security and privacy." International Journal of Business Information Systems, vol. 42, no. 2, pp. 224-242, 2023, doi: 10.1504/ijbis.2023.128648.

163. K. Aldrawiesh, A. Al-Ajlan, Y. Al-Saawy, and A. Bajahzar, "A comparative study

between computer programming languages for developing distributed systems in web environment." Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, pp. 457-461, 2009, doi: 10.1145/1655925.1656009.

164. R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "A comparative study of language support for generic programming." ACM SIGPLAN Notices, vol. 38, no. 11, pp. 115-134, 2003, doi: 10.1145/949343.949317.

165. M. Al-Mekhlal and A. Ali Khwaja, "A Synthesis of Big Data Definition and Characteristics." 2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC), pp. 314-322, 2019, doi: 10.1109/cse/euc.2019.0006.

166. F. Khan, A. Abubakar, M. Mahmoud, M. Al-Khasawneh, and A. Alarood, "Cotton crop cultivation oriented semantic framework based on IoT smart farming application." International Journal of Engineering and Advanced Technology, vol. 8, no.3, pp. 480-484, 2019.

167. M. I. Uddin, S. A. Ali Shah, M. A. Al-Khasawneh, A. A. Alarood, and E. Alsolami, "Optimal policy learning for COVID-19 prevention using reinforcementlearning." Journal of Information Science, vol. 48, no. 3, pp. 336-348, 2020, doi: 10.1177/0165551520959798.

168. A. A. Alarood, E. Alsolami, M. A. Al-Khasawneh, N. Ababneh, and W. Elmedany, "IES: Hyper-chaotic plain image encryption scheme using improved shuffled confusion-diffusion." Ain Shams Engineering Journ

169. al, vol. 13, no. 3, p. 101583, 2022, doi: 10.1016/j.asej.2021.09.010.

170. R. Rani, "Towards Green Computing Oriented Security: A Lightweight Postquantum Signature for IoE." Sensors, vol. 21, no. 5, p. 1883, 2021, doi: 10.3390/s21051883.

171. E. M. Clarke, O. Grumberg, and D. A. Peled, Model Checking, MIT Press, Cambridge, Massachusetts, 2000.

172. R. May, C. Biermann, X.M. Zerweck, K. Ludwig, J. Krüger, and T. Leich, "Vulnerably (mis) configured? Exploring 10 years of developers' Q&As on Stack Overflow," Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems, 2024, pp. 112-122.

173. M. Murtaza, "S-RealSpec: A Security Extension to Detect SQLI attack and Sensitive Data Exposure." Kurdish Studies, vol.12(5), 2024, pp. 757-769.