ANALYZING THE BEHAVIOURAL SECURITY POSTURE OF SOFTWARE SYSTEMS

BY

JOHN BRETON

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Applied Science
in
Electrical and Computer Engineering

Carleton University
Ottawa, Ontario, Canada

To Alanna

Abstract

Designing secure systems from the start remains a challenge for designers. To address this, we introduce an approach to enhance system security during the early design phase. We derived two sound security metrics known as Critical Element Risk Index (CERI) and Corruption Propagation Potential (CPP). These metrics are used to inform a system's behavioural security posture, which we define as a system's resilience to knowable threats based on its flows, as determined by its security policies and threat model.

To complement our approach, we developed an automated tool that generates optimized mitigation recommendations to bolster a system's security level. By evaluating Unified Modelling Language (UML) activity diagrams, we identify threats through pattern matching and suggest appropriate mitigation patterns. This approach supports designers in incorporating security considerations early in the software development lifecycle (SDLC), resulting in an improved behavioural security posture for the system and thus, a more secure system design.

Acknowledgements

This work would not have been possible without the patience, guidance, and incredible support I received from my thesis supervisor Dr. Jason Jaskolka. I would be remiss if I failed to express my gratitude to Dr. George O. M. Yee, who has selflessly volunteered his time to collaborate and review this work throughout the entire process. Further, I would like to thank Dr. AbdelRahman Abdou for the opportunity to branch out of my area of focus and explore other aspects of cybersecurity.

I would like to express my appreciation towards all members of the CyberSEA Lab. Their support and compassion fostered a great environment to collaborate and share ideas about my research. I would also like to specifically thank Dylan Leveille and Alvi Jawad for their friendship and advice throughout the process of writing this thesis.

I also thank my family and close friends for their support and encouragement during this undertaking. Lastly, I thank my wonderful wife Alanna, whose patience, love, and understanding during my graduate studies have pushed me to achieve my goals.

Contents

List of Tables

List of Figures

List of Abbreviations

ATT&CK Adversarial Tactics, Techniques, and Common Knowledge

BNF Backus-Naur Form

BPMN Business Process Model and Notation

BSP Behavioural Security Posture

CAPEC Common Attack Pattern Enumeration and Classification

CERI Critical Element Risk Index

CIA Confidentiality, Integrity, and Availability

CLI Command-Line Interface

CPP Corruption Propagation Potential

CSSM Conditions for Sound Security Metrics

CVSS Common Vulnerability Scoring System

D3FEND Develop, Deploy, and Defend

DoS Denial-of-Service

GUI Graphical User Interface

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

ISO International Organization for Standardization

NIST National Institute of Standards and Technology

OMG Object Management Group

OSM Online Seller of Merchandise

PDF Portable Document Format

SDLC Software Development Life Cycle

SQL Structured Query Language

SR Security Requirements

SSDLC Secure Software Development Life Cycle

STRIDE Spoofing, Tampering, Repudiation, Information disclosure, Denial-of-service, and Elevation of privilege

UML Unified Modelling Language

WCAG Web Content Accessibility Guidelines

XMI XML Metadata Interchange

XML Extensible Markup Language

Chapter 1

Introduction

The design of software systems has matured to enable the creation of large and complex modern systems. Yet, this maturity has not extended to provide designers with the methods and tools required to secure these systems throughout all phases of development. Section 1.1 introduces the concepts of software design and engineering as they relate to system security. Section 1.2 highlights security concepts that relate to the analysis of system design artifacts. Section 1.3 identifies the lack of support for analyzing behavioural design artifacts, highlighting the necessity to establish a method for capturing a system's behavioural security posture. Section 1.4 presents the problem statement for this work. Section 1.5 outlines our contributions. Section 1.6 lists academic publications related to this work. Lastly, Section 1.7 details the structure of the thesis.

1.1 General Context

Software has evolved to the point of being integral to modern-day life. Far beyond the point of individual applications, *software systems* (or simply systems) are designed to cohesively integrate different components to meet a set of requirements that define how a system should behave. From social media platforms to the latest advancements in artificial intelligence, systems continue to grow in scale, scope, and adoption [2]. To deal with the increasing complexity of systems, software engineering continues to be applied to design safe and complete systems in a regulated and structured manner. While not exclusive to software engineering, the structured nature of the domain informs the approach that is used to develop software systems, known as the Software Development Life Cycle (SDLC).

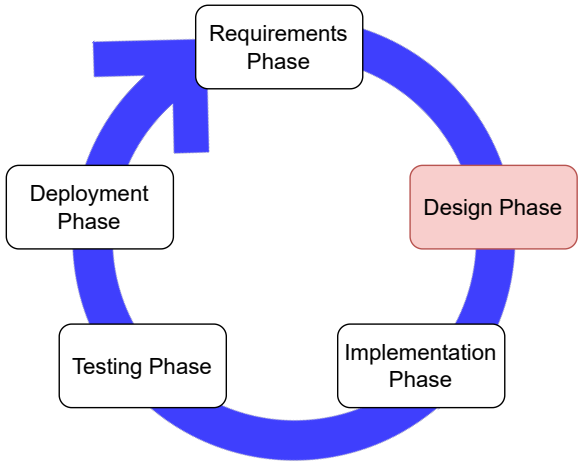


Figure 1.1: An Overview of the SDLC

We will focus on a version of the SDLC with five phases. This simplification is derived from the International Organization for Standardization (ISO)/ International Electrotechnical Commission (IEC)/ Institute of Electrical and Electronics Engineers

(IEEE) 12207 standard for software life cycle processes [3]. In Figure 1.1, we show an overview of the SDLC. For simplicity, we represent the SDLC as a cyclic process. It is important to note that the SDLC is not strictly sequential and there is flexibility when development moves between phases. In certain circumstances, development may return to a prior phase, or activities from multiple phases may occur concurrently.

To briefly explain each phase, the requirements phase deals with the gathering and elicitation of requirements that will define the necessary capabilities of the system. The design phase creates models of the system-to-be, defining the structure, behaviour, and functionality of the system before any code is written. During the implementation phase, models from the design phase are coded into usable systems. Software validation and verification are performed during the testing phase to ensure the system performs correctly and satisfies all requirements derived during the requirements phase. Lastly, the deployment phase deals with the logistics of deploying the completed system and subsequent upkeep activities.

The goal of the SDLC is to develop software systems. It is not designed to result in the creation of secure systems, despite a growing public interest for secure systems by government officials and end users [4, 5]. This shortcoming is nothing new, and attempts to address this issue have been explored, resulting in the conception of the Secure Software Development Life Cycle (SSDLC) [6]. The SSDLC aims to incorporate security throughout the traditional SDLC, rather than attempting to bolt on security after a system has been deployed.

Security considerations can start as early as the requirements phase, but they are best considered throughout each phase. The design phase is especially vulnerable

if security is not considered, as the activities contained within this phase lock in a system in terms of its structure, behaviour, and functionality [7]. Coupled with the fact that security considerations are more effective the earlier they are explored during the SDLC, we choose to focus on the design phase for this thesis [6].

The SDLC design phase generates design artifacts. Typically, these artifacts correspond to a specific system view. Although design artifacts can exist in many forms, the Unified Modelling Language (UML) is commonly used to capture different system perspectives. The UML is a standard published by the Object Management Group (OMG) that allows designers to create system models that can be used to define the structure and behaviour of systems. The latest specification of the UML (version 2.5.1) includes a mapping to show if a diagram is meant to capture structural or behavioural information about a system, which we highlight in Figure 1.2.

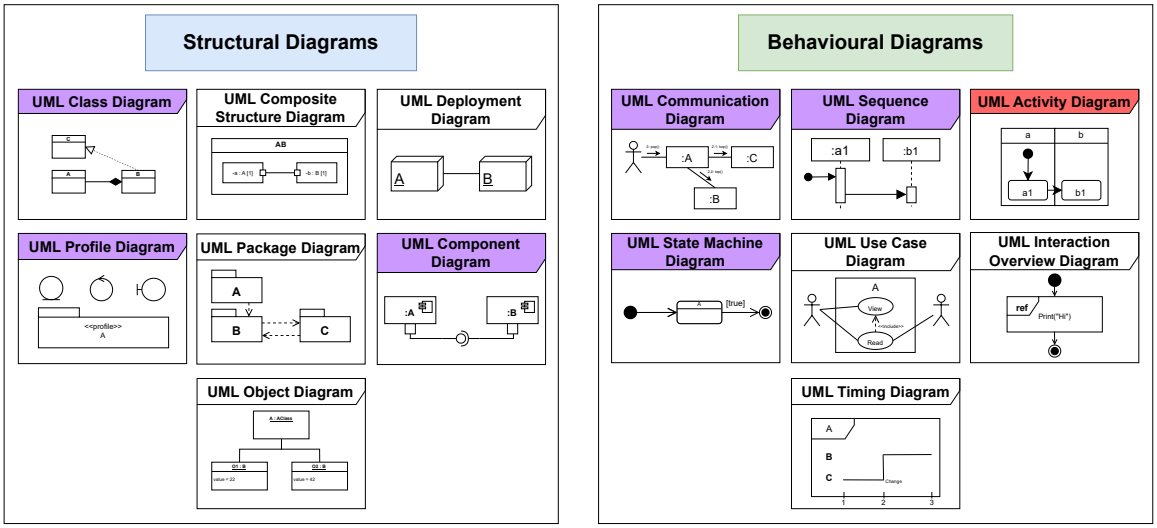


Figure 1.2: The Structural and Behavioural Diagrams of the UML [1]

For both the structural and behavioural categories, the UML defines seven diagrams that aim to capture different information about a system at a high level. The diagrams with purple or red highlights seen within Figure 1.2 are relevant to this thesis and we will provide short overviews for each. Understanding all structural and behavioural diagrams is not necessary for this thesis, but we want to highlight the groupings used to categorize these diagrams. The structural view defines the components of a system, as well as the communication links between these components. It is primarily focused on how the system will serve end users. Importantly, it does not define the precise medium by which these components communicate.

While all structural diagrams listed in Figure 1.2 have their purpose, in this work we only discuss class diagrams, profile diagrams, and component diagrams. Class diagrams are used to show the structure and relationships of classes within a system. Classes are given a name, and they often contain attributes and operations that belong to the class. Relationships between classes are specified using different types of arrows. Profile diagrams extend the capabilities of the UML using visual representations for custom-defined stereotypes, tagged values, and constraints specific to a particular domain or purpose. Lastly, component diagrams depict the structure and relationships between software components in a system, showing how they interact through interfaces and dependencies. Further information regarding all structural diagrams can be found within the latest 2.5.1 specification of the UML [1].

The behavioural view, as the name implies, expresses the behaviour of a system. We define *behaviour* in this context as the control and data flow that move between components within a system. In this view, the nature of the data and the medium by

which this data is transferred is concretely defined. As with the structural diagrams, all of the behavioural diagrams shown in Figure 1.2 serve a distinct purpose. We have chosen to focus on four of the seven behavioural diagrams within this work: communication diagrams, sequence diagrams, activity diagrams, and state machine diagrams.

Communication diagrams illustrate the interactions and message exchanges between objects or instances in a system, emphasizing the order of messages and the direction of communication between objects. Sequence diagrams mirror the description of communication diagrams, all the while supporting additional notation to illustrate conditional execution paths, object creation and deletion, and other advanced concepts that are not supported in communication diagrams. Finally, state machine diagrams depict the various possible states of an object or system and how a system transitions between these states in response to events. They mainly consist of states, transitions, events, guards, and actions, providing a visual representation of the dynamic behaviour of a system.

We note that this thesis is focused on the analysis of activity diagrams, hence its unique highlight colour of red in Figure 1.2. We provide an in-depth look at activity diagrams in Section 3.4. Further information regarding all behavioural diagrams can be found within the latest 2.5.1 specification of the UML [1].

A third view, known as the functional view, also exists but is not explicitly covered by the UML specification. The functional view represents the system from a programmer's or hardware engineer's perspective; viewing the system in terms of its code and the physical components required to function. The hardware components

that make up a system can be specified and linked together, alongside more low-level component communication organization in terms of interfaces and packages. Typically, The functional view of a system is the final view generated during the design phase before development moves onto the SDLC implementation phase. At times, the functional view can also be expressed using the code for system components, rather than higher-level system diagrams.

1.2 Specific Context

In addition to guiding development and serving as an effective means of communication between system designers during the conception of a system, design artifacts can also be analyzed. This analysis has the potential to unveil redundancies within the proposed system design. However, when viewed through a security lens, it can also shed light on threats the system may not be prepared to handle. Moving to a focus on security for our specific context, we define *threats* as avenues that can render damage to a system. Threats are realized through the exploitation of *vulnerabilities*, which can be defined as the weaknesses that exist within a system.

Despite the larger focus on security in recent times, security incidents against systems continue to make headlines. This is not surprising, as systems must be hardened against all potential threats while bad actors need only find a single vulnerability to exploit. This imbalance is all the more reason why novel security research continues to be published. Whether it be highlighting code-level vulnerabilities, methods to identify and respond to security incidents rapidly, or means to harden systems against

attacks that originate from those with direct access to a system, the literature is vast and ever-growing.

Even with this landscape of security research, certain gaps exist that remain to be fully addressed. One such example area involves the inclusion of security considerations during the design phase of software systems. While some research covering this topic exists, studies have found that designers can struggle to implement recommendations suggested by said research due to various factors, such as non-existent automated tool support, a lack of knowledge in the target domain, and other blockers [8]. We believe that the best way to support designers is to provide tools and methods that are shown to be sound and that can be incorporated into their development life cycles.

This need to support designers is continually growing as the threats that plague systems continue to expand. Systems can be subjected to a large number of attacks, which can come from a variety of entry points. This may include disgruntled employees who modify the data of running systems, and state-sponsored actors that launch attacks to overwhelm system resources to the point they can no longer function, among many other possibilities. Systems need to be protected against these threats to avoid security incidents and work should be done to create additional methods to protect systems against exploitation. Such novel methods should also account for the fact that they will need to evolve as new threats are conceived, promoting system security as we know it today and as it may be known in the future.

To understand how systems are protected against threats, it can be helpful to explore some common goals that attackers have when launching attacks against systems. A common teaching in the domain of software security is the Confidentiality, Integrity,

and Availability (CIA) triad, which encompasses the three major security criteria an attacker might target when attacking a system.

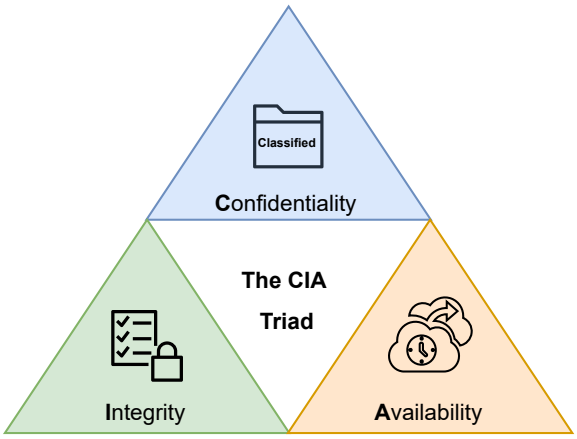


Figure 1.3: The CIA Triad

As shown in Figure 1.3, confidentiality, integrity, and availability exist as three individual but equally important criteria that stakeholders strive to uphold. Confidentiality safeguards data from unauthorized access, integrity ensures data accuracy and completeness, and availability guarantees timely access to resources. Attackers will exploit vulnerabilities in an attempt to compromise these principles. Attackers may aim to breach confidentiality through unauthorized system access via credential stuffing attacks, tamper with data integrity using malicious Structured Query Language (SQL) statements, or disrupt availability via Denial-of-Service (DoS) attacks. These examples are just some of the known threats that can impact the principles of the CIA triad.

These known threats are what make up a system's *threat landscape*, a term used to represent the collection of threats that could pose harm to the security of a system. Given the large number of threats that can exist within a threat landscape, it is

difficult to adequately account for all of them [9]. In an attempt to alleviate this difficulty, the concept of threat categorization exists to group known threats into categories that can be addressed. These categories contain a variety of threats but can help select countermeasures that target multiple threats within a single group, making the task of system hardening a more structured experience. One of the most popular threat model frameworks is known as STRIDE [10].

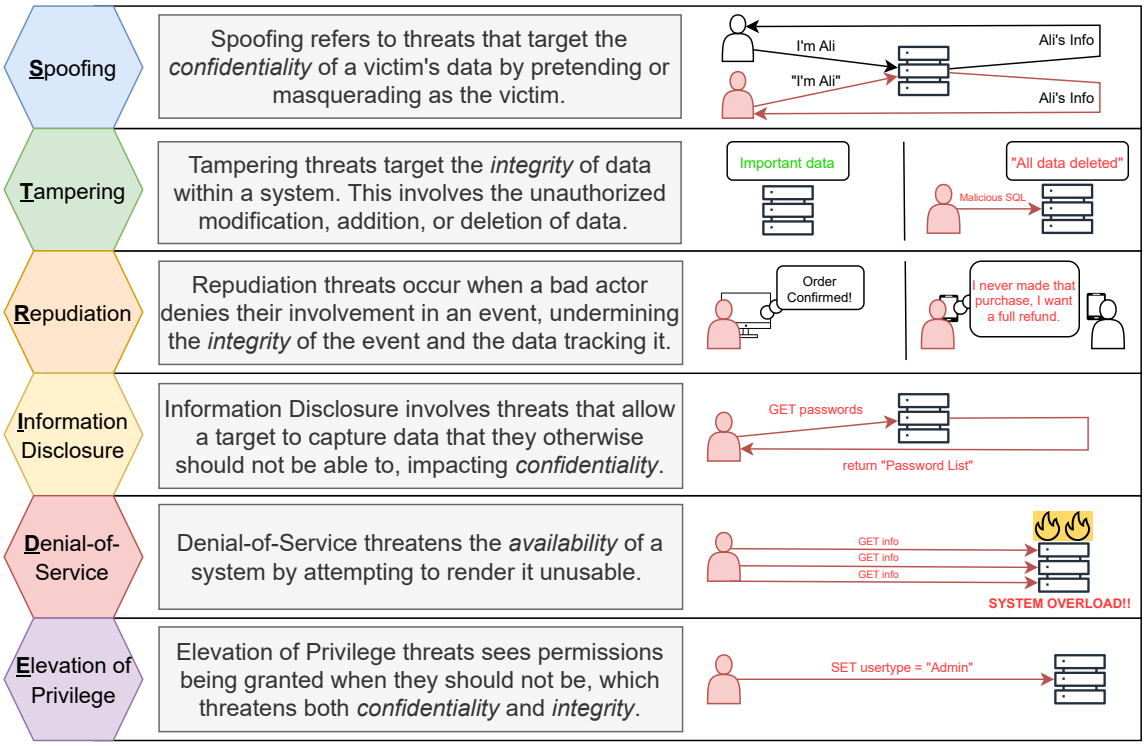


Figure 1.4: The STRIDE Framework of Threat Classification

STRIDE is a mnemonic that stands for six distinct threat categories: **S**poofing, **T**ampering, **R**epudiation, **I**nformation disclosure, **D**enial-of-service, and **E**levation of privilege. Figure 1.4 maps these categories to the aspect of the CIA triad the threats target, while also providing a high-level description and pictorial example

of each threat category. Overall, STRIDE aids in categorizing and understanding threats, providing a structured approach to threat identification and analysis.

While STRIDE is helpful to group threats at a high level, the general mitigation strategies for each category are not enough to address all threats. Further, simply applying a mitigation may not be enough to adequately address a threat, but rather how and where the mitigation is applied can play a role in further hardening a system's security. Nevertheless, knowing what threats a system is susceptible to is an important first step in determining how to best improve a system's security.

1.3 Motivation

As systems continue to grow in size and complexity, we can also notice a growth in security incidents [11]. Systems continue to demand more data from users to operate, and this in turn makes data breaches and security incidents extremely damaging to end users. Bad actors can harvest credit card numbers, personally identifiable data, and other sensitive information through the exploitation of threats in established systems. Below, we highlight security incidents that occurred in 2023.

1. Microsoft's cloud services were exploited in part due to a zero-day flaw, which was a flaw that existed since the conception of the system that allowed access to thousands of email accounts across the world without requiring a victim's interaction [12].

2. Google's Chrome web browser faced an exploit that allowed bad actors to execute arbitrary code on affected systems if a victim visited a maliciously crafted web page [13].

3. Apple's Safari web browser faced a memory corruption vulnerability that allowed bad actors to execute arbitrary code on desktop and mobile systems, a vulnerability which Apple had been aware of for some time before its disclosure [14].

These incidents have and will continue to occur worldwide. Corporations will be faced with costly ventures to correct exploited vulnerabilities in completed systems and have to deal with the fallout of such incidents. It is particularly troubling that some corporations are aware of vulnerabilities yet often neglect to adequately support security measures during system development. The cost of integrating security is perceived to outweigh the potential losses from security incidents, which can motivate corporations to neglect adequate security [15].

To minimize security incidents, we must identify and address gaps in designer support that contribute to these occurrences. Analysis activities can help harden systems against threats and can minimize the costs associated with applying security after a system has moved past the early phases of the SDLC [16]. By providing tool support and techniques that improve system security without significantly disrupting designers' workflow during the SDLC design phase, we can make the cost-benefit analysis more favourable for incorporating security early in system design. In turn, this results in the conception of more secure systems and ultimately, a reduction in overall security incidents.

Capturing a measurable increase in system security is challenging, yet previous work by Samuel addresses this by introducing the concept of a system's structural security posture [17]. A system's *security posture* is defined as a collection of metrics and measurements that can be used to quantitatively represent the security of a software system for comparison-based purposes. Following this, a system's *structural security posture* can be defined as the determination of a system's security posture through its structural view representation. Representing the security level of a system using a collection of security metrics provides the means to capture a system's security using information gathered from its initial design. As we discussed in Section 1.1, a system is comprised of multiple views, and the information in one view may not be present in another. To support designers with the incorporation of security throughout development, we need to provide the tools and techniques to incorporate security for each of these views. In doing so, vulnerabilities that were not perceivable in a system's structural view may still be identified and addressed upon analysis of a system's behavioural or functional view.

For this thesis, we focus on determining a system's security posture given its behavioural view by developing a definition and methodology to determine a system's behavioural security posture. Leveraging information that was not present in a system's structural view, we need to consider information from a system's behavioural view to perform new types of security analysis that can be used to harden systems against a selection of threats across each category of the STRIDE threat framework. Similar to the work done to develop the definition of a system's structural security posture, we hope to provide designers with the techniques and tool support to enable the incorporation of security considerations for additional activities during the early

phases of the SDLC. In short, we must identify and address gaps in designer support surrounding the incorporation of security in behavioural system models, moving us closer to achieving comprehensive security support throughout the SDLC design phase.

1.4 Problem Statement

Following along with the research that suggests that security can be especially effective if considered during the early phases of the SDLC, we highlight the following problem statement:

Despite the increasing emphasis on cybersecurity in recent years, there remains a significant gap in tool and methodological support for system designers to seamlessly integrate security measures during the conception of a system's behavioural view as part of the SDLC design phase. This gap can lead to vulnerabilities and security inefficiencies in the later stages of development.

While this tool and methodological support gap for system designers has been explored in the past, it has not been fully resolved. An effort must be made to address it if we hope to promote the incorporation of security throughout all phases of the SDLC. This leads directly to our research questions for this work:

> ***RQ1***: How can we define a system's behavioural security posture in terms of quantitative and sound security metrics?

RQ2: In what ways can the modification of behavioural design artifacts, informed by a system's behavioural security posture, lead to measurable improvements in system security against different types of threats?

RQ3: How can both the determination of a system's behavioural security posture and the suggested set of improvements based on the calculation of said posture be automated to best support designers in ameliorating the security of their systems?

While it is understood that these questions address only parts of our problem statement, they serve as a starting point for analyzing the security of software systems using the notion of a security posture given a system's behavioural view. We make no claims that through answering these questions, a user analyzing their system's behavioural view will be able to account for all knowable threats. However, we do claim it will help to mitigate a select subset of those threats to improve the security of their system designs.

1.5 Contributions

Through answering our research questions listed in Section 1.4, we present the following contributions:

1. The definition of behavioural security posture, which can capture a system's resilience against knowable threats through the analysis of behavioural design artifacts (**Chapter 4**, addresses ***RQ1***).

2. An approach to identify potential threats within behavioural system design artifacts using attack and mitigation pattern matching, captured as a security metric, with the ability to add patterns to identify new threats (**Chapter 5**, partly addresses ***RQ2***).

3. An approach to minimize the propagation potential of corrupted data, captured using a security metric, augmented with mitigation placement optimizations (**Chapter 6**, partly addresses ***RQ2***).

4. A tool (called Dubhe) that automates the determination of a system's behavioural security posture and suggests locations of threat mitigations to support designers in their efforts to incorporate security throughout the early stages of the SDLC (**Chapter 7**, addresses ***RQ3***).

1.6 Related Publications

The following conference publication relates to the work presented in this thesis:

- J. Breton, G. O. M. Yee, and J. Jaskolka. Hardening Systems Against Data Corruption Attacks Using UML Activity Diagrams. In *Proceedings of the 16th International Symposium on Foundations & Practice of Security*, FPS 2023, pages 391–407, Bordeaux, France, Dec. 2023.

1.7 Thesis Structure

An overview of the remaining chapters in this thesis is as follows:

Chapter 2 details existing literature that covers the state of the art regarding the determination of security at the early design stage for software systems.

Chapter 3 covers preliminary topics spanning modelling and security concepts essential to our approach.

Chapter 4 defines behavioural security posture, introduces two novel security metrics that contribute to this definition, and provides a high-level overview of our methodology for evaluating system designs based on these metrics.

Chapter 5 presents an approach that can be employed to identify threat and mitigation patterns present within UML activity diagrams and how this approach is used to calculate a security metric that captures the risk of critical elements within a system.

Chapter 6 highlights an approach that can be employed to identify corruptible flows present within UML activity diagrams and how this approach is used to calculate a security metric that expresses the corruption propagation potential within a system.

Chapter 7 introduces Dubhe, an automated tool that executes our previously presented approaches to determine a system's behavioural security posture, demonstrated through a scenario walkthrough.

Chapter 8 restates the contributions of the work, discusses open areas of research, and concludes the thesis.

Chapter 2

State of the Art

In this chapter, we discuss the state of the art for evaluating the security of systems at the early design stage. Section 2.1 discusses research related to security metrics applied to early system designs. Section 2.2 explores the notion of security posture and how it relates to the representation of security for early system design. Section 2.3 highlights security tools applicable to the behavioural view of software systems. Section 2.4 concludes our findings from this chapter.

2.1 Security Metrics used for System Security

In this section, we present research that focuses on the definition of security metrics that can be applied to early system designs. We follow this up with an overview of works that present methods and frameworks for evaluating security metrics.

2.1.1 Security Metrics for Early System Designs

Security literature is replete with security metrics. Given the focus of this work on analyzing early system security within the SDLC design phase, our objective is to highlight security metrics applicable to the three views that make up this phase.

In the context of the structural view of systems, Alshammari et al. [18] created seven metrics that could be applied to UML class diagrams for object-oriented systems. Mir and Quadri [19] created metrics that concentrate on dependencies as defined in UML component diagrams, providing a representation of a system's overall availability. Further, Manadhata and Wing [20] devised an attack surface metric that relies on privilege levels and access rights to assess the attack surface across the components of the system. In particular, this metric was later adapted for use with UML class diagrams by Samuel et al. [21].

While metrics that target the behavioural view exist, they are less prevalent. This observation was initially made by Jansen in [22] and was later indirectly reconfirmed by Yee in [23]. More recently, Alshayed et al. [24] introduced a methodology to improve the security of systems by analyzing their UML sequence diagrams. They employed a machine learning model to assess bad design smells as a metric for this purpose.

Chowdhury et al. [25] introduced three security metrics designed for the functional view of software systems, particularly towards the latter stages of the SDLC design phase. These metrics, namely the Stall Ratio, Coupling Corruption Propagation, and Critical Element Ratio offer quantitative insights into a system's security at the

code level. Another metric that can be applied to the functional view of systems was developed by Albrecht [26] and is known as the function point metric. While not initially conceived with security in mind, the information that measuring function points provides regarding the size and complexity of a system has been leveraged by Jones [27] to detect security flaws within a system's functional view.

Although metrics exist that apply to each of the three views, it is worth noting that the behavioural view has received less attention in terms of security metrics, in contrast to the structural and functional views in early system designs. We aim to rectify this imbalance by introducing metrics specifically designed for the behavioural view of early system designs.

2.1.2 Evaluating Security Metrics

Due to the abundance of available security metrics, selecting appropriate metrics can be a challenging task. Consequently, the assessment of security metrics has become a prominent area of research in recent years. Yee [28] updated a previous method [29] for designing sound security metrics based on the evaluation of a security metric using three specific conditions. These conditions covered whether a metric was well-defined, progressive, and reproducible. This approach yields quantitative security metrics that are less susceptible to subjective interpretation by practitioners.

In a different vein, Bhol et al. [30] highlighted the lack of standardization in the taxonomy of security metrics and proposed a taxonomy that encompasses the categories of vulnerabilities, protection mechanisms, threats, users, and encounter outcomes.

Numerous other works [31, 32, 33, 34, 35] have evaluated security metrics custom-tailored to domains within the realm of software systems.

Some research has focused on critiquing established security metrics to shed light on potential issues in their application. For instance, Albanese et al. [36] identified shortcomings in the widely used Common Vulnerability Scoring System (CVSS) metric and proposed a novel framework to aid in the creation of new vulnerability metrics. Nayak et al. [37] reviewed an attack surface metric and found that despite the metric demonstrating that the attack surface of a system was reduced, it did not necessarily result in an improvement in overall system security. Further, Pendleton et al. [38] surveyed and identified several problematic security metrics that lacked any form of evaluation to determine if they could be used to properly reflect changes in a system's security.

The abundance of security metrics has necessitated methods for their evaluation. In our work, we aim to define metrics that can be determined to be sound to avoid potential shortcomings, such as those seen with the CVSS metric.

2.2 Representing the Security of Systems as a Security Posture

In the domain of system security, security posture is a term used to describe a system's security, as assessed by a set of security metrics. Bodeau et al. [39] explored the effectiveness of using metrics to determine a system's security posture, emphasizing

various resiliency metrics that can inform this posture and providing a template for the definition and evaluation of new security metrics. Yee [23] discussed the advantages of using security metrics to assess a system's security posture, including allowing for real-time updates to a system's posture during execution.

When assessing initial system designs, Samuel [17] presented the structural security posture of a system by employing a set of metrics derived from information contained within design artifacts. Using UML class diagrams, metrics that represent the attack surface and eigenvector centrality of components inform designers about the security of their designs. Bakirtzis et al. [40] adopted a more general approach to applying the concept of a security posture to early system designs. Instead of specifying particular design artifacts for analysis, they relied on external data sources to determine the security posture across various representations of early system models.

Other works [41, 42, 43] extended the definition of security posture to capture the human factors that contribute to security inefficiencies within systems. A common theme among these works is the recognition that humans play a role in shaping a system's security posture through their influence on the system's overall attack surface.

To the best of our knowledge, there is a gap in the literature when it comes to defining security postures for different system views during the SDLC design phase. Drawing from prior research, our objective is to leverage design artifacts to compute metrics that collectively depict a system's behavioural security posture.

2.3 UML-Based Security Tools Targeting the Early Design Stage

As illustrated by studies conducted by Luan et al. [8] and Jürjens and Shabalin [44], the development of tools is essential to support designers in integrating security into early system designs effectively. Jürjens [45] created UMLsec, a unique UML profile that enables designers to seamlessly incorporate security objectives into their system designs. This UML profile can be easily imported into existing modelling tools. However, to harness UMLsec's full potential, designers must invest time in learning how to accurately apply the profile within their UML diagrams.

Similarly, Lodderstedt et al. [46] described SecureUML, a modelling language and accompanying tool designed for generating role-based access control architectures. Like UMLsec, SecureUML demands a learning investment to effectively enhance the security of systems at the early design stage. As noted by Ebad [47], this learning investment has been recognized as a significant barrier to the widespread adoption of these tools, despite their ongoing presence in recent security research.

Other tools [48, 49, 50] have concentrated on system security by extending the UML while targeting a specific type of diagram, such as UML activity diagrams. In a different approach, Samuel et al. [21] created Polaris, a tool designed to evaluate the structural security posture of early system designs. To perform this evaluation, Polaris analyzes UML class diagrams alongside designer-specified privilege and access levels. Notably, Polaris does not require the extension of the UML with new elements,

making it more accessible than other existing tools that integrate the UML into their security analysis activities.

While tools exist to support designers in some parts of the SDLC design phase, tools tailored towards the behavioural view of systems that do not extend the UML are lacking. We hope to rectify this gap by focusing our efforts on the development of a tool designed with the behavioural view of systems as the focus without requiring extensions to the UML.

2.4 Chapter Summary

Throughout this chapter, we discussed the state of the art for evaluating the security of systems during the early design phase. We have identified gaps in the literature in terms of sound metrics that target the behavioural view of systems. There is also a need to define a system's behavioural security posture. Additionally, we have emphasized the significance of tool support while recognizing a shortage of tools capable of facilitating security integration throughout the entire SDLC design phase. In the next chapters, we aim to bridge these gaps in the literature to bring us closer to our overarching goal of assisting developers in seamlessly incorporating security throughout the entirety of the SDLC design phase.

Chapter 3

Preliminaries

In this chapter, we cover the background information required to understand the rest of this thesis. Section 3.1 discusses the data analysis technique known as pattern matching. Section 3.2 introduces three external data sources that can be used to gather threat and mitigation data. Section 3.3 describes the XML Metadata Interchange (XMI) format and its syntax. Section 3.4 provides an overview of UML activity diagrams. Section 3.5 outlines an existing methodology that can be used to determine if a security metric is sound. Lastly, Section 3.6 restates the main concepts explained in this chapter.

3.1 Pattern Matching

Pattern matching is a technique in which exact occurrences of a desired pattern can be found within a larger set of data [51]. The technique can be applied to a variety of

data media, including text, pictorial representations, and sound waves, just to name a few. In more recent times, computers make use of pattern matching for activities such as model training in machine learning, to more primitive uses such as input validation using regular expressions. Figure 3.1 shows a text-based example of pattern matching.

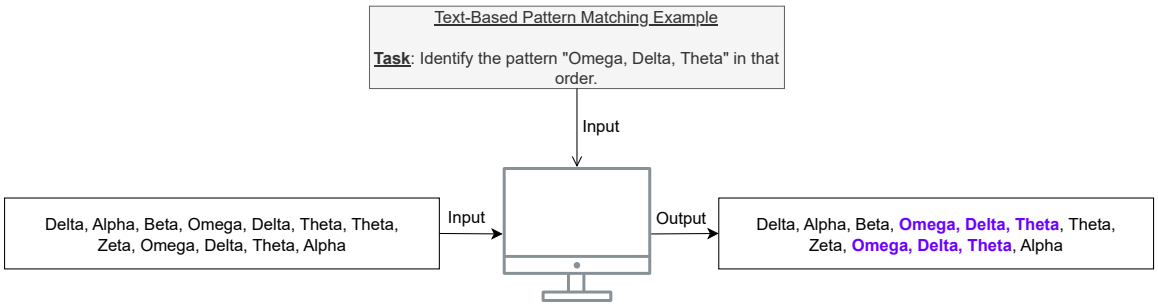


Figure 3.1: The Pattern Matching Process

To clarify, when we use the term pattern matching, we are referring to the process of identifying data points within a pattern in a specific order. This differs from *pattern recognition*, which is a similar technique but is less strict regarding the sequence of the data that participates in a pattern. Pattern recognition identifies a match for a pattern even so long as the exact elements of a pattern are present, regardless of their sequential ordering. The distinction is important to understand as we exclusively use pattern matching in our approach later in this thesis.

3.2 External Data Sources for Threats and Mitigations

In general, analysis activities require data. In the context of security analysis, this data often takes the form of information regarding threats and mitigations. While threats serve as avenues that can render damage to a system, *mitigations* are mechanisms that attempt to prevent that damage from occurring. A single threat may require one or multiple mitigations, and a single mitigation may be capable of mitigating multiple threats. In a sense, threats and mitigations are inherently linked, and both can be leveraged to perform security analysis of a system.

However, before data can be applied to the security analysis of a system, it must be collected. The landscape of threats and mitigations is vast and ever-growing [52]. Due to the evolving nature of threats and mitigations, it can be helpful to rely on external data sources to ensure the data retrieved to perform security analysis is up-to-date and complete [17]. While there are many options to select from, the MITRE corporation is well-respected among security practitioners and offers three data sources from which threat and mitigation information can be retrieved. Brief overviews of the information contained within these three MITRE sources are provided below.

MITRE CAPEC: MITRE Common Attack Pattern Enumeration and Classification (CAPEC) [53] provides a foundational knowledge base of common attack patterns that can be used to pre-emptively recognize and address vulnerabilities. An entry within CAPEC typically includes a description of the attack, a threat likelihood and severity rating, a list of relationships to other attacks, how the attack is performed,

what would be required to have the attack be successful, possible mitigations, and references to related entries in other external data sources.

MITRE ATT&CK: MITRE Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) [54] is a detailed knowledge base that encapsulates the tactics, techniques, and procedures employed by attackers. An entry within ATT&CK typically includes a description of an attack's technique, the associated categorizations for said technique, examples of successful attacks using this technique, detection strategies, mitigation options, and references.

MITRE D3FEND: MITRE Develop, Deploy, and Defend (D3FEND) [55] is a countermeasure knowledge framework that is designed to augment system defences through a structured catalogue of defensive techniques that directly correspond to established attack methodologies, acting as a companion data source to MITRE ATT&CK. These entries serve to inform the selection and creation of mitigations against established attack techniques. An entry within D3FEND includes a definition of the defence, an explanation of how it protects a system, considerations for the defence if applicable, and the relationship to system components (called digital artifacts) within the framework.

3.3 The XML Metadata Interchange (XMI)

The XMI is an established model-interchange language created by the Object Management Group (OMG). As previously mentioned in Section 1.1, OMG is also responsible for the official UML specification [1]. Alongside the UML specification,

the OMG includes rules to be able to represent any UML-compliant model within the XMI's text-based format. While XMI is typically utilized for data exchange between modelling tools, it is possible to parse the data for non-modelling purposes. While there are many XMI tags, we only present a breakdown of the tags that will be explored later in this thesis.

The XMI <groups> tag, as the name implies, is used to group a set of child <node> tags, acting as their parent element. We describe the tag's attributes in Table 3.1.

Table 3.1: The XMI <groups> tag

Attribute Name	**Value Type**	**Default Value**	**Description**
xmi:id	string	nil	A unique lexicographical string used to identify the group. E.g., AAAAAAGKhD-JtzLHih9c=
name	string	nil	A name given to the group by a designer. E.g., Client
visibility	string	public	E.g., public, protected, private, and package
xmi:type	string	nil	A string used to specify the XMI type of the group. It links to the name of the element in the UML specification. E.g., uml:ActivityPartition

The XMI <node> tag is the main element present in UML diagrams. For example, in class diagrams, classes would be the nodes. In activity diagrams, the actions would be considered the nodes. We describe the tag's attributes in Table 3.2.

The XMI <edge> tag is used to capture the connections between nodes. We describe the tag's attributes in Table 3.3.

Table 3.2: The XMI <node> tag

Attribute Name	**Value Type**	**Default Value**	**Description**
xmi:id	string	nil	A unique lexicographical string used to identify the node. E.g., AAAAAAGKhDjGkrQjpT4=
name	string	nil	A name given to the node by a designer. E.g., Begin Client Authentication
visibility	string	public	The visibility of the node. This is independent of the visibility of the parent group. E.g., public, protected, private, and package
xmi:type	string	nil	A string used to specify the XMI type of the node. It links to the name of the element in the UML specification. E.g., uml:OpaqueAction
isLocallyReentrant	boolean	false	Used to specify whether the node is designed to handle interrupts that occur during execution. E.g., true or false
isSynchronous	boolean	true	Determines if the node functions as synchronous or asynchronous. E.g., true or false

3.4 UML Activity Diagrams

As defined in the 2.5.1 specification of the UML [1], activity diagrams are primarily meant to model the flow of control within a system or process. Activity diagrams demonstrate a temporal progression of execution control and data flow as it moves through a system to accomplish a specific use case. It should be noted that while the behaviour of a full system could technically be captured in a single activity diagram, that is not often the norm. Instead, multiple activity diagrams are created to conform to different use cases that the system is expected to accomplish. Like most UML

Table 3.3: The XMI <edge> tag

Attribute Name	**Value Type**	**Default Value**	**Description**
xmi:id	string	nil	A unique lexicographical string used to identify the edge. E.g., AAAAAAGKhDQhErJFcMk=
name	string	nil	A name given to the edge by a designer. E.g., authenticated
visibility	string	public	The visibility of the edge. E.g., public, protected, private, and package
source	string	nil	A reference to the xmi:id of the node this edge originates from. E.g., AAAAAAGKhDjGkrQjpT4=
target	string	nil	A reference to the xmi:id of the node this edge leads into. E.g., AAAAAAGKhDMIcrH8W50=
xmi:type	string	nil	Used to specify the XMI type of the element. It links to the name of the element in the UML specification. E.g., uml:ControlFlow

diagrams, activity diagrams support a large number of elements. While every element of activity diagrams will not be relevant to this thesis, we wish to highlight all of the elements that will be used in our analysis of systems in future chapters. These elements, alongside their pictorial representations, descriptions, and XMI encoding, are shown in Figure 3.2.

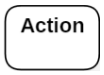
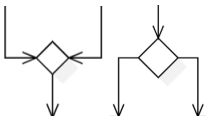
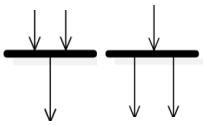
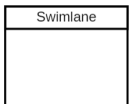
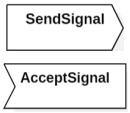
Name	Pictoral Representation	Description	XMI Type
Action		Actions are the primary elements within Activity Diagrams, describing an action performed by a system for a given flow. Actions without a syntactical specification are considered Opaque Actions.	uml:OpaqueAction
Initial Node (Top) Final Node (Bottom)		Initial Nodes act as the entry-point for a flow captured in an Activity Diagram. Final Nodes act as the end-point for the activity in an Activity Diagram. All other flows end when a Final Node reached.	uml:InitialNode uml:ActivityFinalNode
Control Flow		Control Flows link between elements to demonstrate the progression of control through an Activity Diagram.	uml:ControlFlow
Merge Node (Left) Decision Node (Right)		A Merge Node is used to merge multiple non-concurrent flows into a single flow. A Decision Node is used to split off into multiple non-concurrent flows, where only one flow is chosen.	uml:MergeNode uml:DecisionNode
Join Node (Left) Fork Node (Right)		A Join Node is used to merge multiple concurrent flows into a single flow. A Fork Node is used to split into multiple concurrent flows.	uml:JoinNode uml:ForkNode
Swimlane		A Swimlane is used to represent an object that is participating in the flow, parenting Actions and Nodes that it will be responsible for resolving.	uml:ActivityPartition
Send Signal (Top) Accept Signal (Bottom)		A Send Signal is used to send a signal or message within an Activity Diagram. An Accept Signal is used to receive a previously sent signal or message within an Activity Diagram.	uml:SendSignalAction uml:AcceptEventAction
Datastore		A Datastore is denoted with the stereotype <<datastore>> and represents a collection of data (e.g., a database) in an Activity Diagram.	uml:DataStoreNode
Accept Time Event		An Accept Time Event is used to handle instances where a specified amount of time elapses without the completion of the main path within an Activity Diagram.	uml:AcceptTimeEventAction
Flow Final Node		A Flow Final Node indicates the end of a flow within an Activity Diagram. Unlike Final Nodes, they do not indicate the end of an activity and a Flow Final Node can be reached without ending other flows.	uml:FlowFinalNode

Figure 3.2: An Overview of UML Activity Diagrams

3.5 Sound Security Metrics

In the context of cybersecurity, *security metrics* are metrics that are designed to capture and quantitatively express an aspect of a system's security. A security metric is *sound* if it is well-defined, reproducible, and progressive. These characteristics are formally presented by Yee as Conditions for Sound Security Metrics (CSSM) and provide a consistent evaluation basis for determining metric soundness [28]. These CSSM will be used to evaluate security metrics for soundness later in this thesis. Due to their application in our work, we break down the CSSM as presented by Yee [28] in more detail:

Well-Defined: For a security metric to be considered well-defined, four sub-conditions must be satisfied.

1. **The metric must measure the security level of an asset**. As defined by National Institute of Standards and Technology (NIST), an *asset* is anything that can be defined as having inherent value to stakeholders or the operation of a system [56]. Examples include customer data, a proprietary piece of software, or even entire computer systems.

2. **The metric must be meaningful, objective, and unbiased**. This implies that the metric is quantitative and can be determined with fixed data free from human subjectivity. It also implies that the metric should express clear information that is unlikely to be misinterpreted.

3. **The metric must be complete**. This implies that the metric is fully defined to be applicable for its specific purpose. As an example, if a security metric

only measures the number of detected unauthorized system intrusions, it fails to account for the number of unauthorized system intrusions that went undetected. In other words, completeness is meant to ensure that the metric is not missing any consideration that should be included as part of what is it attempting to measure.

4. **The metric must be obtainable without undue hardship or costs**. This will vary depending on the individual or corporation where the security measure will be applied. A large corporation may have a different definition of undue cost compared to an individual security hobbyist, but the sentiment that the metric can be calculated without the need to secure expensive data or data that may put the practitioner at risk is enough to satisfy this condition.

Progressive: For a security metric to be progressive, it must converge towards a value or set of values indicative of an acceptable security level. This convergence is likely only to be observed if the metric is evaluated over a sufficiently long period. Short evaluation periods may be enough to determine a general trend that can be used to suggest the progressiveness of a security metric.

Reproducible: For a security metric to be reproducible, it means that an independent party can perform the metric evaluation in their environment and obtain the same values or similar values as another practitioner in a different environment. Similar values may differ in numerical representation, but the takeaways from the data will be the same regardless.

To promote confidence in the results of a security analysis, the analysis must be informed by sound security metrics [28]. Sound security metrics promote unambiguous representations of a system's security level through logically complete metric definitions [57, 58, 59]. Demonstrating that metrics are sound also reduces the possibility that the metrics are criticized for weaknesses, not too dissimilar to those observed with the CVSS metric [36], which has the potential to undermine the representation of a system's behavioural security posture. To demonstrate how the CSSM are applied, we can test an existing security metric for soundness. As discussed in Section 2.1.2, CVSS is a popular security metric used to assess the criticality of vulnerabilities using a numerical score [60]. We break down the evaluation of the metric for soundness by considering each of the CSSM.

Well-Defined: CVSS indirectly measures the security level of an asset by indicating the severity of vulnerabilities that may be present in a system. In this case, the asset being measured is the system itself. It is meaningful since its value reflects the severity of a vulnerability, with the understanding that a higher score means a vulnerability poses more risk to a system if it is present. It is not objective as the inputs needed to calculate the CVSS of a vulnerability require a practitioner to choose between values of "Low", "Medium", and "High", which will be informed by the practitioner's definition of these labels. This potential for bias prevents the metric from being objective. The purpose of the CVSS metric is to capture the risk of a vulnerability should it be present in a system, and this is expressed through its final numerical score between 0 and 10. This score is calculated using several temporal and environmental factors [60], all of which have been fully defined. As the metric accomplishes its purpose with the values presented in its definition, the metric

is complete. The values needed to calculate the CVSS metric can also be gathered at a low cost, as free online calculators exist to perform the required calculations to generate a CVSS score [61]. Further, the inputs required for the calculation of the metric are based on subjective selections of the practitioners and do not require expensive operations to determine. As mentioned, this subjectivity and potential for bias prevents the metric from being considered unbiased. Thus, the CVSS metric does not satisfy all sub-conditions required to be considered well-defined.

Progressive: CVSS calculates vulnerability severity on a scale of 0 to 10. Higher values indicate more severe vulnerabilities. Through the evaluation of multiple factors over a sufficient period, the CVSS of a vulnerability will converge to a value indicative of the level of severity the vulnerability poses on a system. This convergence informed by data gathered over a sufficiently long time is enough to consider the metric progressive.

Reproducible: As the calculation of a CVSS score only requires the selection of values across different base metrics, it can be argued that security practitioners with similar knowledge sets can obtain final scores that will be close in value when evaluating a vulnerability. However, the subjectivity required when specifying values for the base metrics allows for differences in the final calculated score. Given the sheer number of base metrics involved in the calculation of the CVSS metric, differences across a few of these base metrics due to a practitioner's subjectivity would not severely impact the final calculated score and the same general takeaways could be obtained. As such, the metric can be considered reproducible.

In short, the CVSS metric is progressive and reproducible, but it is not well-defined. Since all three of the CSSM have not been satisfied, the metric cannot be considered sound. This mirrors a previous result from Samuel [17], where a similar but now outdated evaluation framework was used to determine that the CVSS security metric is not sound.

3.6 Chapter Summary

As discussed in this chapter, we covered preliminary concepts that will be explored in our approach in later chapters. We began with an introduction to the technique of pattern matching with a focus on its text-based application. This was followed by an overview of MITRE CAPEC, MITRE ATT&CK, and MITRE D3FEND which serve as external sources that can be used to gather threat and mitigation information. We then explored the XMI format with a focus on the syntax that is used in our approach. Next, we broke down UML activity diagrams by highlighting the key elements that can appear in these diagrams. We introduced CSSM which can be used to determine the soundness of a security metric. In the next chapter, we present our definition of a system's behavioural security posture and the associated security metrics that are used as part of its representation.

Chapter 4

Defining and Capturing Behavioural Security Posture in Systems

In this chapter, we outline our definition of behavioural security posture alongside an overview of an approach that we developed that can be used to determine a system's behavioural security posture. Section 4.1 defines behavioural security posture alongside a breakdown of the definition. Section 4.2 defines a security metric that evaluates the risk of critical elements present within systems. Section 4.3 defines a security metric that evaluates the potential for corruption within the data and control flows of systems. Section 4.4 provides a high-level overview of our approach. Section 4.5 discusses the key takeaways of this chapter.

4.1 Defining Behavioural Security Posture

As highlighted in Section 2.2, the literature defines *security posture* as a means to represent a system's security using a collection of calculated security metrics. We highlight that a security posture is not a single quantifiable representation of a system's security. Rather, it is a collection of security metrics that have been grouped to provide a basis to compare different system design options in terms of their security. The information available to calculate security metrics that can inform a system's security posture varies depending on which view of the system is being analyzed. For example, the information captured in a system's structural view may not be present in a system's behavioural or functional view.

Given this difference in available information, to fully capture the security of a system it is important to analyze it from the perspective of each of its views. While prior work by Samuel [17] has defined a system's structural security posture, definitions for a system's behavioural or functional security posture have yet to be developed. In this work, we limit our focus to the behavioural view of systems conceived during the SDLC design phase. To that end, we present our definition of behavioural security posture below.

> ***Behavioural Security Posture (BSP)***
> *A system's resilience to knowable threats based on its flows, as determined by its security policies and threat model.*

While the terms mentioned in our definition may be known within the general domain of system security, we wish to explain their use concerning a system's behavioural

security posture. We refer to a system's *resilience* as its ability to withstand or prevent the impact of knowable threats. *Knowable threats* refer to a collection of threats retrieved from external data sources. Knowable threats are an explicit way to refer to documented threats, typically including the method the threat may be exploited and optimistically, specific countermeasures or mitigation strategies that can be employed to protect against the threat.

A system is not susceptible to every known threat. As an example, certain threats may target data stores using malicious query statements, but not all systems will contain data stores. This distinction would be covered by a system's *threat model*, which encapsulates the known threats that could harm a system. A threat model helps to inform a system's *security policy*, which is used to define how a system should behave to achieve its security objectives. An example might be to state that one component of a system can only send data to another component through an encrypted communication channel. Security policies will often list these conditions in the form of security requirements, which are similar to software requirements, developed during the requirements phase of the SDLC.

Lastly, *flows* are used to describe the expected object and control paths present within a behavioural representation of a system. Flows highlight how a system progresses, both in terms of data flow and control flow. Attackers may try to interact with flows in a variety of ways, such as trying to discern the data being passed between components or attempting to redirect control to a different component. In the behavioural

representation of systems, flows pass between system elements to accomplish a desired use case. Both the flows and the system elements participate in defining a system's *behaviour*.

Having broken down our definition of a system's behavioural security posture, we now transition to discussing the security metrics that will be used to inform a system's behavioural security posture.

4.2 A Security Metric for Behavioural System Elements

In this section, we present a security metric that represents a system's risk as defined by the threats present in a system's behavioural representation. This metric focuses on the elements present in a system's behavioural view that act in tandem with flows to define a system's overall behaviour.

4.2.1 Critical Element Risk Index (CERI)

To properly evaluate a system given its behavioural view we must consider all aspects of the system that contribute to its behaviour. A system's behaviour is dependent on its flows, but for flows to exist there must be elements for data and control to flow between. As such, we require an approach to evaluate the security of the elements that participate in the definition of a system's behaviour to assess a system's BSP.

We can achieve this evaluation with a security metric that captures the security of behavioural system elements by considering the threat landscape that affects them.

In our work, *behavioural system elements* refer to the elements of a system model that are linked together by control or data flows. Using UML activity diagrams as an example, everything highlighted in Figure 3.2, except for control flows, are considered behavioural system elements. These behavioural system elements (or simply elements) provide the means to create a temporal path using control and data flows to accomplish a desired use case. Given their prevalence in a system's behavioural view, bad actors target these elements for nefarious purposes. Elements in a system can be placed in certain threatening patterns that can allow attackers to exploit them. We provide examples of these threatening patterns in Section 5.3.

However, there are instances where systems cannot avoid including threatening patterns. In some situations, threat patterns will need to be present to satisfy a desired use case. In these scenarios, designers must attempt to mitigate the threats that exist in a system. In the same way that certain elements create threat patterns, a designer can intentionally introduce mitigation patterns into their system designs to address threats. If the mitigations are properly implemented, the elements that participate in the threat pattern are less likely to be exploited by bad actors, as protections are in place to prevent these threats from being realized. To adequately measure the risk an individual element may pose to the system, we must measure its participation in threat patterns while accounting for the threats that have been mitigated.

Of course, all elements of a system can be susceptible to attacks regardless of their participation in threat patterns. However, we argue that elements that participate in

threat patterns are more likely to be exploited than those that do not. Following this logic, we define *critical elements* as the elements present in a system that participate in threat patterns. Measuring the risk that these critical elements pose to a system is vital to represent a system's BSP due to their role in defining a system's behaviour.

Relying solely on an element's participation in threat patterns while accounting for the fact those threats can be mitigated is not enough to adequately capture the risk associated with a critical element. While we are focusing on the elements in a system that contribute to its defined behaviour, it would be misguided to completely disregard the impact of flows. We can utilize information present in a system's behavioural view to further represent the risk a critical element poses in a system. Chowdhury et al. [62] demonstrated evidence that measuring the complexity, cohesion, and coupling of system elements is an effective means of indicating vulnerabilities. While many of the metrics evaluated by Chowdhury et al. [62] can only be applied to a system's functional view, the evaluated cyclomatic complexity metric initially derived by McCabe [63] can be determined using information from a system's behavioural view. Thus, an ideal security metric will consider the cyclomatic complexity of a critical element.

While cyclomatic complexity is normally calculated for a full control-flow graph, it is possible to determine the cyclomatic complexity for a single element by considering the element and the edges leading into it as a subgraph. For reference, the equation for cyclomatic complexity is the following:

$$M = E - N + 2P \tag{4.1}$$

where M is the complexity of the graph, E is the number of edges in the graph, N is the number of nodes in the graph, and P is the number of connected components in a graph [63]. Connected components refer to the number of subgraphs that exist that are not part of any larger graph. When considering a single element, it will only exist as part of a single connected component. This means that the value of P will be 1. Further, N will also always be 1, as we are dealing with a single node, that being the element itself. The only unknown is the number of edges leading into an element. This means the equation for cyclomatic complexity for a single element can be simplified to $M = E + 1$. In other words, the cyclomatic complexity for a single element in a graph is equal to the number of edges leading into the element plus 1.

As highlighted in Section 2.1.1, while the literature is abundant with metrics that can be applied to a system's structural and functional views, there are comparatively fewer metrics that can be applied to a system's behavioural view [22, 23]. To the best of our knowledge, a security metric tailored towards the measurement of critical elements present in a system's behavioural view that considers its cyclomatic complexity and threat participation does not currently exist. To achieve our goal of determining a system's behavioural security posture, we derived a novel security metric that can be applied to a system's behavioural view known as the Critical Element Risk Index (CERI) metric:

$$\text{CERI}_i = c_i \times \left(1 - \frac{m_i}{t_i}\right) \text{ for } t_i > 0 \tag{4.2}$$

where i is the critical element being measured, c_i is the cyclomatic complexity of the i^{th} critical element, t_i is the total number of threats involving the i^{th} critical element,

and m_i is the number of threats involving the i^{th} critical element that have been identified as mitigated.

CERI as a metric can be used to capture the risk of a critical element in a system. The higher the calculated CERI for a critical element i, the higher the risk. When mitigations are added to address threats a critical element participates in, the value of CERI will decrease. This decrease may be offset by an increase in cyclomatic complexity that comes with the introduction of mitigations, as cyclomatic complexity is also a source of risk contribution which we discussed above. We note that CERI is only defined for elements that participate in threats. In other words, CERI is only defined for critical elements. In a situation where all threats in which a critical element i participates are mitigated, its CERI will be equal to 0. Given the diverse range of threats a system element may be susceptible to, it is unlikely that every threat will be adequately mitigated. In situations where it is known that at least one threat a critical element participates in will remain unmitigated, designers may opt to set a target CERI that is close to the critical element's cyclomatic complexity. Minimizing the CERI of an element towards its cyclomatic complexity demonstrates that the risk the critical element poses on a system has been reduced, thus proving an overall improvement to the system's security level. In situations where designers can adequately mitigate all threats that a critical element participates in, a target CERI of 0 would be more appropriate.

To provide additional clarity on how the CERI metric works, we provide an example where the CERI of a critical element from an agnostic directed graph is calculated.

Figure 4.1 illustrates the initial case where a critical element in a graph participates in four unmitigated threats.

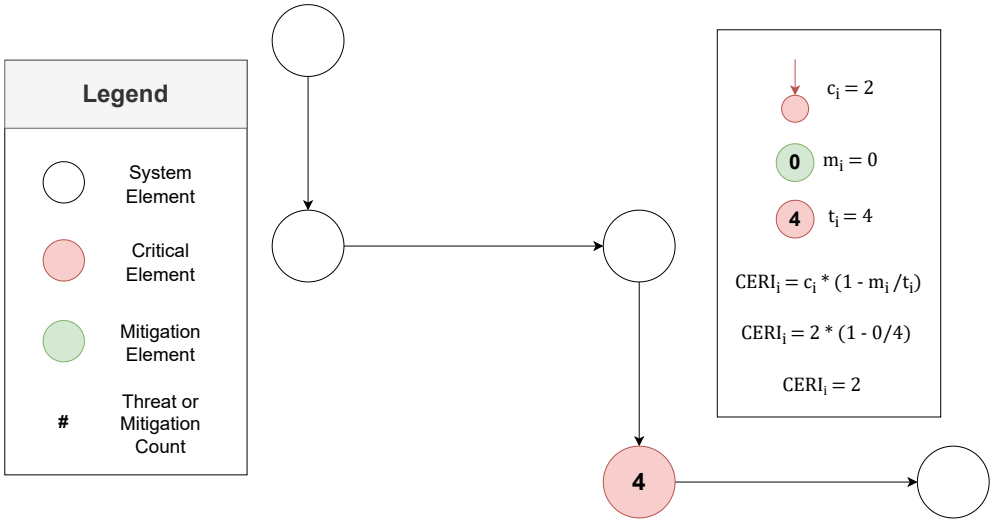


Figure 4.1: Calculating the CERI of a Critical Element With no Mitigations

The critical element (coloured red) in Figure 4.1 is currently susceptible to four threats. In this example, there are no mitigations in place to combat any of the four threats and there is only one edge leading into the critical element, so the cyclomatic complexity is 2. Calculating the CERI in this situation yields a result of 2. Now, we move to look at a revision of this example that includes mitigations, shown in Figure 4.2.

In the revised scenario, the graph has changed. A new mitigation element (coloured green) has been added. To successfully mitigate three of the four threats to which the critical element is susceptible, new nodes need to be added to the graph. For a grounded example, this structural change can arise for mitigations that introduce a decision or check into the system that results in two or more possibilities, as would be

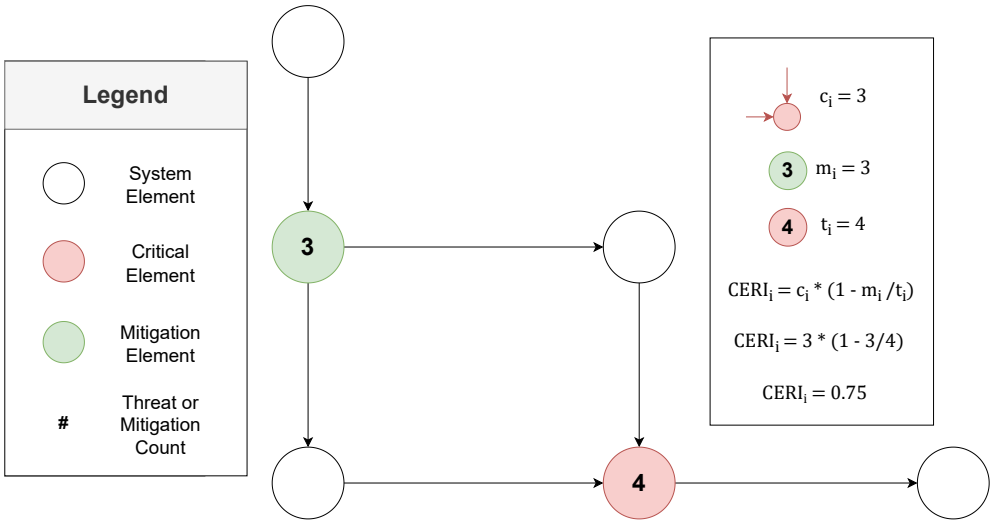


Figure 4.2: Calculating the CERI of a Critical Element With Mitigations

the case with a mitigation that checks for sensitive data and encrypts it to mitigate an information disclosure threat. In code, this would be equivalent to an if-else structure. If we were to recalculate the CERI of the critical element, accounting for the increase in cyclomatic complexity due to the addition of a new edge leading into the critical element, the resulting CERI is 0.75. This is a decrease compared to the initial calculated value of 2, representing an overall increase in system security through a risk reduction of the critical element.

Refocusing on our definition of a system's BSP, we use a system's average CERI to relay the risk that critical behavioural system elements pose in a system design. This information enables designers to harden system designs through the introduction of mitigations to reduce the calculated average CERI. This reduction indicates an increase in the security level of their system designs, which will promote the incorporation of security during the early phases of the SDLC. As CERI is a novel

security metric, we must ensure that the metric is suitable for security evaluation. To determine this, we can test the metric for soundness.

4.2.2 Evaluating CERI for Soundness

As described in Section 3.5, a security metric is sound if it satisfies the CSSM. We test the soundness of CERI by applying the three CSSM.

Well-Defined: CERI measures the security risk of critical system elements, as defined by its cyclomatic complexity and by the number of unmitigated threats in which it participates. It is meaningful since its value decreases as mitigations are introduced that address the threats a critical element participates in. It is objective since the security level of the system is informed by the remaining risks posed by these critical system elements. It is unbiased as its value varies only with the introduction of mitigations to address identified threats a critical element participates in and the critical element's cyclomatic complexity. Thus, this value cannot cannot be overstated or understated. Considering cyclomatic complexity accounts for potential changes to a critical element's security that arise from implementing mitigations. These mitigations can introduce new behavioural system elements that can modify the cyclomatic complexity of a critical element. In the event a mitigation results in an increase in a critical element's cyclomatic complexity, the addition of the mitigation can be evaluated to see if the CERI decreases despite the increase in cyclomatic complexity. This consideration ensures that the metric is effective at expressing a critical element's risk by considering potential small increases in risk associated with the implementation

of certain mitigations. No further considerations are needed, and the metric is complete. Threat and mitigation data can be obtained from various free resources. The detection of these threats and mitigations, as well as the determination of cyclomatic complexity, can be automated using preexisting cost-efficient algorithms, ensuring the values for the metric can be obtained without undue hardship or costs. Thus, the CERI metric satisfies all sub-conditions required to be considered well-defined.

Progressive: Designers can define an acceptable average CERI they are comfortable accepting. By implementing mitigations against identified threats that do not significantly increase the cyclomatic complexity of critical elements, the average CERI will decrease. If enough mitigations are implemented over time to address threats, the value of the average CERI will converge to that designer-defined acceptable value. Due to the metric being capable of converging to an acceptable average CERI as defined by a system designer, the metric satisfies the progressive condition.

Reproducible: While a designer may define different threats and mitigations to check for in a system compared to other designers, the results obtained will be similar. Although it is unlikely the results are the same, so long as the proportion of threats to mitigations is roughly equal the evaluation of CERI for a critical element will be close in final value. This is supported by the fact that the cyclomatic complexity of an element is a defined value with a fixed method of determination, leading to the only ambiguity being in the value of the fraction $\frac{m_i}{t_i}$, which will be a value in the range of (0, 1]. Given this, the metric is weakly reproducible, which satisfies the reproducible condition.

As all CSSM are satisfied, CERI is a sound security metric. While the CERI metric was derived to apply to a system's behavioural view, so long as the necessary information can be captured the metric can be applied to other system views. In our soundness evaluation, we analyzed the metric in the context of its application on a system's behavioural view. The definitions we provided are complete for this context, but they may need to be modified to be applicable in other situations.

4.3 A Security Metric for Data Flows

In this section, we present a security metric that represents a system's susceptibility to data corruption that is used to inform a system's behavioural security posture.

4.3.1 Corruption Propagation Potential (CPP)

As previously discussed, a system's behaviour is defined by two major aspects. The first aspect involves the elements that exist within the system, and the second involves the control and data flows that move between system elements to accomplish a desired use case. In Section 4.2.1 we defined a metric that addresses the critical elements that enable the flow of data and control. In this section, we turn our focus to the flows that contribute to a system's behaviour.

A major difference between a system's structural view and its behavioural view relates to flows. A system's structural view gives the structure of the system elements through the representation of the relationships between elements. The system's behavioural

view is informed by a system's structural view to enable the representation of data and control flows moving through those previously defined structural system elements to accomplish a desired use case. In other words, the structural view links elements together, but there is no ability to represent the temporal progression of data or control through those elements. This progression is the major defining factor when it comes to a system's behaviour and due to its importance to a system's behavioural view, a system's BSP must contain a metric that captures the security level of these flows.

As we defined in Section 4.1, flows represent both data and control progression in a system. To accomplish a desired use case, the flows moving through a system must be deliberate. The data and control are expected to move in a certain and defined way through a system and this expected behaviour is captured in a system's behavioural view. Unfortunately, this expected progression can be abused by bad actors. Through the introduction of corrupted information, attackers can hijack the expected flows of a system to force the system into an undesired or potentially undefined state. This situation leads to a variety of damage vectors the bad actors can explore, which can result in negative impacts in terms of a system's confidentiality, integrity, or availability. This flow-based corruption threat is not captured by specific element patterns like was the case with our definition of CERI, but rather by the existence of flows throughout the entirety of a system. More succinctly, we cannot rely on the same techniques or methods used in our definition of the CERI metric to evaluate how data corruption can impact a system.

To measure the impact corruption has on a system while levering the information present in a system's behavioural view, we can consider its propagation potential. Corruption propagation has been explored in the past by Chowdhury et al. [64] in a system security context through a security metric known as coupling corruption propagation. Using the metric, corruption propagation could be determined using information from a system's functional view. A higher corruption propagation potential was shown to be linked to a decrease in overall system security [64]. Using the coupling corruption propagation security metric as inspiration, we present a revised security metric that is designed to measure the potential of corruption through a system's flows known as Corruption Propagation Potential (CPP).

$$\text{CPP} = \frac{\sum_{i=1}^{n} l_i}{n} \tag{4.3}$$

where i is the flow being evaluated, n is the total number of flows in a system, and l_i is the length of a flow i through which corruption can propagate.

In contrast to the coupling corruption propagation metric defined in [64] where corruption propagation was determined through coupled function calls, CPP analyzes the length of corruptible flows present in a system's behavioural view. We define *corruptible flows* as any flow present in a system that does not have a mitigation in place to address corruption-related attacks. If such a mitigation exists, the length of the corruptible flow is defined as ending at that mitigation. Subsequently, a new corruptible flow begins following the mitigation, as this new flow can be exploited by bad actors through insider attacks or other nefarious methods to propagate corruption through a system while bypassing the placed mitigation.

To clarify how the length of a corruptible flow i is determined, we walk through an example scenario using an agnostic directed graph to represent a system's behavioural view. We begin with a case where no mitigations that deal with corruption-based attacks are present in a system, presented in Figure 4.3.

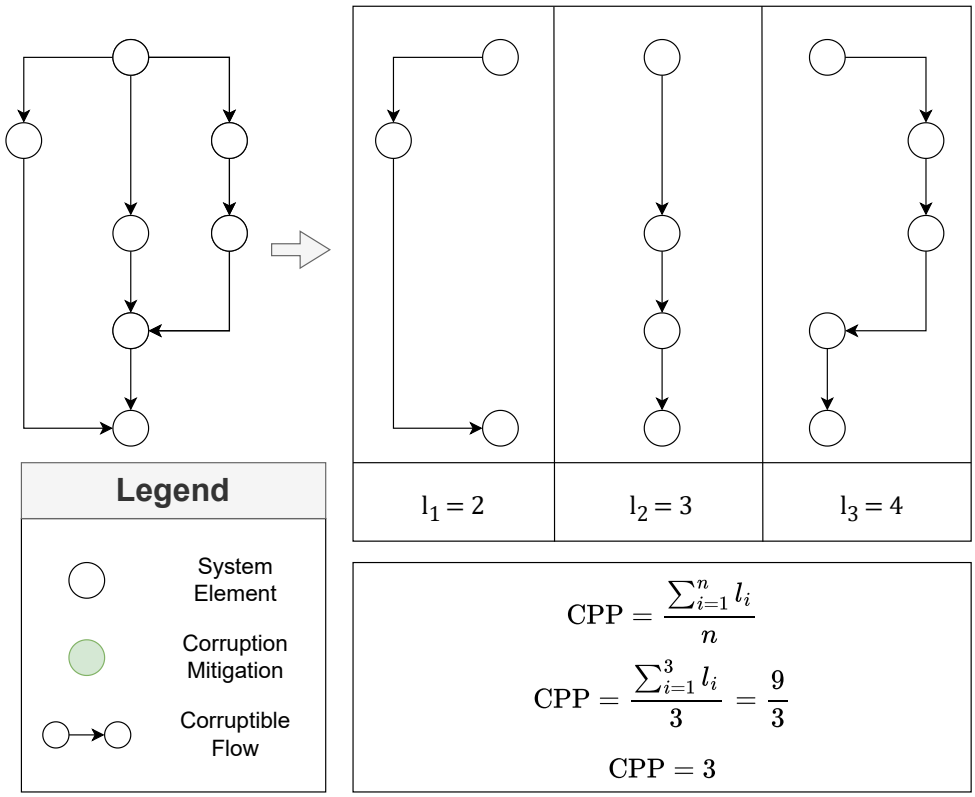


Figure 4.3: Calculating the CPP of a System With no Mitigations

In this example, the directed graph contains three flows that originate from the first node and terminate at the same end node. To determine the length of a flow i, we first isolate the flow. Once this is done, determining the length of the flow follows the same procedure used in graph theory to determine the length of a path [65]. Simply,

we count the number of edges that appear. From Figure 4.3 we can see that the length of the first flow $l_1 = 2$ because there are two edges present. This procedure is repeated for each isolated flow to enable the determination of the system's CPP. In this case, the CPP of the system when no mitigations are present is 3. Next, we consider a transformation where a mitigation is incorporated into one of the flows, illustrated in Figure 4.4.

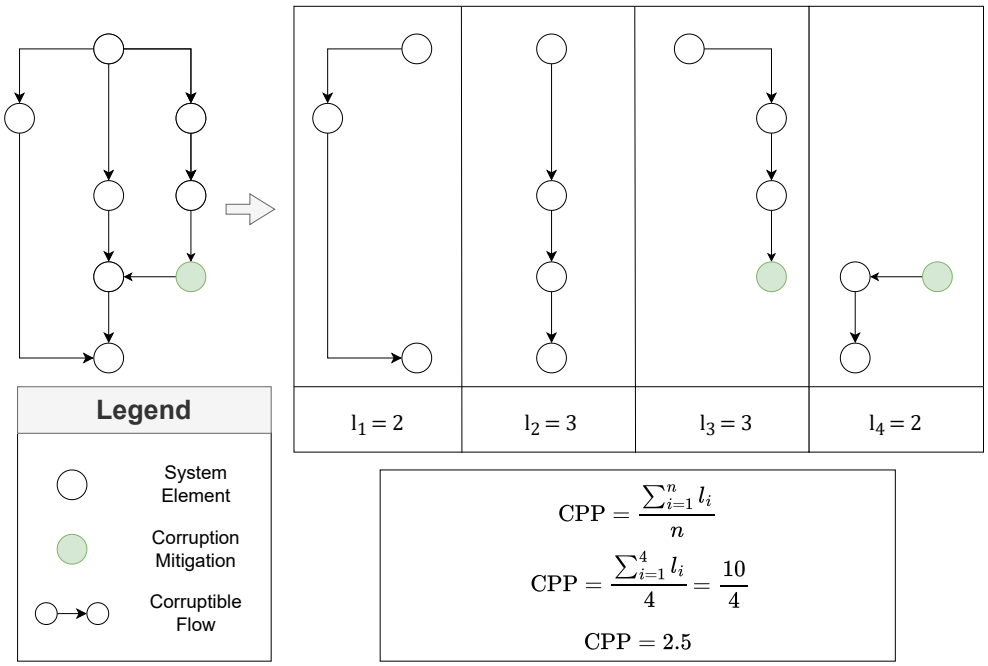


Figure 4.4: Calculating the CPP of a System With a Mitigation

In this revised scenario, the previous flow $i = 3$ highlighted in Figure 4.3 has been split into two smaller flows where $l_3 = 3$ and $l_4 = 2$ respectively. This split is the result of a mitigation that deals with corruption-based attacks being placed in the original $i = 3$ flow, which we defined as an endpoint to a flow in our definition of the CPP metric. Further, the mitigation also acts as the start of a new flow due to the potential for

corruption to be injected by bad actors directly after the mitigation through different advanced attack techniques. The creation of two smaller flows results in an increase in the total number of flows in a system, which in turn increases the value of n in the calculation of a system's CPP. Splitting a flow into two smaller flows also introduces an additional edge into the system, which can be seen when comparing the length of flow $i = 3$ in Figure 4.3, where the length has a value of 4, to the lengths of the two smaller flows $i = 3$ and $i = 4$ in 4.4, where the sum of the lengths has a value of 5.

The creation of two smaller flows will always result in a constant k being added to the numerator and denominator of the metric, which will reduce the metric closer to the value of 1 as long as the sum of flow lengths is greater than the total number of flows in a system [66]. The sum of the flow lengths will be greater than the total number of flows in a system so long as there is at least one flow with an intermediary node. If a system were composed of a single flow of length 1, the calculated CPP of that system would be 1. A case where CPP = 1 does not imply that the system is completely protected against corruption-based attacks, but rather that the introduction of mitigations will not meaningfully increase a system's security level as the mitigation results in the creation of a new flow of equal length as the original flow. In other words, the mitigation does not reduce the corruption potential of the original system as a flow of length 1 cannot be split to be smaller than 1, and mitigations would only create more flows of length 1.

When determining a system's CPP, we only consider flows with a length greater than 1 or what we deem as *meaningful flows*. We argue that flows without intermediary nodes are not capable of performing meaningful system interaction where corruption

would be paramount to prevent and do not contribute towards the definition of a system's behaviour as the flow is defined exclusively by a start and end point with no interaction of system elements, which is required to define a system's behaviour.

In the example shown in Figure 4.4, the value of k is 1. If 2 mitigations were added, the value of k would be 2. In other words, the value of k will be the same as the number of mitigations implemented, which is also the same as the number of new flows added due to the mitigations. Thus, k is added to both the numerator and denominator of CPP. Recalculating the CPP for the system yields a value of 2.5, which is a decrease compared to the result of 3 in the system where no mitigations were present. Through the introduction of a mitigation that deals with corruption-based attacks, the system has been hardened against corruption attacks, resulting in a lower CPP and an increase in the system's security level.

The value of a system's CPP is used to represent the overall security of the flows present in a system's behavioural view. A high CPP indicates a high potential for corruption propagation, implying a low system security level. The lower the CPP, the more hardened a system's flows are against corruption-related attacks. Designers can set a target CPP reflective of the maximum length of corruption they would be comfortable accepting for their system in situations where a limited number of mitigations that target corruption-related threats can be implemented. Using this metric as part of system's BSP alongside the previously defined CERI metric relays vital security information regarding the main factors that define a system's behaviour. Now that we have defined CPP and explained why it would be a good fit to inform a system's BSP, we must demonstrate that it is sound.

4.3.2 Evaluating CPP for Soundness

As we did with CERI, we test the soundness of CPP by applying the three CSSM.

Well-Defined: CPP measures the security of an asset, namely the system itself, by reflecting the number of mitigations that exist to address corruption-based threats. It is meaningful since its value decreases as the number of mitigations increases, with the understanding that a lower CPP equates to a higher system security level. It is objective since it directly reflects changes in the security level due to changes in the number of mitigations. It is unbiased since its value varies only with the number of mitigations present across a system's flows and therefore cannot be overstated or understated. The purpose of the CPP metric is to capture the potential for corruption propagation in a system, and this is expressed through the summation of the corruptible flow lengths present in the system. As the metric accomplishes its purpose with the values presented in its definition, the metric is complete. The values needed for the CPP metric can also be gathered at a low cost, as determining the length of a flow in a system can be efficiently computed using preexisting algorithms. Thus, the CPP metric satisfies all sub-conditions required to be considered well-defined.

Progressive: The introduction of mitigations to harden a system against the propagation of corruption is captured as a reduction in the calculated CPP value of the system. Any time a mitigation is introduced, the sum of the flow lengths for the system is increased by 1, as shown in Figures 4.3 and 4.4. This same constant increase is observed in the value of n as the mitigation splits a corruptible flow into two smaller corruptible flows. Adding the same constant value to both the numerator and denominator results in a reduction of CPP in all meaningful analysis cases where systems

have at least one flow length that is greater than 1, which are the only flows considered when calculating a system's CPP [66]. As more mitigations are introduced, the average will continue to decrease and progress towards a value of 1. Designers can set a goal for the CPP of their system and will observe that the value will converge towards that set goal as more mitigations are introduced, representing an increase in system security. Due to the metric being capable of converging to an acceptable CPP as defined by a designer, the metric satisfies the progressive condition.

Reproducible: The calculation of the metric depends on the placement of mitigations to reduce the corruption potential within a system. Different designers may select different locations or different mitigation strategies which would result in varying CPP values. Regardless, the introduction of mitigations results in a reduction of CPP, improving system security. Through optimizations to maximize the number of flows impacted by a single mitigation, which would be done as a cost-saving method where multiple mitigations that target corruption-based attacks could not be implemented given a system's budget, designers are likely to select very similar locations for mitigations. This would result in similar but not necessarily the same CPP values for a system. Before the introduction of any mitigations, the metric will be strongly reproducible as the determination of corruptible paths will equate to all existing paths within the system per our definition of the metric. However, placing a mitigation to reduce the corruption potential downgrades the metric to being weakly reproducible. As such, the metric is weakly reproducible, which is enough to satisfy the reproducible condition.

As all CSSM are satisfied, CPP is a sound security metric. Like with CERI, CPP was derived to be calculated using information from a system's behavioural view and our soundness evaluation reflects this fact. It may be possible to apply CPP in different contexts, but the justifications in our soundness evaluation of CPP may need to be revised to ensure the metric remains sound. Now that we have presented our definition of a system's BSP, alongside the two security metrics that will be used to inform it, we move to discuss our approach employed to determine a system's BSP.

4.4 Overview of an Approach to Determine a System's BSP

To determine a system's BSP, we must analyze a behavioural system representation that captures the information necessary to calculate CERI and CPP. The UML is a highly regarded standard language that can be used to model systems [1]. As we highlighted in Section 1.1, UML diagrams are divided into two categories, structural diagrams and behavioural diagrams. As the behavioural diagrams are used to represent a system's behavioural view, it follows that a behavioural UML diagram will capture the information necessary to represent a system's behaviour. In short, a behavioural UML diagram will capture the elements and progressing flows between said elements that are used to define a system's behaviour.

So long as a UML diagram contains the behavioural system elements and the flows between these elements, it is possible to calculate the CERI and CPP using the information from the diagram. To promote the adoption of our approach, it is logical

to target behavioural diagrams that are already being created by designers as part of their activities during the SDLC. Two of the most popular behavioural diagrams according to a recent survey by Langer et al. [67] are sequence diagrams and activity diagrams.

Both sequence and activity diagrams contain the necessary information required to calculate the security metrics that are used to inform a system's BSP, so either diagram would be suitable as an analysis target. In our approach, we choose to analyze UML activity diagrams due to their ability to capture the information about a system's behaviour within their elements and flow, coupled with their popularity in both industry and research [67]. We note that our definition of a system's BSP is not directly coupled to activity diagrams and that it would be possible to adapt our approach to other suitable behavioural system representations if desired, but to limit the scope of this work we do not explore these possibilities.

To calculate a system's CERI and CPP, we need to be able to identify patterns indicative of threats and mitigations, as well as be able to calculate the lengths of all flows. Activity diagrams offer graphical representations of a system's behavioural view, but this graphical representation is not standardized. If our approach were to analyze the graphical representation of activity diagrams, we would require a method that analyzes the non-standardized graphical representations of activity diagrams generated by different modelling tools. This would necessitate the implementation of image recognition techniques, increasing the development effort associated with our approach.

Fortunately, the OMG provides a means to represent UML diagrams in a standardized text-based format known as XMI [68]. While not conceived as a means to enable security analysis, the XMI provides a consistent and standardized representation of UML diagrams that can be parsed without requiring advanced techniques such as image recognition. Coupled with the fact that major modelling tools support XMI export options for UML diagrams [69, 70, 71], our approach leverages the XMI to gather the information necessary to calculate a system's BSP.

Understanding that the medium for analysis will be UML activity diagrams exported to the XMI format, we can now discuss our high-level approach to determine a system's BSP. To best support designers towards the incorporation of security into early system designs, adequate tool support that can be used without extensive deviations from traditional design activities must be developed [8]. Any tool must also automate the reporting of pertinent information back to the designer such that they can easily modify their system designs to increase their system's security as demonstrated through an improvement in its calculated BSP. Further, our approach must ensure that the determination of a system's BSP can occur with minimal additional responsibilities from designers [8]. To accomplish these requirements, our approach uses a tool we have developed called Dubhe to automatically calculate a system's BSP using the XMI of an activity diagram. We describe the minimal responsibilities of designers and the responsibilities of Dubhe in Figure 4.5.

As seen in Figure 4.5, our approach requires one activity to be performed by system designers, and five activities to be performed by Dubhe. We provide descriptions for each of these activities below.

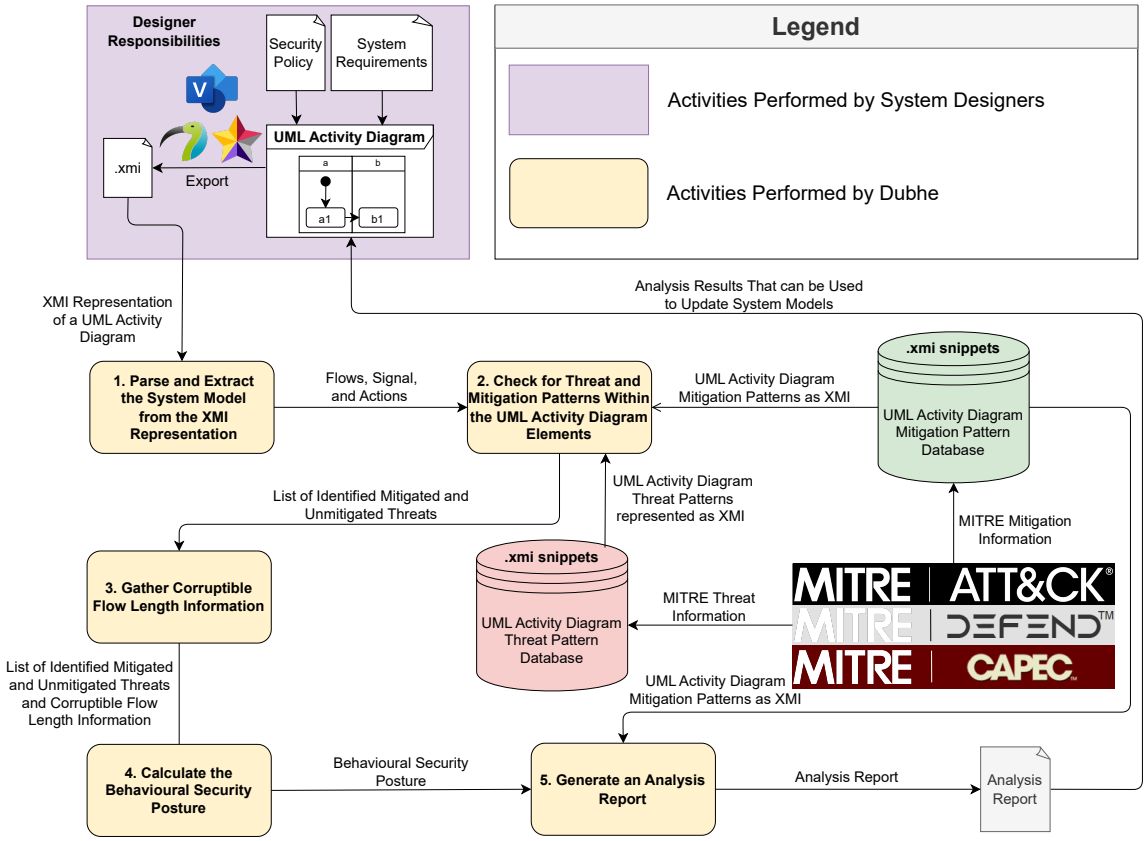


Figure 4.5: Outlining our Approach to Determine a System's BSP

Designer Responsibilities: Before any analysis can begin, system models must first be manually converted to their corresponding XMI representation. As we previously highlighted, this task is supported by popular modelling tools. While it would be ideal to support designers in submitting their unconverted model files directly and performing this XMI transformation as part of Dubhe, the variations in file types from these modelling tools make this a task that would require significant development. Instead, we offload this responsibility to the designers and ensure it is the only step they are required to perform to enable the determination of their system's BSP.

1 - Parse and Extract the System Model from the XMI Representation: After receiving the system model, Dubhe begins to parse the submitted XMI to ensure compliance with the latest specification. It then extracts the necessary data needed to perform pattern matching, the technique we utilize to identify threat and mitigation patterns necessary to calculate a system's average CERI. In this case, this extraction is performed through XMI parsing to gather information related to the elements highlighted in Figure 3.2. This is then stored in an ActivityElement object, a custom data type created to enable the determination of a system's BSP. This data type, along with the parsing logic, will be discussed in more detail in Chapter 5. Following the completion of this step, Dubhe uses the extracted data to begin pattern matching against a repository of threat patterns and mitigation patterns.

2 - Check for Threat and Mitigation Patterns Within the UML Activity Diagram Elements: Using the external MITRE-hosted data sources known as ATT&CK, D3FEND, and CAPEC (see Section 3.2), we created XMI patterns that can be matched against the submitted activity diagram. These data sources are well-respected and provide the data necessary to generate appropriate threat and mitigation patterns. These patterns are used by Dubhe to identify threats and to determine whether or not mitigations against those threats are present and appropriately placed within a designer's submitted activity diagram. Information regarding the detection of these patterns is passed along to enable the calculation of the CERI metric in a future step performed by Dubhe. We outline these created threat and mitigation patterns and the technical details behind the pattern matching performed by Dubhe in Chapter 5.

3 - Gather Corruptible Flow Length Information: Following the detection of threat and mitigation patterns, Dubhe will begin to analyze the flows present in the submitted activity diagram. Our tool will determine the length for each corruptible flow present in a system while checking for mitigations that may break up a corruptible flow, as we described in Section 4.3.1. The algorithms used to perform this determination are described in Chapter 6. Once all flows have been analyzed, the collection of corruptible path lengths and the previously collected set of mitigated and unmitigated threats are passed along to enable the complete determination of a system's BSP.

4 - Calculate the Behavioural Security Posture: Using the previously collected data, Dubhe performs the calculations necessary to determine the values for the CERI and CPP security metrics that make up a system's BSP. These calculations may be performed multiple times depending on the information received from previous steps to generate worst-case and best-case representations of a system's BSP, which we explore in later chapters. The final calculated values for the metrics are passed along to the next step to generate a comprehensive analysis report.

5 - Generate an Analysis Report: Dubhe uses the system's calculated BSP to generate an analysis report. This report includes the values of the two calculated security metrics and provides a list of recommendations based on mitigation patterns for identified unmitigated threats. These suggestions help designers enhance the security of their behavioural system designs by targeting reductions in the values of the calculated CERI and CPP security metrics. This information can then be used by designers to modify their initial system designs, at which point they can regenerate

the XMI representation of their activity diagram to determine if their changes have made their systems more or less secure based on the newly calculated BSP.

This high-level overview of our approach demonstrates the steps required to determine a system's BSP, showing the inputs and outputs of each step and demonstrating the actors responsible for each step. We have attempted to minimize the responsibilities that are required to be undertaken by designers by pushing the majority of responsibilities onto our tool named Dubhe. As we have mentioned above, we break down the technical details of Dubhe's operation concerning this approach in the subsequent chapters.

4.5 Chapter Summary

As seen in this chapter, we presented and broke down our definition of a system's behavioural security posture. We introduced two sound security metrics, CERI and CPP that will be used to express a system's behavioural security posture. Finally, we looked at a high-level overview of the process that will be used to determine a system's behavioural security posture using the XMI representation of UML activity diagrams, alongside the threat landscape and accompanying mitigation strategies retrieved from MITRE ATT&CK, D3FEND, and CAPEC. In the next chapters, we will break down our approach to explain on a technical level how the information required to calculate a system's behavioural security posture is gathered from the XMI representation of UML activity diagrams.

Chapter 5

Identifying Threats and Mitigations in UML Activity Diagrams With Pattern Matching

In this chapter, we outline an approach that leverages pattern matching to identify threat patterns and associated mitigation patterns within the XMI representations of UML activity diagrams. Section 5.1 presents the steps needed to calculate a system's average CERI. Section 5.2 describes how XMI data is transformed to enable security analysis. Section 5.3 presents both the threat and mitigation patterns for six representative STRIDE threats. Section 5.4 highlights the process that our BSP analysis tool Dubhe employs to detect threat and mitigation patterns within a designer's submitted UML activity diagram. Section 5.5 explains how the cyclomatic complexity of critical elements is determined. Section 5.6 shows how the information

that is gathered through our methodology can be used to calculate a system's average CERI. Finally, Section 5.7 summarizes the processes and information presented throughout this chapter.

5.1 An Approach to Resolve the Average CERI of Systems

As we presented in Section 4.2.1, to determine a system's BSP we must be able to calculate the average CERI, which is informed by the CERI values of all critical elements. Critical elements are the system elements that participate in threats, so this implies that an approach to identify the existence of threats in a system's behavioural view must be employed to determine a system's BSP. However, simply identifying the threats in which a critical element participates is not enough, as CERI also depends on whether the threats a critical element participates in have been properly mitigated. Thus, our approach must also detect associated mitigations for identified threats that exist within a system. Lastly, as the CERI metric is dependent on a critical element's cyclomatic complexity, this must also be determined in our approach. If these values are gathered for every critical element that exists within a system, calculating the average CERI to inform a system's BSP can be accomplished.

In Section 4.4, we detailed a high-level overview of our approach to determine a system's BSP. To better support designers in their efforts to incorporate security during the early phases of the SDLC, we developed a tool called Dubhe to perform the tasks required to calculate a system's BSP for a given UML activity diagram.

In this chapter, we provide a technical breakdown of the first two steps listed in Figure 4.5 that are performed by Dubhe, alongside a brief exploration of the fourth step that will be further explored in Chapter 6. A more detailed overview of these steps is shown in Figure 5.1.

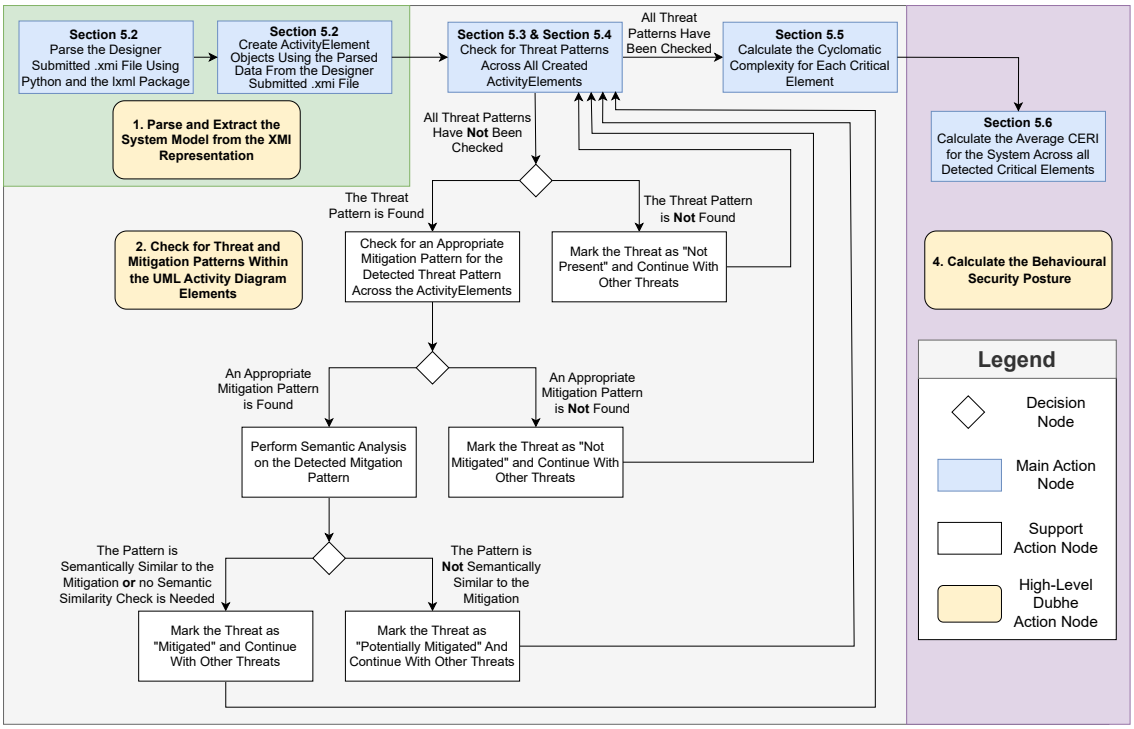


Figure 5.1: The Steps Needed to Calculate a System's Average CERI

In the subsequent sections, we explain the main actions performed by Dubhe, coloured in blue in Figure 5.1. The lower-level actions, coloured in yellow, are mapped to the high-level steps previously presented in Figure 4.5. The specific section each main action refers to is included at the top of each main action respectively. The support actions, coloured in white in Figure 5.1, will also be discussed as part of the explanation of the main action they interact with. Having presented an overview of the steps required to determine a system's average CERI, we move to an explanation

of how Dubhe parses and stores an XMI representation of a submitted UML activity diagram to perform security analysis.

5.2 Transforming XMI Data to Enable Security Analysis

As previously mentioned in Section 4.4, our approach leverages the XMI format to enable the determination of a system's BSP. However, as the XMI format was not conceived to enable security analysis, we must first parse and transform the data into a medium that allows for security analysis. Fortunately, parsing XMI data is simple, as the syntax adheres to the syntax of XML. To develop Dubhe, we decided to utilize the Python programming language and the existing `lxml` package to parse designer-submitted XMI files [72]. Python provides a plethora of support for managing and manipulating data and acts as a suitable choice of programming language for the development of Dubhe. We demonstrate the workflow that Dubhe follows when receiving, parsing, and transforming a designer submitted XMI file in Figure 5.2.

There are three main steps required to transform XMI data into a format suitable for determining a system's BSP. Naturally, the first step is receiving an XMI file that captures a designer's behavioural system model. Dubhe provides support for designers to select their XMI file using their operating system's default file explorer. Designers also have the option to drag and drop their XMI file into Dubhe's Graphical User Interface (GUI), which takes the form of a web application. Upon receipt of a file

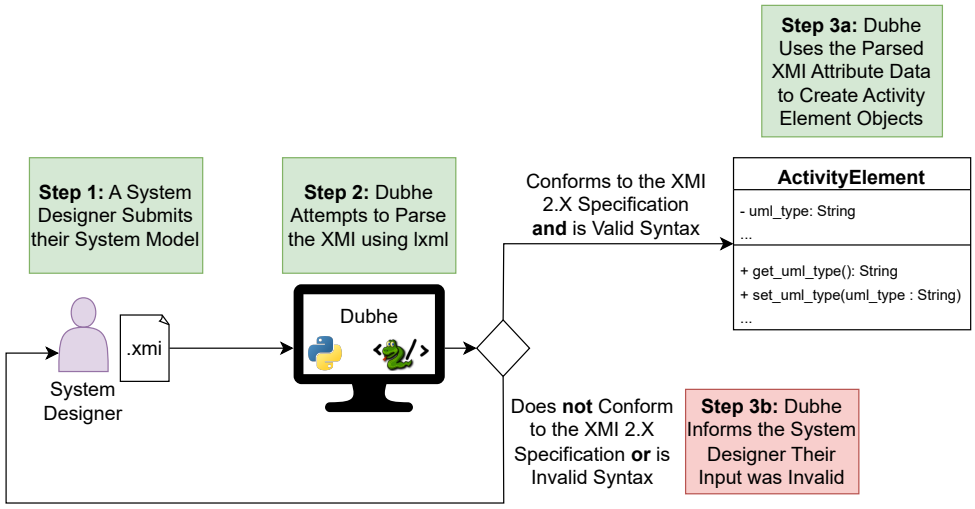


Figure 5.2: Processing Designer Submitted XMI Files Using Dubhe

with a valid XMI file extension, Dubhe proceeds to the next step listed in Figure 5.2 and attempts to parse the file.

Using the `lxml` Python package [72], Dubhe parses the raw XMI from the designer-submitted file. If a parsing error occurs due to incorrect syntax within the submitted file, Step 3b from Figure 5.2 is performed and the designer is informed that their file contains invalid syntax. If no parsing error, Dubhe will perform a basic check across the extracted attributes parsed from the XMI file. This check compares the names of the extracted attributes to a list of attributes that are present within the XMI specification for UML activity diagrams [68]. The specific list of names that are checked is captured in Figure 3.2. If any attributes are parsed that do not appear on the list of valid attributes, Dubhe executes Step 3b and informs the designer that their submitted file does not conform to the latest XMI 2.X specification.

If the file is successfully parsed and the attributes are valid per the XMI 2.X specification, Dubhe proceeds with Step 3a and begins generating ActivityElement objects for each element and flow that existed in the original UML activity diagram before its conversion to XMI. ActivityElement objects are a custom datatype that encompasses an XMI ID, XMI element type, the labelled name for the element (if it exists), the name of the parent element (if it exists), and two lists of XMI IDs corresponding to the elements connected by incoming flows or element and the elements connected by outgoing flows or elements. This data is stored as strings.

Generating ActivityElement objects for every element and flow effectively captures the submitted UML activity diagram while providing a consistent data representation that can be used for security analysis. As we maintain the connections and element hierarchy present in the activity diagram, it is possible to traverse any flow from the original activity diagram, all the while having access to data analysis and manipulation options offered through Python. The added flexibility that comes from Python is especially helpful when it comes time to check for threats across the created ActivityElement objects through pattern matching.

5.3 Representing Threats and Mitigations as XMI Patterns

Before discussing how Dubhe performs pattern matching to identify threats and mitigations, we must first go over the patterns. Over the following subsections, we present

these threat and mitigation patterns, demonstrated using the notation of UML activity diagrams. These pattern overviews include a description of each threat and mitigation pattern, the sources of information that were used to inform the creation of the patterns, and the actual patterns themselves. Before diving into the patterns, we must first briefly touch upon the rationale behind our threat and mitigation pattern selection.

As we've previously discussed in Section 1.2, systems can contain a wide range of threats. These threats can be realized by bad actors to target the confidentiality, integrity, or availability of systems. In our approach, we leverage representative threats for each category of STRIDE to provide a suitable breadth of threats for our security analysis. We selected to use representative threats from STRIDE and not other threat classification schemes due to its popularity and reputation in both academic and industry settings [73, 74]. In total, we created six threat and mitigation pattern pairs, one for each category of STRIDE.

While it would be valuable to consider more threats in our approach, we decided to focus on a threat from each category of STRIDE to maintain a reasonable scope for this work. We have instead focused efforts on building a system that performs pattern matching in a manner that can be expanded to incorporate new patterns in the future, which we describe in Section 5.4. This allows our work to evolve to include newly discovered threats while allowing us to demonstrate the capabilities of Dubhe using a limited but sufficiently diverse set of pattern pairs. The threat and mitigation

patterns presented in the following subsection were sourced from reputable information available in three MITRE data sources (CAPEC, ATT&CK, and D3FEND), and were adapted to fit the context of UML activity diagrams [53, 54, 55].

5.3.1 Spoofing Threat and Mitigation Patterns

Spoofing threats can largely be classified as instances where a bad actor impersonates another user, organization, or device. There are different categories of spoofing threats, including identity spoofing, web spoofing, and Internet Protocol (IP) spoofing. For our choice of a representative spoofing threat, we focus on IP spoofing, informed using information from both MITRE CAPEC (CAPEC-151) and ATT&CK (T1078) entries [75, 76].

Consider a scenario where a bad actor wishes to redirect the mail of a victim to their address. The victim's address is managed by an online system that authenticates account changes using the IP address of the connection. If a bad actor were successfully able to spoof the IP address of a victim in such a system, they would have complete freedom to make changes to the victim's account information. This example scenario is captured in Figure 5.3. The vulnerable elements that constitute the detection pattern are coloured in light red in Figure 5.3. The actions performed by the bad actor are coloured in bright red. The threat present in this scenario is that the server is only performing authentication based on a connection's IP address, which can be easily spoofed by bad actors with existing security tools. The specific critical elements that make up the threat detection pattern are the "AcceptEventAction" element, the "OpaqueAction" element that initiates an account modification,

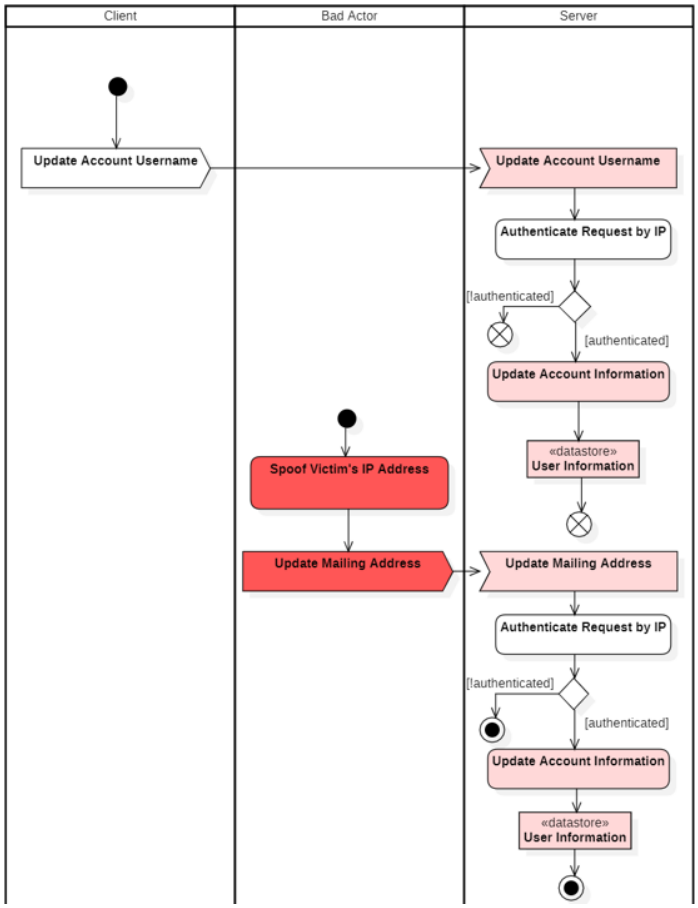


Figure 5.3: A Bad Actor Exploiting the Spoofing Threat

and the "DataStoreNode" element containing the user's account information. There may be additional elements present between the "AcceptEventAction" element and the "DataStoreNode" element, but so long as the three mentioned elements appear in this order within a flow, the threat will exist within a system. Dubhe is capable of performing a check to account for a variable number of extraneous elements between elements that participate in a threat pattern, and we discuss the syntax that enables this feature in Section 5.4.

Any elements that take part in the detection pattern are marked as critical, in this case, that would be the "AcceptEventAction" element, the "OpaqueAction" element that initiates an account modification, and the "DataStoreNode" element containing the user's account information. If extraneous elements are placed in between threat pattern elements, they are not considered critical. For example, the "DecisionNode" structure that contributes to the single-factor IP address authentication in this example is not a member of the threat pattern, as its existence is not what causes the threat to be present in a system. Rather, the "DecisionNode" structure contributes to a mitigation pattern against this representative spoofing threat, but it would not be a complete mitigation as it only considers a single factor for authentication. All marked critical elements must have their CERI values calculated to enable the calculation of a system's average CERI, which is used to inform its BSP.

The system should not rely on a single form of authentication to confirm the correct user is submitting changes to their account information, especially one as insecure as authentication by IP address. The growing standard is to use multi-factor authentication schemes to suitably protect against a variety of threats, including the IP spoofing example demonstrated in Figure 5.3. In other words, the system must include a multi-factor authentication structure to mitigate the presented representative spoofing threat. The information that was used to determine this mitigation was gathered from entries listed in both MITRE ATT&CK (M1036) and D3FEND (D3-MFA) [77, 78]. An implementation of this described mitigation using the UML activity diagram syntax is shown in Figure 5.4.

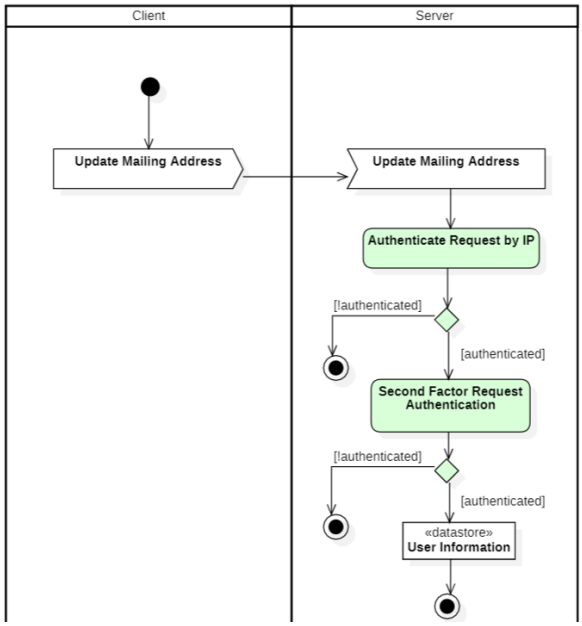


Figure 5.4: A UML Activity Diagram Hardened Against a Spoofing Threat

The elements that participate in the mitigation pattern are coloured in light green. The specific elements that constitute the mitigation pattern are the two "DecisionNode" authentication structures. While the first factor is still the unchanged IP address authentication scheme from the presented example, designers have the freedom to select authentication mechanisms that best suit their system needs. So long as multiple authentication factors are detected in a system, the mitigation pattern will be marked as valid by Dubhe.

If the bad actor were to reattempt the scenario presented in Figure 5.5 with the addition of this mitigation, the attempt to update the victim's mailing address would fail, as the second authentication factor would not be bypassed by the bad actor's IP spoofing attempt. Thus, the mitigation pattern successfully prevents the described representative spoofing threat, allowing the system to reach a higher security level

through the reduction of its average CERI due to the inclusion of an appropriate mitigation.

5.3.2 Tampering Threat and Mitigation Patterns

In STRIDE, it is typically understood that tampering threats deal with the unauthorized alteration of data, threatening a system's integrity. These threats can be realized on the data being passed between system elements, data stored in databases within a system, and other possible system asset targets. For our choice of a representative tampering threat, we focus on indicator removal, informed using information from both MITRE CAPEC (CAPEC-268) and ATT&CK (T1070) entries [79, 80].

Consider a scenario where a bad actor has obtained the username and password of a victim and wishes to confirm if they can successfully login using the credentials. To avoid possible repercussions, the bad actor wishes to remove any trace of this login attempt from the system such that the attempt cannot be traced back to them. This example scenario is captured in Figure 5.5.

The vulnerable elements that constitute the detection pattern are coloured in light red in Figure 5.5. The actions performed by the bad actor are coloured in bright red. The threat present in this example is that the server is not performing proper data sanitization on received signals from a client before it interacts with its internal system log. The specific critical elements that make up the threat detection pattern are the "AcceptEventAction" element, the "OpaqueAction" element, and the

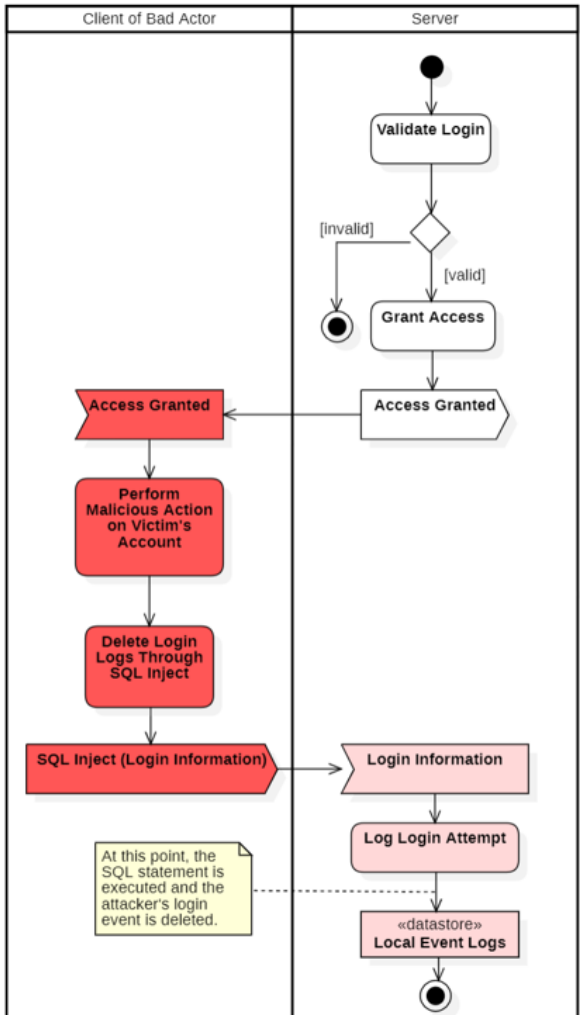


Figure 5.5: A Bad Actor Exploiting the Tampering Threat

"DataStoreNode" element. More elements may appear between the "AcceptEventAction" element and the "DataStoreNode" element without altering the presence of this tampering threat. As such, so long as the flow from the "AcceptEventAction" element eventually goes through an "OpaqueAction" element that deals with system logging and a "DataStoreNode" element that represents a system log, the threat will be detected within a system.

In the example scenario from Figure 5.5, the bad actor is seen logging into the victim's account. After this attempt is successful, they prepare and send a SQL statement designed to delete their login attempt from the local system log. To avoid such situations, the system should perform data sanitization on any received signal before it interacts with a "DataStoreNode" element. Logs should also be kept in multiple locations in the event the local log is corrupted or damaged. The information that was used to determine this mitigation was gathered from entries listed in both MITRE ATT&CK (M1029) and D3FEND (D3-DI) [81, 82]. An implementation of this described mitigation using the UML activity diagram syntax is shown in Figure 5.6.

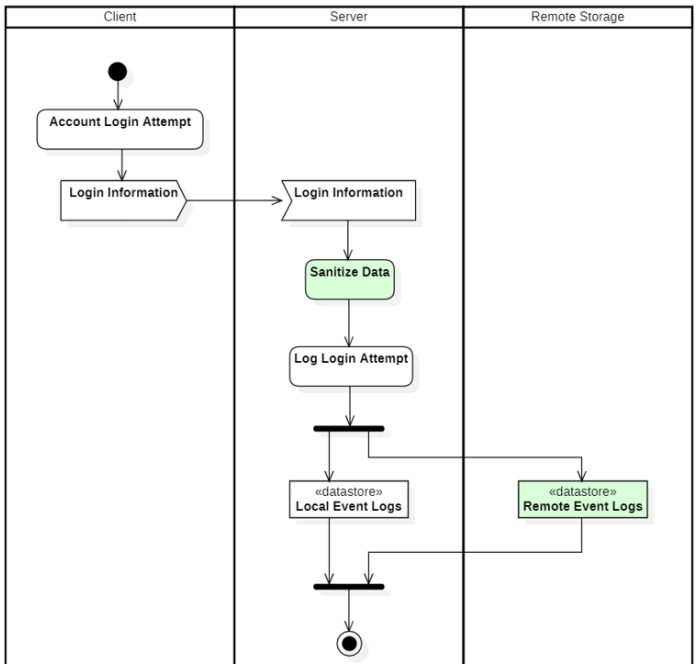


Figure 5.6: A UML Activity Diagram Hardened Against a Tampering Threat

The elements that participate in the mitigation pattern are coloured in light green. The specific elements that constitute the mitigation pattern are the "OpaqueAction"

element which should contain an appropriate name that refers to the sanitization of data, and a "ForkNode" and "JoinNode" structure that simultaneously stores system event information in a local and remote "DataStoreNode" that acts as a system log. Like with the detection pattern, extraneous elements may appear between elements that participate in the mitigation pattern, but if the pattern can be matched in the presented order, the mitigation will be considered valid.

If the bad actor were to reattempt the scenario presented in Figure 5.5 with the addition of this mitigation, the malicious SQL statement would be properly sanitized and the entry in the system log would not be deleted. If the bad actor pursued other actions, such as physically destroying the server where the log was located, the login attempt would still be recorded and accessible on the externally hosted data store. In short, the mitigation pattern successfully prevents the explained representative tampering threat, reducing the average CERI of the system, which in turn leads to a higher system security level.

5.3.3 Repudiation Threat and Mitigation Patterns

The term *repudiation* means to refute a statement or event. In the context of STRIDE, this can be more concretely defined as instances where an individual or system denies having performed an action or transaction. Such instances open up the possibility for fraud or other nefarious actions by bad actors. For our choice of a representative repudiation threat, we focus on insufficient system logging, informed using information from both MITRE CAPEC (CAPEC-196) and ATT&CK (T1134) entries [83, 84].

Consider a scenario where a bad actor purchases an item from an online store to commit fraud. They will make the purchase, await receipt of the item, and then contact the store's customer support and claim they never made such a purchase and their account must have been hacked. As such, they demand a refund for the "fraudulent" purchase. This scenario is captured using the syntax of UML activity diagrams in Figure 5.7.

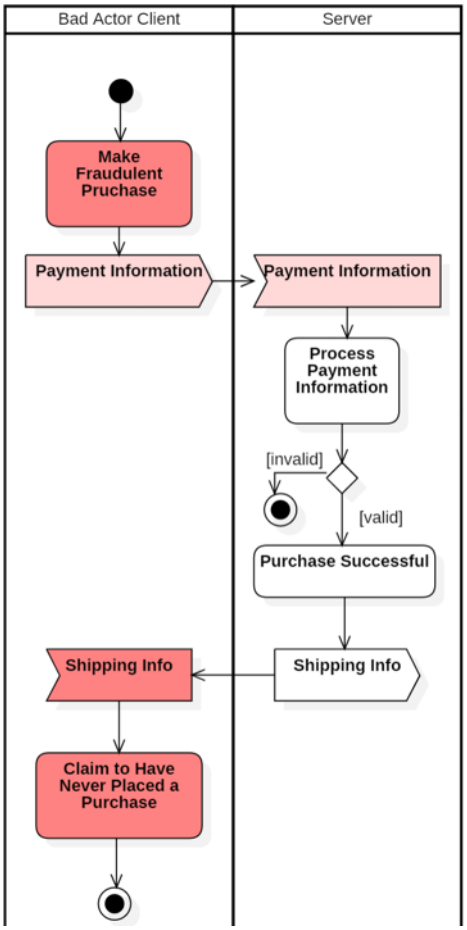


Figure 5.7: A Bad Actor Exploiting the Repudiation Threat

The vulnerable elements that constitute the detection pattern are coloured in light red in Figure 5.7. The actions performed by the bad actor are coloured in bright red. The threat present in this example is that the server has no mechanism for tracking signals received from users. The specific critical elements that constitute the threat detection pattern are the "SendSignalAction" element and the "AcceptEventAction" element. This means to determine a system's average CERI, the elements that were matched against this threat detection pattern would be considered critical and thus would need to have their CERI values calculated.

When the bad actor claims that their purchase was not made by them, the system has no way to refute this claim. If the system logs incoming signals, and these logs include information regarding the location of where a signal originates, system owners could refute the claim made by the bad actor. This could be done by demonstrating that the IP address of the "fraudulent" purchase matches the IP address that has been previously used by the bad actor for other purchases. The information that was used to determine this mitigation was gathered from entries listed in both MITRE ATT&CK (M1018) and D3FEND (D3-LAM) [85, 86]. An implementation of a suitable mitigation using the UML activity diagram syntax is shown in Figure 5.8.

The elements that participate in the mitigation pattern are coloured in light green. The specific elements that constitute the mitigation pattern are the "OpaqueAction" element which should contain an appropriate name that refers to performing a logging action and the "DataStoreNode" element which should have an appropriate name that indicates the data store acts as a log storage for the system. It should be

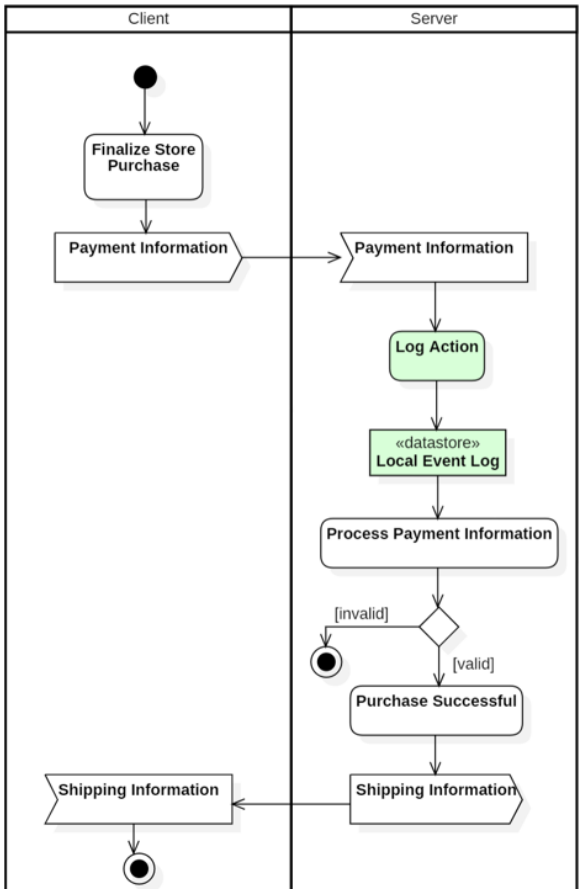


Figure 5.8: A UML Activity Diagram Hardened Against the Repudiation Threat

noted that there may be elements between the "OpaqueAction" element and the "DataStoreNode" element. If this is the case, so long as both elements appear in the system before the end of the flow, the mitigation will be valid. Dubhe is designed to account for this possibility when attempting to identify this mitigation pattern.

If the bad actor were to reattempt the scenario presented in Figure 5.7 with the addition of this mitigation, the system owners could reference the generated logs from the data store and refute the fraudulent transaction claim. In other words, the mitigation effectively addresses the presented representative repudiation threat,

leading to a more secure system through a reduction of the CERI values for critical elements that participate in the representative repudiation threat.

5.3.4 Information Disclosure Threat and Mitigation Patterns

Information disclosure threats allow for situations where bad actors are capable of intercepting, receiving, or gaining access to information that was not intended for their receipt. Depending on the sensitivity or value of the information, this can have significant ramifications on a system's confidentiality. For our choice of a representative information disclosure threat, we focus on the threat of an adversary-in-the-middle, informed using information from both MITRE CAPEC (CAPEC-94) and ATT&CK (T1557) entries [87, 88].

Consider a scenario where a bad actor sits between a client and a server for an online medical service provider system. This bad actor has gained the ability to intercept signals passed between the client and server. The client requests a copy of their medical history, and the system sends the information back to the client. Before it reaches the client, the bad actor intercepts and stores a copy of the information, before passing the information along to the client such that they remain unaware that their information has been disclosed to an authorized party. This scenario is captured using the syntax of UML activity diagrams in Figure 5.9.

The vulnerable elements that constitute the detection pattern are coloured in light red in Figure 5.9. The actions performed by the bad actor are coloured in bright red. In this scenario, the data was not encrypted before being sent to the client, allowing

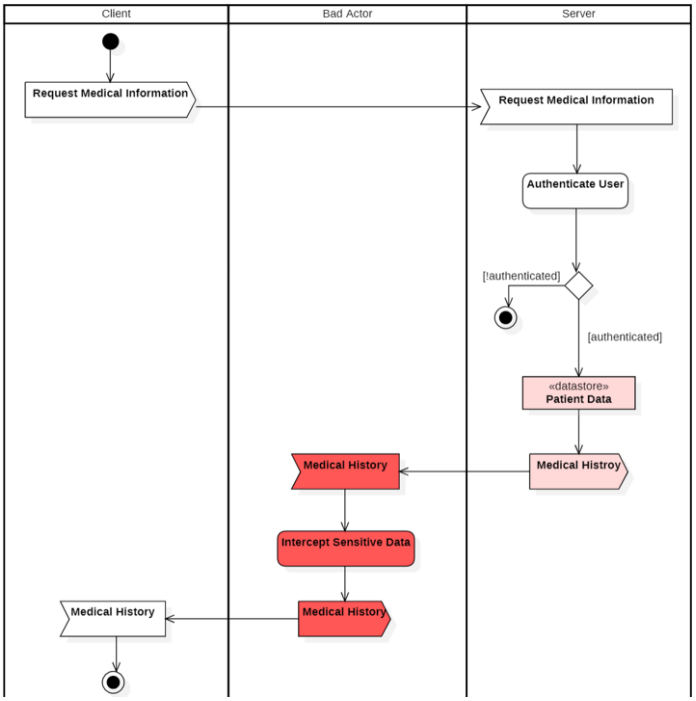


Figure 5.9: A Bad Actor Exploiting the Information Disclosure Threat

the bad actor to successfully intercept and read the sensitive information. The specific critical elements that constitute the threat detection pattern are the "DataStoreNode" element containing sensitive information and the "SendSignalAction" element. This means to determine a system's average CERI, the elements that were matched against this threat detection pattern would be considered critical and thus would need to have their CERI values calculated. Extraneous elements can appear between the "DataStoreNode" containing sensitive information and the "SendSignalAction" element, but the detection pattern will be identified as present so long as both elements appear within a flow.

There are a few ways to mitigate the threat of an adversary-in-the-middle. Since the system is dealing with sensitive information, a mitigation that should already

be in place deals with the encryption of said data before it travels over communication channels. The information that was used to determine this mitigation was gathered from entries listed in both MITRE ATT&CK (M1041) and D3FEND (D3-MENCR) [89, 90]. An implementation of an adequate mitigation using the UML activity diagram syntax is shown in Figure 5.10.

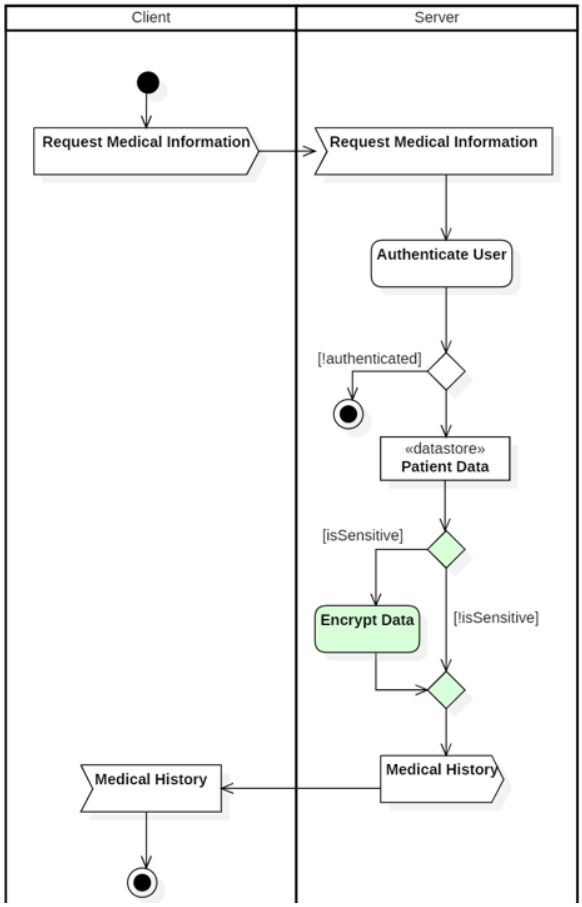


Figure 5.10: A UML Activity Diagram Hardened Against an Information Disclosure Threat

The elements that participate in the mitigation pattern are coloured in light green. The specific elements that constitute the mitigation pattern are the “DecisionNode”

and "MergeNode" data encryption structure, where one path proceeds without any modification to the data if it is not evaluated as sensitive, and the other path will encrypt the data through an "OpaqueAction" element if the data is sensitive. We leave the specific method of data encryption agnostic as it will vary from system to system. Returning to the example shown in Figure 5.9, even though the adversary can intercept the signal in transit the information will not be immediately understandable. The bad actor can attempt to break the encryption, but depending on the strength of the scheme employed by the medical system, this will likely require an amount of time that would surpass the value of the information. As such, the representative information disclosure threat has been successfully mitigated, resulting in a more secure system through a reduction in CERI values of critical system elements.

5.3.5 Denial-of-Service Threat and Mitigation Patterns

When discussing threats that fall under the category of Denial-of-Service (DoS), the main risk for a system having such threats realized is towards the system's availability. Such instances can cause serious financial harm to both system owners and users depending on the criticality of the system being targeted. For our choice of a representative denial-of-service threat, we focus on system resource exhaustion, informed using information from both MITRE CAPEC (CAPEC-125) and ATT&CK (T0814) entries [91, 92].

Consider a scenario where a bad actor is engaged in an online multiplayer video game with other clients. The bad actor is currently losing and does not wish to have their ranking in the game impacted. Thus, they try to flood the server with game requests

in an attempt to cause it to crash, resulting in the current match of the game not impacting their ranking in the game. This scenario is captured using the syntax of UML activity diagrams in Figure 5.11.

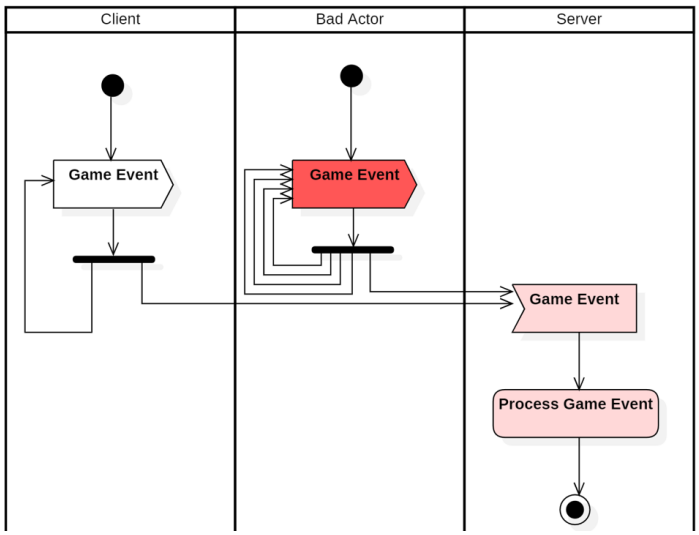


Figure 5.11: A Bad Actor Exploiting the Denial-of-Service Threat

The vulnerable elements that constitute the detection pattern are coloured in light red in Figure 5.11. The actions performed by the bad actor are coloured in bright red. In this scenario, the game server has no mechanism to prevent a flood of game requests from a single client. The specific critical elements that constitute the threat detection pattern are the "AcceptEventAction" element and the "OpaqueAction" element which performs some action using the received signal. To properly determine a system's average CERI, the elements identified as participating in this threat detection pattern would be considered critical and would need to have their CERI values calculated.

To mitigate the threat of message flooding from a disgruntled bad actor, the system could explore different mitigation options. For our purposes, the mitigation we choose to best address the representative denial-of-service threat involves rate limiting. The information that was used to determine this mitigation was gathered from entries listed in both MITRE ATT&CK (M1037) and D3FEND (D3-NTF) [93, 94]. An implementation of an adequate mitigation using the UML activity diagram syntax is shown in Figure 5.12.

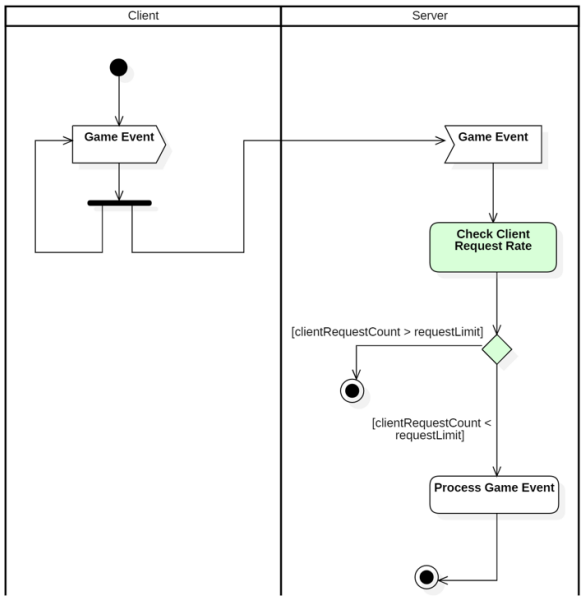


Figure 5.12: A UML Activity Diagram Hardened Against a Denial-of-Service Threat

The elements that participate in the mitigation pattern are coloured in light green. The specific elements that constitute the mitigation pattern are the "DecisionNode" element structure where one path proceeds without any modification to the data if the request rate from the client does not exceed a defined level that would indicate abuse, and the other path will end the connection for the client if abuse is detected to avoid further signal flooding. We leave the specific rate limit value and disconnection means

to designers as they will vary from system to system. Returning to the example shown in Figure 5.11, now that the game server has a rate-limit check in place, the attacker will be disconnected from their match if abuse is detected. This allows the match to complete as expected and the bad actor has their video game ranking adjusted accordingly. The representative denial-of-service threat has been mitigated, yielding an improvement to the system's security level through a reduction of its average CERI value.

5.3.6 Elevation of Privilege Threat and Mitigation Patterns

Elevation of privilege threats are typically understood to refer to instances where a bad actor can achieve higher access rights or permissions than they are supposed to have for a given system. If a bad actor manages to achieve root access, they could cause significant damage to a system that could have major impacts on its confidentiality, integrity, and availability. For our choice of a representative elevation of privilege threat, we focus on improper privilege management, informed using information from both MITRE CAPEC (CAPEC-233) and ATT&CK (T1548) entries [95, 96].

Consider a scenario where a bad actor interacts with an existing system. The bad actor has a limited set of actions they can perform based on their permissions. One such action includes the creation of directories to store files, which requires the system to temporarily raise the permission of the bad actor to create the directory before the permissions are lowered after the operation is completed. However, the bad actor crafts a directory name using malformed characters to force the directory creation operation to raise an exception, resulting in the permission level for their account not

being reset to its initial lower state. This hypothetical scenario is captured using the syntax of UML activity diagrams in Figure 5.13.

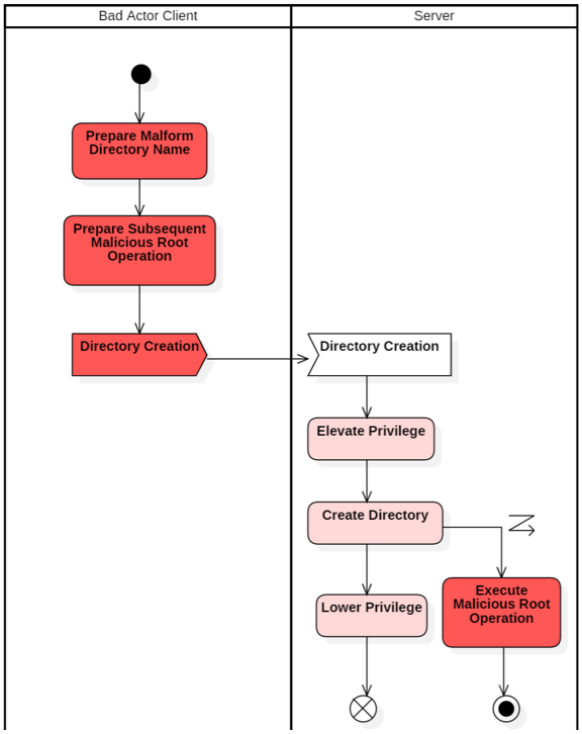


Figure 5.13: A Bad Actor Exploiting the Elevation of Privilege Threat

The vulnerable elements that constitute the detection pattern are coloured in light red in Figure 5.13. The actions performed by the bad actor are coloured in bright red. In this scenario, the "OpaqueAction" element that deals with the directory creation process is not designed to handle possible exceptions, which forces the system to follow an unexpected flow defined by the commands the bad actor wishes to execute using their newly raised permission level. The specific critical elements that constitute the threat detection pattern are the "OpaqueAction" element which refers to the elevation or raising of permissions, the second "OpaqueAction" which involves an operation that requires elevated permissions to be successfully executed, and the

third "OpaqueAction" element that is meant to lower the permission level. All three are needed to be able to determine a critical region within a system, bounded by the "OpaqueAction" elements that raise and lower the permissions of the flow.

There is the possibility for multiple "OpaqueAction" elements or other elements within the critical region. So long as a critical permission region exists by the bounded "OpaqueAction" elements, Dubhe will successfully identify the presence of this threat within a system's behavioural view. In this case, any element within a critical region will be detected as part of the threat pattern as they each present the opportunity for the elevation of privilege threat if they are not designed to handle exceptions being raised. To mitigate the threat of improper privilege management, the system must handle possible exceptions for all elements within critical system regions. The information that was used to determine this mitigation was gathered from entries listed in both MITRE ATT&CK (M1038) and D3FEND (D3-EHPV) [97, 98]. An implementation of this mitigation using the UML activity diagram syntax is shown in Figure 5.14.

The elements that participate in the mitigation pattern are coloured in light green. The specific elements that constitute the mitigation pattern are the "InputNode" element (denoted as a small box containing a downwards arrow in Figure 5.14) for an "OpaqueAction" element contained within a critical element and an "OpaqueAction" element that resets the permission level of the flow in the event an exception is raised. Returning to the example shown in Figure 5.13, now that the system is prepared to handle possible exceptions in the critical section of the flow where the permission level has been raised, no unexpected flow state is achieved. Because of this, the

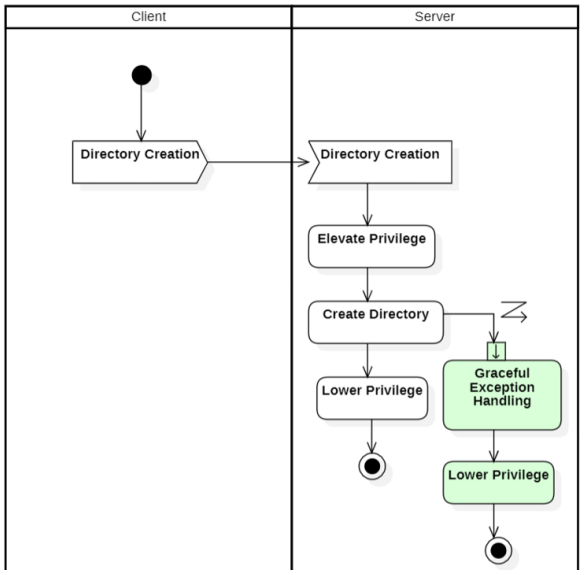


Figure 5.14: A UML Activity Diagram Hardened Against an Elevation of Privilege Threat

subsequent commands the bad actor wished to execute will not be successful, as the system will lower the permission level as part of the graceful exception handling before terminating the flow due to the exception. Adding proper exception handling has successfully hardened the system against the representative elevation of privilege threat, which will in turn lower the average CERI of the system.

5.4 Detecting Threats and Mitigations With Pattern Matching

Having presented our selection of pattern pairs for representative threats for each category of STRIDE, we now discuss the detection process that is involved in identifying

```
%
THREAT TECHNIQUE: Adversary-in-the-Middle
TECHNIQUE NUMBER: T1557
THREAT MITIGATION: Encrypt Sensitive Information
MITIGATION NUMBER: M1041
DETECT PATTERN: [["DataStoreNode", "...", "SendSignalAction"]]
MITIGATION PATTERN: [["DecisionNode", "MergeNode"],
                     ["DecisionNode", ("*Action", "Obfuscate Data"),
                     "MergeNode"]]
MITIGATION INDEX: 1
```

Listing 5.1: The Contents of a .dubhe File

these patterns within a designer's UML activity diagram. Referring back to Figure 5.1, the support action nodes and decision nodes present the logic Dubhe employs throughout its application of pattern matching. However, an important preliminary step that is not included in Figure 5.1 involves the medium in which the threat and mitigation patterns are captured to enable Dubhe to perform pattern matching.

To determine which patterns to check for, Dubhe makes use of information stored within .dubhe files. In total, six .dubhe files are used to check for patterns, one for each category of STRIDE. The format of the .dubhe file that contains the threat and mitigation patterns for the information disclosure threat presented in Section 5.3.4 is shown in Listing 5.1.

As can be seen in Listing 5.1, a single entry in a .dubhe file contains 7 lines of data. The initial % symbol seen at the top of the file is used as a delimiter for different threat and mitigation pattern pairs that fall under the same STRIDE category. We will explain each line's purpose and function before proceeding with a discussion of the process Dubhe follows while pattern matching across a designer's submitted activity diagram.

Threat Technique: Line 1 from Listing 5.1 refers to the name of the threat that this pattern pair is associated with on MITRE ATT&CK. This data is read in by Dubhe as a string. As we've mentioned previously, we made use of MITRE CAPEC, ATT&CK, and D3FEND when sourcing the patterns presented in Section 5.4. However, we choose to only include reference data to the relevant MITRE ATT&CK listing as we believe it presents the relevant information to a user the best out of the three options. Further, MITRE ATT&CK entries can be used to find associated CAPEC and D3FEND listings, so designers can seek out additional information if desired.

Technique Number: Line 2 from Listing 5.1 refers to the listing number on MITRE ATT&CK associated with the threat technique listed on the previous line. This data is read in by Dubhe as a string.

Threat Mitigation: Line 3 from Listing 5.1 refers to the name of the mitigation that this pattern pair is associated with on MITRE ATT&CK. This data is read in by Dubhe as a string.

Mitigation Number: Line 4 from Listing 5.1 refers to the listing number on MITRE ATT&CK associated with the mitigation technique listed on the previous line. This data is read in by Dubhe as a string.

Detect Pattern: Line 5 from Listing 5.1 presents the threat detection pattern Dubhe will reference when attempting to identify this threat within a designer's submitted activity diagram. The data is read by Dubhe as a Python list, where the outer list contains items that are themselves lists and the inner lists contain items of type string or type tuple. In the event an item is of type tuple, the elements of the tuple are

of type string. For tuples, the first item in the tuple represents the XMI type of the element, while the second item in the tuple specifies the name that the XMI element should have been given to suggest the presence of this threat. This is to account for cases where a threat detection pattern may be present in a system but it performs actions that are unrelated to the associated threat. To avoid forcing a specific name for an element, Dubhe makes use of a pre-trained semantic analysis model to determine if the name of the detected element is semantically equivalent to the name included within the threat detection pattern. We discuss this model in more detail when we highlight how matches are classified by Dubhe later in this section. The syntax available in defining a detection pattern refers to the XMI types of elements within a UML activity diagram. We drop the "uml:" prefix as the context is not needed for the analysis activities performed by Dubhe. Instead, if a detection element requires a specific parent swimlane to be valid, the name of swimlane can be appended to the front of the XMI type of the element. As an example "Client:SendSignalAction" indicates that an element of XMI type "SendSignalAction" should be contained within a swimlane with the name "Client". The only examples of this syntax in our work are contained within the spoofing and repudiation .dubhe files, which can be viewed in Appendix A. The "*" symbol seen before the "Action" entries in Listing 5.1 acts as a wild card, where one "*" is used per word that can be in place of the wild card. In this example, the "*" ensures that any valid type of action element from UML activities diagrams are considered as part of the detection pattern. The specific detection pattern shown in Listing 5.1 refers to the activity diagram from Figure 5.9, with the modification explained in Section 5.3.4 that elements may appear between the "DataStoreNode" and the "SendSignalAction" element. This modification is

captured with the "..." entry within the list, which Dubhe will interpret to allow for a variable number of elements between the concrete elements listed in the detection pattern. Lastly, multiple lists can be included in the detect pattern line in case the same threat can arise through different patterns. In such instances, Dubhe will consider a match to any pattern listed in the detect pattern line as an indication of the threat existing within a system.

Mitigation Pattern: Line 6 from Listing 5.1 presents the mitigation pattern Dubhe will reference when attempting to identify this mitigation within a designer's submitted activity diagram. The data is read by Dubhe as a Python list, where the outer list is a list of lists, and the inner lists contain items of type string or type tuple. In the event an item is of type tuple, the elements of the tuple are of type string. The syntax options described in the detect pattern line also apply to the mitigation pattern line.

Mitigation Index: Line 7 from Listing 5.1 refers to the index that the mitigation pattern should appear relative to the detection pattern. This data is read in by Dubhe as an integer. As an example, a value of 0 would indicate the mitigation pattern should appear before the first element in the detection pattern. A value of 1 would indicate the mitigation pattern should appear after the first element in the detection pattern, and so on. We note that a value of -1 indicates that the mitigation patterns should appear after the last element of the detection pattern, to follow the expected Python syntax for indices. Additionally, if a mitigation would be valid at multiple indices, this can be indicated by using a comma-separated list of integers that correspond to the various valid indices. In this situation, Dubhe will read the data as a string and

process the input into a list of integers. Lastly, for the cases where the mitigation index is 0 or -1, the mitigation does not need to appear immediately before the first element or after the last element respectively. As long as it is present before the start of a flow (when the index is 0) or before the end of the flow (when the index is -1), the mitigation pattern will be detected as valid.

While we only covered a single .dubhe file to facilitate the explanation of the file's structure, the remaining .dubhe files for the 5 remaining threat and mitigation pattern pairs presented in Section 5.3 can be found in Appendix A. Appendix A also contains the grammar for .dubhe files, expressed in Backus-Naur Form (BNF). Although every line in a .dubhe file contributes to the generation of a comprehensive security analysis report, Dubhe primarily utilizes the last three lines of each entry for pattern matching. The detection process matches the high-level overview captured by the support action nodes in Figure 5.1, which we present in more detail in Figure 5.15.

Given the complexity of the pattern-matching algorithm used by Dubhe, including its recursive nature and multiple validation layers, it is not presented as pseudo-code. Instead, the algorithm is described in detail throughout this section to highlight the step-by-step processes involved in threat detection and mitigation pattern identification. While explaining Dubhe's pattern matching, we will refer back to the example cases present in Figure 5.15. Note that the detection pattern and mitigation pattern listed at the top of Figure 5.15 are used across all cases, the specific case number is seen to the left of each example in grey, and the outcomes are listed to the right of each example in orange. The remaining elements follow the standard syntax for UML activity diagrams.

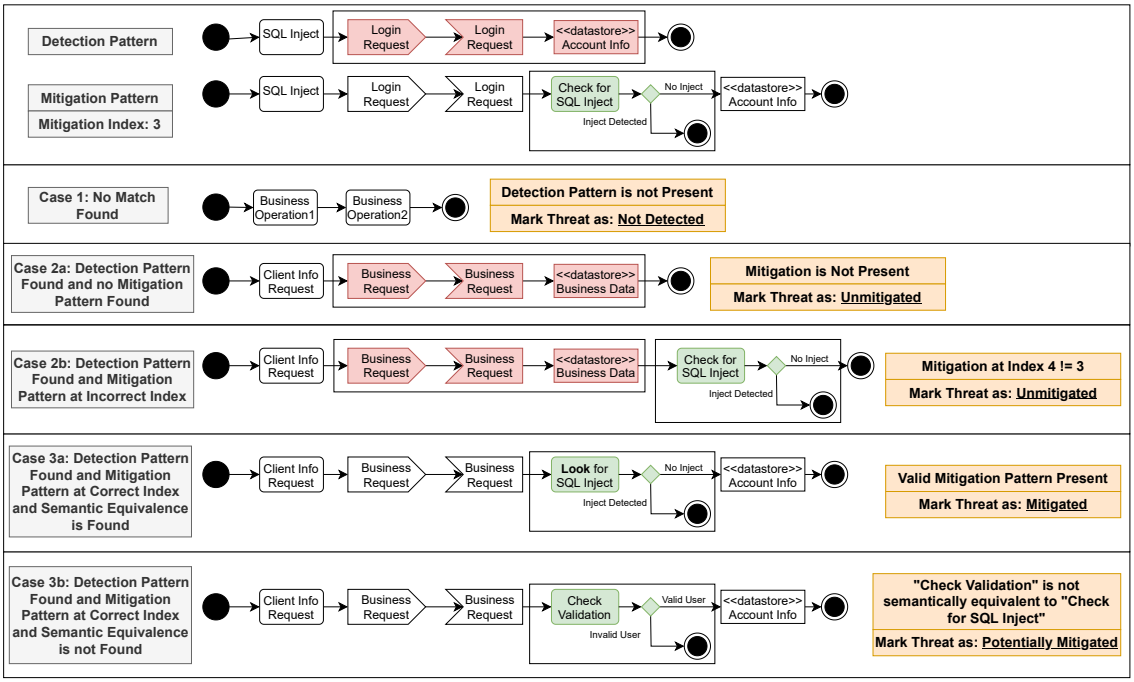


Figure 5.15: A Visual Example of Dubhe's Pattern Matching Process

Returning to Dubhe's implementation of pattern matching, Dubhe begins by collecting a list of ActivityElements that have no incoming flows. In other words, Dubhe creates a list of elements that represent the start points of flows within a system. For each collected element that has no incoming flows, Dubhe will check to see if the element's XMI type matches the XMI type of the first element from the current detection pattern. If the element is not a match, Dubhe will proceed to the subsequent elements connected by flows and continue the detection check process. This will repeat until all elements across the system have been checked. If no elements match the first element of the detection pattern, or if no elements match the expected name of the first element from a detection pattern, Dubhe marks the threat as *not detected*. An example of this scenario is listed as **Case 1** in Figure 5.15.

If at any point an element is encountered that matches the XMI type of the first element of the detection pattern, Dubhe proceeds to check if the full detection pattern is present in the subsequent elements. If during this check an element's XMI type does not match what is listed in the detection pattern, Dubhe returns to the point where it began pattern matching against the detection pattern and continues checking subsequent elements. In the event the pattern allows for extraneous elements, Dubhe will continue down the flow looking for the next listed element in the threat detection pattern. As long as the next element in the pattern appears before the end of the flow, Dubhe can proceed with its attempt to fully match the threat detection pattern.

If the threat detection pattern is fully matched, Dubhe utilizes the mitigation index from the .dubhe entry to identify where to check for a mitigation pattern. If the mitigation pattern is not identified at the specified mitigation index, Dubhe marks the threat as *unmitigated*. An example where a threat pattern is detected but no mitigation pattern is present is listed as **Case 2a** in Figure 5.15. The case where a mitigation pattern for the threat is present but appears at the wrong mitigation index is listed as **Case 2a** in Figure 5.15. In both instances, the threat is marked as *unmitigated*.

Whenever a threat pattern is fully detected, Dubhe makes a note of each element that participates in the pattern. As these elements participate in a threat, they are considered critical elements. Dubhe tracks these elements using a Python dictionary, where the key is the XMI ID of the critical element, while the value is a tuple with the following data:

$$(m_i, pm_i, t_i)$$

Where m_i is the number of threats a critical element i participates in marked as *mitigated*, pm_i is the number of threats a critical element i participates in marked as *potentially mitigated*, and t_i is the total number of threats a critical element i participates in. If an entry for a critical element does not already exist, Dubhe adds it to the dictionary as a new key-value pair. Otherwise, Dubhe updates the appropriate values depending on how the current threat has been marked. In the case of a threat that is marked as *unmitigated*, this means increasing the value of t_i by 1.

After the critical elements have been appropriately updated, Dubhe proceeds with the remaining elements from the current flow in case the same threat is present at multiple points in the system. If the mitigation pattern is fully matched and the mitigation pattern does not specify any named elements, Dubhe marks the threat as *mitigated*. If a threat is marked as *mitigated*, critical elements will have their associated values of t_i and m_i increased by 1.

If the detected mitigation pattern has specified named elements, Dubhe performs a semantic analysis check. Dubhe makes use of a pre-trained spaCy semantic comparison model named "en_core_web_md" [99]. This model compares the semantic similarity between two text strings and determines if the meanings of the texts are similar from a linguistic perspective. As an example, the words **encrypt** and **encipher** are semantically equivalent as they express the same meaning, despite being different words. The model requires a similarity threshold between 0 and 1 to function, which we left at the default value of 0.7. If semantic equivalence is determined for all elements with names specified in the mitigation pattern, Dubhe marks the threat as *mitigated*. An example of this scenario is shown as **Case 3a** in Figure 5.15. In the

case where no semantic analysis is required, all participating critical elements will see their values of t_i and m_i increased by 1.

In the event semantic equivalence is not found, Dubhe marks the threat as *potentially mitigated*. We cannot guarantee that the mitigation is not in place given our use of the spaCy model for recall to check for semantic equivalence, but we flag the occurrence to inform the designer and ultimately let them decide if the threat has or has not been properly mitigated [99]. An example of this scenario is shown as **Case 3b** in Figure 5.15. In the case where semantic equivalence is invalid, all participating critical elements will see their values of t_i and pm_i increased by 1. Conversely, when detecting threats, we opted not to use a *potentially detected* status. By not including a *potentially detected* status, we reduce the likelihood of false positives, ensuring that the identified threats are actionable and relevant.

Finally, we wish to highlight one additional example scenario for pattern matching performed by Dubhe, which we present in Figure 5.16.

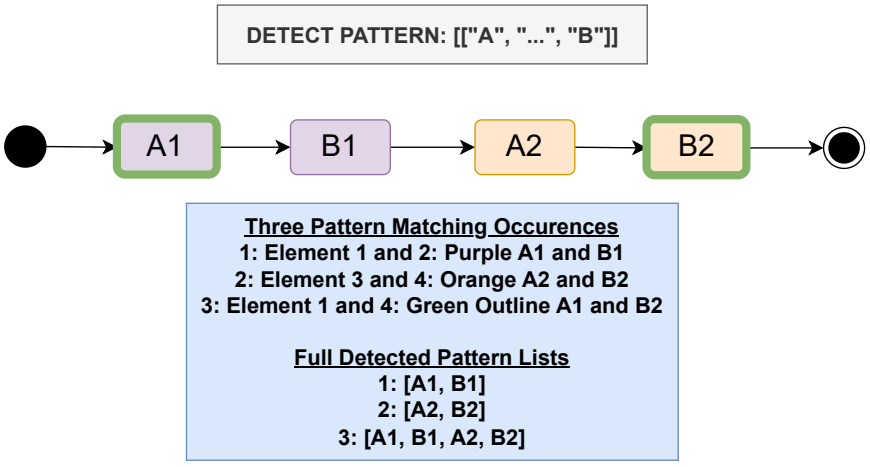


Figure 5.16: Matching Multiple Occurrences of a Single Pattern in Dubhe

In this scenario, the detection pattern comprises three elements, “A”, “...”, and “B”. When a pattern such as “A”, “...”, and “B” is used, and the model includes elements like A1, B1, A2, and B2, there is a potential for multiple matches, including combinations like A1-B1, A2-B2, and A1-B1-A2-B2. Our approach is designed to recognize all valid matches within the model. Specifically, the pattern-matching algorithm evaluates each possible pairing independently through backtrack indexing when a match is found, ensuring that all potential matches are identified. For example, in the given scenario, the system would match the pattern three times: once for A1-B1, once for A2-B2, and once for A1-B1-A2-B2, including all elements as part of the detection pattern due to the use of the variable “...” notation within the detection pattern. This comprehensive matching process is crucial for ensuring that all relevant patterns are detected, even when they appear in varying sequences or combinations within the model.

Now that we have described the process Dubhe follows to perform pattern matching, we move on to the processes required to determine the remaining variables needed to calculate a critical element’s CERI.

5.5 Calculating the Cyclomatic Complexity of Critical Elements

To be able to calculate a critical element’s CERI, we must first determine its cyclomatic complexity. As mentioned in Section 4.2.1, the equation to determine the cyclomatic complexity of a single element within a graph can be simplified to $M = E + 1$.

Thus, the only value we need to be able to determine a critical element's cyclomatic complexity is the number of edges, or in our case flows, leading into it.

Fortunately, as part of the data transformation performed when parsing the designer submitted XMI file, all ActivityElement objects contain two lists of XMI IDs with incoming and outgoing flows. When calculating the CERI for a critical element i, Dubhe determines the length of the list of incoming flows using Python's `len()` function and adds 1 to the value to determine its cyclomatic complexity. This value is appended to the existing tuple for the critical element i that was explained in Section 5.4. With this, Dubhe now has all of the required information to calculate the average CERI across all critical elements in a system.

5.6 Calculating the Average CERI for Different Cases

With all the required information gathered to calculate a system's average CERI, we now highlight how these final CERI calculations are performed for each critical element within a system. It should be noted that Dubhe does not immediately calculate a system's average CERI after all required values are gathered, as reflected in Figure 5.1. Rather, Dubhe proceeds to gather the information required to calculate the CPP of a system before performing all final calculations for the metrics that define a system's BSP in a single step. This was done to incorporate similar responsibilities into distinct modules, enabling quicker debugging during the development of Dubhe.

Dubhe determines the CERI for each identified critical element to determine a system's average CERI. This is done using the values from the tuple for each previously identified critical element as described in Section 5.4. However, due to the categorization of threats as *mitigated* or *potentially mitigated*, this average calculation occurs twice. The first time, the CERI of critical elements is determined using the number of threats it participates in that have been classified as *mitigated*, which would be the value of m_i from a critical element i's tuple. The second time the calculation is performed, it includes both *mitigated* and *potentially mitigated* threats, so the sum of the values m_i and pm_i from a critical element i's tuple. This yields two final values for a system's average CERI, one where we assume the potentially mitigated threats have not truly been mitigated and another where we assume that all potentially mitigated threats have been fully mitigated. Dubhe reports both values back to the designer, representing a worst-case scenario when only the threats marked as *mitigated* are considered, and a best-case scenario where both *mitigated* and *potentially mitigated* threats are considered for a system's average CERI. Ultimately, the value a designer decides best reflects their system will come from their evaluation of the threats that have been marked as potentially mitigated, but we believe presenting both cases provides a more complete representation of a system's average CERI.

5.7 Chapter Summary

In this chapter, we detailed the steps Dubhe employs to identify a system's average CERI. We began with an overview of how a designer's submitted XMI representation of their system's UML activity diagram is transformed into ActivityElement

objects. Next, we explained our decision to leverage representative STRIDE threats to create detection and mitigation patterns. We also presented a detection and mitigation pattern pair for each category of STRIDE and explained how they were sourced. We followed this with a technical breakdown of the pattern matching Dubhe employs to mark threats into different categories. We ended with a brief explanation of how a critical element's cyclomatic complexity is determined before explaining how the final calculated average CERI of a system is calculated. In the next chapter, we explore the other metric required to inform a system's BSP, known as Corruption Propagation Potential (CPP).

Chapter 6

Revealing and Mitigating Corruptible Flows in UML Activity Diagrams

In this chapter, we explain an approach that determines the corruption potential of system flows within the XMI representations of UML activity diagrams. Section 6.1 presents the steps needed to calculate a system's CPP. Section 6.2 demonstrates how Dubhe gathers the corruptible flows through ActivityElement objects. Section 6.3 introduces data corruption attacks and their role in corruption propagation. Section 6.4 highlights how to effectively harden systems against data corruption attacks in different scenarios. Section 6.5 explains how Dubhe computes the CPP of a system after gathering all required information. Section 6.6 concludes the chapter with a summary of the information presented.

6.1 An Approach to Resolve the CPP of Systems

As explained in Section 4.3.1, to determine a system's BSP we must identify and determine the lengths of all corruptible flows to calculate a system's CPP. *Corruptible flows* are flows present in a system that are susceptible to corruption-based attacks. The susceptibility to corruption will be present in all system flows due to the possibility of insider attacks, but the length of a corruptible flow can be reduced through the introduction of mitigations that specifically target known corruption-based attacks. This reduction in length reduces the overall corruption potential from a single corruption-based attack, yielding a lower calculated CPP and an overall increase in a system's security level. As such, to determine a system's CPP we must identify corruptible flow lengths while accounting for reductions to these lengths with the presence of mitigations that combat corruption-based attacks.

Like we did when we explained the process to determine a system's average CERI, we provide a technical breakdown for the actions performed by Dubhe in the determination of a system's CPP. A more detailed overview of Steps 3 and 4 from the high-level overview explained in Section 4.4 is shown in Figure 6.1.

We explore the main actions performed by Dubhe, coloured in blue in Figure 6.1, in the subsequent sections. The lower-level actions, coloured in yellow, are mapped to the high-level steps previously presented in Figure 4.5. The section each main action maps to is included within Figure 6.1 at the top of each main action respectively. The concurrent support actions, coloured in white in Figure 6.1, will be explained alongside the main action they are initially forked from. Having presented an overview

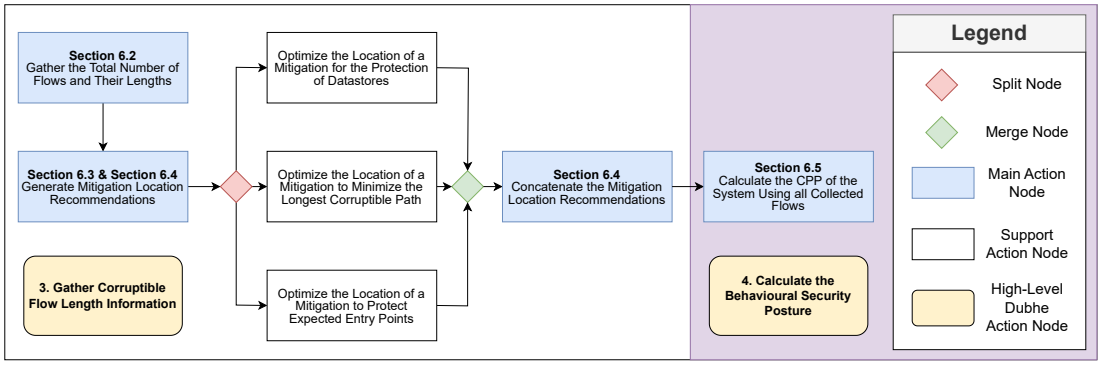


Figure 6.1: The Steps Needed to Calculate a System's CPP

of actions that are performed by Dubhe to determine a system's CPP, we proceed with an exploration of the process involved in gathering and determining the length of the corruptible flows that exist within the designer submitted XMI representation of a UML activity diagram.

6.2 Gathering System Flow Information

As the CPP metric is exclusively dependent on the length and number of corruptible flows present within a system, a natural first step to its determination is identifying and measuring corruptible flows within a system. The flow generation algorithm used here is a depth-first search, which ensures that all possible flows are explored before backtracking. Through the creation of ActivityElement objects generated through the translation of the XMI data that captures a designer's UML activity diagram, checking all flows within a system is a straightforward task. Dubhe begins by identifying ActivityElements that have no incoming flows, implying they are elements that constitute the start of a flow within a system. Dubhe will then proceed to check the

outgoing flows from start points to begin building a list of all flows within a system. This process continues until Dubhe encounters an element that has no outgoing flows, indicating the end to the current flow being analyzed. In the event an element has multiple outgoing flows, Dubhe makes a shallow copy of the current flow being analyzed. Upon reaching the end of one outgoing flow, Dubhe will return to the element where multiple outgoing flows were detected and will begin moving down the unexplored direction until another endpoint is reached. This occurs until all flows have been traversed and fully captured.

The net result is a list of all flows within a system, captured as a list of XMI IDs within Python. This parent list contains individual lists of flows, whose lengths can be easily determined using the Python `len()` function. However, before determining the length of each flow, Dubhe analyzes each collected flow to determine if any mitigations against corruption-based attacks are present. As mentioned, a corruptible flow would be split by such a mitigation, essentially converting one long corruptible flow into two smaller corruptible flows on either side. While the ability to cover a variety of corruption-based attacks would ultimately be preferred, to maintain scope we choose to focus on data corruption attacks. Like with the representative STRIDE threats chosen when determining a system's CERI, selecting a representative corruption-based attack allows us to demonstrate the application of our approach while leaving the opportunity for future attacks to be considered by Dubhe through the inclusion of additional mitigation checks.

For data corruption attacks, Dubhe searches for a data sanitizer by checking an ActivityElement's XMI type and name through `get()` methods. A *data sanitizer* is an

object capable of sanitizing corrupted data to prevent its propagation throughout a system. In an activity diagram, a data sanitizer has an XMI type of "*Action", and the name given to the element should be semantically equivalent to "data sanitizer". Semantic equivalence is once again determined using the pre-trained spaCy semantic comparison model named "en_core_web_md" [99]. If found, Dubhe splits the current flow at the point of the mitigation and continues with this process until all flows have been analyzed in their entirety. We still need to consider both split flows as corruptible despite the mitigation as the flow that exists after the mitigation can still be targeted by insiders through memory data corruption attacks. Due to their importance in our approach, we provide a brief explanation of data corruption attacks in the following section.

6.3 Data Corruption Attacks

Data corruption attacks manipulate legitimate data within a system to propagate corruption for malicious purposes. While the goal of corruption attacks can be to render a system inoperable by corrupting the code of the system, other attacks are focused on memory tampering to be able to execute malicious code and steal sensitive information. The potential for harm makes data corruption attacks a topic of research interest, with numerous works exploring such attacks and their potential countermeasures [100, 101, 102].

For this work, a surface-level understanding of data corruption attacks is sufficient. The methods by which data corruption attacks are employed can vary dramatically

and fall outside of the scope of this thesis. While existing literature on data corruption attacks is expanding, it is important to note that research thus far has been focused on the analysis of these attacks using completed systems. This enables the analysis of source code and executable models that are created as development shifts toward the implementation of the functional view of systems. Our approach considers data corruption attacks at an earlier stage of system development, focusing on techniques that target a system's behavioural view to harden it against these attacks.

To best protect systems against data corruption attacks, mitigations must be implemented. In our work, we consider the implementation of a data sanitizer object as a suitable mitigation for data corruption attacks. For our purposes, a data sanitizer is a component responsible for checking the integrity of data, whether by sanitizing the input in the case of data injection attacks, comparing the data against a set of expected data that should flow between elements, among other possible forms of detection [103]. The exact nature of the data sanitizer object is outside of the scope of this work and will depend on the needs of the current system. Now that we have discussed data corruption attacks and their associated mitigation, we can shift our focus toward a unique facet of Dubhe that deals with mitigation placement optimization.

6.4 Protecting Systems Against Data Corruption Attacks

Unlike the case where Dubhe checks for mitigations when determining a system's CERI, the placement of a data sanitizer does not need to be at a specific location within

a system to effectively reduce the impact of data corruption attacks. The inclusion of a data sanitizer anywhere in a system is enough to yield a reduction in the CPP metric, which in turn represents an increase in the security level of a system. That being said, while the CPP metric demonstrates an improvement to a system's security level through the introduction of mitigations, there are further refinements that can be done to better harden systems against data corruption attacks. While this information is not captured in the metric itself, Dubhe does perform a variety of scenario analyses to help guide designers toward the optimal placement of a data sanitizer with regards to protecting a system's data stores, protecting the expected system entry points, and minimizing the longest corruptible flow or flows that exist within a system.

To demonstrate how the recommended location of data sanitizers varies to harden systems against data corruption attacks under different scenarios, we highlight an example using an Online Seller of Merchandise (OSM) system originally defined in [104]. For our purposes, we only use five components of those defined in the UML class diagram of the OSM system, which are the components coloured in white in Figure 6.2.

As we are interested in analyzing a system's BSP we need to analyze the OSM system's behavioural view. Using the five components specified in Figure 6.2, we created a UML activity diagram that represents an account login use case, which is shown in Figure 6.3.

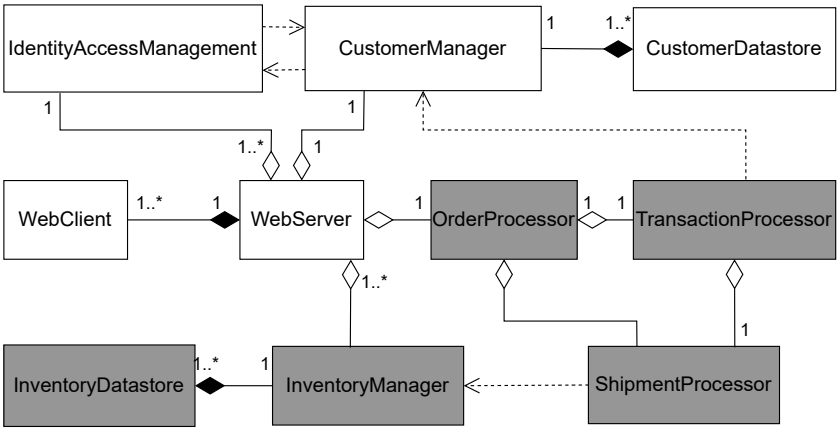


Figure 6.2: A UML Class Diagram of the OSM System

The activity diagram created for the OSM system demonstrates the expected use case for when a customer tries to login to their account. There are certain conditional checks in place to ensure the customer is authenticated before granting access to their account. This is done through the querying of account information from the "CustomerDatastore" within the system. If the account information is not found or the submitted login information is not validated against the expected data, access to the account does not occur. A third scenario will cause a login to fail, which occurs if customer information is not retrieved from the "CustomerDatastore" within 15 seconds of receiving the "Login Information" signal. This condition is in place in an attempt to prevent system stalls that could threaten the availability of the system. Beyond these cases, the system has not been meaningfully hardened against other forms of attacks and mitigations are not present for any representative STRIDE threats that may be present in the system. In other words, the UML activity diagram for the OSM system has not undergone any form of security analysis or system hardening.

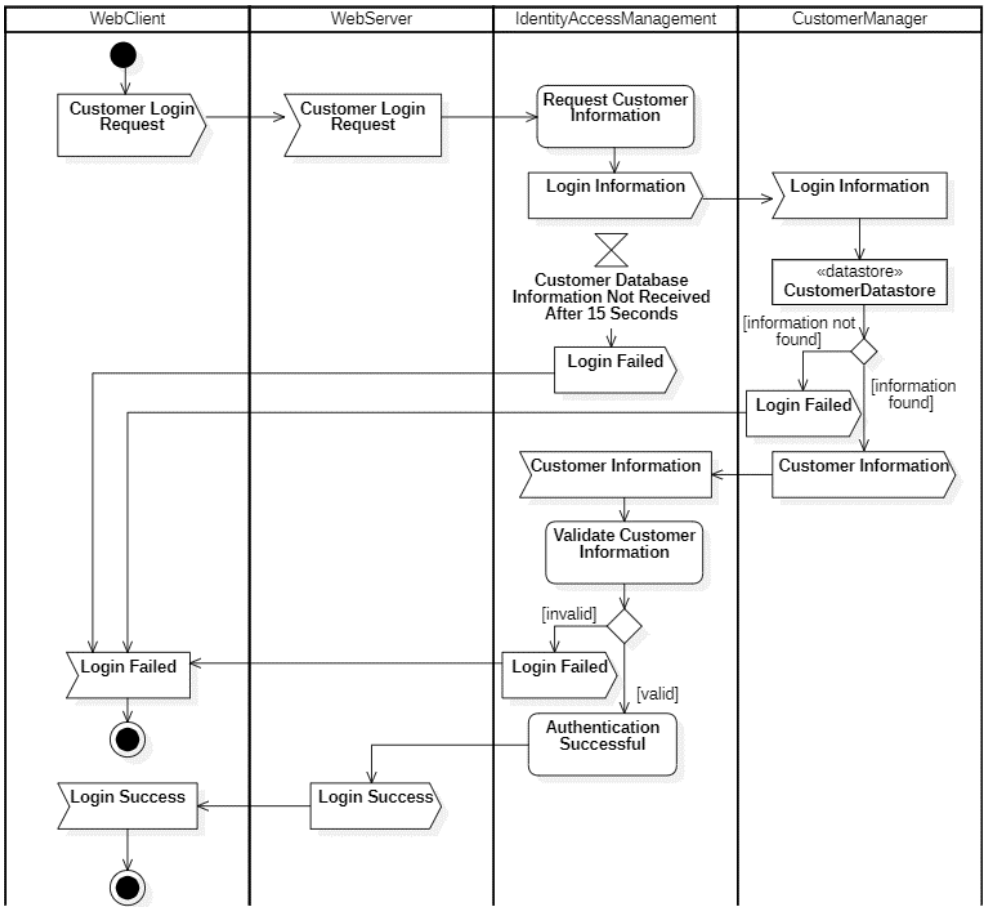


Figure 6.3: A UML Activity Diagram for an Account Login Use Case for the OSM System

As captured in Figure 6.1, Dubhe performs 3 concurrent support actions when determining optimal placements for data sanitizers within a system. These support actions prioritize the placement of a data sanitizer to accomplish different objectives designers may have when choosing to harden their systems against data corruption attacks. In an ideal world, it would be desirable to place a data sanitizer at each location that Dubhe recommends. However, the time and effort associated with such an undertaking may lie outside the scope of feasibility for a project. Thus, we provide

designers with all recommendations but do not specifically prioritize or suggest one recommendation over another.

Over the next three subsections, we describe the algorithms involved in each support action that are used to recommend the locations of data sanitizers to harden systems against data corruption attacks. Using the presented algorithms, we will reference back to the UML activity diagram from Figure 6.3 to highlight the recommendations each algorithm generates for the OSM system.

6.4.1 Prioritizing the Protection of Data Stores

The first support action prioritizes the protection of data stores. As seen in Algorithm 1, each element e in the set of ActivityElements E is checked to see if it is of type "DataStoreNode", which is the standard for such elements per the XMI 2.X standard [1]. If no "DataStoreNode" is present in the XMI, this analysis activity does not generate any suggestions. If exactly one "DataStoreNode" is detected, the analysis activity will recommend that a data sanitizer object be placed immediately before the data store element and the element that has a flow entering the data store. If multiple "DataStoreNode" elements are detected, Dubhe works backward from each "DataStoreNode" element to determine the flow of elements that lead into it. Once all of the flows have been gathered, Dubhe will find which element is the most common or central across all flows and in the event of a tie, will determine which element among those with the highest occurrences is closest to the data stores. For our purposes, we rank the centrality of an ActivityElement higher the more flows it appears in. In either case, this analysis activity generates a suggestion for the location of a data

sanitizer that will protect the most data stores possible by ensuring that data enters the data sanitizer before it reaches a data store. This will harden a system against data corruption attacks that target data stores with the goal of either gaining access to or corrupting the information contained within.

Algorithm 1 Prioritize the Protection of Datastores

```
1: procedure ProtectDatastores(ActivityElements)
2:     DataNodes ← ∅                                  ▷ Initialize as an empty set
3:     FlowElements ← ∅                               ▷ Initialize as an empty set
4:     RecommendationPart1 ← ""                       ▷ Initialize as an empty string
5:     RecommendationPart2 ← ""                       ▷ Initialize as an empty string
6:     for each element ∈ ActivityElements do
7:         if element.type == "DataStoreNode" then
8:             DataNodes ← DataNodes ∪ {element}
9:         end if
10:    end for
11:    if ||DataNodes|| == 1 then                     ▷ Only one data store node exists
12:        for each node ∈ DataNodes do
13:            RecommendationPart1 ← node.entryFlow[0].name
14:            RecommendationPart2 ← node.name
15:        end for
16:    else                                           ▷ Multiple data store nodes
17:        for each node ∈ DataNodes do
18:            while ||node.entryFlow|| > 0 do
19:                FlowElements ← FlowElements ∪ {node.entryFlow}
20:                node ← node.entryFlow
21:            end while
22:        end for
23:        result ← Node with max centrality closest to data store in FlowElements
24:        RecommendationPart1 ← result.entryFlow[0].name
25:        RecommendationPart2 ← result.name
26:    end if
27: end procedure
```

Having outlined the algorithm, we now present an example of its application using the OSM system. Consider a scenario where a designer is interested in hardening

```
1 ----Protecting Data Stores----
2 It is recommended to place a Data Sanitizer object between the
    following elements:
3     - AcceptEventAction: Login Information (parented by
        CustomerManager)
4     - DataStoreNode: CustomerDatabase (parented by CustomerManager)
5 This recommendation is beneficial if you want to maximize the
    protection of your data stores against corrupted data that would
    be damaging if destroyed or leaked to an attacker (e.g., data
    injection attacks).
```

Listing 6.1: Output Snippet to Protect the CustomerDatabase in the OSM System

the OSM system against data corruption attacks while prioritizing the protection of the data store present within. Through the application of Algorithm 1, the following recommendation is determined, shown in Listing 6.1.

The recommendation and associated text in Listing 6.1 are representative of the information Dubhe provides to designers following its analysis. We will provide an in-depth walkthrough of Dubhe in Chapter 7, but for now, we simply use the Dubhe-generated recommendations for these example scenarios with the OSM system. Using the recommendation generated from Algorithm 1, the designer places a data sanitizer object between the "AcceptEventAction" element named *Login Information* and the "AcceptEventAction" element named *Login Information*. The inclusion of this recommendation in the OSM system's activity diagram is shown in Figure 6.4.

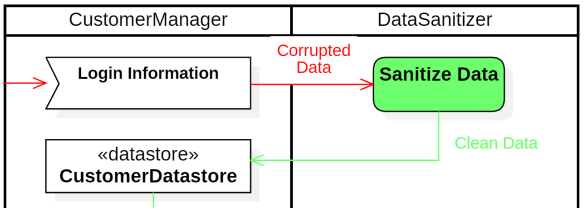


Figure 6.4: Modified Activity Flow to Protect the CustomerDatabase in thee OSM System

Thus, the designer was able to successfully harden the OSM system against data corruption attacks by leveraging the recommendation generated from Algorithm 1. While this example does highlight one case covered by Algorithm 1, it does not adequately demonstrate the case for when a system contains multiple data stores that require protection. To cover this case, consider a revision where the OSM system instead queries two different data stores concurrently during the login flow, one for login information and another for client profile information, demonstrated in Figure 6.5.

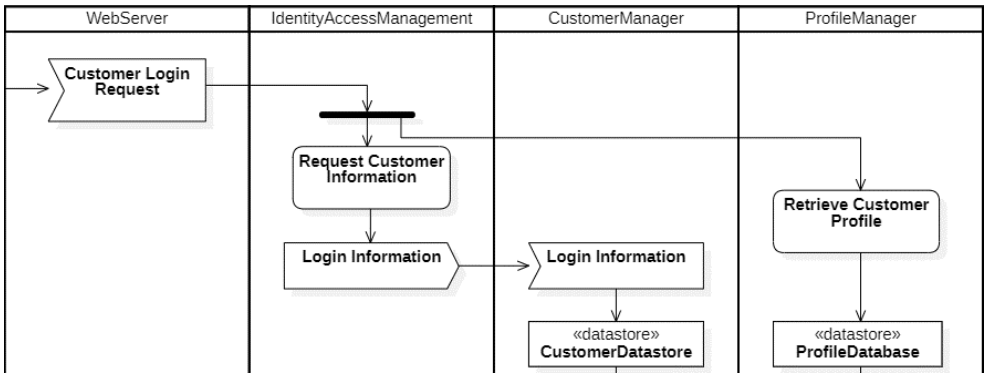


Figure 6.5: A Revised Multi-Data Store OSM Activity Flow

In contrast to the case where a single data store is present, Algorithm 1 attempts to protect the largest number of data stores possible with a single data sanitizer. In this case, since multiple data stores share an initial flow, the recommendation will be different than the one shown in Listing 6.1. This revised recommendation is shown in Listing 6.2.

In this case, the recommendation is to place a data sanitizer between the "AcceptEventAction" element named *Customer Login Request* and the "JoinNode" element named *JoinNode1*. The inclusion of this recommendation in the revised OSM system's activity diagram is shown in Figure 6.6.

```
----Protecting Data Stores----
It is recommended to place a Data Sanitizer object between the
   following elements:
    - AcceptEventAction: Customer Login Request (parented by
       WebServer)
    - JoinNode: JoinNode1 (parented by IdentityAccessManagement)
This recommendation is beneficial if you want to maximize the
   protection of your data stores against corrupted data that would
   be damaging if destroyed or leaked to an attacker (e.g., data
   injection attacks).
```

Listing 6.2: Output Snippet for the Revised Multi-Data Store OSM Activity Flow

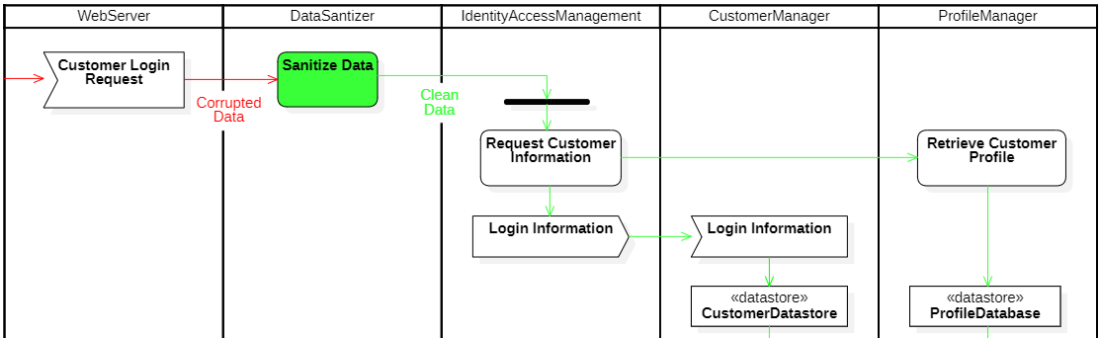


Figure 6.6: Modified Activity Flow to Protect Multiple Data Stores in the OSM System

Thus, the designer was able to successfully harden the OSM system against data corruption attacks by leveraging the recommendation generated from Algorithm 1. As both flows are now impacted by a mitigation against data corruption attacks, this modification will also reduce the system's calculated CPP value.

6.4.2 Prioritizing the Minimization of the Longest Corruptible Flow

The second support action makes the minimization of corruption propagation its goal. As illustrated in Algorithm 2, analysis begins by iterating through the set of

all generated ActivityElements E. For each element e in E, Dubhe checks to see if it has no incoming flows, indicating that the element appears at the start of a flow. These elements are then collected and iterated through. For each of these elements, Dubhe will determine their total flow length. Once every flow length has been determined, Dubhe will calculate which flow is the longest and attempt to place a data sanitizer object in the middle of the flow. In this sense, the largest corruptible flow that previously existed in the system will be eliminated in favour of two shorter flows. This accomplishes the goal of minimizing the worst-case corruption propagation scenario for systems, making full system takeovers by data corruption attacks more difficult to accomplish.

Algorithm 2 is designed to minimize the potential for total system corruption in scenarios where both insider attacks and external threats are likely and need to be addressed. By systematically analyzing the flows in a UML activity diagram to identify and mitigate corruption-based attacks that could compromise a system, this algorithm minimizes the potential for total system corruption to occur by strategically placing a data sanitizer object in the middle of the longest corruptible flow. This ensures that said flow cannot be fully corrupted, sufficiently hardening the system against data corruption attacks. The focus on addressing both internal and external threats makes Algorithm 2 particularly effective in environments where comprehensive security is essential.

Now that we have explained the algorithm, we present an example of its application using the OSM system. Consider a scenario where a designer is interested in hardening the OSM system against data corruption attacks while minimizing the longest

Algorithm 2 Prioritize the Minimization of Corruption Flow Lengths

```
1: procedure MinimizeCorruption(ActivityElements)
2:     StartNodes ← ∅                                  ▷ Initialize as an empty set
3:     AllFlows ← ∅                                    ▷ Initialize as an empty set
4:     CurrentFlow ← []                                ▷ Initialize as an empty list
5:     MidPoint ← null                  ▷ Initialize object for later assignment
6:     RecommendationPart1 ← “”                        ▷ Initialize as an empty string
7:     RecommendationPart2 ← “”                        ▷ Initialize as an empty string
8:     for each element ∈ ActivityElements do
9:         if ‖element.entryFlow‖ == 0 then            ▷ Find nodes with no entry flow
10:            StartNodes ← StartNodes ∪ {element}
11:        end if
12:    end for
13:    for each node ∈ StartNodes do
14:        CurrentFlow ← [node]
15:        while ‖node.exitFlow‖ > 0 do
16:            CurrentFlow ← CurrentFlow.append(node.exitFlow)
17:            node ← node.exitFlow
18:        end while
19:        AllFlows ← AllFlows ∪ {CurrentFlow}
20:    end for
21:    MidPoint ← Middle element of the longest flow in AllFlows
22:    RecommendationPart1 ← MidPoint.entryFlow[0].name
23:    RecommendationPart2 ← MidPoint.name
24: end procedure
```

corruptible flow present in the system. Through the application of Algorithm 2, the following recommendation is determined, shown in Listing 6.3.

Using the recommendation generated from Algorithm 2, the designer places a data sanitizer object between the “DataStoreNode” element named *CustomerDatastore* and the “DecisionNode” element named *DecisionNode1*. The inclusion of this recommendation in the OSM system’s activity diagram is shown in Figure 6.7.

```
----Minimizing Corruption Propagation----
It is recommended to place a Data Sanitizer object between the
   following elements:
    - DataStoreNode: CustomerDatabase (parented by CustomerManager)
    - DecisionNode: DecisionNode1 (parented by CustomerManager)
This recommendation should be applied if you have the goal of
   minimizing the longest flow of corruption within your system,
   making system-wide data corruption attacks more difficult.
```

Listing 6.3: Output Snippet for Minimizing the Corruption Propagation of the OSM Activity Flow

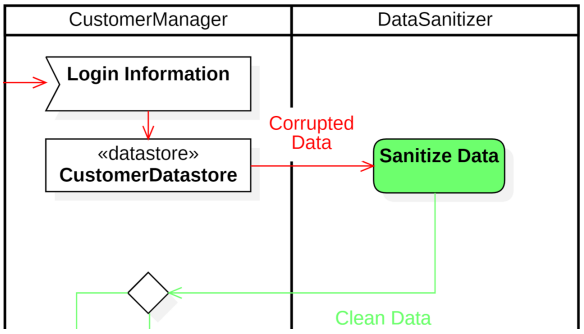


Figure 6.7: Modified Activity Flow to Minimize Corruption Propagation in thee OSM System

Through the inclusion of the recommendation generated from Algorithm 2, the designer is successful in hardening the OSM system while minimizing the longest corruptible flow present in the system. Interestingly, this recommendation sees a data sanitizer placed after the data store containing customer information. A designer may be willing to sacrifice the minimization of the longest corruptible flow in place of moving the location of the data sanitizer to appear before the data store (as was the recommendation from Algorithm 1). However, if the true desire of a designer is to minimize the longest corruptible flow present in a system, this recommendation accomplishes that goal.

Algorithm 3 Prioritize the Protection of Expected Entry Points

```
1: procedure ProtectEntry(ActivityElements)
2:     RecommendationPart1 ← “”                ▷ Initialize as an empty string
3:     RecommendationPart2 ← “”                ▷ Initialize as an empty string
4:     for each element ∈ ActivityElements do
5:         if element.type == “InitialNode” then
6:             RecommendationPart1 ← element.name
7:             RecommendationPart2 ← element.exitFlow[0].name
8:         end if
9:     end for
10: end procedure
```

6.4.3 Prioritizing the Protection of Expected Entry Points

The third and final support action prioritizes the protection of expected system entry points. As shown in Algorithm 3, Dubhe iterates through the set of ActivityElements E to determine which ActivityElement e has a value of “InitialNode”, which is the standard for such elements per the XMI 2.X standard [1]. Once this is identified, Dubhe prepares the recommended location for a data sanitizer object to be placed between the identified InitialNode element and the element connected to the InitialNode element by an outgoing flow. Application of this suggestion would result in the sanitization of data entering the system through the expected system entry point, effectively hardening the system against data corruption attacks that use defined entry points as their initial attack vector.

Algorithm 3 provides mitigation recommendations specifically in cases where insider threats are not a primary concern. Instead, it focuses on minimizing the corruption potential originating from a system's expected entry points. By attempting to address corruption as soon as it enters a system, Algorithm 3 helps to ensure that external threats are effectively managed. This targeted approach reduces the potential impact

```
1 ----Protecting Expected Entry Points----
2 It is recommended to place a Data Sanitizer object between the
     following elements:
3      - InitialNode (parented by WebClient)
4      - OpaqueAction: Client Login Request (parented by WebClient)
5 This recommendation is useful if the threat of insider attacks is
     sufficiently small compared to the threat of external attacks.
     Examples of such external attacks include attempting to harm your
      system by threatening its availability or attempting a forceful
     takeover using arbitrary code execution via corrupted data.
```

Listing 6.4: Output Snippet for the Protection of Expected Entry Points for the OSM Activity Flow

of corruption that could compromise critical data flows, providing a focused defence against external bad actors.

In contrast to the other algorithms presented thus far, Algorithm 3 is very straightforward. For the sake of completeness, we still wish to highlight an example of its application. Consider a scenario where a designer is interested in hardening the OSM system against data corruption attacks while protecting the expected entry points into the system. Through the application of Algorithm 3, the following recommendation is determined, shown in Listing 6.4.

Using the recommendation generated from Algorithm 3, the designer places a data sanitizer object between the "InitialNode" element, which in this case does not have a name attribute, and the "OpaqueAction" element named *ClientLoginRequest*. The inclusion of this recommendation in the OSM system's activity diagram is shown in Figure 6.8.

Placing a data sanitizer directly after the expected entry point of the system has successfully hardened it against data corruption attacks. Yet, this scenario leaves a large corruptible flow length in the system that can be exploited by bad actors

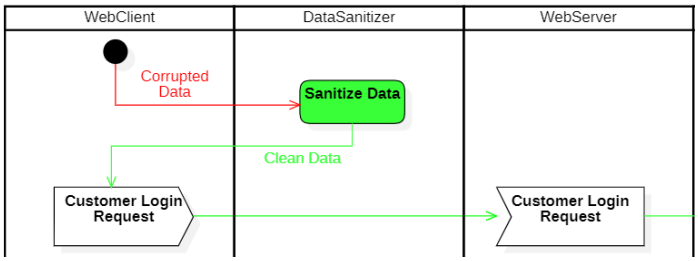


Figure 6.8: Modified Activity Flow to Protect the Expected Entry Points of thee OSM System

through data corruption attacks that involve direct memory system access. As we mentioned with Algorithm 2, if the goal of the designer is to protect the expected entry points of a system, Algorithm 3 accomplishes this task. The CPP value of the system will still be reduced, and the system's security level will increase in turn. However, a designer may opt to select one of the other recommendations if they are concerned about insider attacks that can bypass expected entry points to propagate corruption throughout the remainder of the system.

6.4.4 Aggregating the Mitigation Location Recommendations

Now that all of the support actions have been explained, the final step is to summarize the recommendations. Due to Dubhe utilizing multiple concurrent processes to determine the optimal locations for data sanitizers within a system, the next step from Figure 6.1 does not occur until all three processes have written their analysis results to a thread-safe Python list. Once this is done, Dubhe prepares a designer-readable summary of the analysis results, including the names of elements in their UML activity diagram to help with the placement of a data sanitizer. Dubhe also includes a brief description highlighting why a designer may be interested in each recommendation,

but as stated before this does not include any sort of suggestion or promotion of one recommendation over another.

While it would also be possible for Dubhe to automatically modify the submitted XMI to place a data sanitizer for designers, this has the risk of removing the designer's agency. Designers may wish to make use of Dubhe without the desire to place a data sanitizer, or they may take the suggestions for placement and modify them to account for their specific security requirements. These factors contributed to our decision to omit such a feature from Dubhe and this is a topic we discuss more in Section 8.3.

6.5 Calculating the CPP for a System

Now that all flows have been collected and mitigation placement suggestions have been generated, Dubhe moves to perform the final calculations needed to determine a system's CPP. As mentioned in Section 5.6, all information required to determine a system's average CERI and CPP is collected before final metric calculations occur. In other words, a system's average CERI and CPP are calculated concurrently within Dubhe, mapping to step 4 in Figure 4.5. For the CPP calculation, Dubhe accesses the previously created list containing all system flows. This list is traversed, with Dubhe calling `len()` on each flow to determine its length which is then summed together. Once all flows have had their lengths determined, Dubhe calls `len()` on the list that contains the system flows to determine how many flows were present in the system. Afterwards, a simple division between the sum of the lengths and the total number of flows yields the CPP for a system. Once both the CPP and CERI metrics

have been calculated for a given system, Dubhe moves to the next and final high-level step from Figure 4.5 and begins generating an analysis report, which will be explored in the next chapter.

6.6 Chapter Summary

In this chapter, we presented and walked through the approach Dubhe uses to determine a system's CPP. We discussed how system flows are gathered accounting for mitigations that may split corruptible flows. We also introduced data corruption attacks and explained how data sanitizers can be optimized to more effectively harden systems. To demonstrate this, we highlighted three scenarios Dubhe considers to generate locations for data sanitizers that target varying security objectives. Lastly, we explained how the results of the analysis are aggregated and covered how the final CPP value of a system is determined. In the next chapter, we present a full tool overview of Dubhe and cover its application with a scenario walkthrough.

Chapter 7

Dubhe: A Behavioural Security Posture Analysis Tool

In this chapter, we formally present Dubhe, a tool we developed to automate the process of determining a system's BSP. Section 7.1 outlines the features and GUI of Dubhe. Section 7.2 presents a sample design scenario where specific security goals are achieved using Dubhe. Section 7.3 discusses the results of the scenario walkthrough and highlights some key takeaways. Lastly, Section 7.4 concludes and summarizes the chapter.

7.1 An Overview of Dubhe

Dubhe is an automated tool written in Python that can be used to determine a system's BSP. Using the Flask framework, Dubhe offers both a Command-Line Interface (CLI) for terminal use and a GUI that can be accessed through modern web browsers [105]. We developed Dubhe intending to create an open-source security analysis tool that would be easily accessible to designers. To meet this goal, we ensured that Dubhe could be properly containerized using Docker to enable a future deployment to the Compass toolkit, an online platform of security analysis tools supported by the CyberSEA Research Lab at Carleton University [106, 107]. As Dubhe is completely open source, its code repository can be publicly accessed at `https://gitlab.com/CyberSEA-Public/dubhe`. In the following subsections, we will highlight the main features of Dubhe before exploring a use case scenario where a designer uses Dubhe to improve the security level of their system.

7.1.1 The "New Report" Page

Upon proceeding from the landing pages, users will be met with Dubhe's "New Report" page, shown in Figure 7.1.

This page is designed to enable the receipt of XMI files up to version 2.5.1. These files capture a user's UML activity diagram, enabling the analysis approach described in Section 4.4. Should the submitted XMI file pass all required parsing checks, Dubhe immediately begins to execute all of the steps outlined in the approach overview highlighted in Figure 4.5. While developing Dubhe, we took steps to ensure that

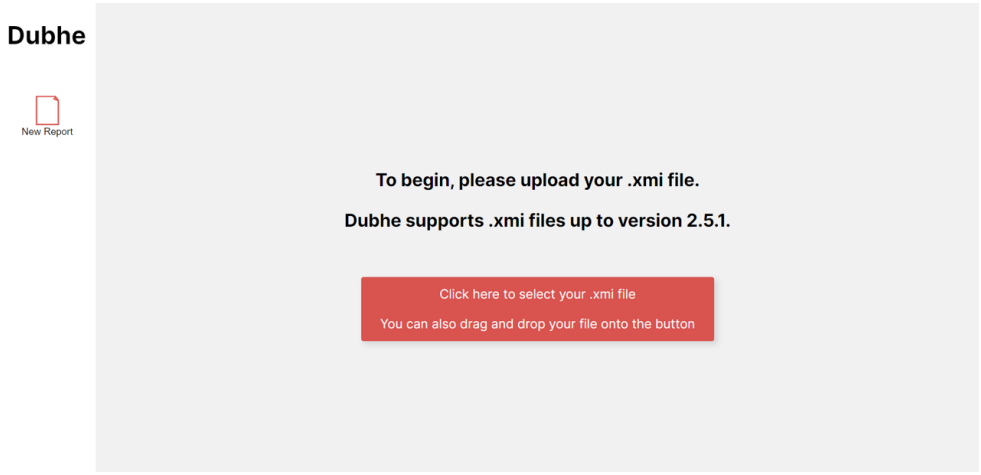


Figure 7.1: The "New Report" Page of Dubhe

the analysis activities required to determine a system's BSP could be completed in a reasonable amount of time. We achieved this by using concurrent programming techniques to simultaneously determine a system's average CERI and CPP.

Once analysis is complete, users can explore the results using two new buttons that appear on the sidebar, enabling navigation to a page that covers the analysis results from a high-level overview and another page that provides detailed recommendations to improve the BSP of the analyzed system.

7.1.2 The "Analysis Highlights" Page

Dubhe's "Analysis Highlights" page becomes accessible following the successful receipt and analysis of a user's XMI file. This page acts as a high-level analysis overview and serves as a dashboard that users can reference to gather important information about the security level of their system. We demonstrate the expected layout of the "Analysis Highlights" page in Figure 7.2.

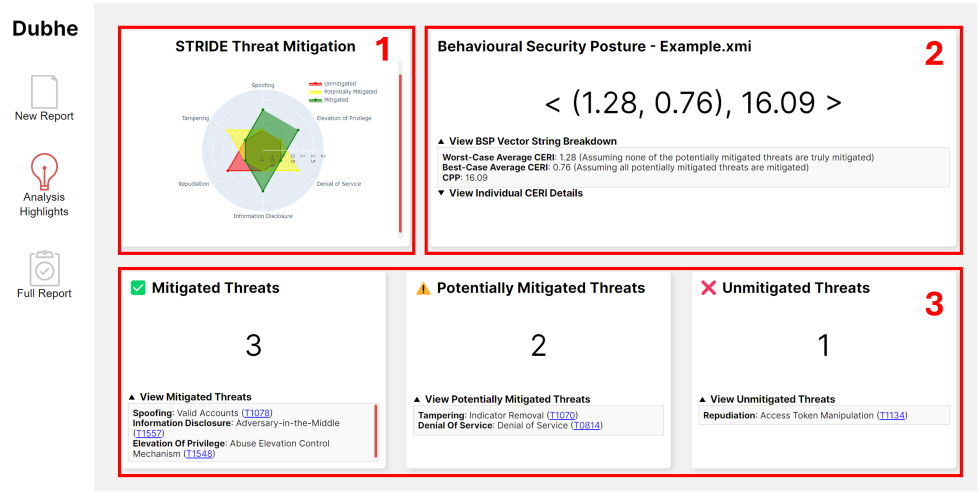


Figure 7.2: The "Analysis Highlights" Page of Dubhe

The "Analysis Highlights" page is separated into three distinct sections, denoted by the red boxes on Figure 7.2. In the section labelled '1' at the top left of the page, we include an interactive Plotly chart that visually captures a system's threat landscape based on the categories of STRIDE [108]. Each colour in the graph represents a different threat categorization. Green is used to indicate threats that have been detected as mitigated by Dubhe. Yellow is used for potentially mitigated threats and red is used for unmitigated threats. Dubhe determines these categorizations following the process explained in Section 5.4. In our current iteration of Dubhe, we only have one representative STRIDE per category. If more threat patterns were to be added in the future, this graph would provide a quick visual representation of the categories that contain the most unmitigated, potentially mitigated, and mitigated threats. This could serve to help prioritize security tasks should an organization have security requirements targeting specific categories of STRIDE.

In the section labelled '2' in Figure 7.2, a system's BSP is displayed, captured as a vector. As seen in the expanded drop-down text shown in Figure 7.2, the first value

of the vector represents the worst-case average CERI for the system. This value is calculated assuming none of the identified potentially mitigated threats have truly been mitigated. The second value in the vector is the best-case average CERI for the system, where the assumption is flipped to assume that all potentially mitigated threats have been mitigated. The third and final value in the vector is a system's CPP, determined using the approach described in Section 6.1. This part of the page also has a drop-down that is used to list every identified critical element present in a designer's system, alongside the specific worst-case and best-case CERI values for each of these critical elements.

Lastly, in the section labelled '3' in Figure 7.2, identified threats are grouped by their assigned detection category as determined by Dubhe. In the "Analysis Highlights" page, only the threats categorized as mitigated, potentially mitigated, and unmitigated are displayed. For each listed threat within the three categories, the relevant MITRE ATT&CK technique listing is included as a clickable hyperlink.

7.1.3 The "Full Report" Page

Dubhe's "Full Report" page also becomes accessible following the successful receipt and analysis of a user's XMI file. Similarly to the "Analysis Highlights" page, the "Full Report" page is divided into multiple distinct sections. We highlight the expected layout of the "Full Report" page in Figure 7.3.

In the section labelled '1' in Figure 7.3, users can read about suggested mitigations to threats categorized as either potentially mitigated or unmitigated. We include

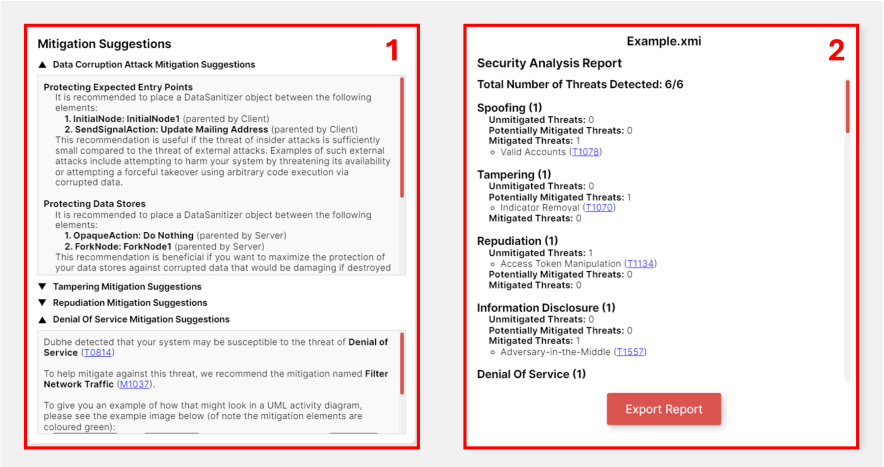


Figure 7.3: The "Full Report" Page of Dubhe

hyperlinks to the relevant MITRE ATT&CK technique and mitigation listings for each threat, and we also provide users with an example reflecting the implementation of the suggested mitigation using the UML activity diagram syntax. Further, users can view recommended locations for data sanitizer objects for three distinct scenarios, which are generated following the process described in Section 6.4. We provide the UML type and the assigned name for the two elements Dubhe determined should surround a data sanitizer object. While we would have liked to have included images for the data sanitizer recommendations, due to the varying nature of a user's submitted system compared to the fixed threat scenarios, such a task would require additional XMI rebuilding which fell out of the scope of this work.

In the section labelled '2' in Figure 7.3, users can view a full security analysis report. The report includes the total number of identified threats compared to the total number of threats available for detection within Dubhe, the number of flows detected within a system, and an indication of whether or not the system contains a data sanitizer object. Nearly all other previously presented information that was described

throughout Section 7.1 is also included within the final security analysis report. The only element not present in the report is the STRIDE threat mitigation graph shown in the section labelled '1' from Figure 7.2 due to restrictions associated with the exporting process of the report to the Portable Document Format (PDF). Any security analysis report can be exported from Dubhe to be used as a comparison reference to observe how a system's BSP has changed between design revisions.

7.2 Analyzing the BSP of a System using Dubhe

To properly illustrate the iterative security amelioration process that can occur through the use of Dubhe, we will explore a designer-focused scenario walkthrough. In this scenario, we follow a system designer named Ali, who is working on the Online Seller of Merchandise (OSM) system previously presented in Section 6.4. This system was initially defined by Yee in [104]. For this scenario walkthrough, the OSM system definition will be extended to include security requirements that Ali is working towards satisfying while designing the behavioural view of the system.

Ali is not a security expert. They have designed systems in the past, but due to a recent security incident at the company Ali works for, upper management has updated their software design process such that it now includes security considerations in the form of security requirements. Previously, systems at Ali's company were designed without explicitly-defined security requirements, mirroring a disturbing trend pervasive throughout real-world companies [15]. While working on the design of an

upcoming OSM system for their company, Ali needs to ensure their design satisfies the following list of Security Requirements (SR):

- **SR1**: If a client request is not satisfied in 15 seconds, the system must stop servicing the request to ensure the availability of the system.
- **SR2**: Any user interacting with the system must be authenticated before access is granted.
- **SR3**: Any sensitive client information retrieved from system data stores must be encrypted before being transmitted.
- **SR4**: System data stores must be hardened against data corruption attacks.
- **SR5**: The system must be hardened against knowable repudiation threats to minimize the risk associated with false refund requests.

This list of security requirements is not meant to reflect any industry standards and has been devised with the explicit purpose of guiding this scenario walkthrough. In reality, the security requirements for a system will vary dramatically to conform to the needs of stakeholders among other contributing factors. In this scenario, Ali does their best to satisfy the list of security requirements, but they lack the expertise to satisfy SR4 and SR5. We present a slightly modified version of the OSM login activity diagram representative of Ali's design efforts in Figure 7.4.

In contrast to the activity diagram shown in Figure 6.3, this updated version includes an additional timeout condition, an additional decision node to encrypt sensitive data flowing from the **CustomerDatastore**, and notes from Ali showing the element

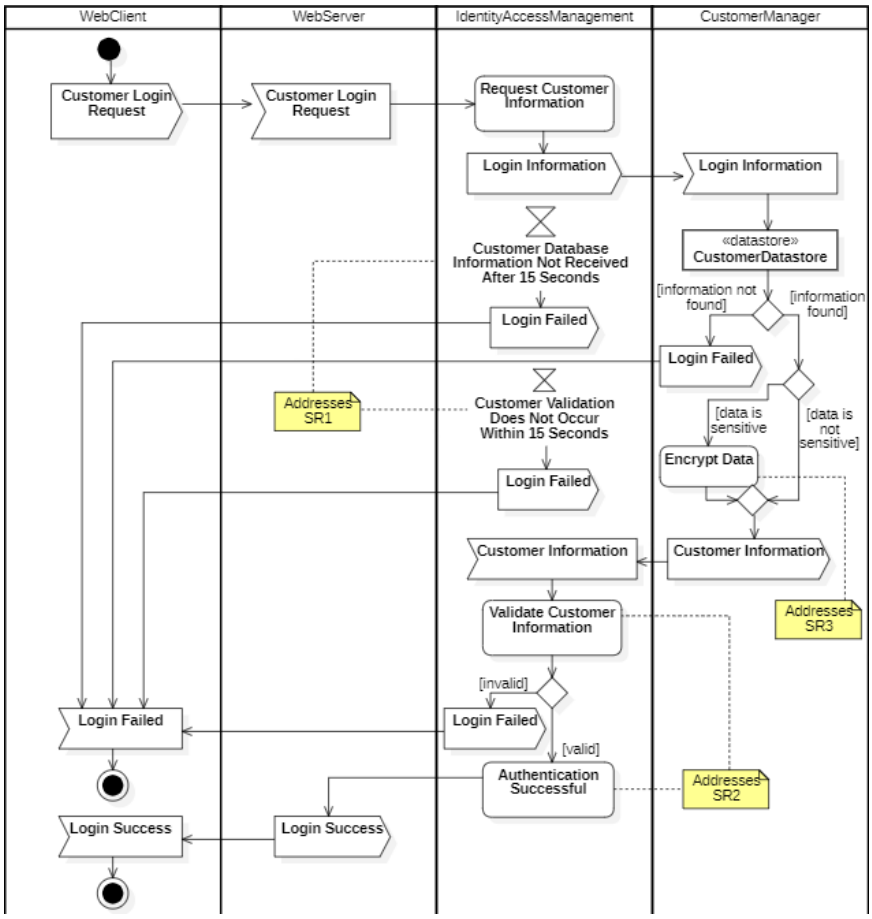


Figure 7.4: The Initial UML Activity Diagram for an Account Login Use Case

structures that relate to the list of presented security requirements. These changes result in a slightly more complex iteration of the system than what was previously presented. Regardless, both iterations accomplish the required account login use case for the OSM system.

7.2.1 System Information

Before delving further into the scenario walkthrough, we will examine each object from the OSM system that participates in the account login use case. As illustrated in Figure 7.4, the **WebClient**, **WebServer**, **IdentityAccessManagement**, **CustomerManager**, and **CustomerDatabase** components are involved in the login request flow for the OSM system. The first four are depicted as swimlanes, while the **CustomerDatabase** is represented as an embedded data store object within the **CustomerManager** swimlane.

- **WebClient** acts as the expected interface for users who interact with the OSM system. The system can be accessed via web browsers, and **WebClient** is responsible for handling user input and rendering the system to users.
- **WebServer** is responsible for mediating between **WebClient** and the rest of the OSM system. It handles input from multiple local **WebClient** components and routes requests to core components of the OSM.
- **IdentityAccessManagement** is responsible for authenticating users if authentication is required for an action requested by the **WebServer**.
- **CustomerManager** is responsible for handling requests related to the retrieval of customer information. Whenever user data is needed, **CustomerManager** will query for the data from **CustomerDatabase** and return the data to the requesting component.

- **CustomerDatabase** is a data store that contains customer information, such as their account usernames, encrypted passwords, and stored cart information. It can only be accessed via **CustomerManager**.

Having explained the objects participating in the use case, let us return to the scenario walkthrough featuring the system designer named Ali.

7.2.2 Evaluating the Initial System Design

To help ensure SR4 and SR5 are satisfied, Ali submits their activity diagram to Dubhe for analysis. Using their UML modelling tool, Ali first exports their model to an XMI file before passing that file to Dubhe for analysis. The analysis highlights from this initial system evaluation are shown in Figure 7.5.

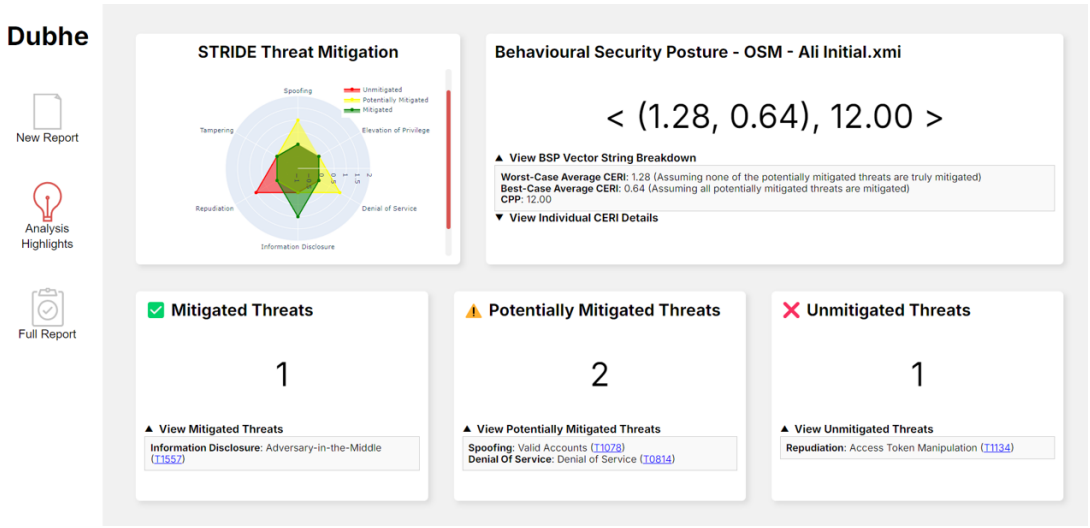


Figure 7.5: The Analysis Highlights of the OSM's Account Login Use Case

During this initial system analysis, Dubhe detected a total of four threats within Ali's design. Dubhe successfully identified the representative information disclosure risk as mitigated due to the pattern introduced by Ali to address the third listed security requirement. Dubhe also categorized the representative repudiation threat as unmitigated, which Ali is eager to address to satisfy the security requirements imposed on the OSM system. Finally, Dubhe identified two potentially mitigated threats, namely the representative spoofing and the representative denial-of-service threats. This makes sense given the patterns associated with these threats, which we presented in Section 5.3. Although it may be fairly obvious to an observer that these threats are not properly mitigated in the current iteration of the activity diagram shown in Figure 7.5, Dubhe's pattern matching cannot make the same conclusions.

To err on the side of caution, whenever Dubhe detects a mitigation pattern for a threat that fails the associated semantic equivalence checks, it categorizes that threat as potentially mitigated. Ultimately we choose to mark these threats as potentially mitigated because of the uncertainty associated with using a machine learning model to analyze the meaning of text. We would rather provide an optimistic outlook for a system's security level using both the best-case and worst-case calculations of its average CERI than simply present the worst-case average when it may not accurately reflect the system. We provide the MITRE ATT&CK technique and mitigation entries to inform the judgment of designers regardless of their backgrounds to allow them to decide whether a potentially mitigated threat is or is not truly mitigated.

In this case, Ali explores the information present in the full analysis report, alongside the information on the linked MITRE ATT&CK pages for the two potentially

mitigated threats, and decides that they are not mitigated. This means the initial iteration of the OSM system's account login use case has a final BSP vector in the form $\langle 1.62, 12.00 \rangle$, where 1.62 is the system's average CERI and 12.00 is the system's CPP. Using the best-case average CERI of the OSM system as a reference point, a value of 0.87, Ali wants to improve the OSM system to a state where the worst-case average CERI is no greater than 1.00. Additionally, Ali believes that they will be able to achieve a CPP score of less than 10.00 after attempting to harden their system against data corruption attacks.

7.2.3 Improving the BSP of the System Using Dubhe

To attempt to lower the CERI of the system, Ali will revise the design of their activity diagram to include the mitigations recommended for the potentially mitigated and unmitigated threats categorized by Dubhe. To lower the CPP of the system, Ali plans to place a data sanitizer object to interrupt the potential for corruption within the OSM system. Recalling the previously presented security requirements, Ali observes the three data sanitizer location recommendations generated by Dubhe and decides to select the recommendation associated with the protection of data stores. We show the output of the recommendation generated by Dubhe for the initial iteration of the OSM system in Listing 7.1.

For this revision, Ali redesigns the activity diagram to ensure the previously presented security requirements are satisfied without focusing on the other detected threats for the time being. Using the information provided by Dubhe during the initial analysis

```
1 ----Protecting Data Stores----
2 It is recommended to place a Data Sanitizer object between the
     following elements:
3      - AcceptEventAction: Login Information (parented by
          CustomerManager)
4      - DataStoreNode: CustomerDatabase (parented by CustomerManager)
5 This recommendation is beneficial if you want to maximize the
     protection of your data stores against corrupted data that would
     be damaging if destroyed or leaked to an attacker (e.g., data
     injection attacks).
```

Listing 7.1: The Generated Recommendation to Protect Data Stores Within the OSM System

of the system, Ali modifies the activity diagram associated with the account login use case for the OSM system, which we show in Figure 7.6.

Ali places a data sanitizer object before the **CustomerDatastore**, as recommended by Dubhe, to address SR4. Ali also adds a mitigation for the detected unmitigated repudiation threat within the **WebServer** swimlane to address SR5. This mitigation involves logging signals received from the **WebServer**, which Ali decides to store in a new data store within the **WebServer** itself. Following completion of the model redesign, Ali once again exports the activity diagram to an XMI file and resubmits the file to Dubhe. The analysis highlights from this revised system model are shown in Figure 7.7.

While this revision has satisfied all of the presented security requirements, Ali is not satisfied with the new BSP vector of $\langle(1.41, 0.83), 7.83\rangle$. While the worst-case average CERI has been reduced and the CPP has also fallen below Ali's target of 10.00, Ali recognizes further improvements could be made. Additionally, it seems that through the introduction of a mitigation for the repudiation threat, a new unmitigated tampering threat has been introduced into the system. Having reviewed

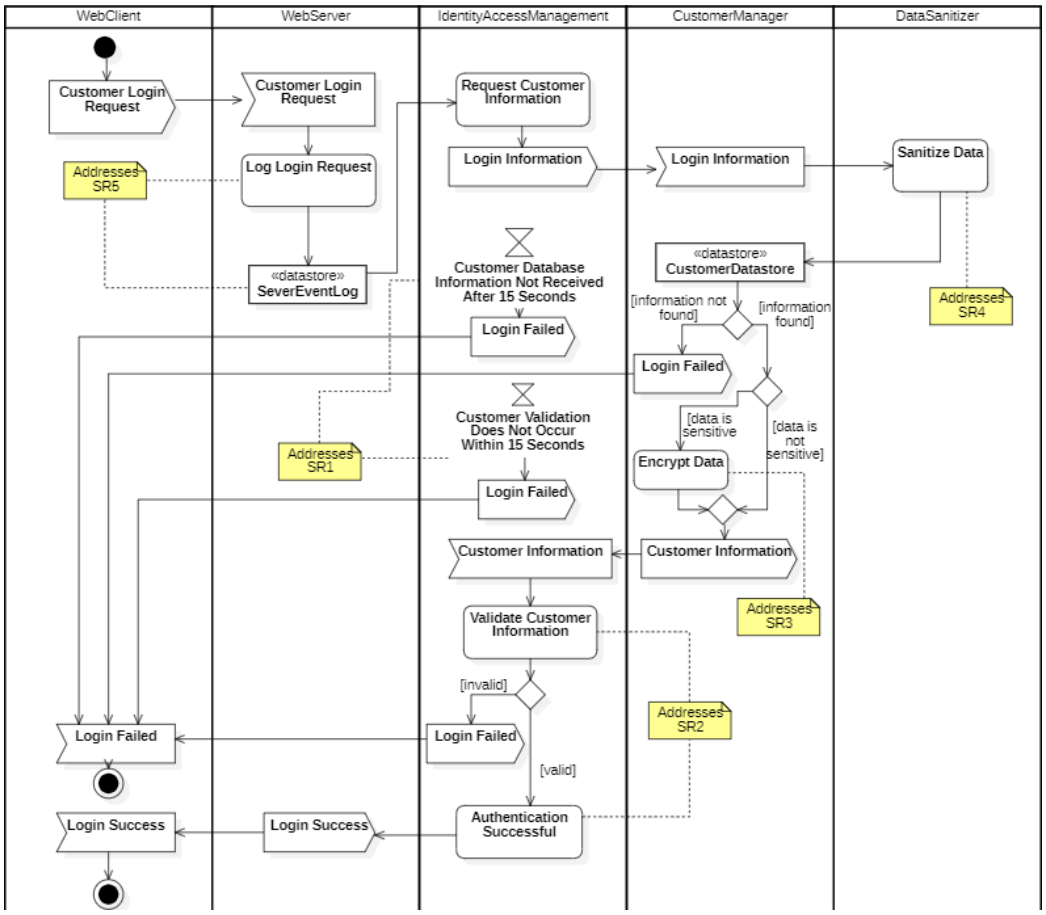


Figure 7.6: A Revised UML Activity Diagram for an Account Login Use Case

the mitigations for the remaining unmitigated and potentially mitigated threats, Ali makes a final revision to their activity diagram, shown in Figure 7.8.

As can be seen, Ali has successfully mitigated all of the threats detected by Dubhe. Through the incorporation of multiple logs, the routing of a flow through a second data sanitization action, and a modification to the authentication sequence to incorporate multi-factor authentication, the OSM system's account login use case has had its security level increased. The new BSP vector is now $\langle(0.84, 0.84), 7.40\rangle$, with the reduction in CPP stemming from an additional data sanitization action to adequately

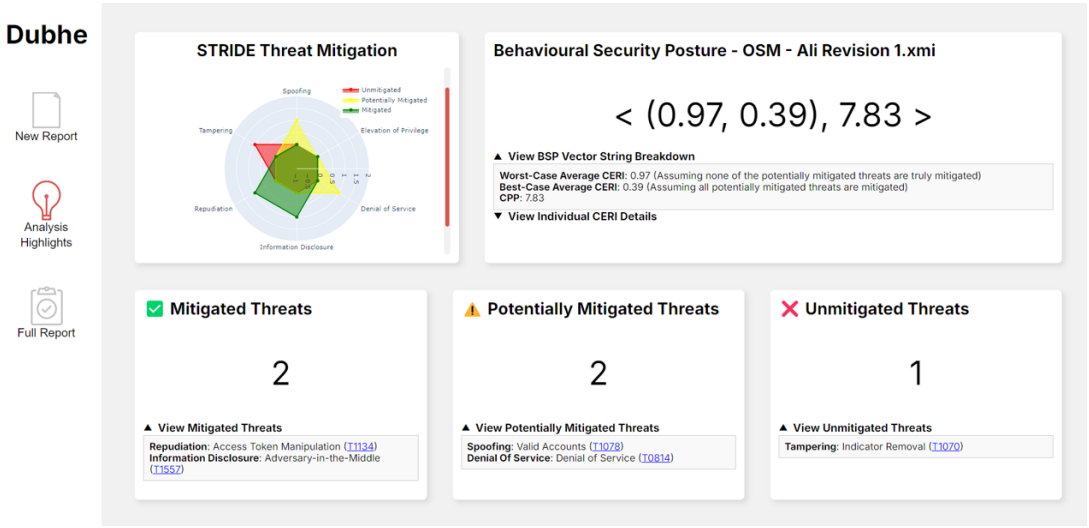


Figure 7.7: The Analysis Highlights of the Revised OSM's Account Login Use Case address the tampering threat. With this latest revision, both of Ali's goals for the system's BSP have been achieved, and the reduction in value of the system's BSP can be tied directly to the work undertaken by Ali to address threats with their system model. In other words, the reduction of the OSM system's BSP has resulted in a system model with a higher level of security. Satisfied with their results, Ali begins preparing other OSM system use cases that they will later analyze using Dubhe.

7.3 Discussion and Takeaways

Now that we have concluded with the scenario walkthrough, we wish to discuss the results and potential takeaways that can be gathered from the use of Dubhe. To begin, it is clear that each iteration in the scenario walkthrough resulted in a reduction in the system's BSP. However, this reduction did not occur free of consequences. As

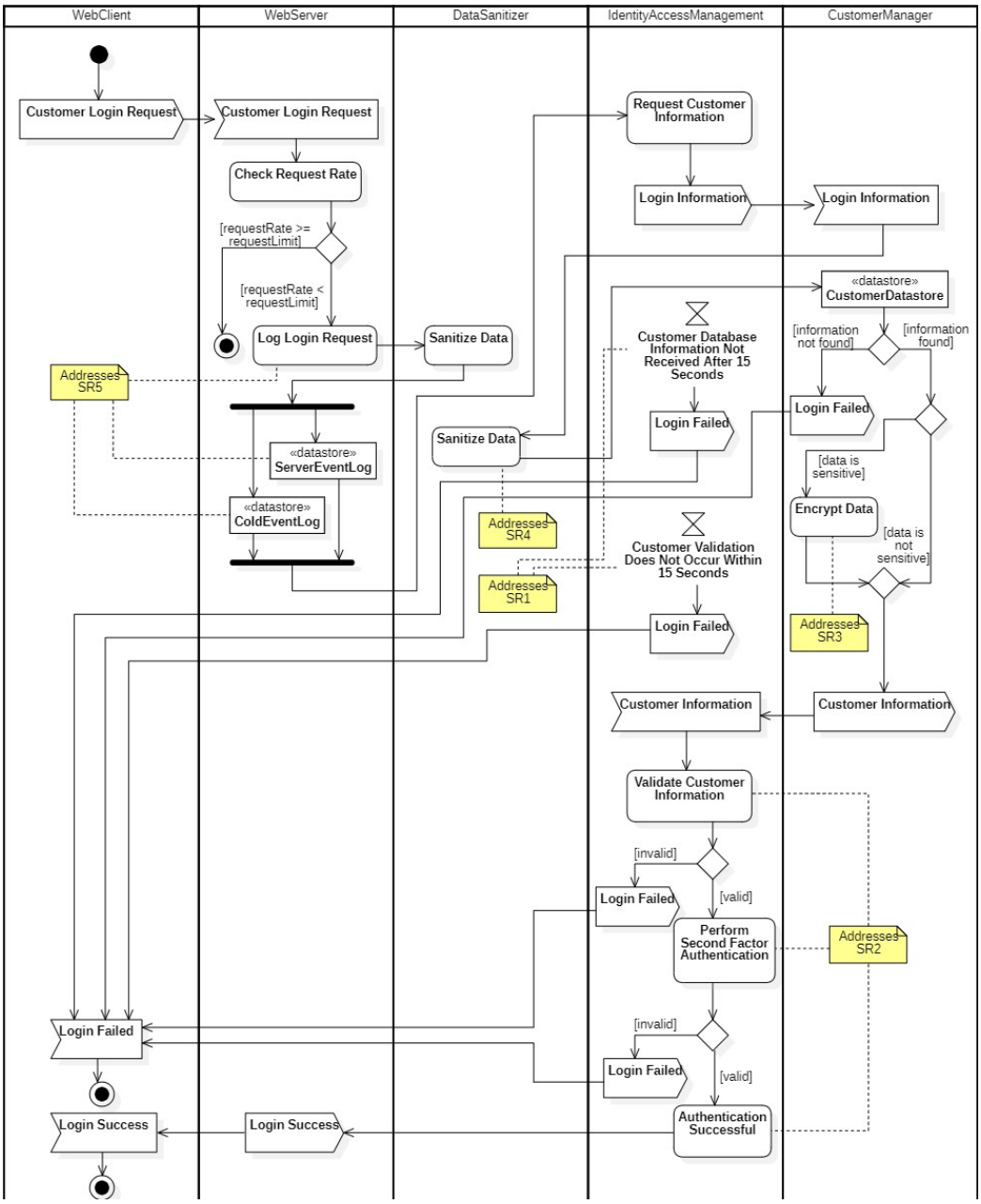


Figure 7.8: Ali's Final UML Activity Diagram for an Account Login Use Case

the BSP of the system was reduced, the number of elements and flows present within the system increased. We summarize this data in Table 7.1.

Table 7.1: OSM System Metrics Across Revisions of its Account Login Use Case

	Worst-Case Average CERI	**Best-Case Average CERI**	**CPP**	**Total # of ActivityElements**	**Total # of Flows**
Initial Design	1.28	0.64	11.43	27	7
Rev. 1	0.97	0.39	7.83	30	12
Rev. 2	0.18	0.18	7.40	40	45

As revisions were performed, more elements were added to the system. Through the addition of "ForkNode" and "DecisionNode" elements integral to many mitigation patterns, the number of flows was also increased. In fact, due to the addition of these elements when attempting to address all detected representative STRIDE threats, the total number of flows more than tripled with an overall increase of 275% between the first and second revision, while the increase in the total number of elements was only ∼33.33% between the first and second revision. This data suggests that as systems improve their BSP, their complexity in terms of elements and flows will increase. This fact may deter some designers from mitigating all detected threats. However, even if a few threats are addressed the BSP of the system will still be reduced. In turn, this results in a system with a higher security level than before the inclusion of any mitigations.

An interesting result from the scenario walkthrough is observed in the average CERI values, detailed in Appendix B. Despite detecting 4-5 threats in all iterations, the average CERI did not exceed 2.00. This is because few critical elements were involved in multiple threats. Essentially, the scenario highlighted a broad involvement of

critical elements, rather than a variation in their criticality. This finding underscores why it would be irresponsible to set a target average CERI or CPP for designers to aim for within Dubhe; these metrics should be specific to the system under analysis and are only useful for comparative purposes across system revisions. For instance, a system with an average CERI of 7.00 is not necessarily less secure than one with an average CERI of 1.00; the security assessment depends on the particular factors and behaviours identified in each system, which together define the system's BSP.

To summarize the purpose of the walkthrough, it aimed to demonstrate how Dubhe can be used to iteratively improve the security of a system's behavioural view during the early stages of the SDLC. Requiring only the XMI of a system design, designers can receive descriptive feedback detailing how to mitigate identified threats within their systems through MITRE ATT&CK technique and mitigation listings, alongside example mitigation implementations provided by Dubhe using the UML activity diagram syntax. Through reductions to a system's BSP as designs are revised, designers can observe quantifiable improvements to the security level of their system designs, which will lead to a more secure final system once it is fully developed.

While not explicitly highlighted in the scenario walkthrough, the performance of Dubhe was evaluated based on execution times, which never exceeded 5 seconds during testing. However, the execution time is directly proportional to the size of the UML activity diagram being analyzed. Since these diagrams conventionally address single-use cases, the resulting execution time remains within acceptable bounds. A full performance evaluation across different model sizes and scenarios was outside the

scope of this work, but the current results suggest that the tool is scalable within the typical constraints of UML-based system design.

7.4 Chapter Summary

Throughout this chapter, we explored Dubhe, a tool we developed to determine a system's BSP. We began with a brief overview of the features present within Dubhe alongside a presentation of its GUI. Next, we followed along with a scenario walk-through featuring a system designer named Ali as they utilized Dubhe to improve the security of the OSM system. Lastly, we discussed the results from the scenario walk-through and how iterative security improvements can be quantified and visualized using Dubhe. In the next chapter, we will revisit the objectives of this work before concluding the thesis.

Chapter 8

Conclusion

In this thesis, we presented an approach that can be integrated within the early stages of the SDLC to enable designers to improve the security of their systems by analyzing their behavioural view. To do this, we defined Behavioural Security Posture (BSP) and derived two sound security metrics that can be used to inform a system's BSP in Chapter 4. In Chapter 5, we outlined an approach to be able to detect threat patterns within the XMI representations of UML activity diagram. Using this approach, we demonstrated how a system's average CERI could be determined, informing part of a system's BSP. This was complemented by the approach to determine a system's CPP presented in Chapter 6. Through the identification of flows in a system's XMI representation, not only did we completely inform a system's BSP, but we also extended the approach to optimize the locations of mitigations under different scenarios. Finally, in Chapter 7, we presented Dubhe, our tool-based solution to be able to automate the determination of a system's BSP given nothing but an XMI file as input. Using

a scenario walkthrough on an OSM system, we demonstrated how Dubhe supports designers through guided mitigation suggestions to iteratively improve both the BSP and consequently the security level of their early system designs.

8.1 Research Outcomes

Throughout this thesis, we presented different approaches that can be utilized to inform the BSP of a system, and we can revisit the research questions initially posed in Section 1.4.

RQ1: How can we define a system's behavioural security posture in terms of quantitative and sound security metrics?

- This was the main focus of Chapter 4. In Section 4.1, we identified that a system's behavioural view is quantified by both the elements and flows that are linked in a temporal progression to define a system's behaviour. As such, to ensure that a system's BSP is quantitatively represented, it must be informed by sound security metrics that are well-defined, progressive, and meaningful. These metrics must also be determined using the information that defines a system's behaviour. We accomplished this by deriving and demonstrating the soundness of two novel security metrics. One targets the elements of a system's behavioural view known as CERI which we defined in Section 4.2, and the other targets the flows of a system's behavioural view known as CPP which we defined

in Section 4.3. When taken together, both the average CERI and CPP values for a system are used to inform the system's BSP.

RQ2: In what ways can the modification of behavioural design artifacts, informed by a system's behavioural security posture, lead to measurable improvements in system security against different types of threats?

- This was addressed in both Chapter 5 and Chapter 6. In Section 5.4, we explained how through the introduction of appropriate mitigation patterns to address representative STRIDE threats with behavioural design artifacts, the average CERI of a system will be reduced. Similarly, in Section 6.2 we explored how interruption to corruptible flows through the implementation of data sanitizer objects can quantitatively harden a system against corruption-related threats through the reduction of its calculated CPP. By reducing a system's average CERI and CPP through the implementation of mitigations, systems can attain a higher level of security.

RQ3: How can both the determination of a system's behavioural security posture and the suggested set of improvements based on the calculation of said posture be automated to best support designers in ameliorating the security of their systems?

- This was addressed in Chapter 7. To effectively support designers, we must develop and provide tools that are both easy to use and do not significantly interrupt the expected activities of designers during the early phases of the SDLC [47]. To that end, we developed our automated BSP analysis tool named Dubhe in

Section 7.1, which we presented in Section 7.1. Through a scenario walkthrough in Section 7.2, we demonstrated how mitigation suggestions are determined and communicated back to designers through automated XMI analysis. When implemented, these mitigation suggestions result in a reduction of a system's calculated BSP, yielding an improvement to the overall security level of the system.

8.2 Limitations

Although our work addressed our problem statement and answered our research questions, certain limitations remain in our approach to ensure the scope of this thesis remained manageable.

Pattern Naming Conventions: In developing the threat and mitigation patterns used in this work, which are all explicitly presented in their .dubhe formats in Appendix A, we encountered challenges in selecting action names that are both descriptive and intuitive for designers. For example, names like "Sanitize Data", "* Log", "Encipher Data", "Check Message Rate", and "Lower Permission" were chosen for their clarity and brevity. However, these may not fully align with the terminology commonly used in industry or academic settings, potentially leading to misunderstandings. To address this, we used wildcard terms like "*", but further improvements could involve refining action names, allowing for customizable terminology options, or incorporating a list of alternative names for each action. This list approach could

broaden the semantic matching capability, enabling the system to recognize a wider range of equivalent terms that designers might use.

Mitigation Labelling and Reusability: Current, Dubhe uses predefined mitigation pattern suggestions that may not consider existing labels or mitigations already applied within the system. This could result in redundancy or unnecessary complexity. A more advanced approach would involve detecting and reusing existing labels, allowing the system to reference these instead of introducing new ones. These could then be used to automatically generate example UML activity diagrams with these existing labels to further illustrate how mitigations can be incorporated into a designer's existing system, further improving the ease of use of Dubhe.

Graph Matching and Query Language Considerations: Our approach to pattern matching utilizes a list of lists structure, where each list is designed to capture one branch of a decision node. This method effectively represents multiple branches between a fork and a join by treating each path as a distinct sequence to be matched. However, while this approach is tailored to our specific needs, it may still have limitations when dealing with more complex graph structures that could be more naturally handled by graph-matching technologies or query languages like XQuery, Neo4J, and SPARQL [109, 110, 111]. These technologies were considered during development, but they did not fully align with the requirements of our UML-based model analysis. As a result, our approach, though novel, may face challenges in scenarios requiring more advanced or flexible graph representations for the behavioural view of systems.

Handling of Loops and Concurrency: In the current implementation of Dubhe, loops are traversed a single time when detected, simplifying the analysis but potentially overlooking the full implications of iterative processes. Concurrency, such as that introduced by forks, is handled similarly to decision and merge node structures, with each concurrent path being treated as a separate flow. However, Dubhe does not account for complexities like race conditions, which could impact the accuracy of security assessments in highly concurrent environments. These limitations suggest areas for future enhancement to more comprehensively address the nuances of loops and concurrent processes.

Further Validation of Patterns: As of now, our work has not undergone empirical testing with industry-standard UML activity diagrams or feedback from real users. While the patterns and metrics were derived from reputable MITRE sources and based on theoretical frameworks, their representation as UML activity diagrams has not been empirically validated. This lack of testing introduces potential risks regarding the generalizability and practical applicability of the patterns, especially when applied to diverse system models beyond our examples. Future work should focus on validating Dubhe in real-world scenarios through testing with various UML activity diagrams and gathering feedback from designers, ensuring the system aligns with industry needs and increasing its overall usability.

Inconsistent XMI Generation Across Modelling Tools: The approach outlined in this work has been tested using different XMI exporters from various UML tools, including Star UML, Papyrus, and Microsoft Visio [112, 69, 71]. While Star UML produced usable XMI that was compatible with our pattern-matching algorithms,

other tools, such as Papyrus and Microsoft Visio, generated XMI files that were not directly usable. This limitation highlights the challenges of relying on XMI as an interchange format, as the variability in how different tools generate XMI can lead to inconsistencies in compatibility. These discrepancies limit the broader applicability of the approach across multiple UML tools, suggesting that future work may need to explore more robust handling or standardization of XMI exports to ensure cross-tool compatibility.

8.3 Open Research Areas

While we have answered all of our previously presented research questions, there nonetheless exists an opportunity to improve and expand on the contributions presented in this work.

Support Automatic XMI Editing: As it stands, the current implementation of Dubhe only provides written recommendations to designers. If we aim to maximize support for designers and enhance the likelihood of widespread adoption of our developed approaches, it is essential to automate as many activities as possible [8]. Since a designer will be providing an XMI file as input to Dubhe, the possibility of modifying this XMI file automatically to incorporate the suggested mitigations is a feasible endeavour. This feature should ensure designers retain the ability to decide whether or not they want this automatic XMI editing to occur. Preserving their agency is crucial, as designers may be dissuaded from using Dubhe if they feel their judgment will be overridden by the results of the security analysis. Assuming this stipulation

is properly considered, such a feature would serve to minimize the required designer responsibilities even further, ultimately making the tool more attractive to potential users.

Investigate Additional Security Metrics Applicable to a System's Behavioural View: While we derived two novel security metrics, one that targets the elements and another that targets the flows that are used to define a system's behaviour, a system's overall security level can likely be informed by additional metrics. For example, in Section 7.3 we discussed complexity in terms of the number of elements and flows present in a diagram. It could be beneficial to capture this complexity and quantify it in terms of a system's behaviour to enable Dubhe to provide additional recommendations regarding the improvement of a system's security level. The amount of data from the XMI representation of a system is expansive, and we have only used a small portion of that data in this work. Even more security metrics can likely be determined through the creative use of XMI data, enabling further system security improvement opportunities.

Perform Usability and Accessibility Studies for Dubhe: While outside the scope of this work, Dubhe's usability and accessibility from a human-interaction perspective must be studied to maximize potential adoption. If users struggle with certain aspects of the approach presented in this work, a usability study would reveal those issues and allow for refinements that would broadly benefit users of the tool. This usability study could then be repeated to further improve the user experience associated with using Dubhe, resulting in a robust security analysis tool for

users. Alongside usability, determining the accessibility of Dubhe must also be explored. Security tools should be accessible to all and those that rely on accessibility tools must not be excluded as potential users. Ensuring Dubhe conforms to Web Content Accessibility Guidelines (WCAG) 2.1 standards alongside data from accessibility studies should be used to refine the tool even further to ensure a system's BSP can be determined by anyone [113].

Add Additional Threat and Mitigation Pattern Pairs to Dubhe: As we discussed in Section 5.3, we chose to focus on an approach targeting a breadth of threats over a depth of single types of threats. That being said, the functionality to support additional threats and mitigation patterns within .dubhe files already exists. To cover a wider area of the knowable threat landscape, additional threat and mitigation patterns can be conceived and added to the .dubhe files by exploring the same MITRE data sources used in this work. This could also include expanding to different types of attacks, such as ransomware and social engineering attacks, which often exploit human factors or external vulnerabilities. By creating additional patterns for these attacks, designers can harden their system against even more threats that may exist within their system design, providing the means to further improve the security of their system through a reduction in its average CERI.

Investigate the Relationship Between a System's Structural Security Posture and its BSP: Previous work by Samuel [17] defined a system's security posture given its structural view. We continued that work by defining a system's security posture given its behavioural view, however a connection between them has yet to be explored. Ideally, the suggestions derived from the analysis of one view should

agree with the suggestions generated from the analysis of the other. Should this be explored, the opportunity to determine additional metrics using data from multiple system views could enable security improvements that may not be possible otherwise.

Define a System's Functional Security Posture: Now that a system's structural and behavioural security postures have been defined, it is only natural to extend this to a system's functional view. Defining a system's functional security posture enables designers to understand a complete picture of a system's security throughout the evolution of the system across the early phases of the SDLC. The functional view presents additional sources of data to pull from, such as interface definitions and deployment overviews. Further, the relationship between a system's functional security posture and its structural and behavioural security postures could be investigated enabling the creation of additional security metrics.

Session Saving in Dubhe: To enhance the usability of Dubhe, it would be desirable to implement a mechanism for persistent tracking of mitigations. Currently, Dubhe does not provide support to designers that enables them to mark potentially mitigated threats as either mitigated or unmitigated, based on their analysis and judgment. Introducing a feature that remembers such user-defined mitigation classifications across multiple iterations of the analysis could significantly streamline the workflow for designers. This could be achieved through session tracking mechanisms, such as cookies or an account-based system, allowing Dubhe to maintain a consistent record of flagged mitigations. Such an enhancement would improve usability, reduce redundant assessments, and better support the iterative nature of system design and security analysis.

Adapting the Approach to BPMN: While this work focuses on UML activity diagrams, it is worth considering the potential for adapting our approach to Business Process Model and Notation (BPMN). BPMN is often regarded as more popular in certain industries for modelling business processes and workflows, and aligning our security analysis techniques with BPMN could increase the applicability and adoption of Dubhe [114]. Future research could explore the feasibility of extending the pattern matching and mitigation suggestions to BPMN diagrams, ensuring the tool remains relevant to a wider range of system designers and industries.

8.4 Final Remarks

When the conversation of software security arises, the current norm is to think in terms of completed systems. Whether that be network security, exploiting vulnerabilities in running software, or a plethora of other options, the conversation surrounding security rarely focuses on the early design of systems. However, we hope to see this perspective change in the future. While our work is only a step towards that goal, considering security during the early stages of system design will enable the creation of more secure systems [115]. The tool we have developed can be used during the SDLC design phase to improve a system's security level. As more approaches targeting this improvement are developed, the more we may begin to see the conversation about system security evolve into a more accurate and complete meaning of all it truly encompasses.

Bibliography

[1] Object Management Group. Unified Modeling Language. `https://www.omg.org/spec/UML/2.5.1/PDF`, 2017. Version 2.5.1.

[2] Eric J Braude and Michael E Bernstein. *Software engineering: modern approaches*. Waveland Press, 2016.

[3] Institute of Electrical International Organization for Standardization, International Electrotechnical Commission and Electronics Engineers. ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes. *ISO/IEC/IEEE 12207:2017(E) First edition 2017-11*, pages 1–157, 2017.

[4] Joe Kim. Cyber-security in government: reducing the risk. *Computer Fraud & Security*, 2017(7):8–11, 2017.

[5] Kane J Smith, Gurpreet Dhillon, and Lemuria Carter. User values and the development of a cybersecurity public policy for the IoT. *International Journal of Information Management*, 56:102123, 2021.

[6] Malik Imran Daud. Secure software development model: A guide for secure software life cycle. In *Proceedings of the international MultiConference of Engineers and Computer Scientists*, volume 1, pages 17–19, 2010.

[7] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[8] Kaiying Luan, Ragnhild Halvorsrud, and Costas Boletsis. Evaluation of a Tool to Increase Cybersecurity Awareness Among Non-experts (SME Employees). In *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*, pages 509–518, 2023.

[9] Michael J Assante. Infrastructure protection in the ancient world. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2009.

[10] Microsoft. The STRIDE Threat Model. `https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN`, Nov 2009.

[11] Matteo Iaiani, Alessandro Tugnoli, Sarah Bonvicini, and Valerio Cozzani. Analysis of cybersecurity-related incidents in the process industry. *Reliability Engineering & System Safety*, 209:107485, 2021.

[12] Microsoft. Microsoft Outlook Elevation of Privilege Vulnerability. `https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2023-23397`, Mar 2023.

[13] Google. CVE-2023-3079. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-3079`, June 2023.

[14] Apple. CVE-2023-32435. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-32435`, June 2023.

[15] Marshall A Kuypers, Thomas Maillart, and Elisabeth Paté-Cornell. An empirical analysis of cyber security incidents at a large organization. *Department of Management Science and Engineering, Stanford University, School of Information, UC Berkeley*, 30, 2016.

[16] Sasha Romanosky. Examining the costs and causes of cyber incidents. *Journal of Cybersecurity*, 2(2):121–135, 2016.

[17] Joe Frederick Samuel. *A Data-Driven Approach to Evaluate the Security of System Designs*. PhD thesis, Carleton University, 2021.

[18] Bandar Alshammari, Colin Fidge, and Diane Corney. Security Metrics for Object-Oriented Class Designs. In *2009 Ninth International Conference on Quality Software*, pages 11–20. IEEE, 2009.

[19] Irshad Ahmad Mir and SMK Quadri. Analysis and Evaluating Security of Component Based Software Development: A Security Metrics Framework. *International Journal of Computer Network and Information Security*, 4(11):21, 2012.

[20] Pratyusa K. Manadhata and Jeannette M. Wing. An Attack Surface Metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[21] Joe Samuel, Jason Jaskolka, and George OM Yee. Analyzing Structural Security Posture to Evaluate System Design Decisions. In *2021 IEEE 21st International*

Conference on Software Quality, Reliability and Security (QRS), pages 8–17. [IEEE], 2021.

[22] Wayne Jansen. *Directions in Security Metrics Research*. Diane Publishing, 2010.

[23] George OM Yee. Security Metrics: An Introduction and Literature Review. *Computer and Information Security Handbook*, pages 553–566, 2013.

[24] Mohammad Alshayeb, Haris Mumtaz, Sajjad Mahmood, and Mahmood Niazi. Improving the Security of UML Sequence Diagram Using Genetic Algorithm. *IEEE Access*, 8:62738–62761, 2020.

[25] Istehad Chowdhury, Brian Chan, and Mohammad Zulkernine. Security Metrics for Source Code Structures. In *4th International Workshop on Software Engineering for Secure Systems*, SESS 2008, pages 57–64. ACM, 2008.

[26] Allan J Albrecht. Measuring application development productivity. In *Proc. joint share, guide, and IBM application development symposium*, pages 83–92, 1979.

[27] Capers Jones. Function points as a universal software metric. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–27, 2013.

[28] George O. M. Yee. Improving the Derivation of Sound Security Metrics. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1804–1809, 2022.

[29] George O.M. Yee. Designing Sound Security Metrics. *International Journal of Systems and Software Security and Protection (IJSSSP)*, 10(1):1–21, jan 2019.

[30] Seema Gupta Bhol, JR Mohanty, and Prasant Kumar Pattnaik. Taxonomy of cyber security metrics to measure strength of cyber security. *Materials Today: Proceedings*, 80:2274–2279, 2023.

[31] Matthew Zaber and Suku Nair. A framework for automated evaluation of security metrics. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–11, 2020.

[32] Alex Ramos, Marcella Lazar, Raimir Holanda Filho, and Joel JPC Rodrigues. Model-based quantitative network security metrics: A survey. *IEEE Communications Surveys & Tutorials*, 19(4):2704–2734, 2017.

[33] Andreas Aigner and Abdelmajid Khelil. A Benchmark of Security Metrics in Cyber-Physical Systems. In *2020 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*, pages 1–6. IEEE, 2020.

[34] Zachary A Collier, Mahesh Panwar, Alexander A Ganin, Alexander Kott, and Igor Linkov. Security Metrics in Industrial Control Systems. *Cyber-security of SCADA and other industrial control systems*, pages 167–185, 2016.

[35] Daniel R Thomas, Alastair R Beresford, and Andrew Rice. Security Metrics for the Android Ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 87–98, 2015.

[36] Massimiliano Albanese, Ibifubara Iganibo, and Olutola Adebiyi. A framework for designing vulnerability metrics. *Computers & Security*, 132:103382, 2023.

[37] Kartik Nayak, Daniel Marino, Petros Efstathopoulos, and Tudor Dumitraş. Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild. In *International Workshop on Recent Advances in Intrusion Detection*, pages 426–446. Springer, 2014.

[38] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Computing Surveys (CSUR)*, 49(4):1–35, 2016.

[39] Deborah J Bodeau, Richard D Graubart, Rosalie M McQuaid, and John Woodill. Cyber Resiliency Metrics, Measures of Effectiveness, and Scoring: Enabling Systems Engineers and Program Managers to Select the Most Useful Assessment Methods. *MITRE Corp Bedford Ma Bedford United States*, 2018.

[40] Georgios Bakirtzis, Brandon J. Simon, Aidan G. Collins, Cody Harrison Fleming, and Carl R. Elks. Data-Driven Vulnerability Exploration for Design Phase System Analysis. *IEEE Systems Journal*, 14(4):4864–4873, 2020.

[41] Brian M Bowen, Ramaswamy Devarajan, and Salvatore Stolfo. Measuring the Human Factor of Cyber Security. In *2011 IEEE International Conference on Technologies for Homeland Security (HST)*, pages 230–235. IEEE, 2011.

[42] Samuel Moses and Dale C Rowe. Physical security and cybersecurity: Reducing risk by enhancing physical security posture through multi-factor authentication and other techniques. *International Journal for Information Security Research (IJISR)*, 6(2):667–676, 2016.

[43] Nicola d'Ambrosio, Gaetano Perrone, and Simon Pietro Romano. Including insider threats into risk management through Bayesian threat graph networks. *Computers & Security*, 133:103410, 2023.

[44] Jan Jürjens and Pasha Shabalin. Tools for secure systems development with UML. *International Journal on Software Tools for Technology Transfer*, 9(5-6):527–544, 2007.

[45] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In *International Conference on The Unified Modeling Language*, UML 2002, pages 412–425. Springer, 2002.

[46] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *International Conference on the Unified Modeling Language*, pages 426–441. Springer, 2002.

[47] Shouki A Ebad. An Exploratory Study of Why UMLsec Is Not Adopted. In *ICISSP*, pages 357–364, 2022.

[48] Alfonso Rodríguez, Eduardo Fernández-Medina, and Mario Piattini. Capturing Security Requirements in Business Processes Through a UML 2.0 Activity Diagrams Profile. In *International Conference on Conceptual Modeling*, pages 32–42. Springer, 2006.

[49] Guttorm Sindre. Mal-Activity Diagrams for Capturing Attacks on Business Processes. In *Requirements Engineering: Foundation for Software Quality*, REFSQ 2007, pages 355–366. Springer, June 2007.

[50] Samir Ouchani, Otmane Ait Mohamed, and Mourad Debbabi. A security risk assessment framework for SysML activity diagrams. In *2013 IEEE 7th International Conference on Software Security and Reliability*, pages 227–236. IEEE, 2013.

[51] JC Reynold. COGENT–PROGRAMMING MANUAL. Technical report, Argonne National Lab., Ill., 1965.

[52] Abdulla Hussain, Azlinah Mohamed, and Suriyati Razali. A review on cybersecurity: Challenges & emerging threats. In *Proceedings of the 3rd International Conference on Networking, Information Systems & Security*, pages 1–7, 2020.

[53] The MITRE Corporation. CAPEC - Common Attack Pattern Enumerations and Classifications. `https://capec.mitre.org/`.

[54] The MITRE Corporation. ATT&CK - Adversarial Tactics, Techniques, and Common Knowledge. https://attack.mitre.org/.

[55] The MITRE Corporation. D3FEND - Develop, Deploy, and Defend. `https://d3fend.mitre.org/`.

[56] Ron Ross, Victoria Pillitteri, Richard Graubart, Deborah Bodeau, and Rosalie McQuaid. Developing cyber resilient systems: a systems security engineering approach. Technical report, National Institute of Standards and Technology, 2019.

[57] David A Chapin and Steven Akridge. How can security be measured. *Information Systems Control Journal*, 2(1):43–47, 2005.

[58] Debra S Herrmann. *Complete guide to security and privacy metrics: measuring regulatory compliance, operational resilience, and ROI*. Auerbach Publications, 2007.

[59] Tim Klaus. Security metrics-replacing fear, uncertainty, and doubt, 2008.

[60] FIRST. Common vulnerability scoring system: Specification document. `https://www.first.org/cvss/specification-document/`, November 2023.

[61] NIST. CVSS v3 Calculator. `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator/`, June 2019.

[62] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[63] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 4:308–320, 1976.

[64] Istehad Chowdhury, Brian Chan, and Mohammad Zulkernine. Security metrics for source code structures. In *Proceedings of the fourth international workshop on Software engineering for secure systems*, pages 57–64, 2008.

[65] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[66] J Michael Steele. *The Cauchy-Schwarz master class: an introduction to the art of mathematical inequalities*. Cambridge University Press, 2004.

[67] Philip Langer, Tanja Mayerhofer, Manuel Wimmer, and Gerti Kappel. On the Usage of UML: Initial Results of Analyzing Open UML Models. In *Modellierung 2014*, pages 289–304. Gesellschaft für Informatik e.V., Bonn, 2014.

[68] Object Management Group. XML Metadata Interchange. `https://www.omg.org/spec/XMI/2.5.1/PDF`, 2015. Version 2.5.1.

[69] The Eclipse Foundation. Eclipse Papyrus. `https://www.eclipse.org/papyrus/`, 2023. Version 6.5.0 [Software].

[70] MKLabs Co.,Ltd. StarUML. `https://staruml.io`, 2023. Version 6.0 [Software].

[71] Microsoft Corporation. Microsoft Visio. `https://products.office.com/en/visio/flowchart-software`. Version 16.0 [Software].

[72] lxml Development Team. lxml: XML and HTML with Python. `https://lxml.de/`, 2023. Version 4.9.3 [Software library].

[73] Anton Konev, Alexander Shelupanov, Mikhail Kataev, Valeriya Ageeva, and Alina Nabieva. A survey on threat-modeling techniques: protected objects and classification of threats. *Symmetry*, 14(3):549, 2022.

[74] Mouna Jouini and Latifa Ben Arfa Rabai. Threats classification: state of the art. *Handbook of Research on Modern Cryptographic Solutions for Computer and Cyber Security*, pages 368–392, 2016.

[75] The MITRE Corporation. CAPEC-151: Identity Spoofing. `https://capec.mitre.org/data/definitions/151.html`.

[76] The MITRE Corporation. Valid Accounts, Technique T1078. https://attack.mitre.org/techniques/T1078/.

[77] The MITRE Corporation. Account Use Policies, Mitigation M1036. https://attack.mitre.org/mitigations/M1036/.

[78] The MITRE Corporation. Multi-factor Authentication - Technique D3-MFA. https://d3fend.mitre.org/technique/d3f:Multi-factorAuthentication/.

[79] The MITRE Corporation. CAPEC-268: Audit Log Manipulation. https://capec.mitre.org/data/definitions/268.html.

[80] The MITRE Corporation. Indicator Removal, Technique T1070. https://attack.mitre.org/techniques/T1070/.

[81] The MITRE Corporation. Remote Data Storage, Mitigation M1029. https://attack.mitre.org/mitigations/M1029/.

[82] The MITRE Corporation. Data Inventory - Technique D3-DI. https://d3fend.mitre.org/technique/d3f:DataInventory/.

[83] The MITRE Corporation. CAPEC-196: Session Credential Falsification through Forging. https://capec.mitre.org/data/definitions/196.html.

[84] The MITRE Corporation. Access Token Manipulation, Technique T1134. https://attack.mitre.org/techniques/T1134/.

[85] The MITRE Corporation. User Account Management, Mitigation M1018. https://attack.mitre.org/mitigations/M1018/.

[86] The MITRE Corporation. Local Account Monitoring - Technique D3-LAM. https://d3fend.mitre.org/technique/d3f:LocalAccountMonitoring/.

[87] The MITRE Corporation. CAPEC-94: Adversary in the Middle (AiTM). https://capec.mitre.org/data/definitions/94.html.

[88] The MITRE Corporation. Adversary-in-the-Middle, Technique T1557. https://attack.mitre.org/techniques/T1557/.

[89] The MITRE Corporation. Encrypt Sensitive Information, Mitigation M1041. https://attack.mitre.org/mitigations/M1041/.

[90] The MITRE Corporation. Message Encryption - Technique D3-MENCR. https://d3fend.mitre.org/technique/d3f:MessageEncryption/.

[91] The MITRE Corporation. CAPEC-125: Flooding. https://capec.mitre.org/data/definitions/125.html.

[92] The MITRE Corporation. Denial of Service, Technique T0814. https://attack.mitre.org/techniques/T0814/.

[93] The MITRE Corporation. Filter Network Traffic, Mitigation M1037. https://attack.mitre.org/mitigations/M1037/.

[94] The MITRE Corporation. Network Traffic Filtering - Technique D3-NTF. https://d3fend.mitre.org/technique/d3f:NetworkTrafficFiltering/.

[95] The MITRE Corporation. CAPEC-233: Privilege Escalation. https://capec.mitre.org/data/definitions/233.html.

[96] The MITRE Corporation. Abuse Elevation Control Mechanism, Technique T1548. https://attack.mitre.org/techniques/T1548/.

[97] The MITRE Corporation. Execution Prevention, Mitigation M1038. https://attack.mitre.org/mitigations/M1038/.

[98] The MITRE Corporation. Exception Handler Pointer Validation - Technique D3-EHPV. https://d3fend.mitre.org/technique/d3f:ExceptionHandlerPointerValidation/.

[99] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. *GitHub*, 2020.

[100] Long Cheng, Hans Liljestrand, Md Salman Ahmed, Thomas Nyman, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 114–128. IEEE, 2019.

[101] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[102] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*, volume 7462 of *LNCS*, pages 86–106. Springer, 2012.

[103] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *2012 International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[104] George O. M. Yee. Reducing the Attack Surface for Private Data. In *13th International Conference on Emerging Security Information, Systems and Technologies*, SECURWARE 2019, pages 28–34, October 2019.

[105] Pallets. Flask. `https://github.com/pallets/flask/`, 2024.

[106] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux journal*, 2014(239):2, 2014.

[107] CyberSEA Research Lab. Compass. `https://compass.carleton.ca/`, 2021.

[108] Plotly Technologies Inc. Collaborative data science. `https://plot.ly`, 2015.

[109] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

[110] Justin J Miller. Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, pages 141–147, 2013.

[111] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.

[112] Minkyu Lee and C Davis. XMI extension for StarUML. `https://github.com/staruml/staruml-xmi`, 2018.

[113] Ben Caldwell, Michael Cooper, Loretta Guarino Reid, Gregg Vanderheiden, Wendy Chisholm, John Slatin, and Jason White. Web content accessibility guidelines (wcag) 2.0. *WWW Consortium (W3C)*, 290:1–34, 2008.

[114] Ivan Compagnucci, Flavio Corradini, Fabrizio Fornari, and Barbara Re. Trends on the usage of BPMN 2.0 from publicly available repositories. In *International Conference on Business Informatics Research*, pages 84–99. Springer, 2021.

[115] Jason Jaskolka. Recommendations for effective security assurance of software-dependent systems. In *Intelligent Computing: Proceedings of the 2020 Computing Conference, Volume 3*, pages 511–531. Springer, 2020.

Appendix A

Exploring .dubhe Files

In this section, we present the contents and grammar of all .dubhe files that are used by the current version of Dubhe. The contents of each file are based on the threat and mitigation pattern pairs presented in Section 5.3.

The contents of the .dubhe file that contains the representative spoofing threat mentioned in Section 5.3.1 are shown in Listing A.1.

```
%
THREAT TECHNIQUE: Valid Accounts
TECHNIQUE NUMBER: T1078
THREAT MITIGATION: Account Use Policies
MITIGATION NUMBER: M1036
DETECT PATTERN: [["*Server:AcceptEventAction", "...", "*Action",
                 "DataStoreNode"]]
MITIGATION PATTERN: [["DecisionNode", "...", "ActivityFinalNode"],
                     ["DecisionNode", "...", ("*Action",
                     "Authenticate *"), "...",
                     "DecisionNode", "...", "ActivityFinalNode"]]
MITIGATION INDEX: 3
```

Listing A.1: The Contents of the Spoofing .dubhe File

The contents of the .dubhe file that contains the representative tampering threat mentioned in Section 5.3.2 are shown in Listing A.2.

```
%
THREAT TECHNIQUE: Indicator Removal
TECHNIQUE NUMBER: T1070
THREAT MITIGATION: Remote Data Storage & Data Sanitization
MITIGATION NUMBER: M1029
DETECT PATTERN: [[("*Action", "Log *"), "...",
                 ("DataStoreNode", "* Log")]]
MITIGATION PATTERN: [[("*Action", "Sanitize Data"),
                      "...", "ForkNode", "...", "DataStoreNode",
                      "...", "JoinNode"]]
MITIGATION INDEX: 0
```

Listing A.2: The Contents of the Tampering .dubhe File

The contents of the .dubhe file that contains the representative repudiation threat mentioned in Section 5.3.3 are shown in Listing A.3.

```
%
THREAT TECHNIQUE: Access Token Manipulation
TECHNIQUE NUMBER: T1134
THREAT MITIGATION: User Account Logging
MITIGATION NUMBER: M1018
DETECT PATTERN: [["*Client:SendSignalAction", "*Server:
    AcceptEventAction"]]
MITIGATION PATTERN: [[("*Action", "Log *"), "...",
                      ("DataStoreNode", "* Log")]]
MITIGATION INDEX: -1
```

Listing A.3: The Contents of the Repudiation .dubhe File

The contents of the .dubhe file that contains the representative information disclosure threat mentioned in Section 5.3.4 are shown in Listing A.4.

```
%
THREAT TECHNIQUE: Adversary-in-the-Middle
TECHNIQUE NUMBER: T1557
THREAT MITIGATION: Encrypt Sensitive Information
MITIGATION NUMBER: M1041
DETECT PATTERN: [["DataStoreNode", "...", "SendSignalAction"]]
MITIGATION PATTERN: [["DecisionNode", "MergeNode"],
                     ["DecisionNode", ("*Action", "Encipher Data"),
                     "MergeNode"]]
MITIGATION INDEX: 1
```

Listing A.4: The Contents of the Information Disclosure .dubhe File

The contents of the .dubhe file that contains the representative denial of service threat mentioned in Section 5.3.5 are shown in Listing A.5.

```
%
THREAT TECHNIQUE: Denial of Service
TECHNIQUE NUMBER: T0814
THREAT MITIGATION: Filter Network Traffic
MITIGATION NUMBER: M1037
DETECT PATTERN: [["AcceptEventAction", "*Action"]]
MITIGATION PATTERN: [[("*Action", "Check Message Rate"),
                     "DecisionNode", "ActivityFinalNode"],
                     [("*Action", "Check Message Rate"),
                     "DecisionNode", "*Action"]]
MITIGATION INDEX: 1
```

Listing A.5: The Contents of the Denial of Service .dubhe File

The contents of the .dubhe file that contains the representative elevation of privilege threat mentioned in Section 5.3.6 are shown in Listing A.6.

```
%
THREAT TECHNIQUE: Abuse Elevation Control Mechanism
TECHNIQUE NUMBER: T1548
THREAT MITIGATION: Execution Prevention
MITIGATION NUMBER: M1038
DETECT PATTERN: [[("*Action", "Raise Permission"),
                 "...", ("*Action", "Lower Permission")]]
MITIGATION PATTERN: [["*Action", "...", "ActivityFinalNode"],
                     "InputPin", ("*Action", "Lower Permission"),
                     "...", "ActivityFinalNode"]
MITIGATION INDEX: 1
```

Listing A.6: The Contents of the Elevation of Privilege .dubhe File

Lastly, the grammar of .dubhe files presented in Backus-Naur Form (BNF) is shown in Listing A.7.

```
<dubhe_file> ::= "THREAT TECHNIQUE:" <technique_name> "\n"
                 "TECHNIQUE NUMBER:" <technique_number> "\n"
                 "THREAT MITIGATION:" <mitigation_name> "\n"
                 "MITIGATION NUMBER:" <mitigation_number> "\n"
                 "DETECT PATTERN:" <detect_pattern> "\n"
                 "MITIGATION PATTERN:" <mitigation_pattern> "\n"
                 "MITIGATION INDEX:" <mitigation_index> "\n"

<technique_name> ::= <string>
<technique_number> ::= "T" <number>
<mitigation_name> ::= <string>
<mitigation_number> ::= "M" <number>
<detect_pattern> ::= "[" <list_of_patterns> "]"
<mitigation_pattern> ::= "[" <list_of_patterns> "]"
<mitigation_index> ::= <number>

<list_of_patterns> ::= <pattern> | <pattern> "," <list_of_patterns>
<pattern> ::= "[" <list_of_elements> "]"
<list_of_elements> ::= <element> | <element> "," <list_of_elements>
<element> ::= "AcceptEventAction" | "OpaqueAction" | "DecisionNode"
   | "ActivityFinalNode" | "DataStoreNode" | <string>

<string> ::= '"' <any_characters> '"'
<number> ::= <digit> | <digit> <number>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
   "9"
<any_characters> ::= <character> | <character> <any_characters>
<character> ::= any alphanumeric character or punctuation mark
```

Listing A.7: The Grammar of .dubhe Files Expressed in BNF

Appendix B

CERI Values for all Critical Elements from the OSM System Scenario Walkthrough

Throughout the scenario walkthrough in Section 7.2, the worst-case and best-case CERI values for each identified critical element were calculated. We present these values for each of the three revisions of the OSM system's account login use case.

B.1 CERI Values for the Initial OSM System Account Login Use Case

The CERI values for the critical elements determined for the initial OSM system account login use case are shown in Table B.1.

Table B.1: The Worst-Case and Best-Case CERI Values for Critical Elements from the Initial OSM System Account Login Use Case

Element Name	**XMI Type**	**Worst-Case CERI**	**Best-Case CERI**
Customer Login Request	AcceptEventAction	1.67	1.0
Request Customer Information	OpaqueAction	1.6	0.8
Login Information	AcceptEventAction	1.67	1.0
CustomerDatabase	DataStoreNode	1.67	1.0
Customer Information	AcceptEventAction	1.67	1.0
Validate Customer Information	OpaqueAction	1.6	0.8
Authentication Successful	OpaqueAction	1.6	0.8
Login Success	AcceptEventAction	1.67	1.0
Customer Login Request	SendSignalAction	1.6	0.8
Login Information	SendSignalAction	1.6	0.8
Customer Information	SendSignalAction	1.6	0.8
Login Success	SendSignalAction	1.6	0.8

B.2 CERI Values for the First Revision to the OSM System Account Login Use Case

The CERI values for the critical elements determined for the first revision of the OSM system account login use case are shown in Table B.2.

Table B.2: The Worst-Case and Best-Case CERI Values for Critical Elements from the First Revision of the OSM System Account Login Use Case

Element Name	**XMI Type**	**Worst-Case CERI**	**Best-Case CERI**
Customer Login Request	AcceptEventAction	1.43	0.86
Log Login Request	OpaqueAction	1.43	0.86
ServerEventLog	DataStoreNode	1.5	1.0
Request Customer Information	OpaqueAction	1.43	0.86
Login Information	AcceptEventAction	1.43	0.86
Sanitize Data	OpaqueAction	1.43	0.86
CustomerDatabase	DataStoreNode	1.5	1.0
Customer Information	AcceptEventAction	1.43	0.86
Validate Customer Information	OpaqueAction	1.43	0.86
Authentication Successful	OpaqueAction	1.43	0.86
Login Success	AcceptEventAction	1.43	0.86
Customer Login Request	SendSignalAction	1.33	0.67
Login Information	SendSignalAction	1.33	0.67
Customer Information	SendSignalAction	1.33	0.67
Login Success	SendSignalAction	1.33	0.67

B.3 CERI Values for the Final Revision to the OSM System Account Login Use Case

The CERI values for the critical elements determined for the second revision of the OSM system account login use case are shown in Table B.3. Of note, while it appears as though the element with the name **Sanitize Data** appears twice, that is not the case, but rather two occurrences of distinct objects that happen to share a name and XMI type, as can be seen in Figure 7.8.

Table B.3: The Worst-Case and Best-Case CERI Values for Critical Elements from the Second Revision of the OSM System Account Login Use Case

Element Name	XMI Type	Worst-Case CERI	Best-Case CERI
Customer Login Request	AcceptEventAction	0.75	0.75
Check Request Rate	OpaqueAction	0.75	0.75
Log Login Request	OpaqueAction	0.75	0.75
Sanitize Data	OpaqueAction	0.75	0.75
ColdEventLog	DataStoreNode	0.89	0.89
Request Customer Information	OpaqueAction	0.75	0.75
Login Information	AcceptEventAction	0.75	0.75
Sanitize Data	OpaqueAction	0.75	0.75
CustomerDatabase	DataStoreNode	0.89	0.89
Customer Information	AcceptEventAction	0.75	0.75
Validate Customer Information	OpaqueAction	0.75	0.75
Authentication Successful	OpaqueAction	1.43	0.86
Login Failed	AcceptEventAction	2.25	2.25
Customer Login Request	SendSignalAction	0.67	0.67
Customer Information	SendSignalAction	0.67	0.67
Login Failed	SendSignalAction	0.67	0.67

Appendix C

Sample Security Metric Calculations

In this section, we perform sample calculations for the two security metrics introduced in this work. For these sample calculations, we will use a selection of the data collected for the initial OSM system analysis from the scenario walkthrough in Section 7.2.1.

In the initial iteration of the OSM system's account login use case, the identified critical element with the name **Login Success** with a XMI type of "AcceptEventAction" is found to be participating in 2 threat detection patterns. Of these 2 threats, 1 has been identified as mitigated or potentially mitigated. The critical element has a single flow leading into it, resulting in a cyclomatic complexity of 2. Using these values, we can calculate the CERI of the element:

$$\text{CERI}_{LoginSucc.} = c_{LoginSucc.} \times \left(1 - \frac{m_{LoginSucc.}}{t_{LoginSucc.}}\right) = 2 \times \left(1 - \frac{1}{2}\right) = 2 \times 0.5 = 1.0 \quad \text{(C.1)}$$

In the initial iteration of the OSM system's account login use case, 7 flows were identified. The sum of all paths l_i for i in the range $[1, 7]$ is 91. Using these values, we can calculate the CPP of the system:

$$\text{CPP} = \frac{\sum_{i=1}^{n} l_i}{n} = \frac{91}{7} = 12 \tag{C.2}$$