ANALYZING THE SECURITY OF C# SOURCE CODE USING A HIERARCHICAL QUALITY MODEL

by

Payton Rae Harrison

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2022

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor, Dr. Clemente Izurieta, for the mentorship and support throughout the entirety of this thesis. I would also like to acknowledge my committee members Dr. Derek Reimanis and Dr. Ann Marie Reinhold for their support and guidance throughout this project. Montana State University Software Engineering Lab (MSUSEL) members provided many valuable insights and discussions that helped advance this work. The Construction Engineering Research Laboratory (CERL) provided the financial support for me to complete the research conducted in this thesis.

Finally, I would also like to thank my close friends and family for their belief in me to finish this thesis and for constantly supporting me. Thank you all.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

Table Page

LIST OF TABLES – CONTINUED

Table Page

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

Figure Page

NOMENCLATURE

CERL	Construction Engineering Research Laboratory
CISA	Cybersecurity and Infrastructure Security Agency
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DHS	U.S. Department of Homeland Security
INL	Idaho National Lab
MSUSEL	Montana State University Software Engineering Lab
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
OWASP	Open Web Application Security Project
PIQUE	PIQUE is a collection of library functions and runner entry points designed to support experimental software quality analysis from a language-agnostic perspective
PIQUE-Bin	This project is an operationalized PIQUE model for the assessment of security quality in binary files
PIQUE-C#	This project represents a C# actualization of the PIQUE quality analysis platform. This project integrates the C# static analysis framework tool, Roslynator, and provides example extensions of the default weighting, benchmarking, normalizing, and evaluation strategies provided by PIQUE
PIQUE-C#-Sec	This project is an operationalized PIQUE model for the assessment of security quality in C# source code

ABSTRACT

In software engineering, both in government and in industry, there are no universal standards or guidelines for security or quality. There is an increased need for evaluating the security of source code projects, which is made apparent by the number of real-world cyber attacks that have taken place recently.

Our research goal is to design and develop a security quality model that helps stakeholders assess the security of C# source code projects. While there are many analysis tools that can be used to identity security vulnerabilities, the use of a model is beneficial in integrating multiple analysis tools to have better coverage over the number of security vulnerabilities detected (compared to the use of a single tool) and to aggregate these vulnerabilities upward into a broader security quality context. We accomplished our goal by developing and validating a hierarchical security quality model (PIQUE-C#-Sec) to evaluate the security quality of software written in C#. This model is an operationalized model using PIQUE, or the Platform for Investigative software Quality Understanding and Evaluation. PIQUE-C#-Sec improves upon previous security quality models and quality models that precede it by focusing on being specific, flexible, and extensible.

This thesis introduces the model design for PIQUE-C#-Sec and examines the results from the efforts of validating the PIQUE-C#-Sec model. This model was validated using sensitivity analysis, which consisted of collecting data on benchmark repositories and observing if and how the PIQUE-C#-Sec model output varied as a function of these repository attributes. Additionally, the model was analyzed by testing to see how the PIQUE-C#-Sec model node values changed because of the tools reporting additional vulnerabilities. Based on these results, we conclude that the PIQUE-C#-Sec model is effective for stakeholders to use when evaluating C# source code, and the model can be used as a security quality gate for evaluating these projects.

1

INTRODUCTION

Government organizations often hire contractors to assist in writing their software. Government organizations are interested in evaluating the security quality of contractor source code to ensure that the code does not have overall poor security quality or major security vulnerabilities. There are currently limited security or quality standards or guidelines for contractors to follow when writing their code. When the government organizations examine the code to be released, it is difficult to properly evaluate the security quality of the project.

Internet usage and the development of software have both been on the rise, which has led to an increase of virtual attacks and many new threats to security [18][35][4]. The increase of virtual attacks places an increasing priority and importance on information security [6]. The increase of virtual attacks has also resulted in software companies concerning themselves with security-related threats connected to their products' source code in recent years as customers demand high security in these software products [19]. Successful cyber attacks can cost organizations large amounts of resources because of losses caused by the attack, which can be in confidentiality, integrity, and availability [17].

The increased need for a focus on security vulnerabilities in source code projects is made apparent by the number of cyber threats and attacks that have taken place over the course of the last several years. Some examples of these cyber threats and attacks include the Colonial Pipeline[1] cyber incident in April 2021 and the SUNBURST[2] cyber attack, which was discovered in December 2020.

The Colonial Pipeline cyber incident consisted of hackers breaching the pipeline using

[1] `https://www.energy.gov/ceser/colonial-pipeline-cyber-incident`
[2] `https://www.solarwinds.com/sa-overview/securityadvisory`

a single compromised password, according to the Office of Cybersecurity, Energy Security, and Emergency Response[3]. The incident took down the largest fuel pipeline in the United States and led to shortages across the entire east coast.

In the SUNBURST cyber attack, SolarWinds[4] and their customers were the victims of a cyber attack to their systems that inserted a vulnerability within their Orion Platform builds. This vulnerability when present and activated could potentially allow an attacker to compromise the server on which the Orion products run. This cyber attack was a supply chain attack which is a disruption in a standard process resulting in a compromised result with a goal of being able to attack subsequent users of the software, according to the SolarWinds Security Advisory[5].

Both cyber attacks had huge impacts on both the government and citizens. Using hierarchical models to evaluate security quality presents an opportunity to lower these expected impacts and losses by alerting government agencies to these potential security threats.

The hierarchical models developed by the Montana State University Software Engineering Lab (MSUSEL) can aid in the process of lowering the impacts of cyber attacks by being used by users and stakeholders to evaluate the security quality and quality of software. The Platform for Investigative software Quality Understanding and Evaluation, or PIQUE[6], is a collection of library functions and runner entry points designed to support language-agnostic software quality analysis. PIQUE allows users to create hierarchical models that define which aspects of security and quality are important to them. By employing static analysis tools to search for and aggregate these security and quality vulnerabilities into a security or quality score, this allows users to have a better idea about the security or quality of their code.

[3] https://www.energy.gov/ceser/colonial-pipeline-cyber-incident
[4] https://www.solarwinds.com/
[5] https://www.solarwinds.com/sa-overview/securityadvisory
[6] https://github.com/msusel-pique/msusel-pique

PIQUE is designed to build operational quality models from its framework. This brings us to an instance of such an operationalized model, PIQUE-C#-Sec. PIQUE-C#-Sec[7] is an actualization of the PIQUE quality analysis platform and can be used to evaluate the security quality of real-world government contractor source code since much of this code is written in C#. PIQUE-C#-Sec has two C# static analysis tools integrated into the model, and provides default weighting, benchmarking, normalizing, and evaluation strategies from PIQUE.

A use case for the PIQUE-C#-Sec model is that government organization management can use PIQUE-C#-Sec as a security quality gate for government contractor source code. Government organization management can choose to set a threshold security quality score that will require the source code to return to the government contractors for modification if the score is not met [1].

The overarching aim of this thesis is to design and develop a hierarchical security quality model for projects with C# source code. A series of objectives were addressed to achieve this aim. To design and develop our own security quality model, we needed to research and become familiar with the background concepts of security quality modeling, security metrics, and static analysis tools (Chapter 2) and previous quality and security quality models (Chapter 3). Once we had this foundational knowledge, we needed to define our own research goals to ensure we knew what questions we wanted to address and answer with our research (Chapter 4).

Once we had our problem framed by our research questions, we could begin starting to design and develop our security quality model (PIQUE-C#-Sec) to address these research questions (Chapter 5). After the model was developed, we needed to address validating our model to help user and stakeholders build trust in our model's ability to evaluate C# source code and to help us investigate what the model output means (Chapter 6). Once the model

[7] `https://github.com/MSUSEL/msusel-pique-csharp-sec`

validation is completed, we need to discuss the implications of these results (Chapter 7) and detail the threats to validity from this thesis (Chapter 8). Finally, we will detail the overall conclusions of this thesis (Chapter 9).

5

BACKGROUND

Security Quality Modeling

The purpose of quality modeling centers around providing a systematic approach for modeling quality requirements, analyzing and monitoring quality, and directing quality improvement measures [42]. The evaluation of quality is subjective, and the evaluation needs will vary widely between users. The perception of security quality can differ significantly even when the quality models are based on the same hierarchical model structure, and this may create confusion when stakeholders evaluate the security quality of their software [21].

Quality modeling helps with this by providing a hierarchical system that helps structure and visualize these needs. A quality model specifies the meaning of quality of software in a way that it can be used in several scenarios. For example, to improve or assess the quality of software [44]. Many models have been proposed to support stakeholders in evaluating their software quality [25].

As software programs and computer systems usually have multiple vulnerabilities, it is desirable to aggregate the score of these individual vulnerabilities [17]; this is where hierarchical models are useful.

Cyber security models can enable users to make more well-informed choices that reduce the risk and impact of security vulnerabilities and incidents [16].

We can use our security quality model (PIQUE-C#-Sec) to provide these benefits of security quality modeling to our users and stakeholders. We will provide a brief overview of the PIQUE-C#-Sec hierarchical model structure to demonstrate how our model helps structure and visualize a user's subjective quality needs, and enable users to be more informed about the security of their software. Chapter 5 will go into more detail about the design of this PIQUE-C#-Sec model.

The PIQUE-C#-Sec model has a standard taxonomy for tree traversal, from the Total

Security Index (TSI) root node to the leaf nodes. The layers in the hierarchical model are TSI, Security Aspects, Product Factors, Measures, and Diagnostics. An example of this hierarchical structure is shown in Figure 2.1 [37].

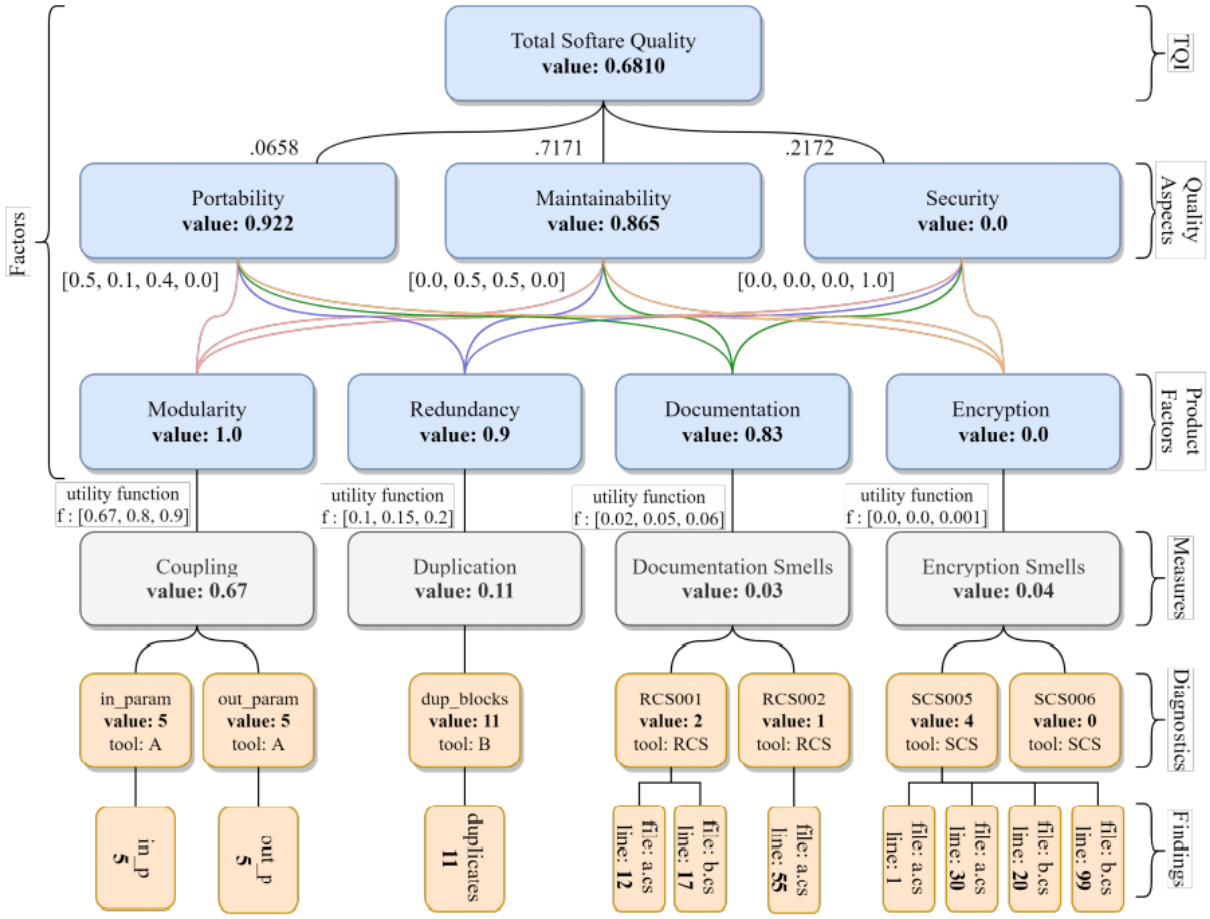

Figure 2.1: An example of a derived PIQUE quality model structure [37].

The PIQUE-C#-Sec model is created using a top-down approach. The root note is Total Security Index, or TSI. This node decomposes into Security Aspect nodes, which are nodes taken from both the ISO/IEC 25010 standard [20] and the Microsoft STRIDE model[1].

The ISO/IEC 25010 standard divides software quality into eight different attributes, as

[1] `https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats`

shown in Figure 2.2. These different attributes are meant to best represent stakeholder needs that can be included in a quality model [20]. For the PIQUE-C#-Sec model, we are only interested in the Security attribute. This consists of the properties confidentiality, integrity, non-repudiation, authenticity, and accountability.

Figure 2.2: ISO/IEC 25010 Standard for Software Quality [20]. These eight attributes are meant to best represent stakeholder needs that can be included in a quality model.

The ISO/IEC 9126 standard preceded the ISO/IEC 25010 standard, and the ISO/IEC 9126 standard forms a basis from which many quality models derived [2]. The ISO/IEC 9126 standard describes a model for software product quality that dissects the overall notion of quality into six main characteristics which are further subdivided into 27 sub-characteristics [15].

However, ISO/IEC 25010 has largely replaced ISO/IEC 9126. ISO/IEC 25010 has expanded the standard to include eight characteristics and 31 sub-characteristics [20]. Additionally, ISO/IEC 25010 improved upon and extended ISO/IEC 9126 by including computer systems and quality in use from a system perspective [2]. Another large improvement from ISO/IEC 9126 to ISO/IEC 25010 that impacts our decision to use the ISO/IEC 25010 standard for our PIQUE-C#-Sec model is that ISO/IEC 25010 added Security as a characteristic rather than a sub-characteristic, as it was in ISO/IEC 9126 [2].

The Microsoft STRIDE Threat modeling chart maps security properties to their

respective threats. STRIDE consists of the properties authentication, integrity, non-repudiation, confidentiality, availability, and authorization as shown in Table 2.1[2].

Table 2.1: The Microsoft STRIDE threat modeling chart maps security properties to their respective threats.

Property	Threat	Definition
Authentication	Spoofing	Impersonating something or someone else
Integrity	Tampering	Modifying data or code
Non-repudiation	Repudiation	Claiming to have not performed an action
Confidentiality	Information disclosure	Exposing information to someone not authorized to see it
Availability	Denial of service	Deny or degrade service to users
Authorization	Elevation of privilege	Gain capabilities without proper authorization

These Security Aspects are then further decomposed into Product Factors. The Product Factors in this model are the categories from the rules documented by the PIQUE-C#-Sec static analysis tools. These Product Factor nodes will be defined and detailed in Chapter 5.

These Security Aspects continue to be decomposed into low-level nodes until the concepts are at a measurable level. At this point, measurement tools are attached to populate these nodes with numerical values from the evaluated product [37]. Finally, the model needs to define how these bottom-level node values aggregate upward back to the TSI value.

Security Metrics

Security quality modeling can be used to evaluate the security attributes of a software project by using multiple types of static analysis tools and analysis, or security metrics. Security metrics allow users to measure the success of security policies, mechanisms, and

[2] `https://www.microsoft.com/security/blog/2007/09/11/stride-chart/`

implementations [33]. The use of security metrics is important because the security of a software project cannot be improved if it cannot be measured.

Metrics assist in the identification of a system's vulnerabilities, which leads to the facilitation of correcting these vulnerabilities while also raising the level of consciousness regarding the system's security [33]. The usage of security metrics in software projects allows security to be evaluated at the code implementation level. Defining these metrics earlier allows for security vulnerabilities to potentially be recognized earlier within a software project. This is valuable because security vulnerabilities that are discovered later in the development cycle are more expensive to fix than ones discovered earlier [11].

The actual severity of a vulnerability is difficult to capture accurately as there is no standard method for this purpose available [16]. However, the vulnerabilities found in the software can be mapped to vulnerability catalogs that are lists developed by the community.

The Common Weakness Enumeration[3] (CWE) and Common Vulnerabilities and Exposures[4] (CVE) are catalogs that capture information about weaknesses and vulnerabilities [22]. A vulnerability is defined as a weakness in some aspect or feature of a system that makes a threat possible [41]. CWEs are representative of general software and hardware weaknesses, while CVEs are representative of specific vulnerabilities in platforms and products.

CWE [29] provides a set of software weaknesses that contain a description, selection, and use of software security services and tools, which permits these weaknesses to be discovered in the software's source code [33]. This also creates a better understanding and management of these weaknesses.

To provide a high-level summary of detecting software vulnerabilities, users can run software vulnerability scans. These vulnerabilities are enumerated with a CVE identification number [30]. These are then stored in the National Vulnerability Database[5] (NVD) which

[3] https://cwe.mitre.org/
[4] https://cve.mitre.org/
[5] `https://nvd.nist.gov`

is maintained by the National Institute of Standard and Technology[6] (NIST) [4].

Static Analysis Tools

The most basic first step in assessing the security of software is to run static analysis tools on the source code to identify flaws within the software [26]. Static analysis tools are external resources that audit the product under assessment by parsing source code, byte code, or compiled source code to retrieve metrics or finding data [37]. The number (i.e., count) of vulnerabilities that exist in software is one of the most practical metrics to use for measuring security [17].

The measures and diagnostics in the PIQUE-C#-Sec model are obtained by running static analysis tools on the software. The tools can search for security vulnerabilities by examining the code directly; automated static analysis tools examine software by evaluating the code without executing it [13][8].

Using static analysis tools is important because as modern code bases are increasingly growing, it becomes more difficult to find security vulnerabilities [7]. Static analysis tools aid in this process and make it easier for users to efficiently identify potential weaknesses within the code base.

For the PIQUE-C#-Sec model, security quality is measured strictly by static code quality. There are two tools used in the PIQUE-C#-Sec model, Security Code Scan[7] and Insider[8]. Security Code Scan detects security vulnerabilities within source code such as SQL injections, cross-site scripting, and cross-site request forgery. Insider is focused on covering the OWASP Top 10[9] vulnerabilities within source code.

Both are static analysis tools that run on C# source code, assess security-related

[6] https://www.nist.gov/
[7] https://security-code-scan.github.io/
[8] https://github.com/insidersec/insider
[9] https://owasp.org/www-project-top-ten/

findings and vulnerabilities, are free and open source, and are command line tools. As discussed earlier, the leaf nodes of the security quality model are from any tooling output.

We decided to implement multiple static analysis tools into the PIQUE-C#-Sec model because using a combination of several techniques is an effective way to find additional vulnerabilities [7].

Different tools have different objectives [26], and for this reason it is likely that integrating more than one tool into our model will be beneficial. Since there is a large selection of static analysis tools, they have obvious value, and they are easily available, there is little reason to not include them in a project's development cycle [26], or in our case, a security quality model.

While our PIQUE-C#-Sec model does implement multiple static analysis tools, it currently only has two tools integrated. This is because of the tools available to us based on our search criteria, which will be detailed in Chapter 5.

There is no single technique or single set of rules that will detail all security vulnerabilities [32]. Therefore, using a combination of factors is beneficial, and why we are using a combination of static analysis tools in our model.

McGraw and Stephen [31] published an article about comparing multiple static analysis tools. They found that two tools perform differently on the same code bases because of internal rules used by the tools and coding style. They also claim that tool configuration and operators can greatly influence the discovery of vulnerabilities [31].

The tools currently used within PIQUE-C#-Sec identify security-related findings and will be implemented into the security quality model. The diagnostics found by each tool map to CWEs, which allows us to easily include these CWEs at the Measure layer within the PIQUE-C#-Sec model. We chose to use CWEs at the Measure layer because having our Diagnostics map to CWEs increases the trust that stakeholders and users can place in our model since this layer maps to this well-known standard for weakness cataloging that is

maintained by MITRE.

Tools that scan the source code of a program to find vulnerabilities and weaknesses are the first line of defense in assessing the security of software [26]. This makes it important to choose the best tools available for static analysis. For the exploration of new tools, it is viable to continue exploring static analysis tools that identify CWEs and CVEs in source code. This would allow any potential tool output to have a clear mapping to the upper layers in the PIQUE-C#-Sec model.

13

SUPPORTING WORK

The supporting work in quality modeling has many previously existing models with efforts that are useful. Our security quality model (PIQUE-C#-Sec) uses elements from these models and improves upon them to achieve the research goals we have defined for our model, which will be detailed in Chapter 4.

PIQUE-C#-Sec was designed and built from the PIQUE framework [37]. There are two hierarchical quality models that influenced the design of PIQUE. They are Quamoco [43] and QATCH [40].

Additionally, there are other hierarchical quality models that were developed using the PIQUE framework, which are PIQUE-C#[1] and PIQUE-Bin[2] [22]. This chapter serves to provide an overview to all these pre-existing models that aided in the design and development of the PIQUE-C#-Sec model.

Quamoco

The Quamoco model is a model that aims to bridge the gap between abstract quality aspects and concrete measurements [43]. The base model uses the ISO 25010 quality attributes, which continues to be a good starting point for quality modeling [43].

The Quamoco model evaluates the quality of a system by aggregating the issues and measures that affect the system [21]. These form the lowest level of the hierarchy and provide input at the measure level.

The Quamoco project consists of the base model, a GUI quality model editor, and quality adaption tools provided by the framework. The base model hierarchical structure consists of four layers under the root node: Quality Aspects, Product Factors, Measures,

[1] `https://github.com/MSUSEL/msusel-pique-csharp`
[2] `https://github.com/MSUSEL/msusel-pique-bin`

and Instruments.

Within the Quamoco model, there are several key terms that are important to note. These terms are explained below and can be seen in the model diagram in Figure 3.1.

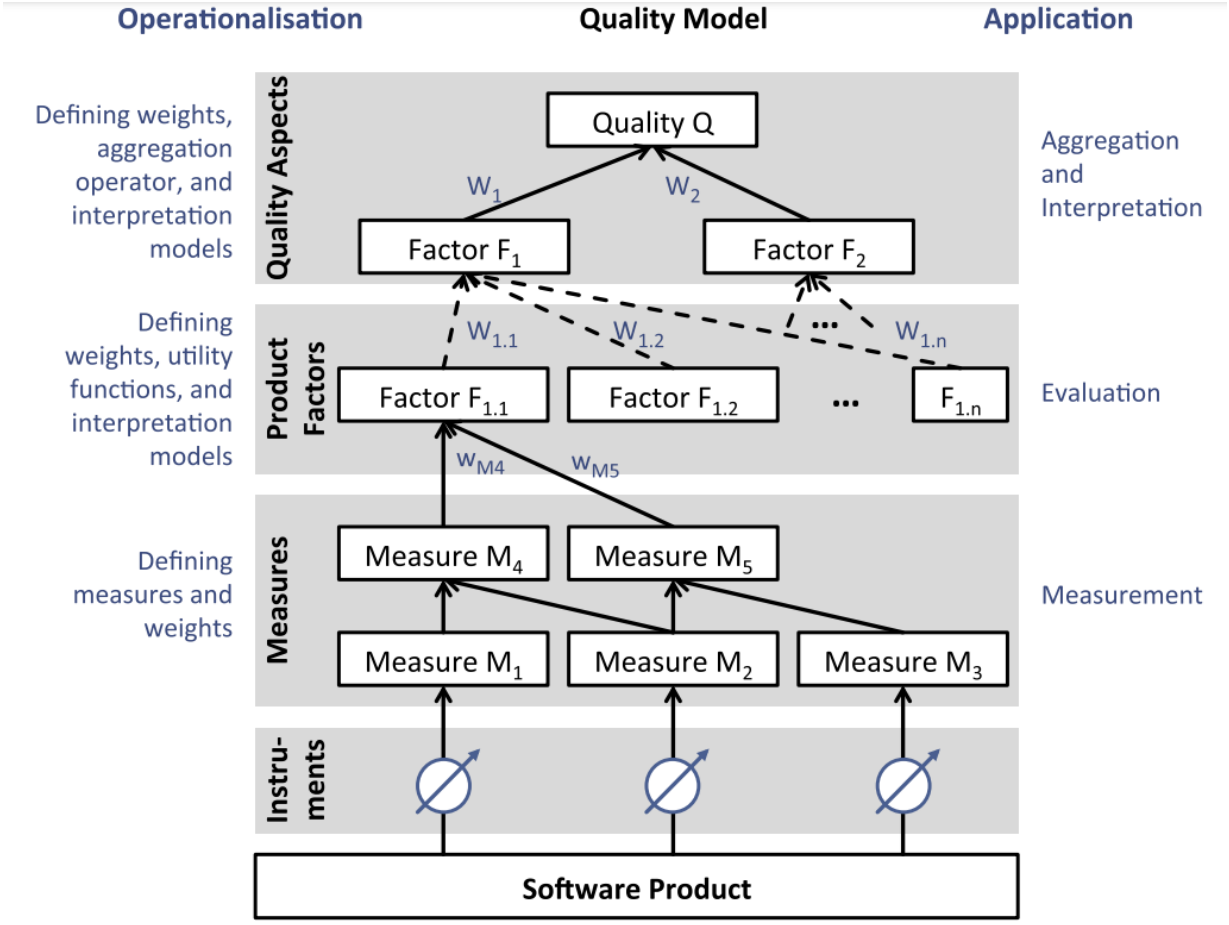

Figure 3.1: An example of a Quamoco quality model structure [43].

- **Factors**: express a property of an entity, where an entity is something that is important to quality and property is the attribute of what we are interested in. A factor constitutes a property of the software that is related to its quality [44]. The central

concept of the model is a factor, which is meant to represent a property or attribute of an entity (an entity being an important aspect of quality we want to measure) [21].

- **Quality Aspects**: these express abstract quality goals and can map to the quality attributes from a scientific standard; for example, ISO 25010 [20]. Quality Aspects provide information on the quality focused on within the model [25].
- **Product Factors**: measurable attributes that make up a part of the product.
- **Measures**: can be associated to more than one Product Factor. Measures are collected by instruments, and multiple measurements can collect values for a single measure. Defines how a specific entity is measured and provides a means to quantify factors that characterize this entity [44].
- **Instruments**: these are separated from the measures. The instruments are used to describe a concrete implementation of their measure. Used to determine the value directly using an external tool or a manual assessment [44].
- **Aggregation**: this is used to aggregate values of other measures [44].

Quamoco identified several areas for potential future work. These included (1) taking further technologies and contents for the base model into account and (2) working on further empirical studies to understand the weaknesses of the approach that still exist and improving these [43].

QATCH

QATCH, or the Quality Assessment Tool CHain, is a quality model that investigates an automated way to derive quality models that are responsive to subjectivity [40]. This is a valuable characteristic because the quality model will change with each new stakeholder and domain.

The QATCH model consists of three layers under the root node: the Characteristics layer, the Properties layer, and the Measures layer.

QATCH also uses the ISO/IEC 25010 standard, as the Quamoco model does.

One of the ways that automated subjectivity is integrated into the QATCH model is through an Analytical Hierarchy Process (AHP) [40]. AHP is an approach followed for decision making that reduces the decision complexity for pairwise comparisons. AHP is used to assist decision makers in choosing the best option among a set of alternatives.

AHP provides the ability for users and stakeholders to input their prioritized values into a quality model, and it is typically used to help these decision makers in scenarios where there are several objectives [38]. AHP enables the usage of pairwise comparisons between criteria to derive an order of importance for decisions [22]. Stakeholders can weigh specific aspects within the model as being more important than others with regards to the overall quality. With AHP, this can be done without requiring extensive knowledge from the user about quality modeling or hierarchical structures.

The pairwise comparisons for AHP may be expressed linguistically, allowing the comparisons to be more intuitive for non-technical stakeholders to express their values and prioritization [22]. QATCH presents a fuzzy AHP that allows stakeholders to express uncertainty about certain comparisons [22]. This decreases the total time for a stakeholder to tune the model since AHP helps stakeholders decide which nodes in each layer are most important.

The current support in QATCH is only for Java software. One area of future work is to expand the QATCH system so that it will be able to assess the quality of software products developed in other programming languages other than Java [40]. Future work also could include adding other static analysis tools or adding dynamic analysis metrics.

Within the QATCH model, there are several key terms that are important to note. These terms are explained below and are shown in the model diagram in Figure 3.2.

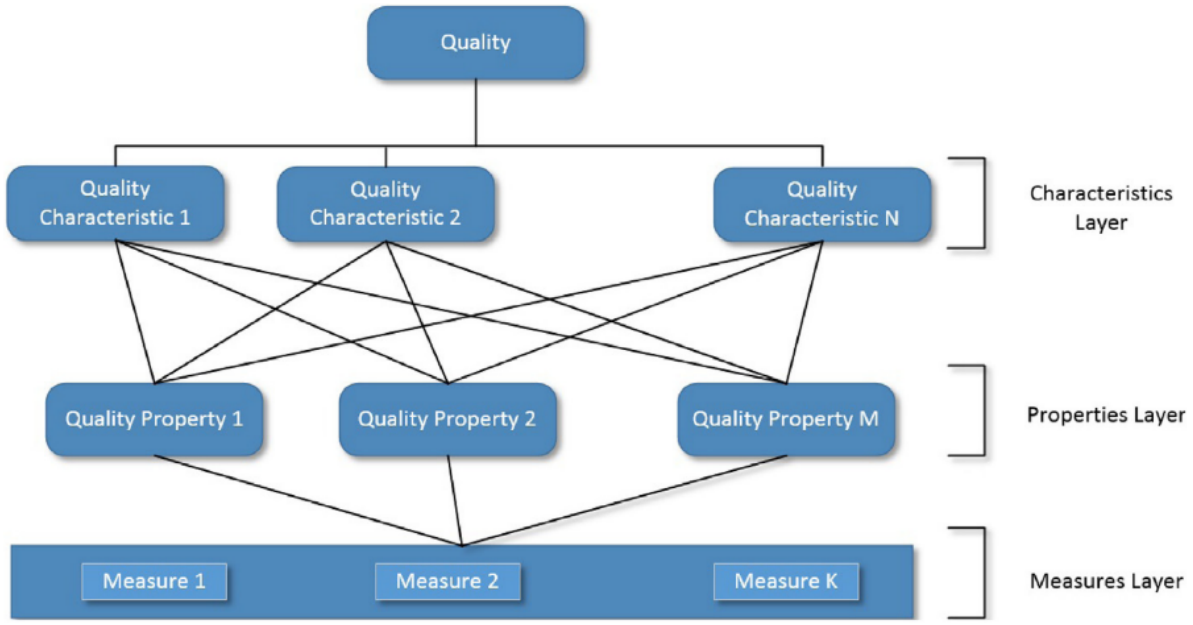

Figure 3.2: An example of a QATCH quality model structure [40].

- **Characteristics**: these are defined by a quality model definition that helps assess the overall software quality through aggregation and normalization.
- **Properties**: these are the decomposed components from software, and each can be given quality properties. Some examples of properties include programs, modules, objects, and variables [40].
- **Measures**: these are a set within the model, and are well-chosen to evaluate the properties.

In the validity experiment that the QATCH model underwent, the QATCH model reported perfect correlation with expert opinions on the evaluated projects. This indicates that QATCH is able to perform comparably, or possibly even better, than Quamoco in

its project evaluation compared to expert judges [22]. This also indicates that the loss of complexity in the QATCH model compared to Quamoco did not result in a loss of correlation with the expert opinions during the validity experiment [22].

PIQUE

PIQUE[3], or Platform for Investigative software Quality Understanding and Evaluation, is a collection of library functions and runner entry points designed to support language-agnostic software quality analysis.

PIQUE was built with nine design goals [22]:

1. Improve benchmarking, utility functions, and adaptive edge weighting
2. Improve default model mechanisms
3. Extend or modify the model mechanisms
4. Allow models to be easy to derive
5. Allow these derived models to be easy to operationalize
6. Allow simple addition, removal, or modification of tool support
7. Input and output are easy to interact with
8. Facilitate automation and continuous integration
9. Facilitate trustworthy models

PIQUE supports two processes needed for quality control: quality model derivation and quality assessment. PIQUE, like Quamoco and QATCH, uses the ISO/IEC 25010 standard.

[3] `https://github.com/msusel-pique/msusel-pique`

The PIQUE model derivation calculates the edge weights and the threshold values in the model, while the quality assessment populates each individual node value and the TQI value.

The PIQUE model consists of four layers under the root node: Quality Aspects, Product Factors, Measures, and Diagnostics. This structure is very similar to the Quamoco base model.

The PIQUE model is designed to have actualizations of the PIQUE model plug into PIQUE (for example, PIQUE-C# and PIQUE-Bin). PIQUE enables small scale teams to build quality models without the resources required for models such as the Quamoco model in [43]. This enables models to be created by single developers, which is the case for both the PIQUE-C# and PIQUE-Bin models [22]. PIQUE models are designed to be more flexible, extensible, and specific than Quamoco and QATCH models. PIQUE also realizes the future work described in the QATCH paper by developing a model that can evaluate software in other languages.

Quamoco, QATCH, and PIQUE all use the ISO/IEC 25010 standard as abstract guidelines for their models. Quamoco, the first of the three models, was found to be effective but slow and complex. QATCH improved upon this by being effective and simple. PIQUE improved upon both Quamoco and QATCH by creating a framework that allows for easily building quality models by focusing on reducing the resources needed to build quality models. PIQUE focuses more on flexibility and extensibility.

PIQUE Mechanisms

Thresholding Functions: Threshold values are created through a benchmarking process that begins with a group of similar projects to the one which the model will be applied. The various utility functions that PIQUE offers will be detailed below. The tools of the model are then applied to all the projects within this benchmark repository; this provides an estimate

of the number of findings that can be expected in the average project [22].

The benchmarking creates threshold values. These are the maximum and minimum values found in the benchmark repository that PIQUE uses by default. Custom utility functions can be written to change the calculation of these threshold values, as was done in PIQUE-Bin. In PIQUE-Bin, the threshold values are calculated as the mean plus and minus the standard deviation of the values in the benchmark repository [22].

We will discuss how the thresholding functions in the PIQUE-C#-Sec model in Chapter 5.

Weighting Strategy: Edges between the TQI/Quality Aspect nodes and the Quality Aspect/Product Factor nodes can be custom weighted which will affect how they are aggregated into the next layer of the model. This process is done using comparison matrices to rank the importance of the various nodes relative to each other [22].

We will discuss which weighting strategy is used in the PIQUE-C#-Sec model in Chapter 5.

Normalizing Process: Normalization in PIQUE divides the value of a Measure node in the model by the lines of code in the project. The DefaultNormalizer is the default normalizer used in PIQUE, which returns the Measure node value divided by the normalizer value (which is the lines of code of the project).

Normalizing allows us to take the different size of projects being evaluated into account. We normalize due to the size of the system directly influencing the number of findings that occur since we are evaluating projects in C# source code.

Utility Functions: Utility functions in the PIQUE model are created through the benchmarking process. The utility function for a measure takes the measurement of some diagnostic for a specific software project and outputs a value based on how it compares

to other software projects in the benchmark repository [22]. PIQUE uses the Linear Interpolation utility function by default, which reports the minimum and maximum value found within the benchmark repository for each measure.

We will discuss which utility functions are used in the PIQUE-C#-Sec model in Chapter 5.

PIQUE-C#

PIQUE-C#[4] is an operationalized PIQUE model for the assessment of quality in projects with C# source code. This project integrates the C# static analysis framework tool, Roslynator[5], and provides example extensions of the default weighting, benchmarking, normalizing, and evaluation strategies provided by PIQUE.

PIQUE-Bin

PIQUE-Bin[6] is an operationalized PIQUE model for the assessment of security quality in binary files. A few of the ways in which PIQUE-Bin has paved the way for PIQUE-C#-Sec include implementing weighting between the TQI/Quality Aspect and Quality Aspect/Product Factor layers, linking multiple static analysis tools to the quality model, and developing custom utility functions in the calibration and evaluator classes. An example of part of the PIQUE-Bin hierarchical model structure is shown in Figure 3.3.

[4] `https://github.com/MSUSEL/msusel-pique-csharp`
[5] `https://github.com/JosefPihrt/Roslynator`
[6] `https://github.com/MSUSEL/msusel-pique-bin`

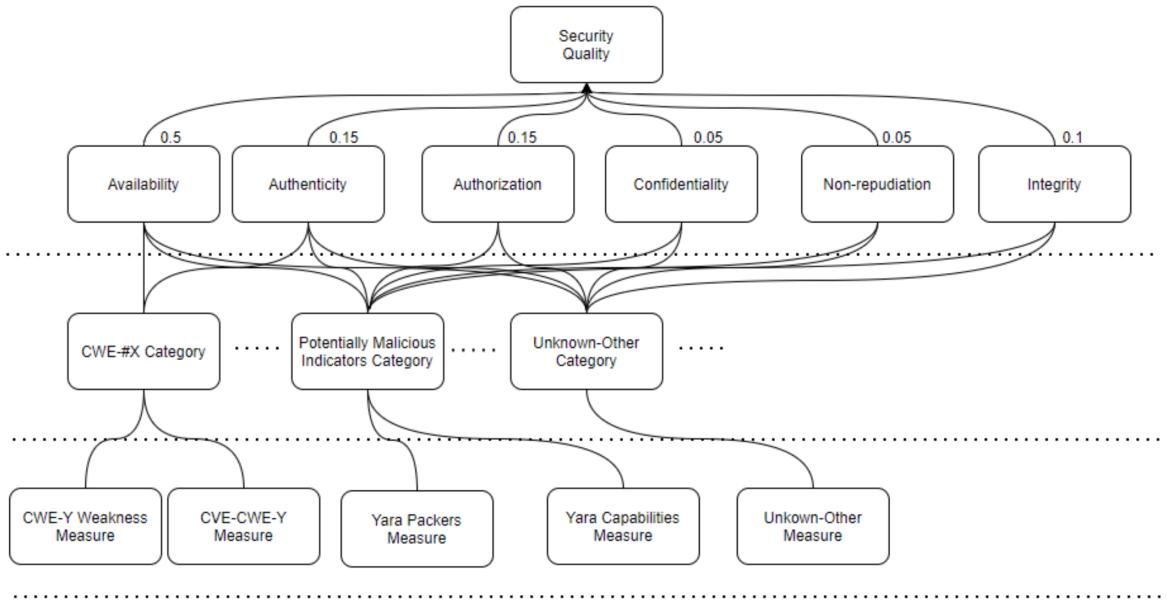

Figure 3.3: An example of a PIQUE-Bin security model structure [22].

Other Quality Models

The Quamoco and QATCH quality models directly shaped the development of PIQUE. However, there are other quality models and operational quality models that make up the supporting work in quality modeling as well. We will summarize some of those here and tie them back into the PIQUE model where applicable.

SQUID

The SQUID model defines the software requirements of the Telescience project [24]. The SQUID model uses the ISO/IEC 9126 standard for software quality.

The key elements in the SQUID model are product behavior, statement of quality requirements, quality characteristics, quality sub characteristics, internal software property, and measures [34]. The distinction of the SQUID model is that in addition to the quality characteristics, it also considers the operational behavior of the product to derive the quality

requirements [34].

When applying the SQUID model to the Telescience project, a number of practical problems were identified that were inherent in the current version of ISO/IEC 9126 [24]. This and problems from other models are likely what caused the shift in more modern quality models (such as Quamoco and QATCH) to use the ISO/IEC 25010 standard.

Factor-Strategy Quality Model

The goal of the Factor-Strategy Quality model is to demonstrate that the gap between qualitative and quantitative statements can be bridged [28]. This model is defined by quality factors being expressed in a set of quantifiable rules that identify violations of design principles, rules, and heuristics [28]. This is similar to the PIQUE model because PIQUE also decomposes quality factors into principles, rules, and heuristics (Product Factors, Measures, and Diagnostics).

The Factor-Strategy model makes the construction of the quality model easier because the quality of the design is naturally and explicitly linked to the principles below [28]. It also allows the quality factors within the model to be described in a concrete manner with respect to the programming paradigm [28].

This model introduces the concept of detection strategies in the quality model to formulate good design rules and heuristics in a quantifiable manner [34].

DEQUALITE

DEQUALITE, or Design Enhanced QUALITy Evaluation is a method to build quality models to measure the quality of object-oriented systems by taking into account both their internal attributes and their designs [23].

There are four steps in their method to build a quality model [23]:

1. Identify a set of high-level quality attributes

2. Identify and classify the most significant, tangible, quality-carrying properties of the system

3. Propose a set of axioms for linking product properties to quality attributes

4. Evaluate the model, identify its weaknesses, and either refine it or scrap it and start again

The key elements in the DEQUALITE model are quality attributes, quality metrics, design pattern/design rule, and the product or system [34].

SQALE

SQALE, or the Software Quality Assessment Based on Lifecycle Expectations, is a model used to estimate both the quality and technical debt of an application source code [27]. The SQALE quality model is used for formulating and organizing the non-functional requirements that relate to code quality [27]. This quality model consists of three hierarchical levels: characteristics, sub-characteristics, and requirements that relate to the source code's internal attributes.

The SQALE quality model asks users or stakeholders to establish their own concrete definition of "right code" [27]. This is similar to PIQUE in which the users or stakeholders must decide which nodes within the hierarchical model structure are of most importance relative to their project under analysis.

One area for future work that SQALE has identified is the fact that their users are expecting a standardized definition on what "right code" is [27]. This could also be something we will see in PIQUE as more stakeholders and users start testing our models.

QMOOD

QMOOD, or the Quality Model for Object Oriented Design, is the hierarchical model that defines relation between quality attributes and design properties with the help of

equations [14].

QMOOD metrics are subjective in nature, but with the relationship between quality attributes and design property being defined, these quality attributes can be calculated and aggregated [14].

There are four levels in the QMOOD model. These layers are Design Quality Attributes, Object oriented design Properties, Object oriented design Metrics, and Object oriented design Components [14]. This is similar to the PIQUE hierarchical model structure, which also has four layers under the root node.

RESEARCH GOALS

Motivation

Modern quality models do not provide adequate support for sole-security characteristics. Many of the current quality models such as Quamoco, QATCH, and PIQUE incorporate security into their quality models as one of the quality aspect nodes, but security is usually evaluated based on a single metric or several metrics. Security is a complex concept, so there exists a need for a dedicated security quality model with more advanced evaluation.

A security quality model is structured similarly to the PIQUE quality model, with a Total Security Index (TSI), Security Aspects, Product Factors, Measures, and Diagnostics.

Goal Question Metric

Basili [10] established an approach called the Goal Question Metric to break up research goals into their respective questions that need to be answered to meet the goal, and the metrics that need to be measured and gathered to answer the questions. We are using this approach as a tool to frame our research goal, questions, and metrics for our security quality model. These goals, questions, and metrics are outlined below:

Goal: Design and develop a security quality model that helps stakeholders assess the security of C# source code projects.

- **Q1** What attributes in C# source code projects result in a different Total Security Index and different Security Aspect scores? (Chapter 6)
  - **M1** Attributes that are tagged in benchmark data (project size in lines of code, project source, and project type)

  - **M2** Production of Benchmark Attribute Table
  - **M3** Production of TSI/Security Aspect Impact Table

- **Q2** How effective are the selected static analysis tools at measuring and reporting security vulnerabilities from the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list? (Chapter 6)
  - **M4** Number of CWEs in model compared to the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list
  - **M5** Production of PIQUE-C#-Sec tool output files
  - **M6** Production of a PIQUE-C#-Sec model description JSON file

- **Q3** What is the impact on the Total Security Index and Security Aspects for each single vulnerability? (Chapter 6)
  - **M7** Production of TSI/Security Aspect Impact Table
  - **M8** Production of PIQUE-C#-Sec tool output files

The tables mentioned within each question's metrics are included in Appendix D and Appendix E. These include the Benchmark Attribute Table (Table D.1) and the TSI/Security Aspect Impact Table (Table E.1).

PIQUE-C#-SEC DEVELOPMENT

This chapter will detail how we created our own security quality model to address our research goal of designing and developing a security quality model that helps stakeholders assess the security of C# source code projects. This security quality model is the PIQUE-C#-Sec model and was developed by using the PIQUE framework. This chapter explains the development of the PIQUE-C#-Sec model over the course of two sections: gather requirements, and development.

Gather Requirements

The requirements for the PIQUE-C#-Sec model were gathered by working with our stakeholders directly. Their approval was obtained on design choices such as the static analysis tools used, the Security Aspects of the model, and the linkage between Security Aspect and Product Factor layers.

Another component of running the PIQUE-C#-Sec model that had to be achieved was defining a set of benchmark repositories to use for the model derivation. The benchmark repository used for the PIQUE-C#-Sec model contains a combination of open source projects written in C# and closed source projects written in C#.

The open source benchmark projects were gathered by searching GitHub[1], filtering by the C# language, and sorting by the most stars, or most popular projects [37]. This gathering was conducted in July of 2019, as these were also the benchmark repositories used for PIQUE-C#. This list of open source benchmark projects can be found at the MSUSEL Benchmarks GitHub page[2].

The closed source benchmark projects were downloaded from our stakeholders' public

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

[2] `https://github.com/MSUSEL/benchmarks/tree/main/csharp-opensource`

GitLab[3] repositories. These closed source projects are similar to what PIQUE-C#-Sec will be used to analyze, so it is important to have closed source projects represented in our benchmark repositories so that they serve as a valid benchmarker for the final model.

Development

There exists a lack of quality and security standards or guidelines for contractors to follow when writing their code.

The development of the PIQUE-C#-Sec model was based on previous work done in the exploratory study PIQUE-C# model and the PIQUE-Bin model [22].

An exploratory study was conducted on developing a hierarchical model that measures the security of a C# project that first focused on a bottom-up model design. This exploratory PIQUE-C# model can be referenced in Appendix A. However, while this hierarchical model allows for great precision with respect to the project repository being analyzed, this specialized PIQUE-C# model is not generalizable. It will not work for any other project, and therefore every project that needs to be analyzed will need to have a model design file manually created. This results in the models being very time-consuming to create and not scalable as the number of projects increases. Therefore, the next and final model design was a top-down approach which allows for more generalizability and maintainability across projects.

The PIQUE-C#-Sec model differs from the PIQUE-C# model because the PIQUE-C# model was designed using a bottom-up approach while the PIQUE-C#-Sec model was designed using a top-down approach. Additionally, the PIQUE-C# model was designed to focus on evaluating the quality of source code while the PIQUE-C#-Sec model was designed to focus on evaluating the security quality of source code.

[3] `https://about.gitlab.com/`

This resulted in different static analysis tools being selected that focus on security-related findings, and the nodes in the hierarchical structure of the model changed to focus on security-related Security Aspects, Product Factors, and Measures. The TQI and Quality Aspects from the PIQUE-C# model changed to TSI and Security Aspects in the PIQUE-C#-Sec model.

The development of the PIQUE-Bin model resulted in some new technical advancements for the PIQUE-C#-Sec model to utilize. These advancements include implementing weighting between the TQI/Quality Aspect and Quality Aspect/Product Factor layers, linking multiple static analysis tools to the quality model, and developing custom utility functions in the calibration and evaluator classes.

However, the PIQUE-Bin model differs from the PIQUE-C#-Sec model because the PIQUE-Bin model was designed to evaluate the security quality of binaries while the PIQUE-C#-Sec model was designed to evaluate the security quality of C# source code. Additionally, PIQUE-C#-Sec differs from PIQUE-Bin in that it utilizes normalization. These differences resulted in different static analysis tools needing to be integrated into each respective model, and the nodes in the lower half of the hierarchical structure of the model (Diagnostic and Measure layers) changed based on the tool output.

Additionally, the PIQUE-Bin model mapped its Quality Aspect nodes to the properties from the Microsoft STRIDE model, which can be reviewed in Table 2.1, while the PIQUE-C#-Sec model maps its Security Aspects to both the Microsoft STRIDE model and the ISO/IEC 25010 standard [20] (Figure 2.2).

Model Structure and Design

The model structure for the PIQUE-C#-Sec model, as mentioned previously, is designed using a top-down approach. A partial view of the PIQUE-C#-Sec model can be seen in Figure 5.1. This is just a snippet from the middle of the model.

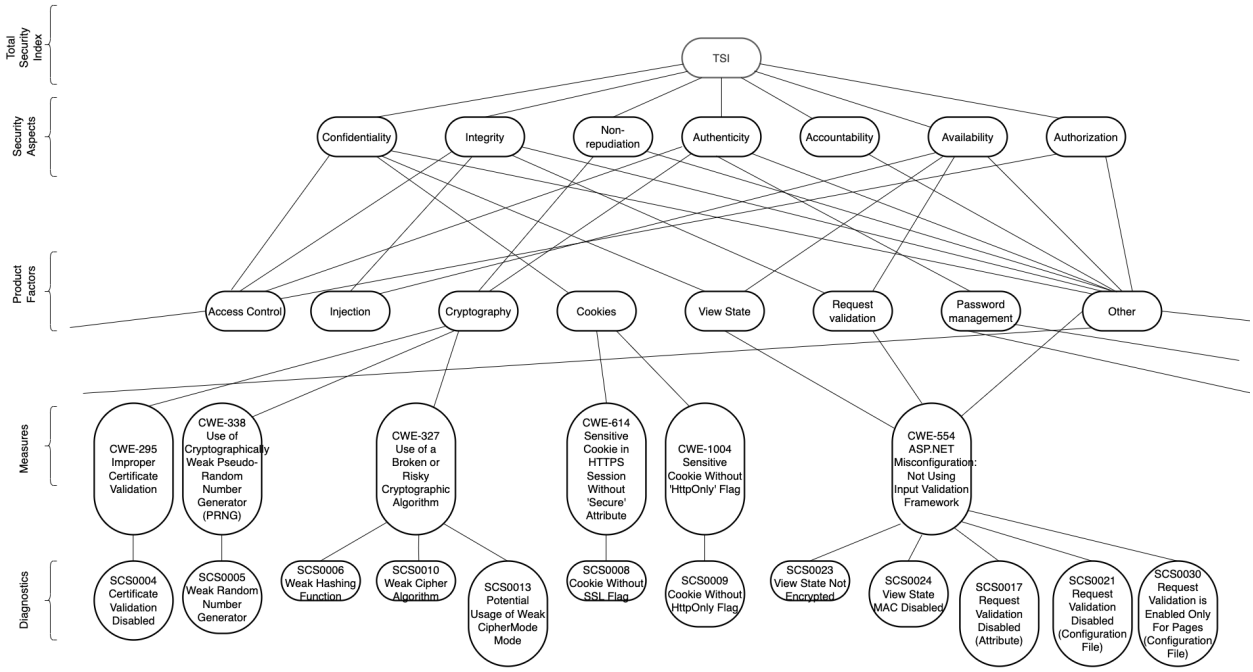

Figure 5.1: Partial PIQUE-C#-Sec Model

At the top level, the root node is Total Security Index (TSI). Below are the Security Aspects which are taken from the Security category in the ISO/IEC 25010 standard [20] and the properties from the Microsoft STRIDE model[4]. There is some overlap between the two sources, and in total they create seven Security Aspect nodes. All seven nodes are connected to the root node or the TSI node. These Security Aspect nodes consist of confidentiality, integrity, non-repudiation, authenticity, accountability, availability, and authorization. These Security Aspect nodes are defined in Table 5.1. These definitions come from the ISO/IEC 25010 standard [20].

The next layer, or Product Factor nodes, are made up from the rules of the Security

[4]`https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats`

Code Scan diagnostics[5]. These Product Factor nodes consist of injection, cryptography, cookies, view state, request validation, password management, and other. One more Product Factor node was added with the addition of Insider[6], our second tool, to fit the mapping from Insider's diagnostics. This additional Product Factor is access control. These Product Factor nodes are defined in Table 5.2.

Research was done on all eight of these Product Factor nodes to determine how to link them to the Security Aspect nodes. This work was approved by our stakeholders, which helps validate these design choices.

Both of our tools find and report diagnostics, and these tools' output make up the PIQUE-C#-Sec Diagnostics layer. There are 31 diagnostics found from Security Code Scan, and 28 diagnostics found from Insider. While all 59 nodes at the Diagnostic layer are too many to show in a figure, they are shown in Appendix B. The diagnostic IDS and descriptions for the Security Code Scan rules are taken from the Security Code Scan documentation[7] and the diagnostic IDs and descriptions for the Insider rules are taken from the Insider documentation[8].

This still leaves one layer left: the Measures layer that connects the Diagnostics to the Product Factors. The Measures layer is made up of the CWEs[9] associated with each Diagnostic. For each Security Code Scan diagnostic, a CWE is linked under the references section which is what Measure that Diagnostic maps to. For each Insider diagnostic, it is mapped to its parent CWE. Each CWE in the CWE documentation lists a "ParentOf" relationship for that CWE unless it is a pillar CWE. None of our CWE diagnostics are pillar CWEs. An example of each of these tool mappings from the Diagnostic to the Measure layer is shown in Figure 5.2.

[5] `https://security-code-scan.github.io/`
[6] `https://github.com/insidersec/insider`
[7] `https://security-code-scan.github.io/#Rules`
[8] `https://github.com/insidersec/insider/blob/master/rule/csharp.go`
[9] `https://cwe.mitre.org/index.html`

Table 5.1: Security Aspect Node Definitions

Node	Description
Confidentiality	Degree to which a product or system ensures that data are accessible only to those authorized to have access
Integrity	Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data
Non-repudiation	Degree to which actions or events can be proven to have taken place so that the events or actions cannot be repudiated later
Authenticity	Degree to which the identity of a subject or resource can be proved to be the one claimed
Accountability	Degree to which the actions of an entity can be traced uniquely to the entity
Availability	Degree to which a system, product or component operates as intended despite the presence of hardware or software faults
Authorization	Individuals, entities, or processes only have access to data and processes they should

Therefore, CWEs make up the PIQUE-C#-Sec model's Measure layer. The list of CWEs are sponsored by national security organizations such as DHS[10], CISA[11], and MITRE[12].

[10] `https://www.dhs.gov/`
[11] `https://www.cisa.gov/cybersecurity-division`
[12] `https://mitre.org/`

Table 5.2: Product Factor Node Definitions

Node	Description
Access control	Dictates who is allowed to access and use company information and resources
Injection	Attacker supplied untrusted input to a program
Cryptography	Prevents unauthorized access to information and keeps data safe
Cookies	Text files with small pieces of data that are used to identify your computer as you use a computer network
View state	Poses a security risk when information of view state can be seen in the page output source directly
Request validation	Feature in ASP.NET that examines HTTP requests and determines whether they contain potentially dangerous content
Password management	Most important feature to look for in a password manager is advanced encryption
Other	Encompasses any topics not covered by other Product Factors

PIQUE-C#-Sec Mechanisms

In Chapter 3, we defined the functions within PIQUE including benchmarking, weighing, normalization, thresholding, and normalizing. Here we will detail which strategies are used for each of these mechanisms.

Thresholding Functions: PIQUE offers a NaiveBenchmarker and a BinaryBenchmarker. The NaiveBenchmarker calculates the lowest and highest of each Measure value (or the minimum and maximum values) for the threshold values. The BinaryBenchmarker calculates the mean plus or minus the standard deviation of each Measure value for the threshold values.

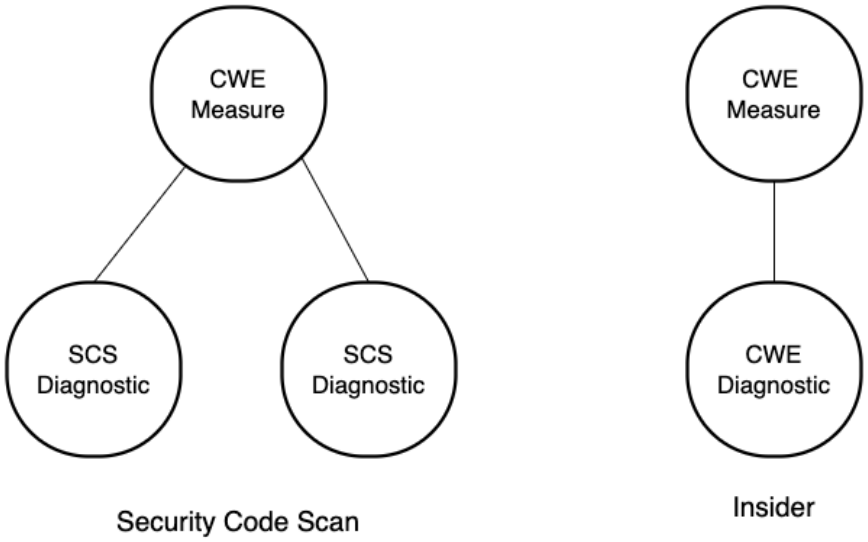

Figure 5.2: Tool mapping examples from the Measure to Diagnostic layer for each of our static analysis tools in the PIQUE-C#-Sec model.

We decided to use the BinaryBenchmarker for PIQUE-C#-Sec because it was the most recently adapted thresholding function at the time that PIQUE-C#-Sec was developed. We thought that the function would provide value in the fact that the calculation was more specific than just using the minimum and maximum values.

However, as we will see with our model validation in Chapter 6, this choice in thresholding function may be limiting the overall range of TSIs our model can produce. By using the NaiveBenchmarker, we can have a wider range in threshold values in our model which could allow for greater flexibility and a more accurate reflection of TSI values.

Weighting Strategy: PIQUE offers a NaiveWeighter and a BinaryCWEWeighter. The NaiveWeighter sets each node's incoming edge weights equal to the average of all incoming edges. The BinaryCWEWeighter is a strategy that completes the edge weighting with comparison matrices.

We are using the NaiveWeighter for PIQUE-C#-Sec based on our stakeholder's request. We asked our stakeholders if they would like to rank the Security Aspects and Product Factors, and we also offered to rank those nodes for them. However, they requested that the nods be left equally weighted.

If this request changes in the future, the BinaryCWEWeighter can be used as the weighting strategy. This is a more effective weighter in the case that there are clear preferences from the user with regards to which nodes should be ranked of higher importance than other nodes.

Normalizing Process: PIQUE offers a DefaultNormalizer and a NoNormalizer. The DefaultNormalizer returns the Measure node value divided by the normalizer value (which is the lines of code of the project). NoNormalizer always returns a value of 1, as no normalization occurs.

PIQUE-C#-Sec uses the DefaultNormalizer as we are dealing with C# source code. This means that each C# source code project does contain its size in lines of code. Therefore, it makes sense for us to normalize in this model so that our model can take the different size of projects being evaluated into account. We normalize due to the size of the system directly influencing the number of findings that occur since we are evaluating projects in C# source code.

Utility Functions: PIQUE offers three utility functions: DefaultUtility, GAMUtilityFunction, and GaussianUtilityFunction. The DefaultUtility provided by PIQUE uses linear interpolation.

For each Diagnostic in the PIQUE-C#-Sec model, we created a histogram with that Diagnostic's number of findings compared to the frequency of the findings. We then went through each Diagnostic and visually observed the histogram to decide which utility function was best based on each Diagnostic's plots.

Out of PIQUE-C#-Sec's 59 Diagnostic nodes, 25 had non-zero findings, so therefore plots could be created for these 25 Diagnostics. Of those 25, 9 were mapped to the DefaultUtility based on histogram shape, and the remaining 16 were mapped to the GAMUtilityFunction.

Since the GAMUtilityFunction was the mapping for most of our observed Diagnostics, we also assigned our zero finding Diagnostics to the GAMUtilityFunction as well.

Tools

This subsection will detail the process of filtering through static analysis tools and describing this selection process.

The effort to obtain and install these tools can be seen as excessive by software developers or support teams [26]. Configuring tools so that the output produced is useful can also be an issue [26]. Some tools have too much documentation, while some tools have little to no documentation, and both are issues when trying to identify tools to use in the model [26].

Whereas the exploratory PIQUE-C# model used Roslynator and the other tools that Roslynator allows to run through its interface (Security Code Scan and VS-Threading), the PIQUE-C#-Sec model only links findings from Security Code Scan[13] and Insider[14]. PIQUE-C#-Sec removed Roslynator as a tool to find diagnostics because Roslynator and VS-Threading diagnostics focused on code quality, while Security Code Scan and Insider

[13] `https://security-code-scan.github.io/`
[14] `https://github.com/insidersec/insider`

focus on security-related findings.

Neither Roslynator nor VS-Threading have findings that are related to security. Roslynator has findings that relate to quality attributes (functional suitability, performance, compatibility, usability, reliability, maintainability, portability) which is why it was chosen for the PIQUE-C# quality model.

However, Roslynator does report lines of code, which the PIQUE-C#-Sec model still utilizes to complete its normalization. While Roslynator is used to report the lines of code, it does not report any diagnostics for the PIQUE-C#-Sec model.

Measures contain a method for normalizing the diagnostic values, and this process is done through a utility function [22]. Normalization is calculated after the measures are derived from a benchmark repository and before the evaluation. The PIQUE-C#-Sec model uses lines of code to normalize the diagnostic values. This divides the value of the node by the lines of code, which normalizes each node by the lines of code in the system [37].

Roslynator counts the physical lines of code in the specified project or solution[15]. This differs from logical lines of code because physical lines of code include the number of lines of code read in the source code, while logical lines of code include the total number of instructions. This means that for logical lines of code, there could be several instructions per physical line of code.

We used two static analysis tools in the PIQUE-C#-Sec model to cover more security vulnerabilities and increase the validity of the model.

Specific criteria were used when searching for static analysis tools. For the PIQUE-C#-Sec model, these criteria were as follows:

- The tool is compatible with source code written in the language C#
- The tool is free or open source

[15] `https://github.com/JosefPihrt/Roslynator/blob/master/docs/cli/loc-command.md`

- The tool is a command line tool (not an analyzer/IDE plugin/extension)
- The tool has security-related rules or findings
- The tool has a way to output results
- The tool has good documentation

A paper that explored assessing security technical debts included a list of selected static analysis tools that are used to help identify design vulnerabilities in smart contracts [3]. The tools the paper explored after filtering out options by a set of criteria were as follows: Slither, SmartCheck, Securify, Mythril, sFuzz, Solhint, Ethlint, and Mythos.

However, none of these tools were viable for the PIQUE-C#-Sec model for two main reasons: almost all the tools in the paper were compatible with source code written in Solidity, not C#, and those tools were targeting design vulnerabilities, not security vulnerabilities.

Many tools were researched and found from static analysis tool lists. The lists that we researched and investigated were OWASP Source Code Analysis Tools[16], NIST Source Code Security Analyzers[17], Analysis Tools Static Analysis Tools[18], and List of Tools for Static Code Analysis[19].

Between these four resources and other research (which consisted of two additional tools found from references in the previously found 27 tools), a total of 29 tools were considered after filtering the list by language (C#) and free/open source tools. After investigating these 29 tools, two were chosen for the PIQUE-C#-Sec model. This reduction was due to filtering out the tools based on our other criteria (the tool is a command line tool, has security-related rules or findings, has a way to output results, and has good documentation).

[16] `https://owasp.org/www-community/Source_Code_Analysis_Tools`
[17] `https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers`
[18] `https://github.com/analysis-tools-dev/static-analysis`
[19] `https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`

The main takeaways after looking at all 29 tools are that there are a lot of analyzers that integrate with IDEs but are not command line tools, and there are a lot of tools advertised as being free or open source when they only are offering a free trial or demo. All 29 tools are listed in Table 5.3, with a legend preceding the table with the detailed reason as to why each tool was ruled out and not used in the PIQUE-C#-Sec model.

Table Legend:

1. Analyzer library/IDE extension
2. Tool has not been updated in at least over a year, the repository is marked as archived, or the tool is retiring in the near future
3. Tool does not appear to have security-focused rules/findings
4. Not a command line tool - may be a GUI or application
5. Not fully free/open source; only offers free demo or trial
6. Tool has too little documentation to be of use
7. Tool depends on registering for an organization account
8. Tool's rules are encompassed by another tool used in the model

Additionally, a cell value of "N/A" indicates that the tool is used in the PIQUE-C#-Sec model, and therefore was not ruled out.

Table 5.3: Static analysis tools researched for potential use in the PIQUE-C#-Sec model and what criteria it violated.

Source	Tool Name	Reason why it was Ruled Out
OWASP Source Code Analysis Tools	LGTM	1
	Microsoft FxCop	2
	Puma Scan Professional	5
	PVS-Studio	5
	SonarCloud	3
	VeraCode	4
	VisualCodeGrepper	6
	Coverity	5
	AppScan	3
	Klocwork	5
	ShiftLeft	7
	Agnitio	6
	SonarQube	5
	HCL AppScan on Cloud	3
	Insider	N/A
NIST Source Code Security Analyzers	Security Code Scan	N/A
Analysis Tools Static Analysis Tools	.NET Analyzers	1
	ArchUnitNet	3
	code-cracker	1
	CSharpEssentials	2
	Infer#	1
	Roslynator	8
	Wintellect.Analyzers	1

Table 5.4: Static analysis tools researched for potential use in the PIQUE-C#-Sec model and what criteria it violated continued.

Source	Tool Name	Reason why it was Ruled Out
List of Tools for Static Code Analysis	ConQAT	2
	.NET Compiler Platform (Roslyn)	1
	Sourcetrail	4
	StyleCop	1
Other Research	SonarAnalyzer.CSharp	1
	Microsoft Security Code Analysis	2

MODEL VALIDATION

The PIQUE-C#-Sec model has now been fully designed and developed, but we must build trust in the PIQUE-C#-Sec model and its ability to evaluate C# source code and investigate what the model output means. This can be achieved by validating the model.

The PIQUE-C#-Sec model has been validated using three approaches. First, we analyzed PIQUE-C#-Sec's benchmark projects and the attribute data associated with those projects to determine if any attributes in C# source code projects result in a different TSI and different Security Aspect scores.

Next, we observed how effective the selected static analysis tools within the PIQUE-C#-Sec model are at measuring and reporting security vulnerabilities by comparing all 59 Diagnostic nodes in our model to the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list.

Finally, we analyzed the impact that each single vulnerability within our model had on the TSI and Security Aspect node values. We can see our study design in Figure 6.1.

Analysis of Benchmark Data

We applied an analysis of the benchmark data to answer Q1 from our Goal Question Metric in the Approach chapter. Q1 asked what attributes in C# source code projects result in a different Total Security Index and different Security Aspect scores?

The first step in analyzing the benchmark data involves identifying project characteristics (size in lines of code, type, and source). The type of the benchmark can either be a library or an application.

The source of the benchmark can either be open source or closed source. The open source benchmark projects were the same ones used in Chapter 5 from the MSUSEL

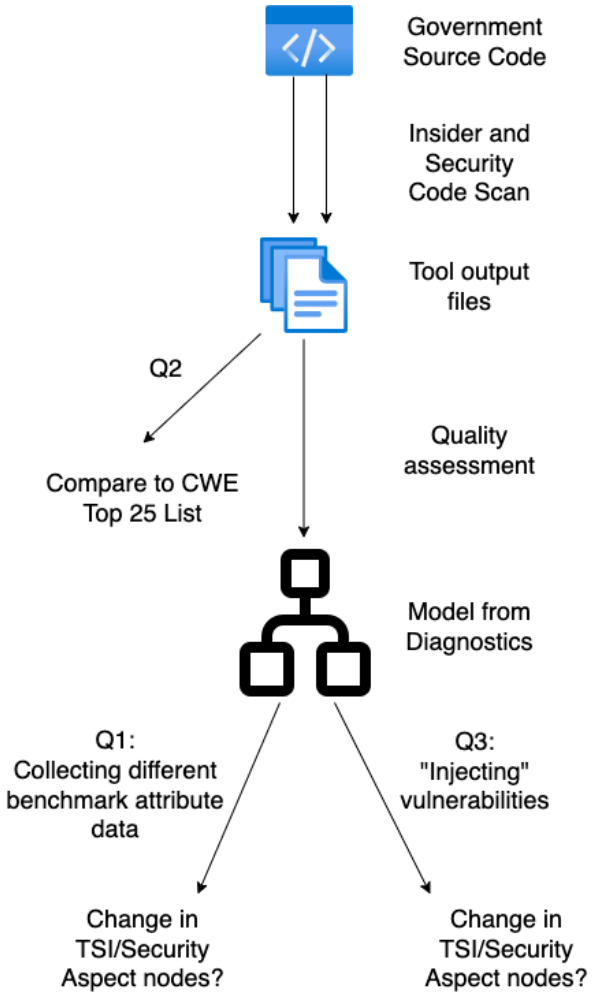

Figure 6.1: Study Design for our PIQUE-C#-Sec Model Validation

Benchmarks GitHub page[1]. The closed source benchmark projects were downloaded from our stakeholders' public GitLab[2] repositories.

To summarize some descriptive statistics from the lines of code attribute of the benchmark projects, a total of 72 projects were used, with sizes ranging from 11 to 286,151 lines of code. Alves [5] presents a methodology for the calibration of mappings from code-level measurements to system-level ratings and in this approach, also measures size using

[1] `https://github.com/MSUSEL/benchmarks/tree/main/csharp-opensource`
[2] `https://about.gitlab.com/`

lines of code. The total lines of code for all benchmark projects was 1,747,620 lines of code.

Of the 72 benchmarks, 26 are open source and 46 are closed source. Additionally, of the 72 benchmarks, 42 were of type library and 30 were of type application.

Appendix D shows a summary of all these results by detailing each project in the benchmark repository, their size in lines of code, their source, and their type. The project names of the closed source projects have been renamed so as not to compromise any sensitive information from the stakeholder that provided us access to these projects.

The first plot that was created with this benchmark attribute data was a plot to observe the TSI of each benchmark project compared with that project's size in lines of code. This plot is shown in Figure 6.2. Since we used the DefaultNormalizer in PIQUE, this means that the TSI is normalized with respect to the size in lines of code.

As we can see in Figure 6.2, the benchmark projects that are below a certain size appear to have little to no influence on the project's TSI. This suggests that with the projects below a certain size, either the PIQUE-C#-Sec model tools are not detecting many vulnerabilities, or that the projects do not contain many vulnerabilities due to their small size.

Additionally, from Figure 6.2, we can see that the lowest TSI we are observing from our benchmark data is a value of around 0.6. This could mean that our benchmark projects are generally of average or good security quality when compared against each other. Based on this theory that all the benchmark projects are representative of good security quality, we could seek out projects that are of known poor security quality and include them within the benchmark repository to see how they compare against the projects we have already selected.

Another approach we could implement to try to observe TSI scores closer to 0.0 could be to use the DefaultBenchmarker instead of the BinaryBenchmarker. Taking the minimum and maximum values as the thresholds (DefaultBenchmarker) allows for more range than taking the mean plus and minus the standard deviation (BinaryBenchmarker), so this could

result in node values aggregating upward into a lower TSI score.

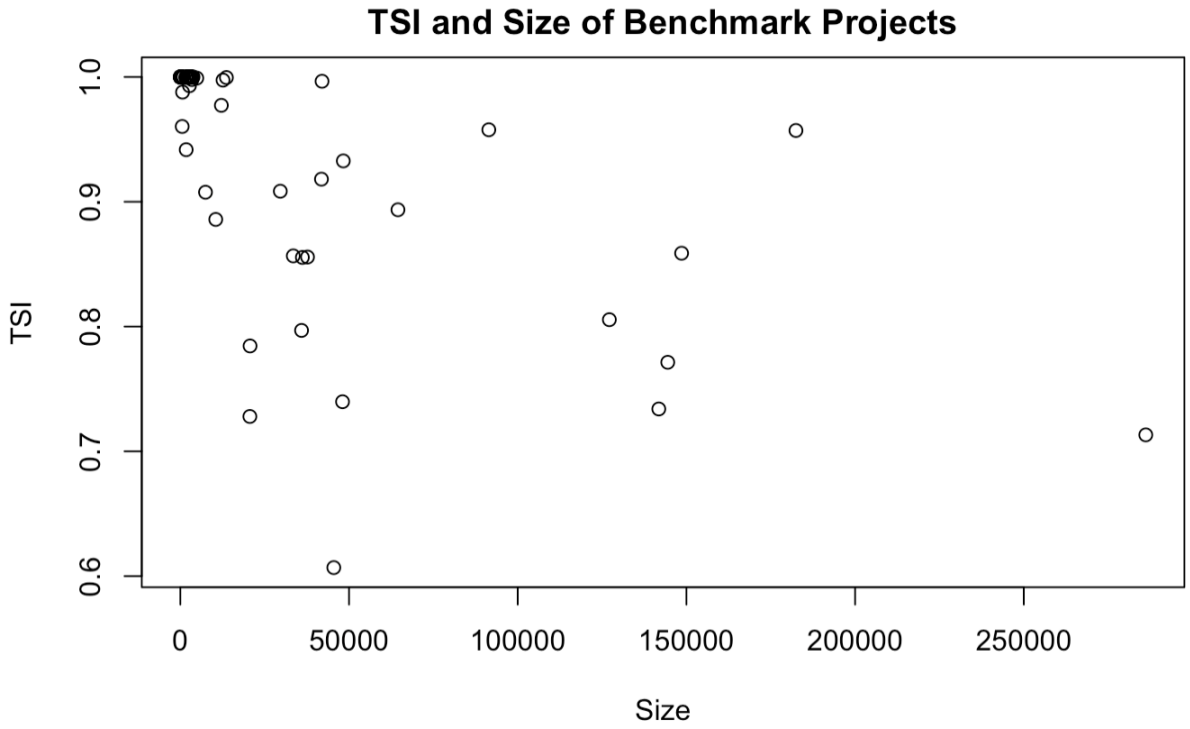

Figure 6.2: Observing the TSI of each benchmark project compared with that project's size in lines of code.

We wanted to break Figure 6.2 out into smaller subsets of the other measured attributes. We created a plot to show the effect of each combination of the categorical attributes as a function of TSI and size. Figure 6.3 shows this data by separating it into four quadrants: closed source projects categorized as libraries (top left), closed source projects categorized as applications (bottom left), open source projects categorized as libraries (top right), and open source projects categorized as applications (bottom right).

After looking at Figure 6.3, what we see is that the open source software appears fundamentally different than the closed source software based on the distribution of the data. The two plots on the right half of Figure 6.3 (the open source projects) have points ranging across the plot, and we can see that it is possible to fit a linear regression line to the

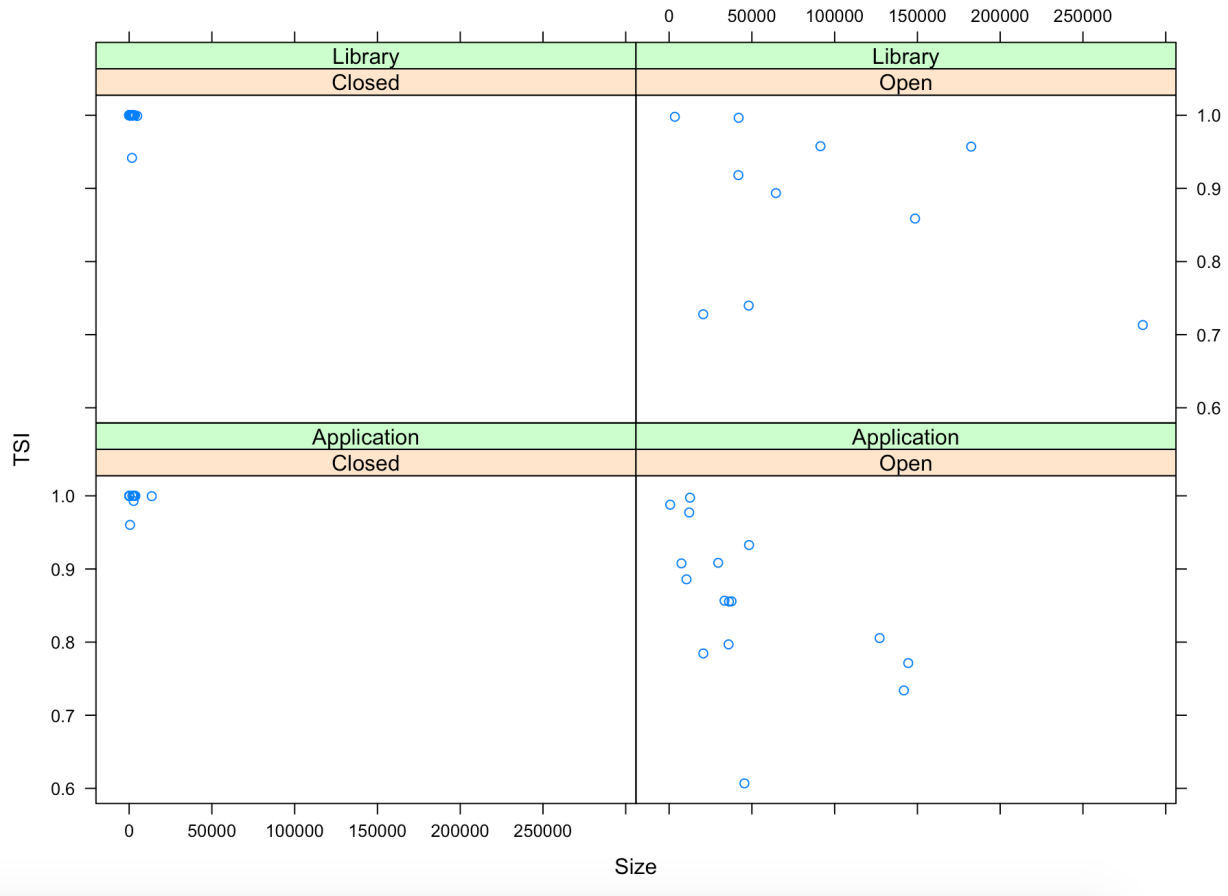

Figure 6.3: Observing Each Attribute Subset as a Function of TSI and Size

data. However, in the two plots on the left half of Figure 6.3 (the closed source projects), all points are clustered in the top left corner of the plot and do not appear to fit a linear regression line. The justification for analyzing open and closed source data separately is that they come from two different populations.

We can see from Figure 6.3 that there was not a lot of variation in the closed source project points. All the projects primarily were of small size and high TSI. This goes back to our earlier observation that projects below a certain size appear to have little to no evidence on the project's TSI.

We have created histograms for the frequency of the TSIs in open source projects

(Figure 6.4) and in closed source projects (Figure 6.5).

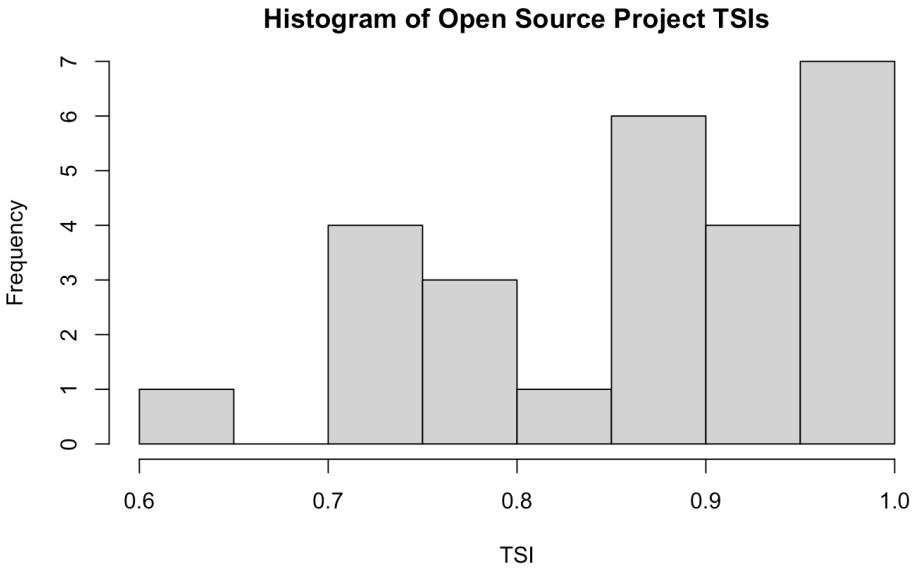

Figure 6.4: Histogram of Open Source Project TSIs

Figure 6.5: Histogram of Closed Source Project TSIs

From these two histograms (Figure 6.4 and Figure 6.5), we can show that the distributions are fundamentally different. The differences between the two histograms shows

that we clearly do not have normally distributed data.

Therefore, we decided to subset our attribute data by either open or closed source. We have fit a linear model to both open and closed source data to observe if there are any significant attributes that have an effect on the TSI.

Before we begin fitting these linear models, we need to define our research question and hypotheses. Our research question is: What attributes in C# source code projects result in different security vulnerabilities and a different Total Security Index? To answer this question, our null hypotheses are as follows:

- $H1_0$: There is no correlation between the size (lines of code) of the C# project under analysis and the TSI.
- $H2_0$: There is no correlation between the type (library or application) of the C# project under analysis and the TSI.
- $H3_0$: There is no correlation between the interaction between size and type of the C# project under analysis and the TSI.

Finding a significant correlation ($p < 0.05$) between an attribute or the interaction of attributes and the TSI will lead us to reject the null hypothesis for that attribute or interaction of attributes [22].

To test our three null hypotheses stated above, we created a linear model for open source projects that is the TSI as a function of the interaction between size and type. We are subsetting the data to only include projects that are open source.

This linear model is as follows:

$$\hat{TSI} = \beta_0 + \beta_1 size + \beta_2 type + \beta_3 size * type, \tag{6.1}$$

Where β_0 is the y-intercept and β_1, β_2, and β_3 are all regression coefficients. This linear model is represented as follows in R code:

```
lm(TSI ~ Size * Type, data = attribute[attribute$Source == "Open",])
```

After fitting the above linear regression model, we must assess to what degree the assumptions are violated since the assumptions can never be fully met. The diagnostic plots for this model are shown in Figure 6.6.

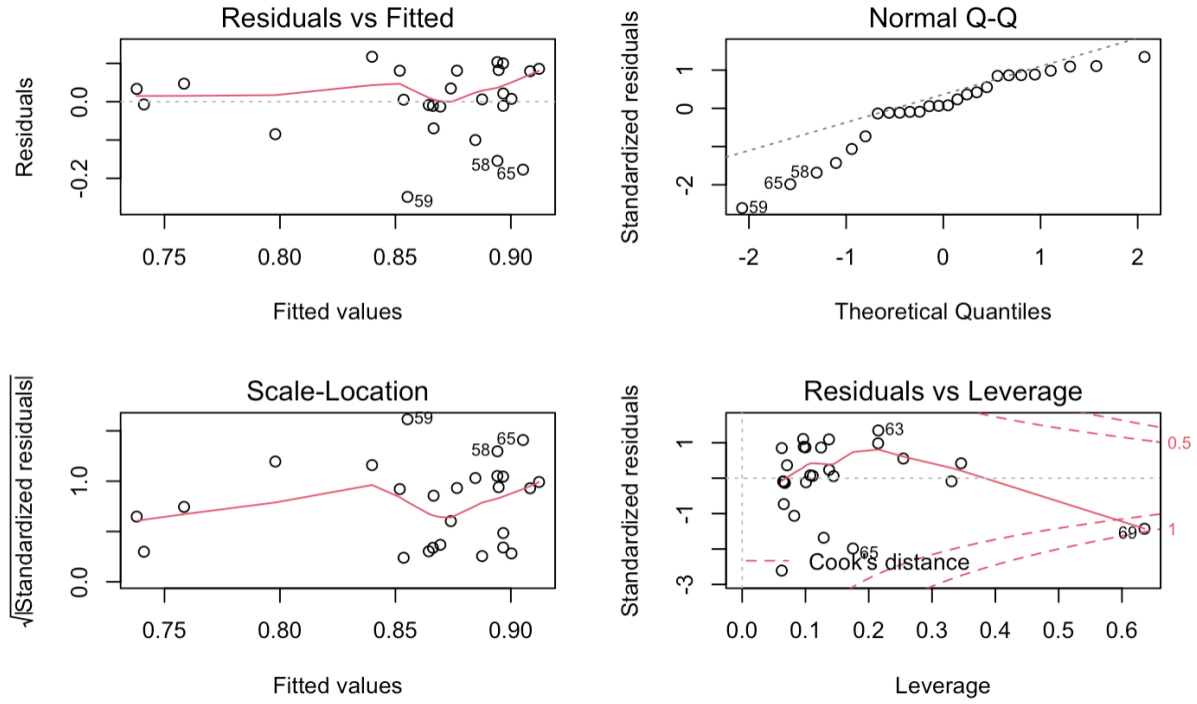

Figure 6.6: Open Source Linear Model Diagnostic Plots

From the context of our research question, we know that we have not violated independence to a great degree.

There is some evidence against normality. From these Q-Q plot in Figure 6.6, we are seeing that we are getting standardized residual values way below -2. The points do seem to generally follow the dashed line in the Q-Q plot, with a bit of left skew or negative skew.

In this asymmetric distribution, since the distribution has negative skew, this indicates that the distribution has a longer left tail [12].

When looking our histogram for the frequency of TSIs in open source projects (Figure 6.4), we can see a longer left tail and that most of the data is concentrated on the far right side of the histogram, with values trailing through the left tail.

There is also some evidence against linearity and equal variance, as neither the residuals vs. fitted plot nor the scale-location plot follow a strictly horizontal line. However, both lines overall closely follow a horizontal line. Additionally, our open source models only contain 26 repositories. This relatively small sample size could explain some of the randomness in these plots. Our model should be robust enough to allow for a small amount of evidence against the assumptions [22].

Now that we have determined that the assumptions have not been violated to a great degree, we fit the linear model to observe if there are any significant variables. We have created a plot to map the linear regression line on the data, separating the open source project data into their two respective types: library and application. This is shown in Figure 6.7.

As we can see in the plot, our y-intercept for open source projects of type application is 0.89 and our y-intercept for open source projects of type library is 0.88. This means that given an open source project of size 0 lines of code, the regression lines predict an average TSI of 0.89 for open source projects of type application and an average TSI of 0.88 for open source projects of type library. In the context of our model, we would not have a source code project with 0 lines of code. Therefore, the y-intercepts do not have practical meaning; they are just used to measure where our regression lines cross the y-axis.

The slope for type application projects is steeper than the slope of type library projects, with the slope of type application being -7.129e-7 and the slope of type library being -1.864e-7. These slopes indicate that for every 100,000 additional lines of code in an open source

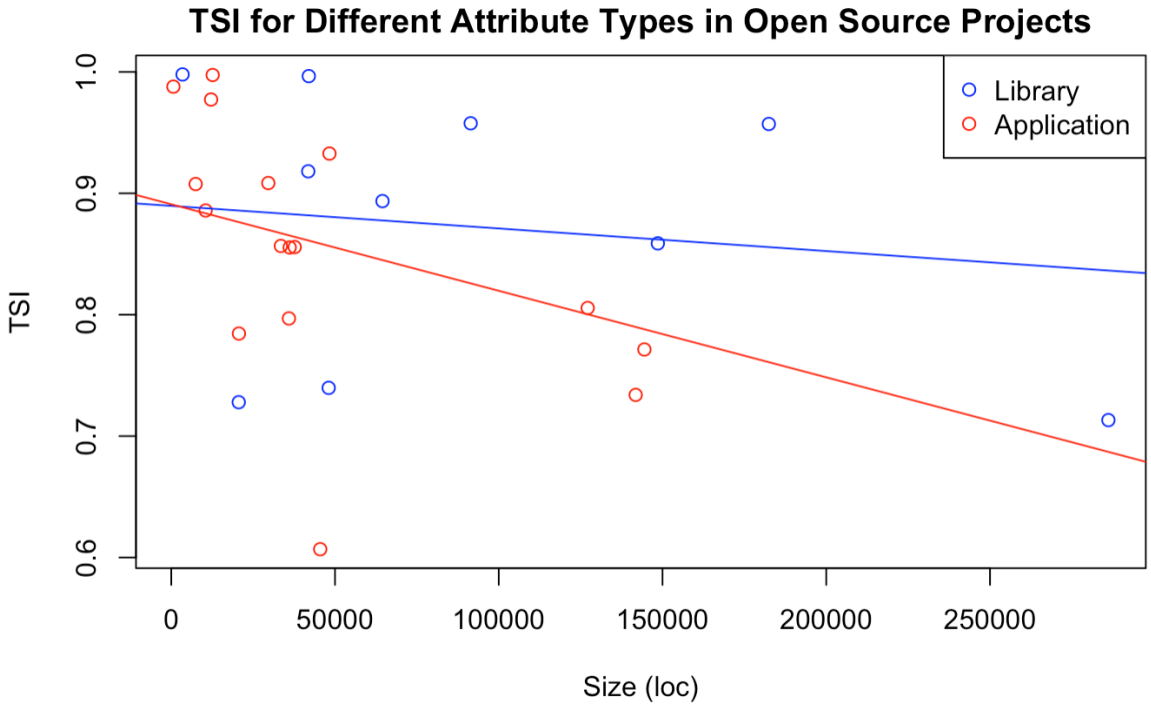

Figure 6.7: TSI for Different Attribute Types in Open Source Projects

project, the predicted TSI decreases by about 0.07 in type application projects and decreases by about 0.02 in type library projects.

After fitting our linear model for open source projects, we can see from Table 6.1 there is only one variable that has a p-value less than 0.05. This is denoted with an asterisk (*) after the value within Table 6.1 for easy identification.

Table 6.1 contains our linear model coefficient p-values from open source projects. These are separated into columns based on the model coefficient: intercept, size, type, and the interaction between size and type.

When looking at the linear regression results in open source projects for the size attribute, we found that there is strong evidence against the null hypothesis that there is no correlation between the size (lines of code) of the C# project under analysis and the TSI (p-value = 0.0373).

Table 6.1: TSI Linear Model Coefficient P-Values from Open Source Projects

	Intercept	Size	Type	Size:Type
TSI	2.00E-16	0.0373*	0.9392	0.2435

When looking at the linear regression results in open source projects for the type attribute, we found that there is there is little to no evidence against the null hypothesis that there is no correlation between the type (library or application) of the C# project under analysis and the TSI (p-value = 0.9392).

Finally, when looking at the linear regression results in open source projects for the interaction between the size and type attributes, we found that there is there is little to no evidence against the null hypothesis that there is no correlation between the interaction between size and type of the C# project under analysis and the TSI (p-value = 0.2435).

Based on these results, we may therefore reject the null hypothesis $H1_0$, but we may not reject $H2_0$ and $H3_0$ for our open source linear models.

Additionally, we have obtained an adjusted R-squared value of 0.1197, which means that our linear model on open source projects accounts for 11.97% of the variance.

To test our three null hypotheses, we created a linear model for closed source projects that is the TSI as a function of the interaction between size and type. We are subsetting the data to only include projects that are closed source.

This linear model is as follows:

$$\hat{TSI} = \beta_0 + \beta_1 size + \beta_2 type + \beta_3 size * type, \tag{6.2}$$

Where β_0 is the y-intercept and β_1, β_2, and β_3 are all regression coefficients. This linear model is represented as follows in R code:

```
lm(TSI ~ Size * Type, data = attribute[attribute$Source == "Closed",])
```

After fitting the above linear regression model, we assessed to what degree the assumptions are violated since the assumptions can never be fully met. The diagnostic plots for this model are shown in Figure 6.8.

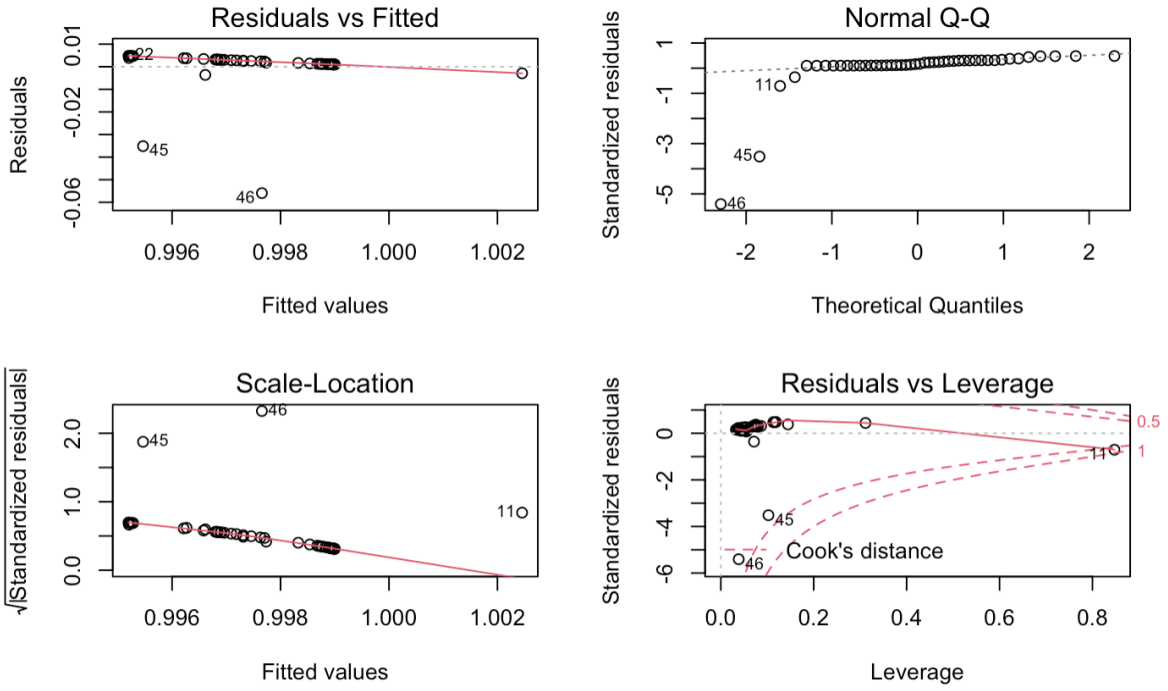

Figure 6.8: Closed Source Linear Model Diagnostic Plots

From the context of our research question, we know that we have not violated independence to a great degree. And from these plots, we can conclude that neither normality nor linearity appear to have been violated to a great degree. The points seem to generally follow the dashed line in the Q-Q plot, with a bit of left skew or negative skew.

In this asymmetric distribution, since the distribution has negative skew, this indicates that the distribution has a longer left tail [12].

When looking our histogram for the frequency of TSIs in closed source projects (Figure 6.5), we can see a longer left tail and that most of the data is concentrated on

the far right side of the histogram, with values trailing through the left tail.

There is some evidence against equal variance, as the scale-location plot does not follow a horizontal line. Our model should be robust enough to allow for a small amount of evidence against the assumptions [22].

Now that we have determined that the assumptions have not been violated to a great degree, we fit the linear model to observe if there are any significant variables. We have created a plot to map the linear regression lines on the data, separating the closed source project data into their respective types: library and application. This is shown in Figure 6.9.

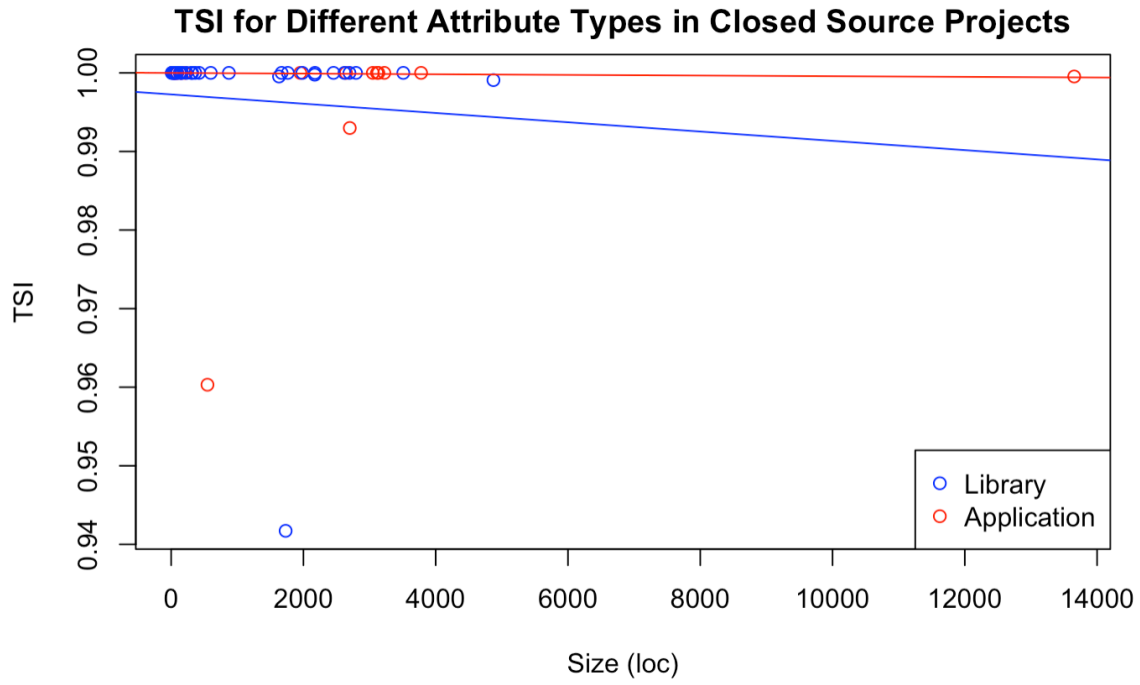

Figure 6.9: TSI for Different Attribute Types in Closed Source Projects

As we can see in the plot, our y-intercept for closed source projects of type application is 1.0 and our y-intercept for open source projects of type library is 0.997. This means that given a closed source project of size 0 lines of code, the regression lines predict an average TSI of 1.0 for closed source projects of type application and an average TSI of 0.997 for

closed source projects of type library. In the context of our model, we would not have a source code project with 0 lines of code. Therefore, the y-intercepts do not have practical meaning; they are just used to measure where our regression lines cross the y-axis.

The slope for type application projects is steeper than the slope of type library projects, with the slope of type application being -4.203e-8 and the slope of type library being -5.913e-7. These slopes indicate that for every 10,000 additional lines of code in a closed source project, the predicted TSI decreases by about 0.0004 in type application projects and decreases by about 0.006 in type library projects.

After fitting our model and looking at Figure 6.9, we decided that it did not make sense to run statistics on this subset of data.

Instead, we found it more interesting to look at the points in Figure 6.9 with the low TSIs and find out what potential differences there are with those projects' source code.

As we can see in the plot, nearly all the closed source project TSIs are very close to 1.0. We have three points in our plot below this TSI (the Project names are referenced from Appendix D):

Table 6.2: Attributes for Closed Source Projects with Low TSIs

Project Name	TSI	Size (loc)	Type
41	0.960	549	Application
42	0.942	1731	Library
43	0.993	2701	Application

We can rule out project type being the difference in these three projects compared to the rest of the closed source projects, as these three projects range in type. They also all are about in the middle of the range for size of closed source projects (11 to 13,654 lines of code for closed source projects) so they are not the smallest or largest projects in size.

We observed the source code to see if these three projects had more C# files to be analyzed than the other projects, but this did not appear to be the case (with many other closed source projects containing more and less C# files than these three). We also observed the number of external library calls in multiple closed source projects, but again, our three projects in Table 6.2 were not unique in this way either.

The three project names are sequential in order, which indicates that they are similar to each other alphabetically. This is an interesting finding because the fact that all three projects are alphabetically sequenced means that they all are projects that relate to similar topics. Without revealing too much about our stakeholder's repositories, all three repositories are named after either security or reports. This indicates that the closed source repositories that relate to security and reports may be more prone to lower TSIs. This is based on our initial observations and there is the potential for other similarities between these projects besides this to be the reason for their low TSIs.

Now that we have created linear models for open and closed source projects to observe potential attribute effects on the TSI, we now want to look at the seven Security Aspect nodes to observe if there are any significant findings. We will only spend time discussing the attributes in each Security Aspect's linear model that had a significant correlation ($p < 0.05$) between the attribute or interaction of attributes and the TSI.

The coefficient p-values for each variable in the linear model for each Security Aspect for the open source projects is in Table 6.3, and the coefficient p-values for each variable in the linear model for each Security Aspect for the closed source projects is in Table 6.4.

Table 6.3 and Table 6.4 contain our linear model coefficient p-values from open and closed source projects, respectively. These are separated into columns based on the coefficient: intercept, size, type, and the interaction between size and type.

As we can see from Table 6.3, there are only two variables across the seven Security Aspect nodes that have p-values less than 0.05. These are the variables of size for authenticity

Table 6.3: Security Aspect Linear Model Coefficient P-Values from Open Source Projects

	Intercept	Size	Type	Size:Type
Authenticity	2.00E-16	0.00524*	0.88011	0.12034
Availability	5.82E-16	0.801	0.392	0.713
Authorization	2.52E-15	0.00619*	0.51575	0.05339
Confidentiality	2.10E-15	0.11	0.805	0.296
Accountability	2.00E-16	0.0779	0.7944	0.274
Non-repudiation	3.79E-14	0.0606	0.7701	0.506
Integrity	2.00E-16	0.138	0.996	0.386

with a p-value of 0.00524, and size for authorization with a p-value of 0.00619. They are denoted with an asterisk (*) after the value within Table 6.3 for easy identification.

The linear models for authenticity and authorization also have higher adjusted R-squared values than the TSI linear model, being 0.2715 and 0.2145 respectively. This means that our linear model for authenticity accounts for 27.15% of the variation in that model, and our linear model for authorization accounts for 21.45% of the variation in that model.

As we can see from Table 6.4, there are also two variables across the seven Security Aspect nodes that have p-values less than 0.05. These are the variables of type for availability with a p-value of 0.0362, and type for accountability with a p-value of 0.0362. They are denoted with an asterisk (*) after the value within Table 6.4 for easy identification.

The linear models for availability and accountability do not have very high adjusted R-squared values, both at 0.0576. This means that our linear models for availability and accountability accounts for 5.76% of the variation in the models.

Table 6.4: Security Aspect Linear Model Coefficient P-Values from Closed Source Projects

	Intercept	Size	Type	Size:Type
Authenticity	2.00E-16	0.444	0.302	0.437
Availability	2.00E-16	0.3793	0.0362*	0.6618
Authorization	2.00E-16	0.984	0.46	0.463
Confidentiality	2.00E-16	0.998	0.868	0.601
Accountability	2.00E-16	0.3793	0.0362*	0.6618
Non-repudiation	2.00E-16	0.401	0.284	0.446
Integrity	2.00E-16	0.984	0.46	0.463

Count-Based Analysis of Chosen Tools

This section of the PIQUE-C#-Sec model validation is conducted to answer Q2 from our Goal Question Metric in the Approach chapter. Q2 asked how effective are the selected static analysis tools at measuring and reporting security vulnerabilities from the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list?

We can use all the PIQUE-C#-Sec model nodes from our model description file and compare those against the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list[3] to get an idea for how many of the top 25 CWEs of 2021 our static analysis tools and model structure provide coverage for.

The CWE Top 25 Most Dangerous Software Weaknesses for 2021 is a list by MITRE of the most common and impactful issues experienced over the previous two calendar years according to CWE's documentation[4]. These weaknesses are dangerous because they are often easy to find and exploit. They can allow adversaries to completely take over a system, steal

[3] https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html
[4] https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

data, or prevent an application from working. This list is a valuable community resource that can help users, testers, developers, researchers, project managers, security researchers, and educators provide insight into current and the most severe security weaknesses according to the CWE documentation[5].

The CWE Top 25 Most Dangerous Software Weaknesses for 2021 list is shown in Appendix C. There is an additional column indicating whether that CWE is represented as either a Diagnostic or Measure node within the PIQUE-C#-Sec model. If the CWE from the table is directly in the model, we indicate that with a "Yes" response in the column. We also traced each Top 25 CWE's relationships in the CWE documentation to see if it has a "ParentOf", "ChildOf", or "CanPrecede" relationship with any nodes in our PIQUE-C#-Sec model. If this was the case, this is indicated with a "By relation" response in the column.

The number of "Yes" responses totaled to 13. This means that our PIQUE-C#-Sec model has direct coverage for 52% of the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list. When we add the "By relation" responses to this count, it increases to 19 which covers 76% of the Top 25 list.

Having this level of coverage over the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list indicates that our selected static analysis tools are effective at measuring and reporting security vulnerabilities since our tools cover over a majority of the Top 25 list.

Sensitivity to Single Diagnostics

We applied an analysis of the sensitivity to single diagnostics within the PIQUE-C#-Sec model to answer Q3 from our Goal Question Metric in the Approach chapter. Q3 asked what is the impact on the Total Security Index and Security Aspects for each single vulnerability?

[5] `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`

To identify the impact that each vulnerability in the PIQUE-C#-Sec model has on the overall TSI and Security Aspects, we first run PIQUE-C#-Sec on a C# source code project. We observe the TSI and Security Aspect node values for this evaluation. Next, we take each of the Diagnostics in the PIQUE-C#-Sec model, as each represents its own vulnerability, and we add one instance of that diagnostic to the project's output file. Now that the PIQUE-C#-Sec model has evaluated the project with one extra diagnostic injected manually, we record the change in TSI and change in each of the Security Aspect values. This will allow us to observe any potential impacts of each Diagnostic.

This method of measuring single vulnerability impacts does not truly inject that vulnerable source code into the project; rather, we are modifying the output file to appear the way it would if our two static analysis tools would have found that vulnerability. This is because we want to measure the impact that finding those vulnerabilities would have on the PIQUE-C#-Sec node values if the vulnerabilities were found.

For this section of the model validation, we are not trying to measure whether our static analysis tools can find the vulnerabilities. We are interested in ensuring that the PIQUE-C#-Sec model can recognize these vulnerabilities from the output files if they were to be found and perform accordingly.

PIQUE-C#-Sec evaluated a closed source project which we will refer to as Project 42. For each of the 59 Diagnostics within the PIQUE-C#-Sec model, a PIQUE-C#-Sec model output file was created as if that vulnerability had been injected into Project 42. This allows us to measure the impact that each vulnerability has on the PIQUE-C#-Sec model nodes without conducting a true injection.

Figure 6.10 shows the change in TSI for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when measuring the change in TSI were CWE-554, SCS0017, SCS0021, SCS0022, SCS0023, SCS0024, and SCS0030. These Diagnostic descriptions are in Table 6.5.

It is also important to note that the bar heights in Figure 6.10 are not additive and do not sum to 1. If this was the case, it would be possible to receive negative TSIs which would not make sense for our model.

Table 6.5: Most Impactful Diagnostics on the TSI

Diagnostic	Description
CWE-554	ASP.NET Misconfiguration: Not Using Input Validation Framework
SCS0017	Request Validation Disabled (Attribute)
SCS0021	Request Validation Disabled (Configuration File)
SCS0022	Event Validation Disabled
SCS0023	View State Not Encrypted
SCS0024	View State MAC Disabled
SCS0030	Request Validation is Enabled Only for Pages (Configuration File)

These seven Diagnostics are the only ones in the PIQUE-C-Sec model that eventually aggregate upward into three Product Factors. They all aggregate upward into View State, Request Validation, and Other. Most of the other Diagnostics only aggregate upward into one Product Factor, and several aggregate upward to two Product Factors.

These seven Diagnostics aggregate upward to CWE-1173 at the Measure layer, which is Improper Use of Validation Framework[6]. CWE-1173 is also a listed member of the OWASP Top Ten 2021 list in the category of Insecure Design for 2021[7]. As a member of the OWASP Top Ten 2021 list, it makes sense that this CWE-1173 would aggregate upward to three Product Factors and impact the TSI so greatly.

The next seven figures show a similar view, but for each individual Security Aspect

[6] https://cwe.mitre.org/data/definitions/1173.html
[7] https://cwe.mitre.org/data/definitions/1348.html

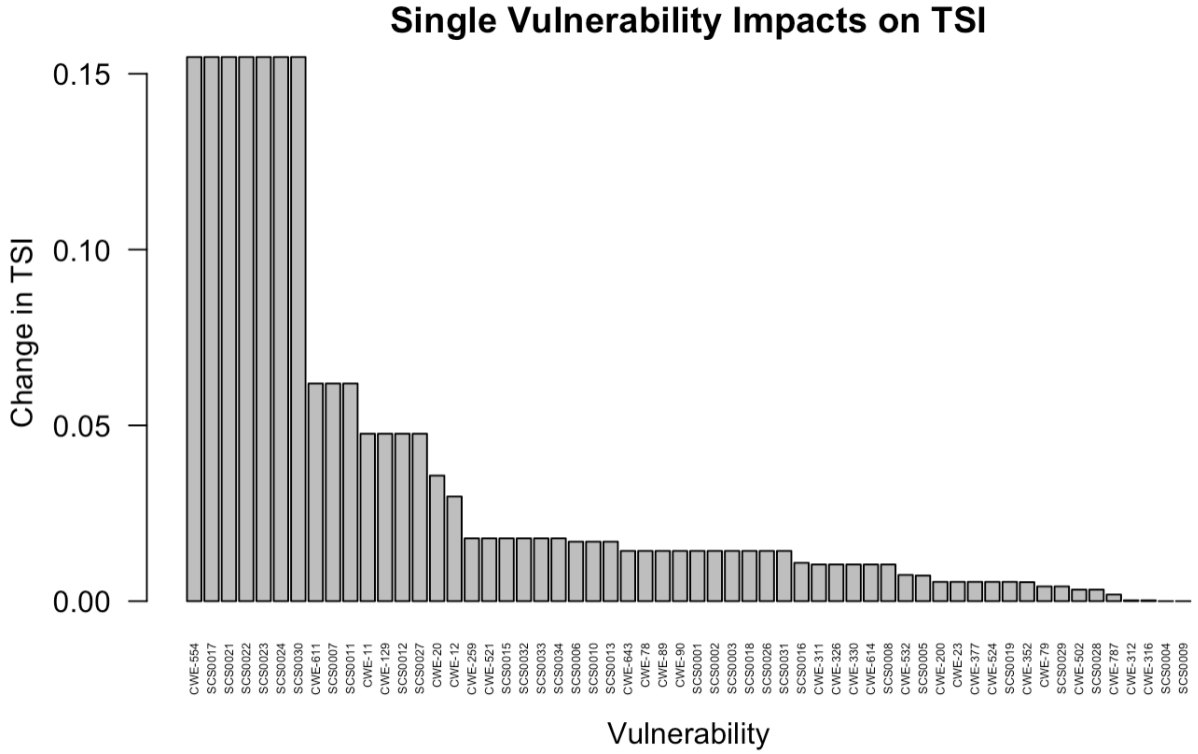

Figure 6.10: Single vulnerability impacts on TSI using the diagnostics in the PIQUE-C#-Sec model to measure the change in TSI.

(Authenticity, Availability, Authorization, Confidentiality, Accountability, Non-repudiation, and Integrity).

Figure 6.11 shows the change in authenticity for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when measuring the change in authenticity were CWE-259, CWE-521, SCS0015, SCS0032, SCS0033, and SCS0034. These Diagnostic descriptions are in Table 6.6.

All these six Diagnostics fall under the Product Factor node Password Management. Therefore, it makes sense that these Diagnostics would impact Authenticity since Password Management links to the Security Aspect node Authenticity.

Figure 6.12 shows the change in availability for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when

Table 6.6: Most Impactful Diagnostics on Authenticity

Diagnostic	Description
CWE-259	Use of a Hard-coded Password
CWE-521	Weak Password Requirements
SCS0015	Hardcoded Password
SCS0032	PasswordRequiredLength Too Small
SCS0033	Password Complexity
SCS0034	Password RequiredLength Not Set

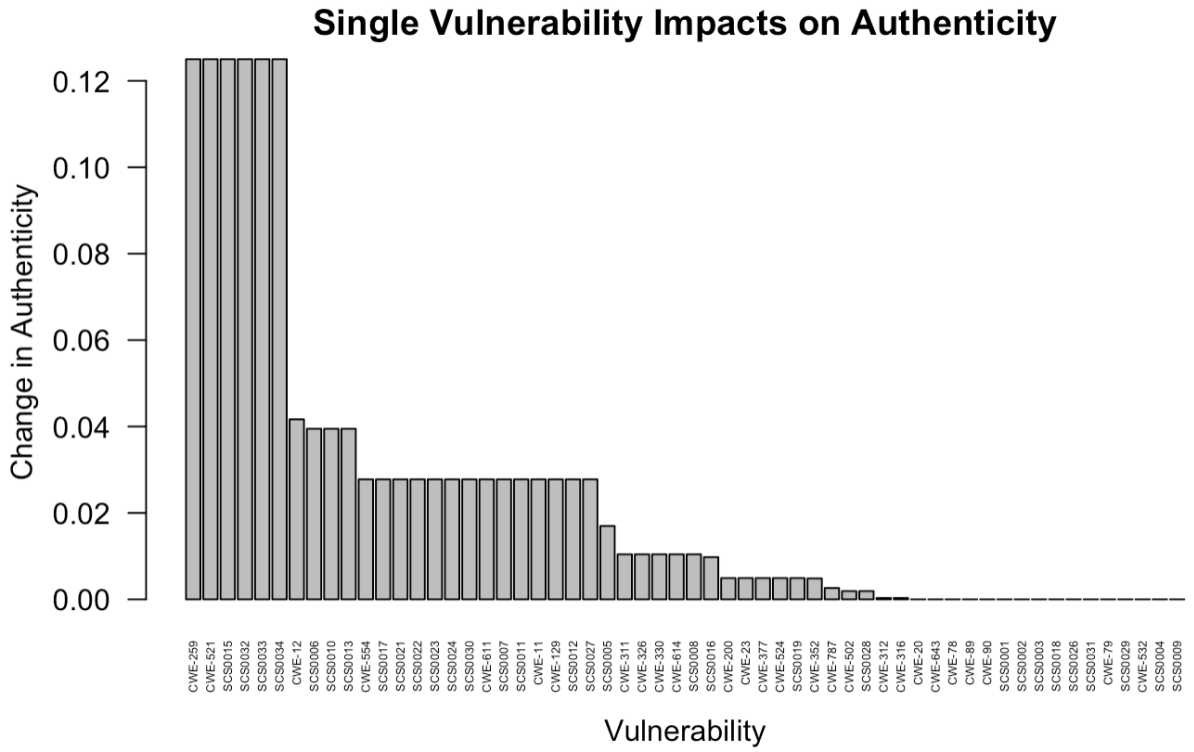

Figure 6.11: Single vulnerability impacts on authenticity using the diagnostics in the PIQUE-C#-Sec model to measure the change in authenticity.

measuring the change in availability were CWE-554, SCS0017, SCS0021, SCS0022, SCS0023, SCS0024, and SCS0030.

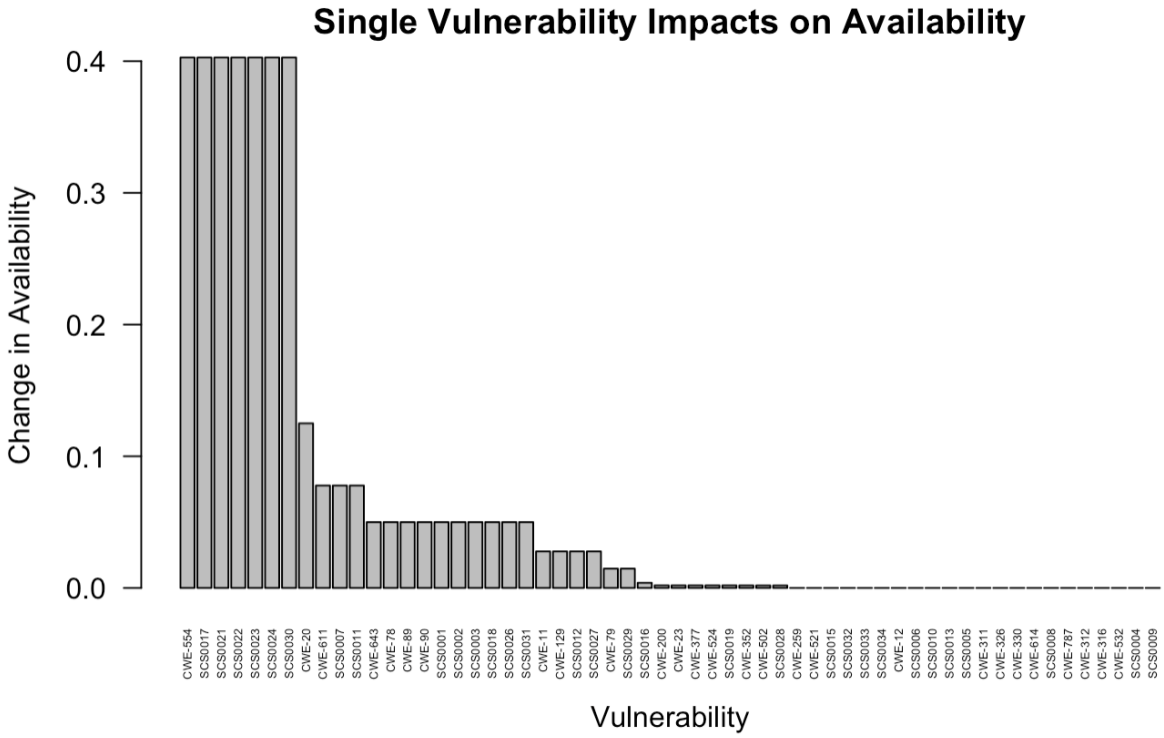

Figure 6.12: Single vulnerability impacts on availability using the diagnostics in the PIQUE-C#-Sec model to measure the change in availability.

The seven Diagnostic nodes that have the greatest impact on Availability are also the same seven that have the greatest impact on TSI. This makes sense because as mentioned earlier, these seven Diagnostics map to the Product Factors View State, Request Validation, and Other. These three Product Factors aggregate upward to be three of the four total Product Factors mapped to Availability (with the other being Injection).

Figure 6.13 shows the change in authorization for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when measuring the change in authorization was CWE-12 (ASP.NET Misconfiguration: Missing Custom Error Page). When taking a closer look at CWE-12's documentation[8], we can see that it is a member of the OWASP Top Ten 2004 for the category of Insecure Configuration Management [9]. As a member of the OWASP Top Ten 2004 list, it makes sense that this CWE-12 would have an impact on Authorization.

[8] https://cwe.mitre.org/data/definitions/12.html
[9] https://cwe.mitre.org/data/definitions/731.html

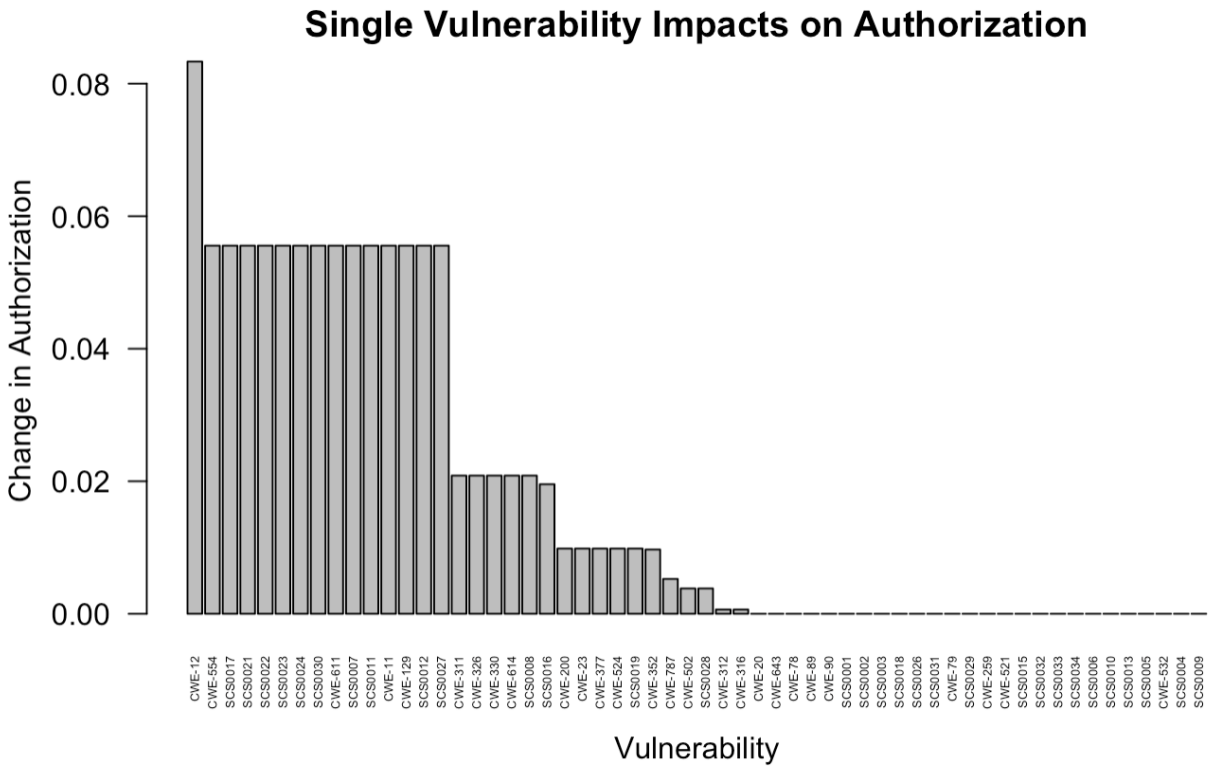

Figure 6.13: Single vulnerability impacts on authorization using the diagnostics in the PIQUE-C#-Sec model to measure the change in authorization.

Figure 6.14 shows the change in confidentiality for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when measuring the change in confidentiality were CWE-554, SCS0017, SCS0021, SCS0022, SCS0023, SCS0024, and SCS0030.

The seven Diagnostic nodes that have the greatest impact on Confidentiality are also the same seven that have the greatest impact on TSI.

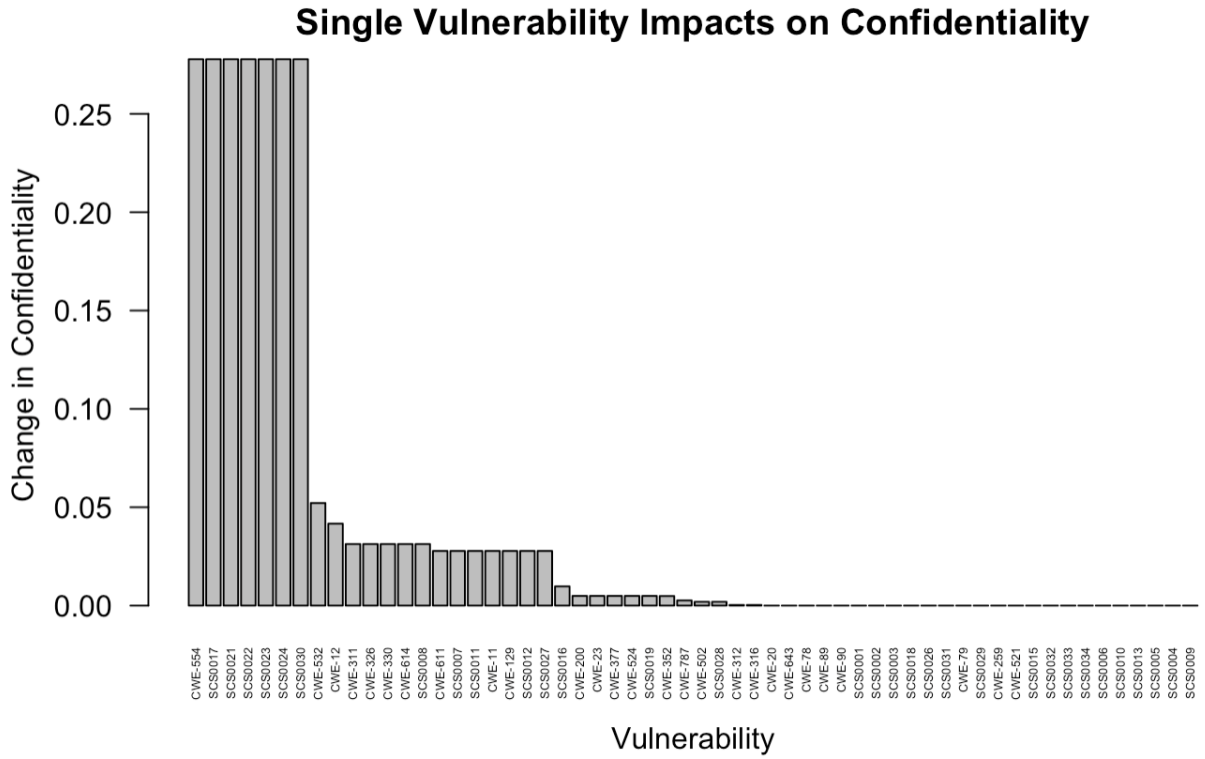

Figure 6.14: Single vulnerability impacts on confidentiality using the diagnostics in the PIQUE-C#-Sec model to measure the change in confidentiality.

Figure 6.15 shows the change in accountability for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when measuring the change in accountability were CWE-554, SCS0017, SCS0021, SCS0022, SCS0023, SCS0024, SCS0030, CWE-611, SCS0007, SCS0011, CWE-11, CWE-129, SCS0012, and SCS0027. The first seven Diagnostics are the same as in Table 6.5, but the next seven Diagnostics descriptions are in Table 6.7.

These Diagnostics are mapped to the Product Factor Other. Accountability is only mapped to one Product Factor node, which is Other. This explains why these Diagnostics impact accountability so greatly.

Table 6.7: Most Impactful Diagnostics on Accountability

Diagnostic	Description
CWE-611	Improper Restriction of XML External Entity Reference
SCS0007	XML eXternal Entity Injection
SCS0011	Unsafe XSLT Setting Used
CWE-11	ASP.NET Misconfiguration: Creating Debug Binary
CWE-129	Improper Validation of Array Index
SCS0012	Controller Method is Potentially Vulnerable to Authorization Bypass
SCS0027	Open Redirect

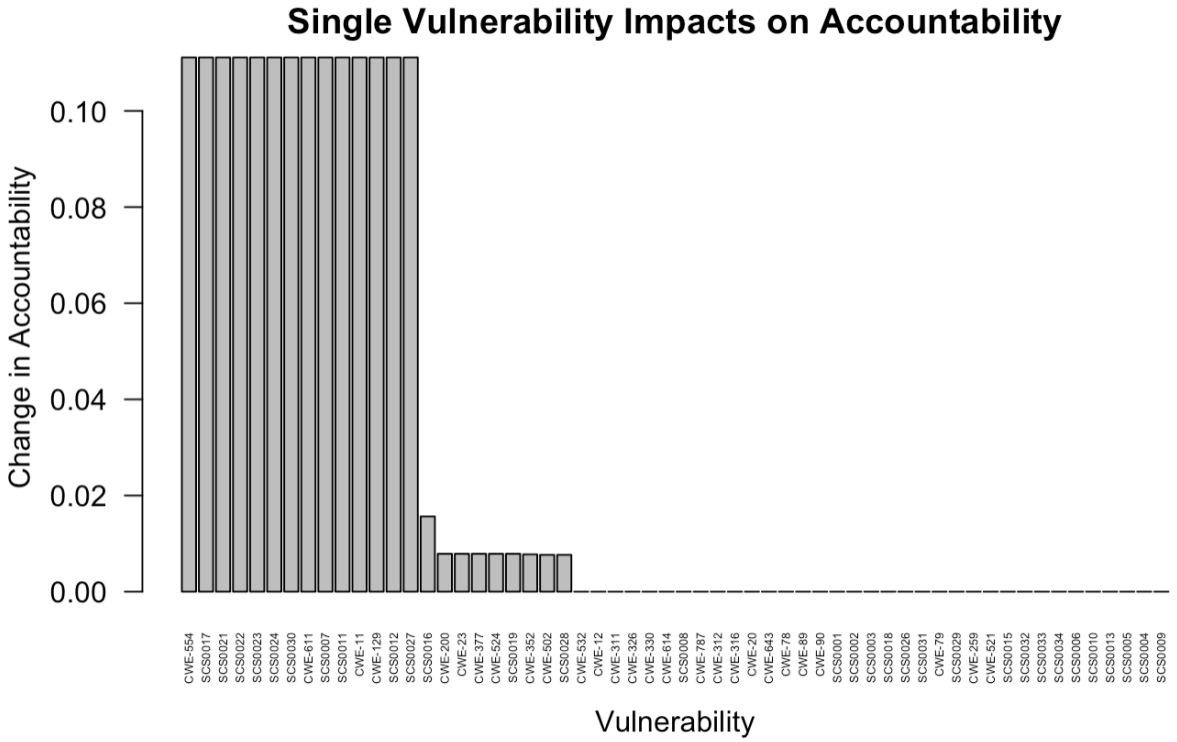

Figure 6.15: Single vulnerability impacts on accountability using the diagnostics in the PIQUE-C#-Sec model to measure the change in accountability.

Figure 6.16 shows the change in non-repudiation for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when measuring the change in non-repudiation were SCS0006 (Weak Hashing Function), SCS0010 (Weak Cipher Algorithm), and SCS0013 (Potential Usage of Weak CipherMode Mode). This makes sense because all these Diagnostics fall under the Product Factor Cryptography, which aggregates upward to the Security Aspect Non-Repudiation.

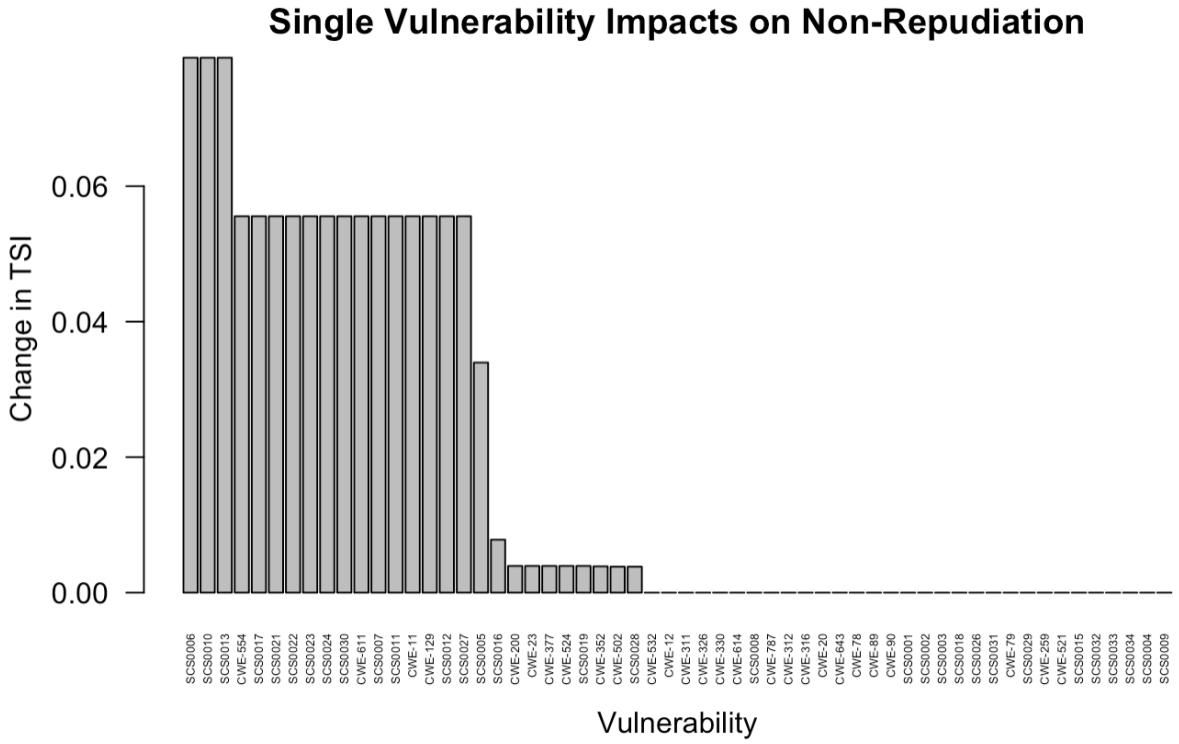

Figure 6.16: Single vulnerability impacts on non-repudiation using the diagnostics in the PIQUE-C#-Sec model to measure the change in non-repudiation.

Figure 6.17 shows the change in integrity for each individual vulnerability within the PIQUE-C#-Sec model when evaluating Project 42. The most impactful Diagnostics when measuring the change in integrity were CWE-554, SCS0017, SCS0021, SCS0022, SCS0023, SCS0024, and SCS0030.

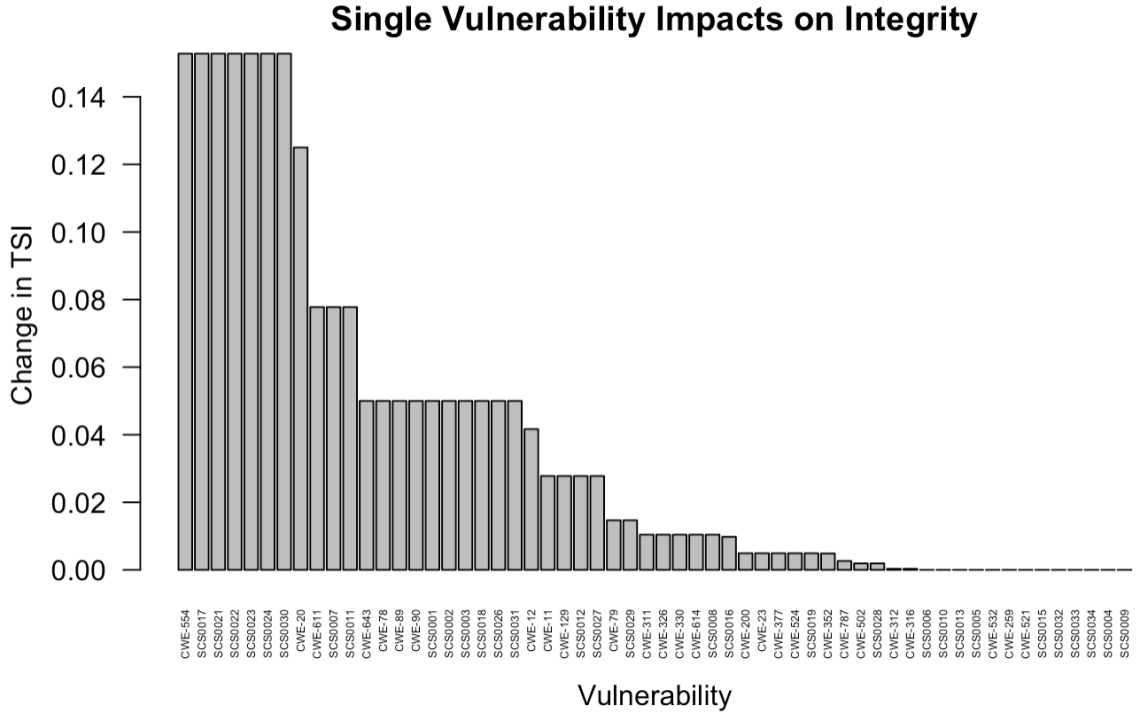

Figure 6.17: Single vulnerability impacts on integrity using the diagnostics in the PIQUE-C#-Sec model to measure the change in integrity.

The seven Diagnostic nodes that have the greatest impact on Integrity are also the same seven that have the greatest impact on TSI.

Now that we have visualized the impacts of single vulnerabilities for the TSI and each Security Aspect, we will isolate the Diagnostics that have the greatest impact on Project 42's TSI and observe in one plot the impact that those Diagnostics have on all the TSI and Security Aspect node values. This is shown in Figure 6.18.

The Diagnostics that had the greatest impact were CWE-554, SCS0017, SCS0021, SCS0022, SCS0023, SCS0024, and SCS0030. Figure 6.18 represents the Change in Values for all seven of these Diagnostics. All these seven Diagnostics have the same impact on all our TSI and Security Aspect node values, so Figure 6.18 is representative of all seven Diagnostics. If we made a separate plot for each Diagnostic, they would all appear the

same as Figure 6.18. We can see that the impact on availability is the highest (change in availability = 0.403), followed by confidentiality (change in confidentiality = 0.278), followed by TSI (change in TSI = 0.155), and then integrity (change in integrity = 0.153), just to list several of the properties in order.

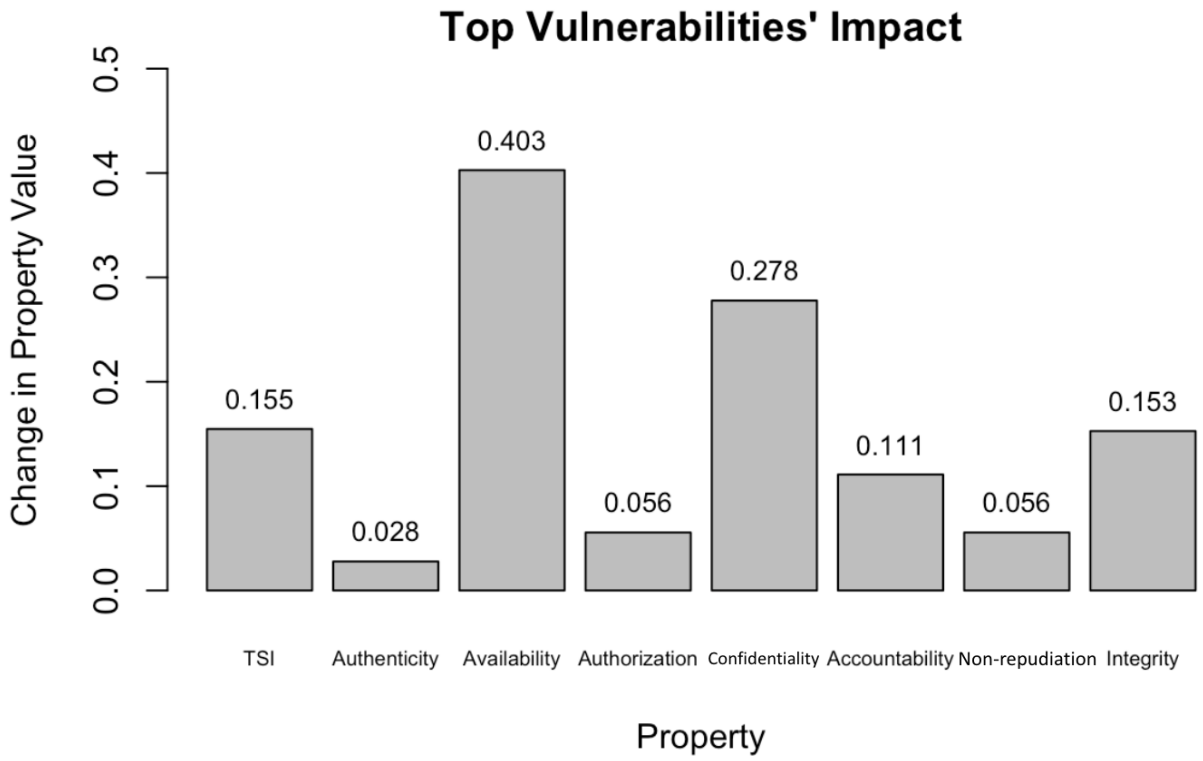

Figure 6.18: Top Vulnerabilities' Impact on PIQUE-C#-Sec TSI and Security Aspect Values

DISCUSSION

The two linear models we have fit for open and closed source projects indicate that for open source projects, the correlation between the size (lines of code) of a C# project under analysis and the TSI is statistically significant (p-value = 0.0373). However, type and the interaction between size and type are not statistically significant. This means that when creating a benchmark repository for open source C# source code projects, the size attribute must be considered because of its statistical significance.

For closed source projects, we did not run statistical analysis on the data as after a visual observation of the linear regression plot, we decided that statistical analysis would not yield any interesting or significant findings. However, we did look at three closed-source projects that had lower TSIs than the rest of the closed source project population. These three projects are all named after either security or reports. This is interesting because it indicates that the closed source repositories that relate to security and reports may be more prone to lower TSIs. This is based on our initial observations and there is the potential for other similarities between these projects besides this to be the reason for their low TSIs.

Our PIQUE-C#-Sec model nodes cover 13 of the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list directly, and 6 additional CWEs with a "ParentOf", "ChildOf", or "CanPrecede" relationship. This indicates that our model has adequate coverage over the current top 25 most common and impactful issues (according to the list by MITRE).

This level of coverage of current dangerous security weaknesses indicates that the PIQUE-C#-Sec model is a good hierarchical model for users because it is successfully detecting these most dangerous software weaknesses.

Additionally, we found that our PIQUE-C#-Sec model TSI is most impacted by the Diagnostics in Table 6.5, which fall under the View State, and Request Validation, and Other Product Factors. This indicates that for the current hierarchical model structure with the

NaiveWeighting, stakeholders especially interested in these Product Factors would likely see high impacts in those areas.

Our work in developing and validating the PIQUE-C#-Sec model is significant because we have created a security quality model for both practitioners and researchers alike to use to evaluate source code projects written in C# and further advance the field of security quality modeling.

Implications

The implication for practitioners is that PIQUE-C#-Sec is an effective security quality model to use to help assess the security of C# source code projects. This is especially true in the case of open source projects. Stakeholders can use PIQUE-C#-Sec as a security quality gate when evaluating C# source code written by government contractors to ensure that a threshold TSI value is being met.

The implication for academics and researchers is that we have developed and validated the PIQUE-C#-Sec security quality model, and other operationalized PIQUE security quality models or quality models can be built based on the PIQUE-C#-Sec technologies for other specific areas of evaluation (for example, source code written in other languages).

Future Work

While the development and validation of the PIQUE-C#-Sec model is at a code complete stage, there are several ways in which further work could be done. This includes adding additional C# source code projects to the PIQUE-C#-Sec benchmark repository. It would be interesting to seek out closed source projects of larger size in lines of code and open and closed source projects of known poor security quality to start. Adding additional static analysis tools to the PIQUE-C#-Sec model also is also a good direction in which to pursue

further research.

Additionally, while the process has started for packaging PIQUE-C#-Sec as a jar file, further work needs to be done on continuing to test this jar file onto stakeholder systems and deploying the packaged jar file within stakeholder continuous integration and continuous deployment (CI/CD) environments.

Finally, further research needs to be conducted on using PIQUE-C#-Sec to evaluate closed source projects. We can add closed source projects into our benchmarks that are of larger size in lines of code as mentioned previously, or change the mechanisms that PIQUE-C#-Sec uses to see if any other strategies are a better fit for closed source projects (thresholding, weighting, utility functions). This may involve continuing to write new custom functions.

THREATS TO VALIDITY

Internal Validity

In this project, the threats to internal validity were the potential for confounding variables that were unaccounted for and the manual tagging of the "type" attribute in the benchmark data.

Threats to the internal validity of this study are threats that are of concern when causal relations are examined. When examining whether one factor affects a second factor there is a risk that the second factor is also affected by a third factor [45]. When this third factor or confounding variable is not identified and taken into account, this creates a threat to the internal validity.

In this thesis, one threat to the internal validity in this project could be confounding variables that we have not accounted for causing us to make conclusions about certain variables that do not truly have an effect on the tools' outputs [22].

One example of this is the manual tagging of the "type" attribute in the benchmark data. For each benchmark project, the documentation and source code were observed to decide whether that project was of type application or library. This manual process could have potentially led to some "type" attributes being incorrectly assigned.

External Validity

In this project, the threats to the external validity were that the C# source code projects used in the benchmark repository were not collected randomly, that the PIQUE-C#-Sec model Diagnostics do not provide coverage for every security vulnerability that exists, that our model may not be applicable to source code projects that are different from the ones we benchmarked, and that the static analysis tool documentation will change as

more findings and rules are added into what the tools can detect.

Threats to the external validity of this study are threats that are concerned with to what extent it is possible to generalize findings, and how relevant those findings are to people outside the investigated case [45].

To start, one inherent threat to the external validity in this thesis is that the C# source code projects that were collected for the benchmark repository were not collected randomly. There is no way to collect the projects randomly, and additionally we wanted to collect benchmarks that would represent the types of projects that PIQUE-C#-Sec is expected to evaluate. However, this means that any conclusions that we make will be limited to the population of projects that we have collected and cannot be extended to a broader population since there was not random sampling.

Another threat to the external validity in this thesis is the fact that the PIQUE-C#-Sec model Diagnostics do not provide coverage for every security vulnerability that exists. There are security vulnerabilities that will not be detected by our PIQUE-C#-Sec model. This threat can be mitigated as the static analysis tools in the POQUE-C#-Sec model update their documentation and rules to include more known security vulnerabilities over time. Additionally, this threat can be mitigated by integrating additional static analysis tools to the model.

The benchmarks chosen for the PIQUE-C#-Sec model derivation reflects the type of source code projects that we expect to apply the PIQUE-C#-Sec model to. However, this model may not be applicable to source code projects that are different from the ones we benchmarked.

Additionally, a threat to the external validity in this thesis is the static analysis tool documentation changing as more findings and rules are added into what the tools can detect. If either Security Code Scan or Insider add vulnerability rules to their documentation, this would put the current PIQUE-C#-Sec model Diagnostics out of date with the full scope of

what the tools are able to find. This affects the study's ability to generalize to other settings and be replicated by other users and researchers [22].

Construct Validity

In this project, the threats to construct validity were the static analysis tools used, modifying the PIQUE-C#-Sec model output files to make it appear as though vulnerabilities were injected but not altering the lines of code of the project, and assessing the assumptions of independence, normality, linearity, and equal variance for our linear regression models.

Threats to the construct validity of this study are threats that reflect to what extent the operational measures that are studied really represent what the researcher has in mind and the research questions [45]. A threat to the construct validity can be when data, measurements, or questions are not clear or measuring what they are intended to measure.

Threats to the construct validity in this thesis are the static analysis tools used. The static analysis tools that we selected to integrate into our PIQUE-C#-Sec model may not be representative of other similar tools. Other tools may find different vulnerabilities. Additionally, developers trust the tool and often assume that the tool is correct in its detection [9]. The false positive rate for a tool is an important factor in its usefulness and is a threat to be considered with the two tools used in our model.

Another threat to the construct validity of this project is that by modifying the PIQUE-C#-Sec model output files to make it appear as though vulnerabilities were injected; we did not alter the lines of code of the project even though it technically contains a new vulnerability. This affects the normalization that the PIQUE-C#-Sec model uses. However, since we are not truly injecting the vulnerabilities into the source code, we do not know for sure how many lines of code should be attributed to each vulnerability type.

Additionally, assessing the assumptions of independence, normality, linearity, and equal variance for our linear regression models is a threat to the construct validity because our

assumptions will never be fully met. Instead, we assess to what degree the assumptions are violated.

CONCLUSION

This thesis detailed a hierarchical security quality model for projects with C# source code, PIQUE-C#-Sec. We presented the design, development, and model validation of PIQUE-C#-Sec.

We began by giving a quick background on relevant topics to the model, including security quality modeling, security metrics, and static analysis tools. Next, we gave an overview on the supporting work in this field, which consists of the prior hierarchical model designs Quamoco, QATCH, PIQUE, PIQUE-C#, and PIQUE-Bin.

The research around creating and validating the PIQUE-C#-Sec model was guided by outlining our research goal in the Goal Question Metric (GQM) format from Basili [10]. Our motivation for designing and developing this security quality model is to provide a modern security quality model that provides adequate support for security characteristics. Our research goal is to design and develop a security quality model that helps stakeholders assess the security of C# source code projects.

The design of the PIQUE-C#-Sec model's hierarchical structure has a root TSI node. At the Security Aspect layer, these nodes are defined by the ISO/IEC 25010 standard and Microsoft STRIDE model. The two static analysis tools in the PIQUE-C#-Sec model are Security Code Scan and Insider, and their documented findings make up the 59 nodes in the Diagnostic layer. Each Diagnostic maps to a CWE, which allows us to aggregate the Diagnostics upward to CWEs at the Measure layer.

Next, we provide an overview to the PIQUE-C#-Sec model validation and the results from exploring our three research questions. Our sensitivity analysis has three main components. First, we analyzed PIQUE-C#-Sec's benchmark projects and the attribute data associated with those projects to determine if any attributes in C# source code projects result in a different TSI and different Security Aspect scores. We found that according to

the two linear models we fit for open and closed source projects, the correlation between the size (lines of code) of a C# project under analysis and the TSI is significant in open source projects. However, type and the interaction between size and type are not. For closed source projects, we did not find any attributes or interactions of attributes to be significantly correlated to the TSI.

Therefore, we can conclude that when creating a benchmark repository for open source C# source code projects, the size attribute must be considered because of its statistical significance.

Next, we observed how effective the selected static analysis tools within the PIQUE-C#-Sec model are at measuring and reporting security vulnerabilities from the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list by comparing all 59 Diagnostic nodes in our model to the CWE Top 25 list. Our PIQUE-C#-Sec model has direct coverage for 52% of the CWE Top 25 Most Dangerous Software Weaknesses for 2021 list. When we add the related responses to this count, it increases to 19 which covers 76% of the Top 25 list.

We conclude that our selected static analysis tools are effective at measuring and reporting vulnerabilities from the Top 25 list since our tools cover over a majority of the list, which represents the most dangerous weaknesses that are commonly experienced. These are impactful weaknesses that we want our PIQUE-C#-Sec model to detect, and our model detects most of them.

Finally, we analyzed the impact that each single vulnerability within our model had on the TSI and Security Aspect node values. We found seven Diagnostics within the PIQUE-C#-Sec model to have the greatest impact on the TSI. These Diagnostics are CWE-554, SCS0017, SCS0021, SCS0022, SCS0023, SCS0024, and SCS0030. All these seven Diagnostics aggregate upward to the Product Factors of Request Validation, View State, and Other.

These seven Diagnostics aggregate upward to CWE-1173 at the Measure layer, which is a member of the OWASP Top Ten 2021 list. These seven Diagnostics also aggregate upward

into the Product Factors View State, Request Validation, and Other.

We then discuss the implications of these results and address some potential areas for future work for the PIQUE-C#-Sec model.

The implication for practitioners is that PIQUE-C#-Sec is an effective security quality model to use to help assess the security of C# source code projects. Stakeholders can use PIQUE-C#-Sec as a security quality gate when evaluating C# source code written by government contractors.

The implication for academics and researchers is that we have developed and validated the PIQUE-C#-Sec security quality model, and other operationalized PIQUE security quality models or quality models can be built based on the PIQUE-C#-Sec technologies for other specific areas of evaluation.

We have successfully addressed our research goal of designing, developing, and validating a security quality model that helps stakeholders assess the security of C# source code projects. While there will always be future work to be done regarding the PIQUE-C#-Sec model, the work done so far has resulted in many improvements in hierarchical security quality modeling.

REFERENCES CITED

[1] Inl qa white paper 1.1 (case study), 2021.

[2] Adewole Adewumi, Sanjay Misra, and Nicholas Omoregbe. Evaluating open source software quality models against iso 25010. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 872–877, 2015.

[3] Sabreen Ahmadjee, Carlos Joseph Mera-Gómez, and Rami Bahsoon. Assessing smart contracts security technical debts. *CoRR*, abs/2103.09595, 2021.

[4] Kenneth Alperin, Allan Wollaber, Dennis Ross, Pierre Trepagnier, and Leslie Leonard. Risk prioritization by leveraging latent vulnerability features in a contested environment. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 49–57, 2019.

[5] Tiago L. Alves, José Pedro Correia, and Joost Visser. Benchmark-based aggregation of metrics to ratings. *Proceedings - Joint Conference of the 21st International Workshop on Software Measurement, IWSM 2011 and the 6th International Conference on Software Process and Product Measurement, MENSURA 2011*, pages 20–29, 2011.

[6] Ross Anderson and Tyler Moore. The economics of information security. *Science*, 314(5799):610–613, 2006.

[7] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. *International Symposium on Empirical Software Engineering and Measurement*, pages 97–106, 2011.

[8] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[9] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities - does experience matter? In *2009 International Conference on Availability, Reliability and Security*, pages 804–810, 2009.

[10] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. The goal question metric approach. *Encyclopedia of Software Engineering*, 2:528–532, 1994.

[11] Barry W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, 1984.

[12] G. Brys, M. Hubert, and A. Struyf. A comparison of some new measures of skewness. In Rudolf Dutter, Peter Filzmoser, Ursula Gather, and Peter J. Rousseeuw, editors, *Developments in Robust Statistics*, pages 98–113, Heidelberg, 2003. Physica-Verlag HD.

[13] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, 2004.

[14] Puneet Kumar Goyal and Gamini Joshi. Qmood metric sets to assess quality of java program. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 520–533, 2014.

[15] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, 2007.

[16] Hannes Holm and Khalid Khan Afridi. An expert-based investigation of the Common Vulnerability Scoring System. *Computers and Security*, 53:18–30, 2015.

[17] Hannes Holm, Mathias Ekstedt, and Dennis Andersson. Empirical analysis of system-level vulnerability metrics through actual attacks. *IEEE Transactions on Dependable and Secure Computing*, 9(6):825–837, 2012.

[18] Chien-Cheng Huang, Feng-Yu Lin, Frank Yeong-Sung Lin, and Yeali S Sun. A novel approach to evaluate software vulnerability prioritization. *The Journal of Systems and Software*, 86:2822–2840, 2013.

[19] Michael T. Hunter, Russell J. Clark, and Frank S. Park. Security issues with the ip multimedia subsystem (ims). In *Proceedings of the 2007 Workshop on Middleware for Next-Generation Converged Networks and Applications*, MNCNA '07, New York, NY, USA, 2007. Association for Computing Machinery.

[20] ISO/IEC 25010:2011 Systems and software engineering: Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models. Standard, International Organization for Standardization, Geneva.

[21] Clemente Izurieta, Isaac Griffith, and Chris Huvaere. An Industry Perspective to Comparing the SQALE and Quamoco Software Quality Models. *International Symposium on Empirical Software Engineering and Measurement*, 2017-Novem:287–296, 2017.

[22] Andrew Johnson. The analysis of binary file security using a hierarchical quality model. Master's thesis, Montana State University, 12 2021.

[23] Foutse Khomh and Yann-Gaël Guéhéneuc. Dequalite: Building design-based software quality models. In *Proceedings of the 15th Conference on Pattern Languages of Programs*, PLoP '08, New York, NY, USA, 2008. Association for Computing Machinery.

[24] Barbara Kitchenham, Steve Linkman, Alberto Pasquini, and Vincenzo Nanni. The squid approach to defining a quality model. *Software Quality Journal*, 6(3):211–233, 1997. Copyright - Chapman and Hall 1997; Last updated - 2021-09-11.

[25] Michael Kläs, Constanza Lampasona, and Jürgen Münch. Adapting software quality models: Practical challenges, approach, and first empirical results. *Proceedings - 37th*

EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, pages 341–348, 2011.

[26] James A. Kupsch, Elisa Heymann, Barton Miller, and Vamshi Basupalli. Bad and good news about using software assurance tools. *Software - Practice and Experience*, 47(1):143–156, 2017.

[27] Jean-Louis Letouzey. The sqale method for evaluating technical debt. 06 2012.

[28] R. Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704, 2005.

[29] R Martin. Common weakness enumeration (cwe v1.8). *National Cyber Security Division, US Dept. Of Homeland Security*, 2010.

[30] R.A. Martin. Managing vulnerabilities in networked systems. *Computer*, 34(11):32–38, 2001.

[31] G. McGraw and J. Steven. Software [in]security: Comparing apples, oranges, and aardvarks (or, all static analysis tools are not created equal), January 2011.

[32] Gary McGraw. Software security: Building security in. In *2006 17th International Symposium on Software Reliability Engineering*, pages 6–6, 2006.

[33] Daniel Mellado, Eduardo Fernández-Medina, and Mario Piattini. A comparison of software design security metrics. *ACM International Conference Proceeding Series*, (c):236–242, 2010.

[34] Padmalata Nistala, Kesav Vithal Nori, and Raghu Reddy. Software quality models: A systematic mapping study. In *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pages 125–134, 2019.

[35] Andreas L. Opdahl and Guttorm Sindre. Experimental comparison of attack trees and misuse cases for security threat identification. *Information and Software Technology*, 51(5):916–932, 2009. SPECIAL ISSUE: Model-Driven Development for Secure Information Systems.

[36] F. Ramsey and D. Schafer. *The Statistical Sleuth: A Course in Methods of Data Analysis*. Cengage Learning, 2012.

[37] David Rice. An extensible, hierarchical architecture for analysis of software quality assurance. Master's thesis, Montana State University, 12 2020.

[38] Thomas Saaty. Decision making with the analytic hierarchy process. *Int. J. Services Sciences Int. J. Services Sciences*, 1:83–98, 01 2008.

[39] Nahm Francis Sahngun. Nonparametric statistical tests for the continuous data: the basic concept and the practical use. *Korean J Anesthesiol*, 69(1):8–14, 2016.

[40] Miltiadis G. Siavvas, Kyriakos C. Chatzidimitriou, and Andreas L. Symeonidis. Qatch - an adaptive framework for software product quality assessment. *Expert Systems with Applications*, 86:350–366, 2017.

[41] K. Sivakumar and K. Garg. Constructing a "common cross site scripting vulnerabilities enumeration (cxe)" using cwe and cve. In Patrick McDaniel and Shyam K. Gupta, editors, *Information Systems Security*, pages 277–291, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[42] Stefan Wagner, Andreas Goeb, Lars Heinemann, Michael Kläs, Constanza Lampasona, Klaus Lochmann, Alois Mayr, Reinhold Plösch, Andreas Seidl, Jonathan Streit, and Adam Trendowicz. Operationalised product quality models and assessment: The quamoco approach. *Information and Software Technology*, 62:101–123, jun 2015.

[43] Stefan Wagner, Klaus Lochmann, Lars Heinemann, Michael Klas, Adam Trendowicz, Reinhold Plosch, Andreas Seidi, Andreas Goeb, and Jonathan Streit. The quamoco product quality modelling and assessment approach. *2012 34th International Conference on Software Engineering (ICSE)*, Jun 2012.

[44] Stefan Wagner, Klaus Lochmann, Sebastian Winter, Florian Deissenboeck, Elmar Juergens, Markus Herrmannsdoerfer, Lars Heinemann, Michael Kläs, Adam Trendowicz, Jens Heidrich, et al. The quamoco quality meta-model. 2012.

[45] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer Berlin Heidelberg, 2012.

APPENDICES

APPENDIX A

EXPLORATORY STUDY

Before designing the current PIQUE-C#-Sec model that is now in place, an initial exploratory study was conducted on a bottom-up design for the hierarchical model. This involved running PIQUE-C# on a single project repository and identifying that project's output findings. Then, the model was manually mapped upward into measures, product factors, and quality aspects that made sense for that specific project.

While this allowed for precision in the hierarchical model with respect to the project repository being analyzed, this specialized PIQUE-C# model is not generalizable. It will not work for any other project, and therefore every project that needs to be analyzed will need to have a model design file manually created.

Approach

The study design for this approach is summarized in Figure A.1. After manually creating and calibrating the PIQUE-C# model, we conduct changes to both the model and the source code and observe if there are changes to the TQI. Our research questions focus on these two types of changes and their results.

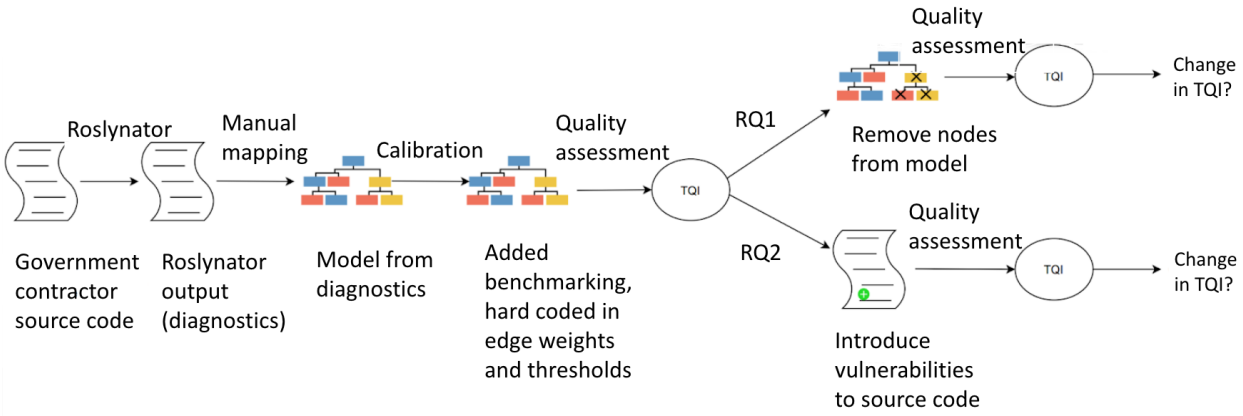

Figure A.1: PIQUE-C# Study Design

The PIQUE-C# model is tuned using two research questions:

- RQ1: Does removing nodes from the model affect the TQI?
- RQ2: Does introducing vulnerabilities into the source code affect the TQI?

Based on these research questions, the following hypotheses have been formulated:
For RQ1:

- H_0: Removing nodes from the model is associated with no observed difference in the TQI score.
- H_a: Removing nodes from the model is associated with an observed difference in the TQI score.

For RQ2:

- H_0: Introducing vulnerabilities into the source code is associated with no observed difference in the TQI score.
- H_a: Introducing vulnerabilities into the source code is associated with an observed difference in the TQI score.

RQ1 is interesting because if a government organization makes a hierarchical model for their source code but then later decides that they no longer want to include a node in the quality assessment, they can remove that node and know the sensitivity that node has on the TQI.

RQ2 is interesting because it allows us to conduct sensitivity testing on the PIQUE-C# model. Introducing vulnerabilities should impact the TQI, as the model's TQI is aggregated upward from tool output.

These hypotheses are tested by manually changing a PIQUE-C# model. After Roslynator executed on the source code, we analyzed the tool output to view the list of diagnostics found. From here, the leaf nodes of the hierarchical model were populated. We manually mapped the PIQUE model diagnostics upward to Measures, Product Factors, and Quality Aspects. This model structure can be seen in Figure A.2.

Validation of the PIQUE-C# model is not complete without calibration. This includes making modifications to the benchmarking and weighting processes which also impacts the model's threshold values. PIQUE-C# utilizes the weighting and weighting strategies from PIQUE.

Table A.1 shows the PIQUE-C# model version numbers and the resulting TQI throughout the calibration process.

Version 1 considered running the original source code with no modifications and using the project root as the benchmark for model derivation. This resulted in a perfect score of 1.0, which is inaccurate because it indicates that the source code had no tool output findings for warnings or vulnerabilities. A TQI score can range between the values of 0.0 and 1.0. Scores of exactly 0.0 or 1.0 are likely due to incorrect model calibration.

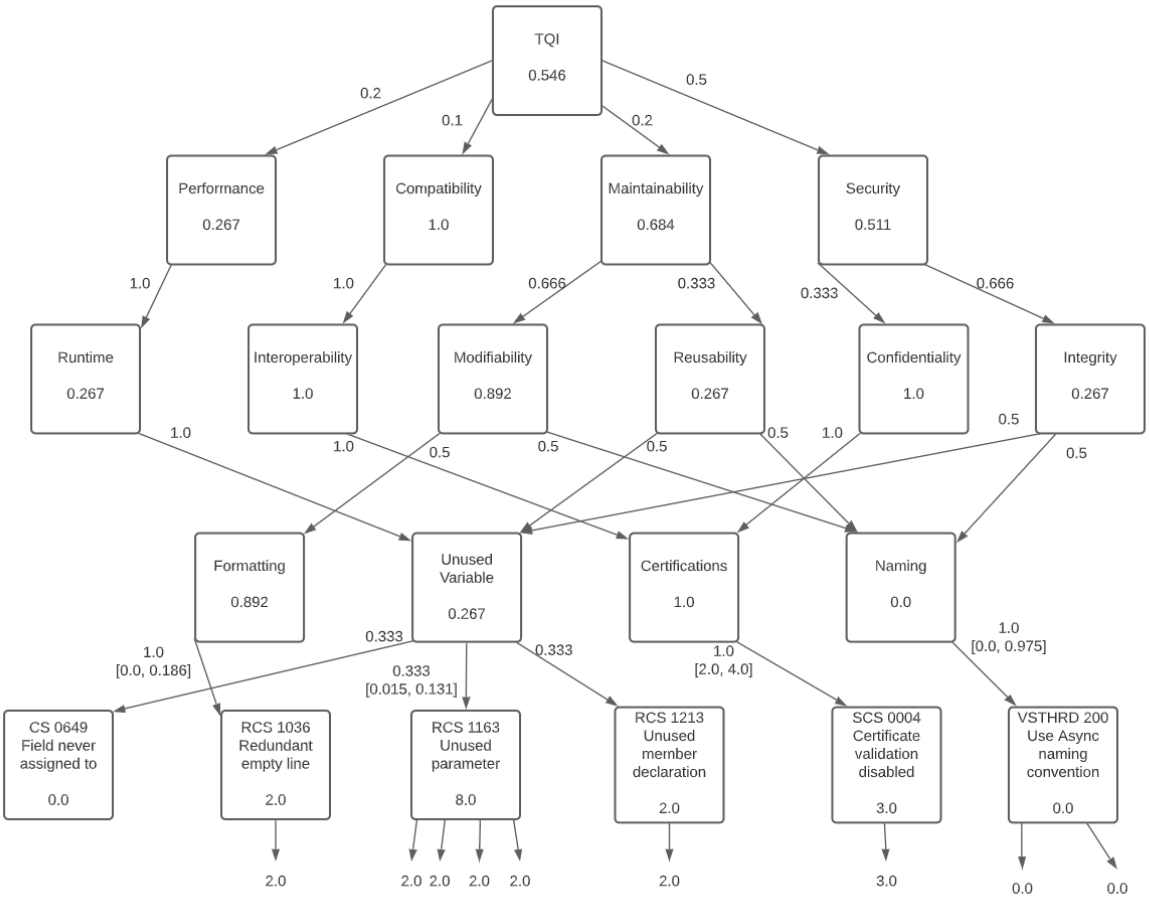

Figure A.2: PIQUE-C# hierarchical model structure based on real-world government source code.

Table A.1: PIQUE-C# calibration versions and the difference in TQI relative to Version 1.

Version	TQI	TQI Difference
1	1	N/A
2	0.175	-0.825
3	0.245	+0.07
4	0.279	+0.034
5	0.546	+0.267

Version 2 included several small benchmark repositories that the project was compared against. Now that the model derivation has been benchmarked against other projects other than just itself, these benchmarked findings can be compared with the project root's findings for a more accurate score. The score did drop 0.825, but this non-0.0 and non-1.0 score is an indicator that this score is more accurate.

Version 3 incorporated more benchmarking repositories. Version 2's benchmarking only resulted in 8 findings. This resulted in three of the project's six diagnostics to have [0.0, 0.0] threshold values, and therefore caused nodes at the Measure layer to have a value of 0.0. By adding larger benchmarking projects, the benchmarking process found 9,116 diagnostics in Version 3. This eliminated two of the [0.0, 0.0] threshold values and therefore increased the TQI.

Version 4 hard coded in the edge weights for the TQI and Quality Aspect layers. The uncalibrated model used equal weighting, which caused all nodes to equally weight their edge weights by the number of their children. In a calibrated model, the stakeholder should be able to weigh certain Quality Aspects and Product Factors more heavily than others, indicating that those aspects and factors are more important.

The edge weights in this model were manually adjusted to test if the new weights made a difference in the TQI. This was completed by making the Security node 0.5 of the TQI's weight, followed by Performance and Maintainability both at 0.2. This leaves Compatibility at 0.1. These were assigned based on which Quality Aspects we deemed most important to the stakeholders for this project.

Then, for Quality Aspects Maintainability and Security, since they have more than one child node, we also manually adjusted these child edge weights based on what we think the stakeholder would view as more important Product Factors.

This process would normally be completed by the stakeholders. This caused a change in the TQI as a result.

Finally, Version 5 hard coded in threshold values for the last remaining diagnostic having [0.0, 0.0] threshold values. This was done by observing the other threshold values within the model and creating a comparable threshold here that made sense for the number of findings associated with this diagnostic. This means that the diagnostic was not found in the 9,116 diagnostics found during the benchmarking process. This can be fixed by including more benchmark projects in hopes of finding this particular diagnostic. Another approach would be to bootstrap the value of the diagnostic to generate more data to use for the threshold values. By hard coding in the threshold values, this introduces a threat to construct validity; however, due to our knowledge with this model, we believe that the values chosen are a good choice for the thresholds. This model can be seen in Figure A.2.

RQ1: The variable of interest is the difference in TQIs among versions in Table A.2. These versions all include removing various nodes from the hierarchical model shown in Figure A.2. A random number generator online is used to choose two nodes from each level (Quality Aspect, Product Factor, and Measure). The random number generator is used to create random assignment so that any differences found in the TQI are attributable to the removal of model nodes and not to a confounding variable.

One version correlates to one node being removed. The TQI Difference column in Table A.2 compares that version's TQI to Version 5's TQI of 0.546.

Table A.2: PIQUE-C# versions and the difference in TQI relative to Version 5 when removing nodes from the model.

Version	Node Removed	TQI	TQI Difference
6	Measure Naming	0.546	0
7	Measure Formatting	0.478	-0.068
8	Product Factor Runtime	0.493	-0.053
9	Product Factor Integrity	0.790	+0.244
10	Quality Aspect Compatibility	0.495	-0.051
11	Quality Aspect Maintainability	0.528	-0.018

The data in Table A.2 has a minimum value of -0.0680, a mean of 0.0090, and a maximum value of 0.2440. The standard deviation of the data is 0.1178. We will discuss more details of the individual points when we conduct a visual observation on a scatterplot of the data.

RQ2: The variable of interest is the difference in TQIs among versions in Table A.3. These versions all include intentionally introducing vulnerabilities into the source code. The calibrated model as shown in Figure A.2 has six findings. Since these findings were already built into the model, these six were each introduced into a version number.

The source code consists of three C# files, so a random number generator online is used to choose which vulnerability would be introduced to which file. The random number generator is used to create random assignment so that any differences found in the TQI are attributable to the introduction of vulnerabilities into the source code and not to a confounding variable. Each file is assigned two vulnerabilities.

The TQI Difference column in Table A.3 is comparing that version's TQI to Version 5's TQI of 0.546.

The data in Table A.3 has a minimum value of -0.091, a mean of -0.022, and a maximum value of 0.044. The standard deviation of the data is 0.0566. We will discuss more details of the individual points when we conduct a visual observation on a scatterplot of the data.

Table A.3: PIQUE-C# versions and the difference in TQI relative to Version 5 when introducing vulnerabilities into the source code.

Version	Vulnerability introduced	TQI	TQI Difference
12	CS0649	0.590	+0.044
13	RCS1036	0.532	-0.014
14	RCS1163	0.455	-0.091
15	RCS1213	0.455	-0.091
16	SCS0004	0.556	+0.01
17	VSTHRD 200	0.556	+0.01

Analysis Plan

The original analysis plan for both research questions was to analyze the data using a one-sample t-test to generate a mean value and a p-value. However, upon reviewing this analysis plan, we decided it was essential to run non-parametric tests on the data as well due to the low sample size of six data points. With such a low sample size, the parametric t-test is not very robust.

Parametric tests require the assumption of normality, which means that the distribution of the sample means is normally distributed [39]. However, nonparametric tests do not require that the normality assumption. This is because nonparametric tests are the statistical methods based on signs and ranks [39].

Therefore, we have added a non-parametric analysis for the data which is a one-sample Wilcoxon signed-rank test, which will be detailed in the Non-Parametric Tests section.

The following section will detail the descriptive interpretations of the results.

Visual Observation

Figure A.3 and Figure A.4 show the results from our two research questions. Figure A.3 correlates to the removal of PIQUE-C# model nodes while Figure A.4 correlates to the introduction of vulnerabilities in the PIQUE-C# project source code.

We created Figure A.3 to visually assess our data and see if there are any points of interest. This scatterplot is difficult to visually assess and gather any insight from as there are only six data points. However, we do want to point out two points of visual interest: Version 6 and Version 9.

Version 6 removed the Measure Naming. Since Naming does not have any Diagnostics, this removal does not change the overall TQI which we can see in the plot by the TQI difference relative to Version 1 being 0. Version 9 removed the Product Factor Integrity. Removing Integrity leads to the Quality Aspect it aggregates to (Security) increasing from a value of 0.511 to 1.0. Additionally, Security is half of the TQI's total weight, so this greatly

impacts the TQI difference relative to Version 1 which we can see in Figure A.3 by Version 9 having a high TQI value in comparison to the other versions in the scatterplot.

Some descriptive statistics for the data in Figure A.3 are that it has a minimum value of -0.0680, a mean of 0.0090, and a maximum value of 0.2440. The standard deviation of the data is 0.1178.

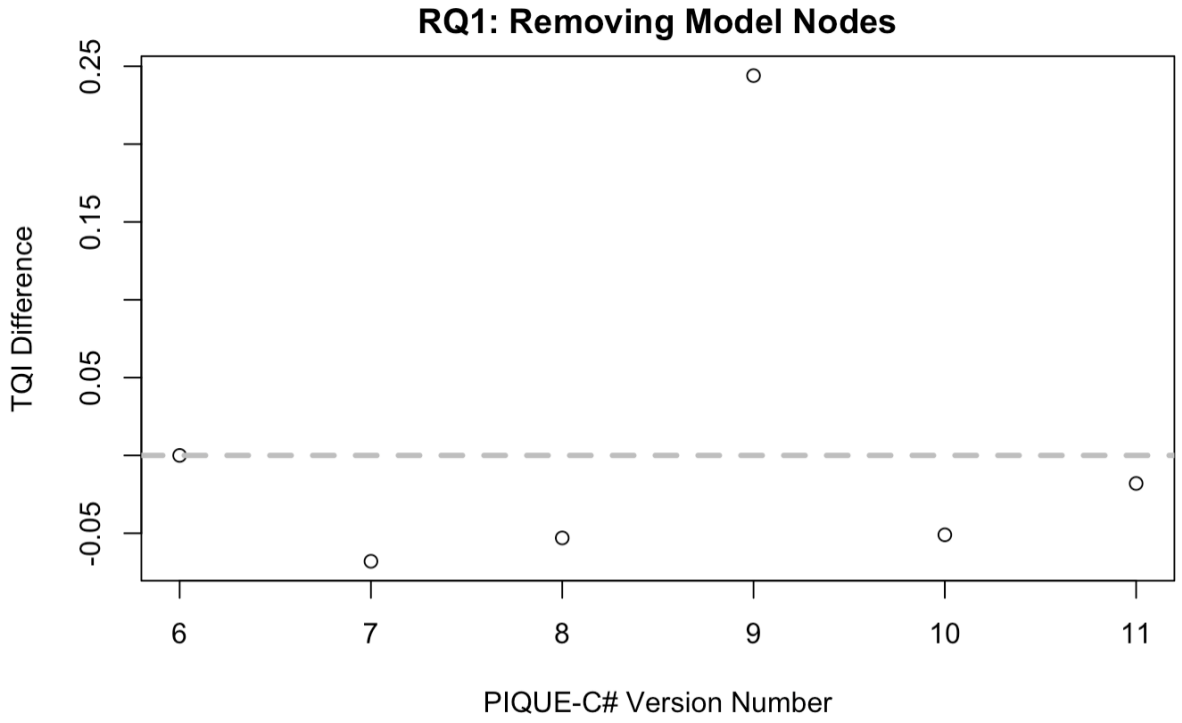

Figure A.3: Scatterplot with the difference in TQI for each PIQUE-C# version number relative to PIQUE-C# Version 1 when removing nodes from the PIQUE-C# model.

We created Figure A.4 to visually assess our data and see if there are any points of interest. This scatterplot is also difficult to visually assess and gather any insight from as there are only six data points. However, we do want to point out two areas of visual interest: the fact that Versions 14 and 15 have the same TQI difference, and the fact that Versions 16 and 17 have the same TQI difference.

Version 14 injected the vulnerability RCS1163, which according to the Roslynator Analyzers documentation[1] is an "Unused parameter". Version 15 injected the vulnerability RCS1213 which is "Remove unused member declaration". These two vulnerabilities, when injected into the model, had the same resulting TQI impact after aggregating upward through the model nodes.

[1] `https://github.com/JosefPihrt/Roslynator/blob/master/src/Analyzers/README.md`

Similarly, this is the case for Versions 16 and 17. Version 16 injected the vulnerability SCS0004, which according to the Security Code Scan documentation[2] is "Certificate Validation Disabled". Version 17 injected the vulnerability VSTHRD 200, which according to the VS-Threading Analyzers documentation[3] is "Use Async suffix for async". These two vulnerabilities, when injected into the model, had the same resulting TQI impact after aggregating upward through the model nodes.

Some descriptive statistics for the data in Figure A.4 are that it has a minimum value of -0.091, a mean of -0.022, and a maximum value of 0.044. The standard deviation of the data is 0.0566.

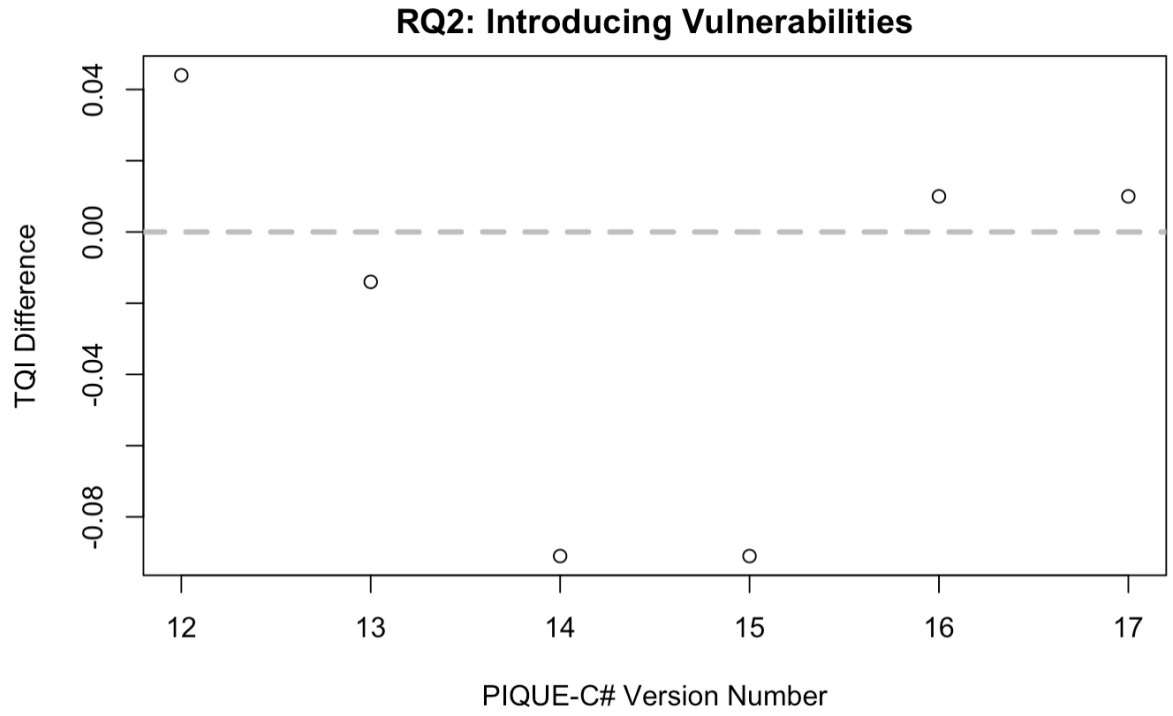

Figure A.4: Scatterplot with the difference in TQI for each PIQUE-C# version number relative to PIQUE-C# Version 1 when introducing vulnerabilities into the source code.

Non-Parametric Tests

The non-parametric analysis plan for both research questions is to analyze the data using a one-sample Wilcoxon signed-rank test to generate a p-value. This test uses the ranks of the magnitudes of the differences in addition to their signs [36]. Since ranks are used, the test is resistant to outliers.

[2] https://security-code-scan.github.io/#Rules

[3] https://github.com/microsoft/vs-threading/blob/main/doc/analyzers/VSTHRD200.md

After running a one-sample Wilcoxon signed-rank test on the RQ1 data, we found that there is little to no evidence against the null hypothesis that there is a difference in the model's TQI when removing nodes from the model (p-value = 0.5896). This would lead us to fail to reject the null hypothesis.

After running a one-sample Wilcoxon signed-rank test on the RQ2 data, we found that there is little to no evidence against the null hypothesis that there is a difference in the model's TQI when introducing vulnerabilities into the source code (p-value = 0.5271). This leads us to fail to reject the null hypothesis.

We attribute the fact that both RQs are not statistically significant to the study's very low sample size. Both research questions had sample sizes of six data points.

The power of a test is the probability of rejecting the null hypothesis, and the probability of finding strong evidence for the alternative hypothesis. One way to increase power is to increase the study's sample size. If we were to repeat this study again, we would obtain a sample size of at least 30 data points for each research question.

Threats to Validity

Internal Validity

In this project, the threats to internal validity were the lack of random assignment in one of our research questions and our small sample size.

Threats to the internal validity are threats that are of concern when causal relations are examined. When examining whether one factor affects a second factor there is a risk that the second factor is also affected by a third factor [45]. When this third factor or confounding variable is not identified and taken into account, this creates a threat to the internal validity.

In this exploratory project, a threat to the internal validity is the lack of random assignment in RQ2. This introduces the potential for confounding variables and means that causation cannot be concluded between introducing vulnerabilities in the source code and the TQI in the second research question.

The tuning of the model from the version numbers in Table A.2 and Table A.3 may not be enough data for proper sensitivity testing. The sample size is small, and other variables may have caused the change in the TQI. The small sample size for both of our research questions is also a threat to the internal validity because the lack of data points makes it difficult to assess the outlier assumption for the analysis. Additionally, all possible combinations of changes were not made.

External Validity

In this project, the threats to external validity were that we did not use random sampling and our small sample of source code is not representative of all source code.

Threats to the external validity are threats that are concerned with to what extent it is possible to generalize findings, and how relevant those findings are to people outside the investigated case [45].

A threat to the external validity in this project is that any cause-and-effect relationship we observe between the changes made to the model or the source code and the resulting TQI difference cannot be extended to a broader population of source code outside of this study because we did not use random sampling. Due to our method of sampling, statistical generalizations will not be possible, so therefore the results will not apply in other contexts.

Another threat to the external validity in this study is that the small sample of government contractor source code used in the case study will not be representative of the real-world number of government contractor source code projects.

Construct Validity

In this project the threat to construct validity was the decision to hard code in one of the threshold values and assessing the assumptions.

Threats to the construct validity are threats that reflect to what extent the operational measures that are studied really represent what the researcher has in mind and the research questions [45]. A threat to the construct validity can occur when data, measurements, or questions are not clear or measuring what they are intended to measure.

The decision to hard code in one of the threshold values is a threat to the construct validity in this project. This could be mitigated by incorporating more benchmark repositories into the project or by using bootstrapping to generate data for the benchmarking to use.

Additionally, assessing the assumptions is a threat to the construct validity because our assumptions will never be fully met. Instead, we assess to what degree the assumptions are violated.

Conclusion

The PIQUE-C# model was calibrated, and then analysis was performed by tuning the model for two research questions: Does removing nodes from the model affect the TQI, and does introducing vulnerabilities into the source code affect the TQI?

We fail to reject our null hypothesis for RQ1 that removing nodes from the model causes no difference in the TQI, meaning that it is likely that what we observe is happening based on chance alone and is not statistically significant.

We also fail to reject our null hypothesis for RQ2 that introducing vulnerabilities into the source code causes no difference in the TQI, meaning that it is likely that what we observe is happening based on change alone and is not statistically significant.

This project could be extended by testing the research questions on multiple models. Additionally, more extensive testing could take place by removing more nodes and by introducing more vulnerabilities to get a larger sample size. The project could also be extended by continuing research on and developing a security quality model.

APPENDIX B

TOOL DIAGNOSTICS

Table B.1: Tool Diagnostics

Tool Name	Diagnostic ID	Description
Security Code Scan	SCS0001	Command Injection
Security Code Scan	SCS0002	SQL Injection
Security Code Scan	SCS0003	XPath Injection
Security Code Scan	SCS0007	XML eXternal Entity Injection (XXE)
Security Code Scan	SCS0018	Path Traversal
Security Code Scan	SCS0029	Cross-Site Scripting (XSS)
Security Code Scan	SCS0026	LDAP Distinguished Name Injection
Security Code Scan	SCS0031	LDAP Filter Injection
Security Code Scan	SCS0004	Certificate Validation Disabled
Security Code Scan	SCS0005	Weak Random Number Generator
Security Code Scan	SCS0006	Weak hashing function
Security Code Scan	SCS0010	Weak cipher algorithm
Security Code Scan	SCS0013	Potential usage of weak CipherMode mode
Security Code Scan	SCS0008	Cookie Without SSL Flag
Security Code Scan	SCS0009	Cookie Without HttpOnly Flag
Security Code Scan	SCS0023	View State Not Encrypted
Security Code Scan	SCS0024	View State MAC Disabled
Security Code Scan	SCS0017	Request Validation Disabled (Attribute)
Security Code Scan	SCS0021	Request Validation Disabled (Configuration File)
Security Code Scan	SCS0030	Request validation is enabled only for pages
Security Code Scan	SCS0015	Hardcoded Password
Security Code Scan	SCS0034	Password RequiredLength Not Set
Security Code Scan	SCS0032	Password RequiredLength Too Small
Security Code Scan	SCS0033	Password Complexity
Security Code Scan	SCS0011	Unsafe XSLT setting used
Security Code Scan	SCS0012	Controller method is potentially vulnerable
Security Code Scan	SCS0016	Cross-Site Request Forgery (CSRF)
Security Code Scan	SCS0019	OutputCache Conflict
Security Code Scan	SCS0022	Event Validation Disabled
Security Code Scan	SCS0027	Open Redirect
Security Code Scan	SCS0028	Insecure Deserialization

Table B.2: Tool Diagnostics Continued

Tool Name	Diagnostic ID	Description
Insider	CWE-78	OS Injection
Insider	CWE-89	SQL Injection
Insider	CWE-643	XPath Injection
Insider	CWE-90	LDAP Injection
Insider	CWE-79	Cross-Site Scripting
Insider	CWE-611	XML eXternal Entity Reference (XXE)
Insider	CWE-310	Cryptographic Issues
Insider	CWE-259	Hardcoded Password
Insider	CWE-521	Weak Password Requirements
Insider	CWE-614	Sensitive Cookie in HTTPS Session
Insider	CWE-330	Use of Insufficiently Random Values
Insider	CWE-326	Inadequate Encryption Strength
Insider	CWE-311	Missing Encryption of Sensitive Data
Insider	CWE-554	ASP.NET Misconfiguration
Insider	CWE-20	Improper Input Validation
Insider	CWE-524	Use of Cache Containing Sensitive Information
Insider	CWE-377	Insecure Temporary Files
Insider	CWE-23	Relative Path Traversal
Insider	CWE-200	Exposure of Sensitive Information to Unauthorized Actor
Insider	CWE-502	Deserialization of Untrusted Data
Insider	CWE-11	ASP.NET Misconfiguration: Creating Debug Binary
Insider	CWE-129	Improper Validation of Array Index
Insider	CWE-316	Cleartext Storage of Sensitive Information in Memory
Insider	CWE-312	Cleartext Storage of Sensitive Information
Insider	CWE-12	ASP.NET Misconfiguration: Missing Custom Error Page
Insider	CWE-787	Out-of-bounds Write
Insider	CWE-352	Cross-Site Request Forgery (CSRF)
Insider	CWE-532	Insertion of Sensitive Information into Log Files

APPENDIX C

CWE TOP 25 MOST DANGEROUS SOFTWARE WEAKNESSES FOR 2021

Table C.1: CWE Top 25 Most Dangerous Software Weaknesses for 2021

Rank	ID	In PIQUE-C#-Sec?
1	CWE-787	Yes
2	CWE-79	Yes
3	CWE-125	By relation
4	CWE-20	Yes
5	CWE-78	Yes
6	CWE-89	Yes
7	CWE-416	No
8	CWE-22	Yes
9	CWE-352	Yes
10	CWE-434	No
11	CWE-306	By relation
12	CWE-190	By relation
13	CWE-502	Yes
14	CWE-287	Yes
15	CWE-476	No
16	CWE-798	By relation
17	CWE-119	Yes
18	CWE-862	No
19	CWE-276	No
20	CWE-200	Yes
21	CWE-522	By relation
22	CWE-732	By relation
23	CWE-611	Yes
24	CWE-918	No
25	CWE-77	Yes

APPENDIX D

BENCHMARK ATTRIBUTE TABLE

Table D.1: Benchmark Attribute Table

Project Name	Size (loc)	Source	Type
AutoMapper	91429	Open	Library
BenchmarkDotNet	64498	Open	Library
choco	37695	Open	Application
Electron.NET	7455	Open	Application
Project 1	319	Closed	Library
Project 2	873	Closed	Library
Project 3	3049	Closed	Application
Project 4	208	Closed	Application
Project 5	11	Closed	Library
Project 6	599	Closed	Library
Project 7	13654	Closed	Application
Project 8	4873	Closed	Library
Project 9	1953	Closed	Application
Project 10	306	Closed	Library
Project 11	3133	Closed	Application
Project 12	423	Closed	Library
Project 13	53	Closed	Library
Project 14	364	Closed	Library
Project 15	46	Closed	Library
Project 16	2643	Closed	Application
Project 17	3510	Closed	Library
Project 18	45	Closed	Application
Project 19	3112	Closed	Application
Project 20	150	Closed	Application
Project 21	93	Closed	Library
Project 22	11	Closed	Library
Project 23	3779	Closed	Application
Project 24	89	Closed	Application
Project 25	3223	Closed	Application
Project 26	1631	Closed	Library
Project 27	2171	Closed	Library
Project 28	2621	Closed	Library
Project 29	2795	Closed	Library
Project 30	87	Closed	Library
Project 31	2174	Closed	Library

Table D.2: Benchmark Attribute Table Continued

Project 32	231	Closed	Library
Project 33	157	Closed	Library
Project 34	1984	Closed	Library
Project 35	48	Closed	Library
Project 36	65	Closed	Library
Project 37	1670	Closed	Library
Project 38	1764	Closed	Library
Project 39	31	Closed	Library
Project 40	130	Closed	Library
Project 41	549	Closed	Application
Project 42	1731	Closed	Library
Project 43	2701	Closed	Application
Project 44	160	Closed	Library
Project 45	2696	Closed	Library
Project 46	2455	Closed	Library
example-voting-app	652	Open	Application
FASTER	41827	Open	Library
FluentTerminal	12645	Open	Application
graphql-dotnet	41998	Open	Library
Hangfire	148574	Open	Library
iotedge	141812	Open	Application
jellyfin	144467	Open	Application
mRemoteNG	48073	Open	Library
Ocelot	45482	Open	Application
Ombi	36168	Open	Application
OpenRA	127159	Open	Application
Opserver	33472	Open	Application
Polly	182463	Open	Library
PushSharp	3448	Open	Library
ql	20607	Open	Library
RestSharp	29629	Open	Application
ScreenToGif	48303	Open	Application
server	35928	Open	Application
ServiceStack	286151	Open	Library
SparkleShare	10497	Open	Application
workflow-core	20661	Open	Application
Wox	12157	Open	Application

APPENDIX E

TSI/SECURITY ASPECT IMPACT TABLE

Table E.1: TSI/Security Aspect Impact Table

Injected Diagnostic	TSI	Authenticity	Availability	Authorization	Confidentiality	Accountability	Non-repudiation	Integrity
CWE-11	0.0476190	0.0277778	0.0277778	0.0555556	0.0277778	0.1111111	0.0555556	0.0277778
CWE-12	0.0297619	0.0416667	0.0000000	0.0833333	0.0416667	0.0000000	0.0000000	0.0416667
CWE-129	0.0476190	0.0277778	0.0277778	0.0555556	0.0277778	0.1111111	0.0555556	0.0277778
CWE-20	0.0357143	0.0000000	0.1250000	0.0000000	0.0000000	0.0000000	0.0000000	0.1250000
CWE-200	0.0054872	0.0049244	0.0019698	0.0098488	0.0049244	0.0078790	0.0039395	0.0049244
CWE-23	0.0054872	0.0049244	0.0019698	0.0098488	0.0049244	0.0078790	0.0039395	0.0049244
CWE-259	0.0178571	0.1250000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
CWE-311	0.0104285	0.0104285	0.0000000	0.0208570	0.0312856	0.0000000	0.0000000	0.0104285
CWE-312	0.0002299	0.0003219	0.0000000	0.0006438	0.0003219	0.0000000	0.0000000	0.0003219
CWE-316	0.0002299	0.0003219	0.0000000	0.0006438	0.0003219	0.0000000	0.0000000	0.0003219
CWE-326	0.0104285	0.0104285	0.0000000	0.0208570	0.0312856	0.0000000	0.0000000	0.0104285
CWE-330	0.0104285	0.0104285	0.0000000	0.0208570	0.0312856	0.0000000	0.0000000	0.0104285
CWE-352	0.0054061	0.0048516	0.0019407	0.0097033	0.0048516	0.0077626	0.0038813	0.0048516
CWE-377	0.0054872	0.0049244	0.0019698	0.0098488	0.0049244	0.0078790	0.0039395	0.0049244
CWE-502	0.0032810	0.0019139	0.0019139	0.0038278	0.0019139	0.0076556	0.0038278	0.0019139
CWE-521	0.0178571	0.1250000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
CWE-524	0.0054872	0.0049244	0.0019698	0.0098488	0.0049244	0.0078790	0.0039395	0.0049244
CWE-532	0.0074497	0.0000000	0.0000000	0.0000000	0.0521480	0.0000000	0.0000000	0.0000000
CWE-554	0.1547619	0.0277778	0.4027778	0.0555556	0.2777778	0.1111111	0.0555556	0.1527778
CWE-611	0.0619048	0.0277778	0.0777778	0.0555556	0.0277778	0.1111111	0.0555556	0.0777778
CWE-614	0.0104285	0.0104285	0.0000000	0.0208570	0.0312856	0.0000000	0.0000000	0.0104285
CWE-643	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
CWE-78	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
CWE-787	0.0018780	0.0026293	0.0000000	0.0052585	0.0026293	0.0000000	0.0000000	0.0026293
CWE-79	0.0041950	0.0000000	0.0146826	0.0000000	0.0000000	0.0000000	0.0000000	0.0146826
CWE-89	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
CWE-90	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
SCS0001	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
SCS0002	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
SCS0003	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
SCS0004	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
SCS0005	0.0072780	0.0169820	0.0000000	0.0000000	0.0000000	0.0000000	0.0339641	0.0000000
SCS0006	0.0169198	0.0394796	0.0000000	0.0000000	0.0000000	0.0000000	0.0789591	0.0000000
SCS0007	0.0619048	0.0277778	0.0777778	0.0555556	0.0277778	0.1111111	0.0555556	0.0777778
SCS0008	0.0104285	0.0104285	0.0000000	0.0208570	0.0312856	0.0000000	0.0000000	0.0104285
SCS0009	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
SCS0010	0.0169198	0.0394796	0.0000000	0.0000000	0.0000000	0.0000000	0.0789591	0.0000000
SCS0011	0.0619048	0.0277778	0.0777778	0.0555556	0.0277778	0.1111111	0.0555556	0.0777778
SCS0012	0.0476190	0.0277778	0.0277778	0.0555556	0.0277778	0.1111111	0.0555556	0.0277778
SCS0013	0.0169198	0.0394796	0.0000000	0.0000000	0.0000000	0.0000000	0.0789591	0.0000000
SCS0015	0.0178571	0.1250000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
SCS0016	0.0108933	0.0097760	0.0039104	0.0195521	0.0097760	0.0156416	0.0078208	0.0097760
SCS0017	0.1547619	0.0277778	0.4027778	0.0555556	0.2777778	0.1111111	0.0555556	0.1527778
SCS0018	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
SCS0019	0.0054872	0.0049244	0.0019698	0.0098488	0.0049244	0.0078790	0.0039395	0.0049244
SCS0021	0.1547619	0.0277778	0.4027778	0.0555556	0.2777778	0.1111111	0.0555556	0.1527778
SCS0022	0.1547619	0.0277778	0.4027778	0.0555556	0.2777778	0.1111111	0.0555556	0.1527778
SCS0023	0.1547619	0.0277778	0.4027778	0.0555556	0.2777778	0.1111111	0.0555556	0.1527778
SCS0024	0.1547619	0.0277778	0.4027778	0.0555556	0.2777778	0.1111111	0.0555556	0.1527778
SCS0026	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
SCS0027	0.0476190	0.0277778	0.0277778	0.0555556	0.0277778	0.1111111	0.0555556	0.0277778
SCS0028	0.0032810	0.0019139	0.0019139	0.0038278	0.0019139	0.0076556	0.0038278	0.0019139
SCS0029	0.0041950	0.0000000	0.0146826	0.0000000	0.0000000	0.0000000	0.0000000	0.0146826
SCS0030	0.1547619	0.0277778	0.4027778	0.0555556	0.2777778	0.1111111	0.0555556	0.1527778
SCS0031	0.0142857	0.0000000	0.0500000	0.0000000	0.0000000	0.0000000	0.0000000	0.0500000
SCS0032	0.0178571	0.1250000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
SCS0033	0.0178571	0.1250000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
SCS0034	0.0178571	0.1250000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000