A Position Study to Investigate Technical Debt Associated with Security Weaknesses

Clemente Izurieta
Montana State University
Bozeman, MT, USA
clemente.izurieta@montana.edu

David Rice
Montana State University
Bozeman, MT, USA
david.rice@msu.montana.edu

Kali Kimball
Georgia College & State University
Milledgeville, GA, USA
kali.kimball@bobcats.gcsu.edu

Tessa Valentien
Georgia Institute of Technology
Atlanta, GA, USA
tvalentien3@gatech.edu

ABSTRACT

Context: Managing technical debt (TD) associated with potential security breaches found during design can lead to catching vulnerabilities (i.e., exploitable weaknesses) earlier in the software lifecycle; thus, anticipating TD principal and interest that can have decidedly negative impacts on businesses. Goal: To establish an approach to help assess TD associated with security weaknesses by leveraging the Common Weakness Enumeration (CWE) and its scoring mechanism, the Common Weakness Scoring System (CWSS). Method: We present a position study with a five-step approach employing the Quamoco quality model to operationalize the scoring of architectural CWEs. Results: We use static analysis to detect design level CWEs, calculate their CWSS scores, and provide a relative ranking of weaknesses that help practitioners identify the highest risks in an organization with a potential to impact TD. Conclusion: CWSS is a community agreed upon method that should be leveraged to help inform the ranking of security related TD items.

CCS CONCEPTS

• General and Reference Surveys and overviews • Software and its engineering

KEYWORDS

quality assurance, software quality, technical debt

1 INTRODUCTION

Various techniques have been used to quantify TD; however, none have specifically focused on measuring the TD of security aspects that affect systems. Given all recent security attacks, and the ever-increasing frequency and severity of breaches, companies are starting to pay significantly more attention to security threats and are shifting resources to address weaknesses earlier in the software development lifecycle (i.e. with developers and managers). Many tools exist that provide metrics based analysis in terms of the number of vulnerabilities found in a system. Furthermore, sets of agreed upon rules have been established by the greater community (i.e. CVE [1], CWE [2], and CERT [3]) to explore these vulnerabilities and weaknesses from different perspectives, and over the last few years, organizations have been investing in ways to measure the quality of systems. ISO [4, 5] continues to evolve these definitions of quality and many companies and academic groups have started operationalizing them in open source and commercial tools. For example, SonarQube [20] (with SQALE [6] and Quamoco [7]), and CAST [8] are amongst the more pervasive. Security is only one of many quality aspects that are assessed, yet assessments are merely done based on counts of issues found by static analysis tools and practitioners are asking for smarter and more intuitive ways to assess the quality of security in a system.

In this position study, we use an operationalization of Security embedded in the Quamoco quality model [9] to identify those entities that are likely to contribute to TD from a security perspective. Further, we offer an approach to help with the analysis and prioritization of TD principal and interest associated with CWE violations.

1.1 Motivation and Research Objective

Our study explores the usage of agreed upon weaknesses (CWEs) as a basis for quantifying TD associated with security issues. Our motivation stems from the fact that a large community effort has already generated a lot of data informed by experts from both industry and academia. Specifically, the Common Weakness Scoring System (CWSS) [17, 22] already provides a mechanism for prioritizing weaknesses according to relevant importance and context. CWSS follows the steps of the Common Vulnerability Scoring System (CVSS), with the former focusing on weaknesses rather than vulnerabilities. This is an important distinction because a weakness is *"a shortcoming or imperfection in the software code, design, architecture, or deployment that, could, at some point become a vulnerability"*[2] and vulnerabilities are manifestations of weaknesses at runtime. Thus, since TD is a phenomenon that is best observed during design (i.e. tradeoffs), then CWSS is the appropriate scoring mechanism that should be leveraged. CWSS offers different approaches to calculate a weakness score, of which the *Aggregated* and *Generalized* methods offer a one-to-one mapping with our implementation [9] of the Quamoco hierarchical quality model. Further, since TD is more relevant to design issues, as opposed to code level (non-design) issues [18], we only focus on those rules (i.e., CWEs)

associated with design at the architectural level [21]. Thus, our goal in this position study is to explore an approach that uses CWSS scores relevant to architectural decisions to help rank TD issues associated with security weaknesses.

1.2 Contribution

Our study provides the following contributions: *i)* the operationalization of a subset of CWEs (i.e., architectural) in the Quamoco framework, implemented as a plug-in into the SonarQube™ platform, and *ii)* an approach to the prioritization of TD associated with common weaknesses using the CWSS scoring system of design related rules.

2 BACKGROUND AND RELEVANT WORK

2.1 Technical Debt Quantification

In 2016, a group of academics and practitioners participated in a Dagstuhl [10] where a new definition for TD was crafted. The definition was repurposed to be more focused. Specifically: *"In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability."*

This definition was needed in order to focus further work in our community. Although a comprehensive synthesis of definitional literature is beyond the scope of this paper, a notable attempt was made by Tom et al. [11]. They found that many aspects make up the field of TD, and were able to build agreed upon definitions of these numerous features. In particular, they found five main components of TD: code debt, design and architectural debt, environmental debt, knowledge distribution and documentation debt, and testing debt. This meant that anything ranging from a poorly written block of code to a programmer having a lack of understanding of the history of the system to issues with the overall design of the program could compound the value of TD for a particular system. Further, additional attempts were made to include socio-technical aspects of organizations as a form of TD. The work of Tamburri et al. [12] serves as an example.

Four prominent approaches to quantify TD are highlighted –all differ in their quantification.

SonarQube [16] implemented a widget into their framework that calculates TD and reports it in terms of days or dollars (i.e. cost) necessary to repay the debt. The TD metric is defined as the effort necessary to fix all maintainability issues and its value is obtained by examining the source code's TD ratio. The ratio is defined as: *Remediation cost / Development cost.*

Nugroho et al. [13] describe TD as occurrences where problems with the quality in software are able to exacerbate and lead to bigger problems if they are not fixed in a timely manner. They propose a formula to measure TD connected to the maintainability of software. By focusing on maintainability, the formula gives a measurement of how much effort will be needed in order to repair the amount of TD in the software, so the software can be easily adapted and improved over time. They use a rating classification of a five-star scale to describe the quality of the maintainability in the system with one star being low quality and five stars being high quality. The TD measurement is found by multiplying a *rework fraction* and a *rebuild value*. The rework fraction is an estimated percentage of the number of lines in the code that contribute to the TD. The rebuild value is the estimated amount of time (*in months*) that needs to be spent fixing the TD. They also provide a formula to calculate the interest of TD. They call this the *maintenance effort*, and this is found by multiplying the percentage of lines of code in a system that will change in a year and the rebuild value. The product of this is then divided by a *quality factor*. This interest value can help an organization with an estimate of how much TD will cost them in terms of repair effort over time.

Letouzey and Ilkiewicz [14] used the SQALE method, which estimates the amount of TD based on a quality model. The quality model used in this method is essentially a set of conditions that a program needs to meet in order to exhibit "quality." The SQALE method requests the organization to pair each condition of the quality model with a remediation function. The remediation function's purpose is to convert the amount of conditions in the model that are not met to a remediation cost. Different companies have different concerns that affect how they configure remediation functions. The SQALE method uses a total of eight quality features in its process: testability, reliability, changeability, efficiency, security, maintainability, portability, and reusability. These features are set up in a pyramid fashion (with testability at the bottom and reusability at the top) to guide the order in which the remediation of the TD issues should be completed. For example, a part of the code that does not meet a condition that is associated with the quality feature of testability should be rectified before one that is associated with maintainability. In addition to requiring the organizations to provide remediation functions associated with each unmet condition, the SQALE method also requires a non-remediation function with each condition. The non-remediation function's purpose is to estimate the consequence of not remediating a condition.

Finally, Curtis et al. [15] introduce a way to measure TD that focuses on converting the amount of TD in code to a quantity in monetary terms. The formula associates TD with an individual's understanding of economic debt. In order to obtain this monetary value, they use the following terms: should-fix violations, principal, interest, and TD. Should-fix violations are issues in the code that contribute to functional problems, principal is how much it will cost to remediate the should-fix violations, interest is how much the should-fix violations will cost the longer they are left unfixed, and TD is defined as a cost that is comprised of the should-fix violations, interest, and principal. They utilize a formula to find the "TD-Principal" with the following variables: should-fix violations, the estimated amount of hours to fix the should-fix violations, and the estimated cost of labor to do so. By classifying each of the should-fix violations to be of either low-, medium-, or high-severity, the formula assigns a higher weight to

the higher severity violations and a lower weight to the lower severity violations in the formula. The "TD-Principal" is calculated by multiplying each level of severity by the number of violations that need to be fixed, the average number of hours it will take to fix them, and $75.00 because this was found to be the average cost per hour for work in IT organizations. After each level of severity is multiplied by these factors to obtain three values (one value for each severity level), the sum of the three values is used to calculate the "TD-Principal."

2.2 Quamoco

The Quamoco quality model is an extensible meta-model based on the ISO/IEC 25010:2011[19]. It allows for quantifiable measures to be tied to more abstract quality attributes. "The central concept of the model is a factor, meant to represent an attribute or property of an entity; where the latter represents an important aspect of quality we want to measure. Two types of factors exist: quality aspects and product factors. The former represents the more abstract qualities found in theoretical models such as the ISO standards. The latter represents the measurable parts of a software component and has an impact on their associated quality aspect. Factors form hierarchies; where factors can further refine some aspect of quality." [9] Because the Quamoco definition and operationalization is hierarchical, it matches the *Aggregated* scoring methods from CWSS.

2.3 The Common Weakness Scoring System

The Common Weakness Scoring System (CWSS) is a recommendation for a community agreed upon set of characteristics and technical impacts of software weaknesses. This recommendation allows practitioners to use a common language when scoring weaknesses in software that could manifest as vulnerabilities when a system is operational. CWSS offers a quantitative approach to measuring potential weaknesses that is based on a formula developed through community involvement. The formula is dependent on three metric groups: *Base Findings (BF)*, *Attack Surface (AS)* and *Environmental (E)*, where each group is made up of sub-factors. Each group is assigned a numeric value, and when multiplied with each other, generate a final CWSS score in the rage of 0-100. For a detailed explanation of the formula see [2] and [22]. Explanations are also complemented by detailed examples.

Although CWSS provides a customizable approach to scoring weaknesses, results could be highly subjective due to the large number of contexts in which software is developed and run. Thus, in conjunction with CWSS, the Common Weakness Risk Analysis Framework (CWRAF) [23] helps remove some subjectivity in scoring by providing vignettes (use cases) in the context of their domains to inform the calculation of a CWSS score. Further, CWRAF helps generate consistent scores that reflect the mission of a specific organization by allowing stakeholders to *i)* define a Business Value Context (BVC), and *ii)* generate a Technical Impact (TI) scorecard. Although defining new BVCs, and generating new TI scorecards is possible, this activity requires significant development; so many practitioners can use existing resources and fine-tune them to their organizations.

3 PILOT STUDY

In order to address our ensuing goal to develop an index for TD associated with security weaknesses, we have developed the following approach (in five steps) depicted in Fig. 1:

1. Define a CWRAF TD vignette by selecting a list of relevant CWE entries and map the CWE hierarchy onto a Quamoco hierarchical model,
2. Run static analysis tools on source code to obtain the list

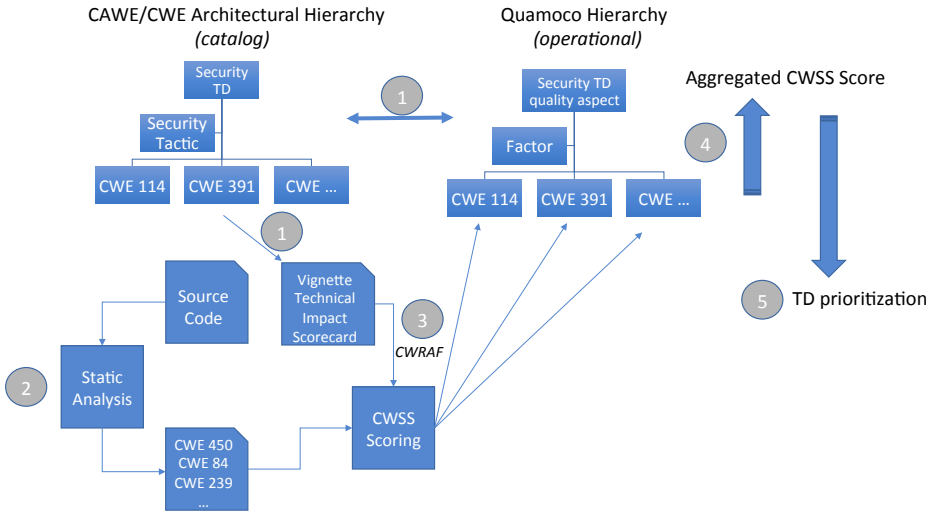

Figure 1. Steps associated with approach to prioritizing security related technical debt items

of relevant potential security weaknesses,
3. Use the CWRAF vignette to inform a CWSS score for each weakness,
4. Aggregate CWSS scores, and
5. Calculate a relative ranking of CWEs based on the CWSS scores to inform the TD prioritization of tasks

3.1 Rationale

The CWE hierarchy is composed of over 1000 separate weaknesses and many views are provided by the community. A view is a way for stakeholders to visualize the hierarchy from a chosen perspective. Three major perspectives are provided by Mitre®: *Research*, *Development*, and *Architectural* concepts. We chose the Architectural view [21] because according to Ernst et al. [18], decisions that affect TD occur during the design stages of software. Further, the security tactics employed at an architectural design level can have significant consequences (in the form of TD principal and interest) if not addressed. Santos et al. [21] state that *"security architectural design decisions are often based on well-known security tactics,"* and these decisions can have decidedly negative consequences on the TD of a system as it evolves. Thus, the first step in our approach uses the Common Architectural Weakness Enumeration (CAWE) [24] that is directly mapped (1-1) to the CWE Architectural view. The CAWE hierarchy is a catalog that contains 224 flaws organized along 11 security tactics. The CAWE hierarchy has many levels, where lower level nodes represent specializations. This hierarchy represents the subset of weaknesses (224/1000+) that influence TD and form part of the CWRAF vignette that is used to influence the CWSS score from our business perspective. Because the organization of these weaknesses is a hierarchy, it facilitates a mapping directly to our implementation of the Quamoco quality model. Security tactics map to either quality aspects or factors in a Quamoco tree. The Quamoco tree represents the operationalization of the CWE/CAWE catalog hierarchy.

The second step in our approach is the static analysis assessment of a target source code. Many static analysis tools exist to help with the identification of potential weaknesses. Each identified weakness has a unique id, and we only focus on those weaknesses that can be found on the CWE hierarchy from step one (i.e., architectural design decisions).

In the third step, we use the CWRAF vignette defined in step one (i.e., that characterizes our TD domain) to inform the CWSS scoring of the node in the tree that represents a specific CWE. Our implementation of Quamoco allows for the aggregation of quality scores up the hierarchy of a tree. Examples of functions for aggregating scores include max, min, average, median, or customized functions. The aggregation of CWSS scores is not a TD calculation; rather, it is a way to generalize scores to ISO defined levels. This is performed in the fourth step and facilitates the relative ranking of weaknesses to inform TD prioritization.

3.2 Example

To demonstrate our approach, we cut a small contour of the CWE hierarchy (see Fig. 2) and followed the steps in our approach as follows:

1. We mapped a subset of the architectural concepts CWE catalogue hierarchy to our Quamoco implementation – *manual step*
2. We ran a static analysis tool (FxCop) to detect security issues, and selected two weaknesses (CWE 114 and 391) *–automated step*
3. We used an existing CWRAF vignette from the financial trading domain to inform scoring for each weakness – *automated step*
4. We aggregated the CWE 114 and CWE 391 scores into categories CWE 1011, and CWE 1020 respectively. The categories represent security tactics *–automated step*
5. Produce relative ranking of CWEs to inform TD prioritization of tasks *–manual step*

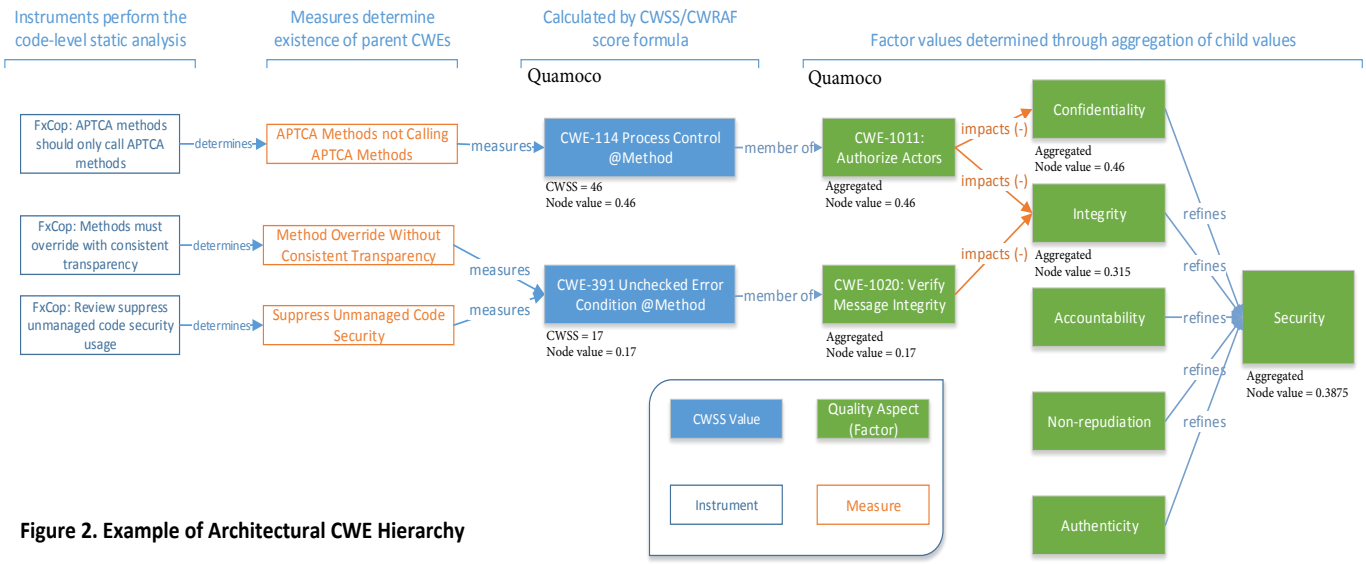

Figure 2. Example of Architectural CWE Hierarchy

4 POSITION ON TECHNICAL DEBT

The position we take on the development of a security index for TD is as follows: the quantification of an index is a difficult problem because it is highly dependent on context; however, taking advantage of a community informed mechanism that specifically takes into account environmental factors and technical impacts according to specific domains is a process that needs to be leveraged. Further, the comprehensive catalogue of weaknesses maintained by Mitre® is robust and our pilot study reveals that using a hierarchical quality model (i.e., Quamoco) to operationalize the CWE catalogue from an architectural perspective (i.e. the CAWE hierarchy) is a natural mapping that provides choices in how measures can be aggregated. The development of a view that focuses on architectural concerns alone narrows down the CWE hierarchy to only capture weaknesses that may impact the design of software and thus TD. Vignettes provide input to the scoring mechanism by removing subjectivity. It does this through technical impact and business value scores. Thus,

$Principal_{TD\text{-}Security}$ = Cost of the maintenance and refactoring associated with fixing architectural CWEs

The scoring generated by CWSS allows for a relative ranking of CWEs; which also allows practitioners to address TD items that may be of higher consequence to their organization. Addressing architectural issues early is directly aligned with reducing TD at design time, before the weaknesses represented by CWEs turn into actionable vulnerabilities. The longer a weakness remains unaddressed in a system, the higher the chances of it becoming a vulnerability. If a weakness turns into a vulnerability, then the technical and business impact have the potential to significantly increase the costs incurred by an organization because cost will not only be measured in terms of maintenance but also in terms of other factors that affect the technical capital of an organization (e.g., market share, reputation, loss of customers, etc.). The interest associated with the TD principal of a CWE is also hard to quantify, however, our position is that regardless of the equation used to model TD interest, there exists a significant event in the lifecycle of a CWE, which occurs when the weakness is exploited (i.e., it turns into a vulnerability). At that point in time, the cost of refactoring the CWE increases significantly due to the potentially irreparable impacts to an organization.

6 CONCLUSION AND FUTURE WORK

Although the Dagstuhl definition of TD limits contingencies to internal quality attributes, it is our position that security is a special case. When security weaknesses are identified in software, it is imperative that they are addressed expediently because although the maintenance associated with fixing a design flaw (i.e., TD principal) may not be cost prohibitive, the potential for damage to a business is. If a weakness is successfully exploited (as a vulnerability), then repairing the damage can be very costly. The TD interest associated with such a weakness can grow significantly at the moment an attacker is successful. The approach we have presented leverages an existing catalog and scoring mechanism to aid practitioners in prioritizing weaknesses as technical debt items hopefully informing the decision making process. We have successfully mapped a CWE hierarchy to an operationalization of Quamoco and have provided an example of using CWSS as a way to help prioritize TD items.

ACKNOWLEDGMENTS

This research is funded by the Construction Engineering Research Laboratories (CERL), Air Force, and the Department of Defense through an intermediary partnership with TechLink. Funding is also provided by NSF grant 1658971.

REFERENCES

[1] [Online] Available: cve.mitre.org
[2] [Online] Available: cwe.mitre.org
[3] [Online] Available: cert.org
[4] Software Product Evaluation—Quality Characteristics and Guidelines for Their Use, ISO/IEC Standard ISO-9126, 1991
[5] "ISO/IEC 25010:2011 Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – Systm and Software Quality Models," Mar. 2011.
[6] "SQALE Software Quality Assessment based on Lifecycle Expectations," Feb. 2017. [Online]. Available: www.sqale.org
[7] S. Wagner, K. Lochmann, L. Heinemann, M. Klas, A. Trendowicz, R. Plosch, A. Seidi, A. Goeb, and J. Streit, "The Quamoco product quality modelling and assessment approach." IEEE, Jun. 2012, pp. 1133–1142.
[8] [Online] Available: castsoftware.com
[9] I. Griffith, C. Izurieta, and C. Huvaere, "An Industry Perspective to Comparing the SQALE and Quamoco Software Quality Models," IEEE ACM 11th International Symposium on Empirical Software Engineering and Measurement (ESEM). Toronto, Canada, Nov. 9-10 2017.
[10] Managing Technical Debt in Software Engineering, Dagstuhl Reports, Vol. 6, Issue 4, April 17-22, 2016. [Online] Available: dagstuhl.de/16162
[11] E. Tom, A. Aurumn, and R. Vidgen. "An Exploration of Technical Debt," Journal of Systems and Software, Vol. 86, Issue 6, pp. 1498-1516, June 2013. https://doi.org/10.1016/j.jss.2012.12.052
[12] D. Tamburri, Philippe Kruchten, P. Lago, and H. van Vliet. "What is Social debt in Software Engineering," CHASE 2013, San Francisco US. https://978-1-4673-6290-0/13
[13] Nugroho, A.; Visser, J.; Kuipers, T., "An empirical model of technical debt and interest," In Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11). ACM, New York, NY, USA, 1-8. doi:10.1145/1985362.1985364
[14] J.L. Letouzey and M. Ilkiewicz, "Managing Technical Debt with the SQALE Method," IEEE Software Vol. 29, Issue 6, Nov-Dec 2012, IEEE doi:10.1109/MS.2012.129
[15] B. Curtis, J. Sappidi, and A. Szynkarski. "Estimating the principal of an application's Technical Debt," IEEE Software Software Vol. 29, Issue 6, Nov-Dec 2012, IEEE doi:10.1109/MS.2012.156
[16] [Online] Available: sonarqube.org
[17] [Online] Available: cwe.mitre.org/cwss/cwss_v1.0.1.html
[18] N. Ernst, S. Bellomo, I. Ozkaya, and R. Nord, "What to Fix? Distinguishing between design and non-design rules in automated tools," IEEE International Conference on Software Architecture (ICSA 2017). April 3-7, Gothenburg, Sweden. doi:10.1109/ICSA.2017.25
[19] "ISO/IEC 25010:2011 Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – Systm and Software Quality Models," Mar. 2011.
[20] "SonarQube Continous Code Quality," Feb. 2017. [Online]. Available: www.sonarqube.org
[21] Santos, J. C. S., Tarrit, K., and Mirakhorli, M. "A Catalog of Security Architecture Weaknesses," 2017 IEEE International Conference on Software Architecture (ICSA). 2017.
[22] Recommendation ITU-T X.1525. Common Weakness Scoring System (CWSS). [Online] Available: www.itu.int/rec/T-REC-X.1525/en
[23] [Online] Available: cwe.mitre.org/cwraf
[24] [Online] Available: design.se.rit.edu/catalog/index.html