Selected Code-quality Characteristics and Metrics for Internet of Things Systems

MATEJ KLIMA1, MIROSLAV BURES1, KAREL FRAJTA1, VACLAV RECHTBERGER1, MICHAL TRNKA1, XAVIER BELLEKENS2, TOMAS CERNY3, and BESTOUN S. AHMED4,1

1Dept. of Computer Science, FEE, Czech Technical University in Prague, Karlovo nam. 13, 121 35 Prague, Czechia
2Lupovis.io, 99 George Street, G1 1XW Glasgow, Scotland, UK
3Dept. Computer Science, Baylor University, One Bear Place #97141, Waco, TX 76798 USA
4Dept. of Mathematics and Computer Science, Karlstad University, Universitetsgatan 2, 65188, Karlstad, Sweden

Corresponding author: Miroslav Bures (e-mail: miroslav.bures@fel.cvut.cz).

ABSTRACT Software code is present on multiple levels within current Internet of Things (IoT) systems. The quality of this code impacts system reliability, safety, maintainability, and other quality aspects. In this paper, we provide a comprehensive overview of code quality-related metrics, specifically revised for the context of IoT systems. These metrics are divided into main code quality categories: Size, redundancy, complexity, coupling, unit test coverage and effectiveness, cohesion, code readability, security, and code heterogeneity. The metrics are then linked to selected general quality characteristics from the ISO/IEC 25010:2011 standard by their possible impact on the quality and reliability of an IoT system, the principal layer of the system, the code levels and the main phases of the project which they are relevant. This analysis is followed by a discussion of code smells and their relation to the presented metrics. The overview presented in the paper is the result of a thorough analysis and discussion of the author’s team with the involvement of external subject-matter experts in which a defined decision algorithm was followed. The primary result of the paper is an overview of the metrics accompanied by applicability notes related to the quality characteristics, the system layer, the level of the code, and the phase of the IoT project.

INDEX TERMS Code Quality, Code Smells, Internet of Things, IoT, Maintainability, Metrics, Quality Characteristics, Reliability, Security

I. INTRODUCTION

Throughout the last decade, the Internet of Things (IoT) systems have grown in popularity and their user base. IoT systems are also increasingly being used to effectively support various enterprise domains [1]. One of the specifics of current IoT systems is a high potential heterogeneity of the technologies used, which could negatively impact the potential interoperability, seamless integration, and general reliability and security of an IoT system [2]–[4]. So far, the problem has been discussed from multiple perspectives, including an overall review [3], [5], integration [6] and security [7], [8]. However, this problem has not been adequately examined from the code-level point of view.

The programming code is an integral part of the IoT system and is usually present in various aspects. Similarly to software systems [9], [10], poor code quality and code heterogeneity in the technologies and coding practice used have the potential to impact the overall reliability and quality of an IoT system. The problem of code quality has been addressed in various software studies and an established set of metrics has been proposed [11]–[13] and consolidated [9], [10], [14], [15]. However, such a consolidated view has not been provided for IoT systems, despite code quality’s relevance in this field.

IoT systems differ from software systems in a number of specific ways [1], [3]. As an example, we can discuss (1) generally larger heterogeneity of used technologies, protocols, devices, and programming languages, which create a significantly higher number of possible system configurations to be tested, (2) as a consequence, problematic interoperability of individual system parts and flawless integration, (3) a currently lower degree of standardization of communication protocols, as there are ongoing discussions and intensive work in this field, (4) increased risk of various privacy
and security issues in the current IoT systems, arising from many factors [16, 17]. Despite certain overlap, established software metrics might be insufficient for current IoT systems and should be revised for all of these differences.

A systematic analysis and review of the problem and available metrics must be performed to reflect the unique technicalities of IoT systems, including the proposal of a new IoT code-specific metric. In this paper, we summarize the relevant available code-quality metrics for IoT systems, and analyze their possible impact on the general system quality characteristics from ISO/IEC 25010:2011 standard. We also discuss the role of code smells in the reliability and quality of the system and their relation to the presented metrics.

The motivation for this mapping is the following. First, it is a useful tool for the definition of a test strategy for an IoT system. Test strategy guidelines typically operate with quality aspects or quality characteristics (e.g., Business Driven Test Management [18], part of TMAP Next [19]). For the definition of static testing levels (code reviews, inspections, etc.), the test engineer is naturally interested in knowing which code quality metrics may relate to particular system quality characteristics. Second, the aspect of test reporting and the impact analysis of poor quality also play a role. When some deficiency in source code quality is detected, the natural question is what the particular impact of this deficiency would be. The quality characteristics of the system serve well in this point, since they are general categories in which the matter can be reported and explained at a managerial level, for example.

The code quality metrics presented and discussed in this paper have an inevitable overlap with the general code quality metrics that can be applied in the case of general software, firmware, control software in an IoT device, infrastructural part or an IoT backend system. This situation is logical, as good coding practices are generally valid. Only specific details would differ for specific types of IoT systems. However, part of the metrics is more specific to distribute IoT systems, reflecting their context.

The paper is organized as follows. Section II discusses related work in the fields of code quality metrics and quality metrics for IoT systems. Section III explains the methodology of this study and provides a consolidated overview of code quality metrics applicable to IoT systems and their components. This section explores the potential impact of the metrics discussed on more general quality characteristics of the IoT system, levels of the IoT system, code levels, and principal project phases. Section IV discusses the issues related to the consolidation of code quality metrics conducted. Section V analyzes the possible threats to validity and limitations of this study. The last section concludes the chapter.

II. RELATED WORK

In the software-related literature, the first suggestions of code-quality metrics appear from the 70s. As a classic example, we can give metrics for sequential code such as cyclical complexity by McCabe [11], or later object-oriented code metrics such as coupling between objects by Chidamber and Kemerer [12], or Lack of Cohesion of Methods by Henderson-Sellers et al. [13]. Later, the published papers aggregated the metrics and demonstrated them in particular case studies. For example, Rosenberg and Hyatt [14] discussed selected aspects of system quality: efficiency, complexity, understandability, reusability, testability, and maintainability. In more recent studies, Heitlager et al. [20] is discussing metrics that impact the maintainability of software systems. The list provided by Heitlager et al. was further extended by Baggen et al. [15] in a study that also focuses mainly on system maintainability aspects. The established metrics such as code volume, code redundancy, unit size, code complexity, unit interface size, and component coupling are discussed [15]. This list partially overlaps with a selection by Pantichina et al., who adds cohesion and code readability to the discussion [9]. Code cohesion and coupling indicators are also the subject of an empirical study by Chaparro et al., which examines the impact of refactoring operations on code quality [10]. These two measures were also adopted by common Software Engineering textbooks, such as by Larman [21].

Nunez et al. [22] has recently presented a systematic overview of software quality metrics. However, the study focuses on software, and particular metrics definitions are not provided. Moreover, as the study focuses primarily on software, a revision is suitable for the IoT context. Also, the code quality for the web page user interface has been subject to investigation, and several quality metrics have been proposed, such as testability [23], maintainability [24] or feasibility of test automation [25].

Although all previously discussed code-related metrics are primarily aimed at software, they can reflect specifics of IoT systems applicable to code in these systems after revision. This applies to procedural code in firmware of various devices and object-oriented code usually used in gateways, server-side applications, or web- or mobile-based front-end user applications.

Kaur et al. [26] considered a broad and comprehensive detail of various metrics for Service-Oriented Architecture (SOA), particularly for their code, which is also an area closely related to IoT systems. Their work provides details of different types of cohesion (e.g., coincidental, logical, temporal, procedural, communication, sequential or functional) and their ratio-based perspectives (e.g., Ratio of Cohesive Interactions, Neutral Ratio of Cohesive Interactions, Pessimistic Ratio of Cohesive Interactions, Optimistic Ratio of Cohesive Interactions). Similarly, various coupling levels are given, ranging from Content Coupling (high), Common Coupling, External Coupling, Control Coupling, Stamp Coupling, Data Coupling to a Message Coupling (low), along with detail to available metrics the Coupling Between Objects and Response for class. Besides, they also consider size-based metrics (size of modules) based on counts such as source lines of code, volume, size, effort, length, etc. In
addition, other metrics are mentioned, such as control-flow-based and data-flow-based metrics. Include this work in our overview as relevant for IoT systems as well. Another related work on SOA by Kumar et al. [27] targeted source code metrics. They highlight that the maintainability prediction of SOA solutions is relatively unexplored and emphasizes maintainability prediction using several source code metrics. They proposed a maintainability prediction model using a Multivariate Adaptive Regression Splines (MARS) method. This method bases on the divide-and-conquer algorithm. In particular, it separates the data into various regions. Compared with other models, in particular, Multivariate Linear Regression (MLR) and Support Vector Machine (SVM) methods provided better results. They underline that software maintainability comprises (1) analysability, (2) changeability, (3) stability, and (4) testability.

To estimate the maintainability of web services, Kumar et al. used eleven source code metrics techniques [27]: Depth of inheritance tree, Weighted method per class, Coupling between object, Number of children, Number of methods, Lack of cohesion among methods, Response for class and McCabe’s Cyclomatic complexity. Norvaiša et al. [28] used McCabe’s cyclomatic complexity, code coupling, Halstead Volume for the SOA maintainability index. They observe that when we distribute application logic, the Halstead Volume metric increases and affects the maintainability index to be increased.

Kreinovich et al. [29] recently proposed a new set of metrics based on complexity and vulnerability of the code; however, a more thorough empirical or experimental evaluation of the proposal is not present in their paper.

The impact of a particular programming language on code quality was recently examined by Berger et al., however, the metrics for this evaluation are based on the density of the defects and the number of commits to a source code repository [30]. Another proposal in this field is the idea of fuzzy software quality metrics by Masmali and Badreddin focusing mainly on code complexity [31].

Wyrich et al. examined code comprehensibility metrics and their relationship with code readability, concluding that the examined metric significantly influences developers’ perception of code readability but, surprisingly, not their performance [32].

In the area of IoT-specific code-related metrics, significantly less work can be found. Most studies focus on IoT quality-related topics, which differ from actual code-level metrics. As an example, we can give the quality models by Kim [33] and Kim et al. [34], which serve to evaluate IoT applications and services. However, metrics that focus on different aspects in the fields of functionality, reliability, efficiency, and portability of an IoT system are presented, however, not explicitly focusing on code level [33].

Voas et al. commented on the general IoT metrology [35]. In their work (affiliated with NIST [1]), they provide multiple perspectives on possible metrics. The driving question of their work is how current knowledge can be leveraged for the Internet of Things (IoT). They mention general options to count items such as nodes, clients, gateways, servers, which can give us a sense of size and complexity. However, in their article, particular suggestions of metrics expressed by formulas are not provided.

Devices connected to IoT systems have also been discussed from the point of view of quality and the respective metrics provided by Arauz and Fynn-Cudjo [36]. However, this work focuses on the network level of an IoT system. Regarding the network level, the quality of service (QoS) for IoT systems has been examined by several studies [37], also discussing relevant metrics for this field. These works discuss general IoT systems [38], or, specifically, wireless sensor networks [39].

From the related areas, cloud service quality, a topic also relevant to the IoT system, is being quantified in a study by Zheng et al. [40], where aspects such as availability, reliability, usability, responsiveness, security, and elasticity are discussed.

In 2022 Fizza et al. published a paper on evaluating data quality in IoT smart agriculture applications [41]. They proposed and defined three sensor data quality metrics: suitability, accuracy, and completeness [41].

There already exist some practical examples of using quality metrics to evaluate IoT system’s quality. Chanhee et al. propose a Device-to-Device Framework for the Internet of Things, for which the metrics, such as Lines of Code, code complexity, duplication lines, maintainability, degree of the branch usages, duplicated source codes, and modification efforts, are measured using SonarQube platform [42].

Another way of using quality metrics were presented by Mishra and Kertesz. Authors discuss taxonomy categories to compare and analyze features of various MQTT protocol implementations use source code metrics. The measurement is performed using a source code analyzer called CLOC and the individual metrics are the number of files, blank lines, comment lines, and lines of code [43].

Minovski et al. introduce the term quality of IoT-experience (QoIoT), comprised from Quality of Experience (QoE) and Quality of Machine Experience (QoME). To measure the QoME the authors propose some code quality metrics, such as reliability or efficiency [44].

Matheus et al. utilized “long established code quality metrics”, without stating which they used, to evaluate a proposed semantic network management prototype, called SNoMAC. This system should help developers or system providers to easily interconnect the variety of heterogeneous devices that make up the IoT [45].

To measure the quality and reliability of information processed by IoT sensors, Kuemper et al. propose in their Valid.IoT framework several quality metrics. The individual definitions are based on the several Quality of Information and Quality of Service metrics, such as completeness, timeliness, plausibility, artificiality, or concordance [46].
Several metrics to evaluate performance of an IoT system were specified by Alexopoulos et al. They divide them into the following categories: (1) Processing speed/capacity, (2) Power Consumption, (3) Memory Requirements, and (4) Functionality Accuracy [47].

Although various aspects of the quality of IoT systems have already been discussed by recent studies and particular metrics suggested for their aspects, no paper specifically focused on a consolidated overview of code-quality metrics for IoT systems.

III. METRICS OVERVIEW
We start the overview of code-related quality metrics by summarizing our methodology. Then, we provide a consolidated overview of the metrics, followed by mapping the metrics to general quality characteristics of an IoT system by their possible impact.

A. METHODOLOGY
In the search for relevant literature on IoT-related metrics and code quality metrics, we followed the simplified methodology, as suggested by Kitchenham et al. [49]. We used six primary databases, IEEE Xplore, ACM Digital Library, Springer Link, Elsevier ScienceDirect, Web of Science, and Scopus. We performed the following general search:

('Code Quality Metrics' AND IoT) OR ('Code Quality Metrics' AND 'Internet of Things') OR ('Code Metrics' AND IoT) OR ('Code Metrics' AND 'Internet of Things') OR ('Code Quality Metrics' AND Software) OR ('Code Metrics' AND Software)

which we further adapted to the specifics of individual databases. Apostrophes denote an exact string to be searched for. No time or media type (e.g., journal or conference paper) restrictions were applied during the search.

In the literature search, we applied the workflow depicted in Figure 1. First, we downloaded all papers returned from the databases after the initial query and preliminarily filtered for relevance by title and abstract. This resulted in an initial set of 371 papers. Subsequently, two members of our lab independently analyzed the acquired papers and selected those that contained relevant information on the scope of this study were selected. This process resulted in a set of 79 papers to be further analyzed. We also added other relevant papers using a snowball sampling procedure during the reading phase. During the snowball sampling, 14 more papers were added. Papers were further analyzed in detail and particular metrics were identified, analyzed for duplicity, and extracted. The identified metrics were then stored in a collaborative spreadsheet. Then, final metrics selection and mapping of impact on quality characteristics were made due to the authors’ thorough discussion and consensus.

In the presented consolidated overview of code-quality metrics applicable to IoT systems, we use the two-level hierarchy:

1) We define **code quality categories**, which captures particular aspects of code quality (e.g., complexity, coupling, or cohesion) and can be expressed by several metrics.
2) For each of the code quality categories, we provide a set of possible **code quality metrics** to quantify the discussed code properties.

In our overview, we use the following code quality categories:

- **Size** determines the extent of source code of examined system in various measures (e.g., total size, size of the unit, size of its interfaces),
- **Redundancy** captures an extent to which the system code duplicates,
- **Complexity** measures a degree of code complexity from various viewpoints,
- **Coupling** measures a degree to which software components depend on each other and how closely are individual components connected,
- **Unit Test Coverage and Effectiveness** captures an extent to which the source code is covered by unit tests and also assessing the effectiveness of these tests,
- **Cohesion** measures the extent to which elements of a software module belong together; a different, but, in principle inversely correlating concept to coupling,
- **Code Readability** captures ease to understand the code and be able to do changes in it,
- **Security** expresses how many vulnerabilities, code security anti-patterns are present in the source code or how large the attack surface is, and,
- **Code Heterogeneity** captures how many languages and coding styles are employed if the complete source code is used in an IoT solution.

In Section III-B each code quality category is dedicated to a separate subsection, in which relevant metrics for this category are suggested. Suggestions are accompanied by references to studies that have previously defined and discussed metrics. Apart from those found in the recent literature, we have also suggested metrics that were not yet mentioned in the recent works. In Section III-C we discuss the impact of the code quality metrics presented on the selected general characteristics of the system [49] as well as their relevance to the general layers of an IoT system, the levels of the source code, and the main development phases of an IoT project. Then, in section III-D we discuss the relation of the presented metrics to code smells, as these two fields overlap.

B. CODE QUALITY METRICS
We discuss the individual code quality metrics in subsections dedicated to each defined code quality category. If no reference is given, the metric is our suggestion.

1) **Size**
Larger systems might be harder to extend, maintain, and more prone to defects [50]. To express the size of the system under test (SUT) code, we have several options:
Line of Code (LOC) metric counts all lines of source code that are not commented or blank lines \[20\]. Several variants of the metric can be used: Physical Lines of Code (PLOC) to capture the total number of lines in the source code or Logical Lines of Code (LLOC), which counts the actual number of instructions (one physical line of code may translate to several instructions). LLOC can be considered as a more accurate variant. Various open tools can be used to calculate LOC \[51\] in which a particular definition of LOC could differ. However, controversies are discussed regarding the application of the LOC metric, as this metric itself can have limits in expressing the quality of the system \[51\], \[52\]. The discussions concluded that the LOC metric could be useful when combined with other source code metrics.

Estimated Rebuild Value (ERV) captures an effort in man-years needed to build the system and could be more accurate than the sole LOC metric, due to the known fact that in different programming languages, the LOC needed to perform a particular task could differ \[15\], \[20\]. This is especially the case for low-level languages like C versus object-oriented programming languages like Java. Therefore, the estimated rebuild value is calculated from LOC taking into account the productivity of a particular programming language. For this, data from Software Productivity Research \[15\] are used \[15\].

Average Unit Size (US) is another option to express the code size of a SUT. The structure of the system units has an impact on its overall maintainability \[53\]. Ideally, individual units should be small, focused, and easy to test and understand. The average size of a unit can be measured as follows.

\[
US = \frac{KLOC}{n},
\]

where \(KLOC\) denotes the number of code lines in thousands and \(n\) is the number of subsystem units \[15\], \[20\].

Unit Interface Size (UIS), i.e., a number of parameters declared in the interface metric, evaluates the unit interfacing property. A unit with an interface with many parameters may be a signal of bad encapsulation \[15\].

Number of Distributed Architectural Components (DAC) (or blocks) used for quantified metrics can be seen as the number of nodes, clients, gateways, or servers in the overall system. Although there is a lack of work on metrology in IoT, NIST recognizes Network of Things (NoT) \[35\], and five primitive blocks for things that can be mapped: sensors, aggregators, communication channels, e-Utility, or decision triggers. Using these metrics for these primitives can be a good fit. Thus, they propose to extend the general countable options with things and highlight that the interconnection perspectives matter as well, since counting things is crude and some things are large and others small.

Number of NoT Architectural Components (NAC) (or blocks) \[35\] extend the number of distributed architectural components/blocks metrics utilizing the number of the following Network of Things (NoT) components:

- sensors - measuring physical properties,
- aggregators - component that transform data from a sensors,
- communication channels - for transmission,
- e-Utility - agents to execute processes,
- decision triggers - producers of final results.

This metric can be easily applied to general IoT systems as well to bring a crude static view of system size and complexity.

2) Redundancy

Source code duplication naturally leads to inefficient maintenance and is the source of possible defects during system updates \[53\].

Code Duplication (CD) metric is one of the options that we can use to quantify code redundancy in a SUT:

\[
CD = \frac{N_{dup}}{KLOC}
\]

where \(N_{dup}\) denotes the number of duplicate code blocks over six lines and \(KLOC\) denotes the number of code lines in thousands \[15\], \[20\].

Percentage of Redundant Code (RC) might be another option to express code duplication. Code is considered duplicated when a group of blocks over six lines appears unmodified more than once in the source code \[15\], \[20\]. The number of code lines from which the block is considered duplicate is a result of some empirical observations \[15\], \[20\] and can be modified by project context or programming language.
3) Complexity
Not only size and duplication might play a role in system maintainability and reliability; complexity is another of these potential factors. To measure complexity, we can employ several options.

**Cyclomatic Complexity (CYC)** is the number of linearly independent paths through a source code [11]. The higher this number, the more complex the code. For example, the cyclomatic complexity of a method \( m \) is defined as:

\[
CYC(m) = E - N + 2P
\]

When constructing a control flow graph for \( m \), \( E \) is the number of its edges (transfers of control), \( N \) is the number of its nodes (a sequential group of statements containing only one transfer of control), and \( P \) is the number of its connected components.

**Cyclomatic Complexity per Unit (CCU)** is defined as the sum of \( CYC(m) \) for all methods of a unit:

\[
CCU(u) = \sum_{m \in u} CYC(m)
\]

Practically, \( CCU(u) \) expresses the number of linearly independent paths in all unit methods at \([9, 14, 15, 20]\). For object-oriented languages, the same metric as \( CCU(u) \) is also known as Weighted Methods per Class \([9, 12]\), where a class is considered as a code unit.

**Depth of Inheritance Tree (DIT)**, relevant for object-oriented languages, expresses the number of ancestor classes of an analyzed class. In terms of the inheritance tree, it is the distance from the discussed class to the root of the tree \([14]\). The maximum depth of the inheritance tree for all classes in the unit can express an inheritance complexity for a unit of code. An averaged depth of the inheritance tree can be employed as a complementary metric to get better accuracy.

**Number of Children (NOC)** is another inheritance-related metric defined as the number of direct descendants of a class in the inheritance tree \([14]\). Interpretation of this metric requires caution: A higher value might indicate misuse of sub-classing. However, and at the same time, it signals code reuse, since inheritance is a form of reuse. At the same time, a more complex code hierarchy might require more tests.

**The number of Thing Interconnections (NTI)** brings insight to reliability in case a thing fails or to see the possible composability \([35]\). It also shows implications for performance. The interconnections are implemented via integration interfaces related to the code level.

4) Coupling
Coupling is the degree to which software components depend on each other. It measures how closely components are connected. Low coupling is often a sign of a well-structured system and good design. When combined with high cohesion, it supports the general goals of high readability and maintainability \([55]\). Several options are available to measure coupling:

**Coupling Between Objects (CBO)** metric can be used to measure coupling based on class usage. Two classes are considered to be coupled when methods of one class call methods or instances of the other class \([10, 12]\).

For a class \( c \), the coupling is defined as

\[
CBO(c) = |C_{coup}|
\]

where \( C_{coup} \) is the set of classes that are coupled to \( c \). The low value of \( CBO(C) \) indicates low coupling \([10, 12]\).

**Response for a Class (RFC)** is defined as the number of distinct methods and constructors invoked by a discussed class \([9, 14]\). This metric can act as a coupling degree indicator as well as might indicate code complexity.

**The number of the incoming call per module (ICM)** is another option to express the coupling property. The module is considered as a delimited group of units (e.g., classes) \([15]\). A higher value indicates a higher degree of coupling in the analyzed module.

5) Unit Test Coverage and Effectiveness
Unit tests’ coverage of source code is not exactly a quality of the analyzed source code itself. However, it is an aspect highly relevant for the final reliability of a system. As Heitlager et al. suggested, including unit testing level in code quality measures \([20]\), we follow their suggestion. Unlike Heitlager et al., we name this discussed property as unit test coverage and effectiveness, because, in our opinion, such a name more complies with standard terminology.

**Line Coverage (LC)** is an elementary metric that captures the execution of the source code by unit tests. A line of code is considered to be covered by a unit test when there exists a unit test that executes the code on that particular line. Line coverage then is defined as

\[
LC(u) = \frac{LOC_{ex}(u)}{LOC(u)} \cdot 100\%
\]

where \( u \) stands for a unit of code (e.g., class, package, or routine), \( LOC_{ex}(u) \) is the number of lines of code executed by a unit test and \( LOC(u) \) is the number of code lines of \( u \). However, caution must be taken when interpreting the line coverage figures, as no information is given about the unit tests’ real power to detect relevant defects (i.e., the number of assertions or amount of exercised test data combinations are not measured) \([56]\). Hence, this metric shall be accompanied by a number of assert method metrics defined later for higher accuracy.

**Branch Coverage (BC)** works on the same principle as line coverage; only the execution branches of the code are measured. A branch is considered to be covered by a unit test when there exists a unit test that executes that particular branch at least once \([57]\). A definition can be made in analogy to \( LC(u) \).

**Method Coverage (MC)** for object-oriented languages is a more high-level metric expressing the ratio of methods, which are called by unit tests to the total number of methods. A definition can be made in analogy to \( LC(u) \).
Density of Assert Methods (DA) is a metric that increases the accuracy of the insight in the probability that unit tests detect defects [20]. We suggest using this metric to complement the line, branch, and method coverage metrics. The density of assert methods can be expressed as

\[ DA(u) = \frac{A(u)}{LOC(u)} \times 100\% \]

where \( u \) stands for a unit of code (e.g., class, package, or routine), \( A(u) \) is the number of assertion methods applied to the unit \( u \) and \( LOC(u) \) is the number of code lines of the \( u \).

6) Cohesion

Cohesion is the assessment of the degree to which the responsibilities implemented in a class belong together. High cohesion is desirable, as it promotes the principle of single responsibility, promoting code maintainability [55]. Cohesion and coupling are in inverse correlation [58]. Several options can be employed to measure cohesion:

**Lack of Cohesion of Methods (LCOM)** metric is based on shared variables among the methods of a class \( C \) and is defined as

\[ LCOM(C) = \frac{1}{2} \sum_{j=1}^{a} \mu(A_j) - m \]

where \( a \) stands for the number of variables in a class \( C \), \( \mu(A_j) \) is the number of methods of \( C \) accessing the variable \( A_j \), and \( m \) stands for the number of methods in \( C \) [9], [13].

The value of \( LCOM(C) \) ranges from 0 to 1, where 0 indicates the highest cohesion of \( C \) and 1 indicates its lowest cohesion.

**Lack of Cohesion of Methods 1 (LCOM1)** is an alternative possible definition of lack of cohesion property. Given \( n \) methods, \( M_1, M_2, \ldots, M_n \) contained in a class \( C_1 \) that also contains a set of instance variables \( \{I_i\} \), then for any method \( M_i \) we can define the partitioned set of \( P = \{ (I_i, I_j) \mid I_i \cap I_j = \emptyset \} \) and \( Q = \{ (I_i, I_j) \mid I_i \cap I_j \neq \emptyset \} \) then

\[ LCOM1 = |P| - |Q|, \] if \( |P| > |Q|, 0 \) otherwise

otherwise \( LCOM1 \) is the count of the number of method pairs whose similarity is zero [26].

**Lack of Cohesion of Methods 5 (LCOM5)** metric as another possible alternative is based on the field usage of methods of class \( C \) and is defined as

\[ LCOM5(C) = \frac{TMA(C) - NA(C)}{1 - NA(C)} \]

where \( NA(C) \) stands for the number of attributes of \( C \), \( NM(C) \) stands for the number of methods of \( C \) and \( TMA(C) \) is the total number of field accessses to these methods. \( TMA(C) = \sum_i NA(a_i) \), where \( NA(a_i) \) is the number of methods that use a field \( a_i \) of class \( C \). The zero value of \( LCOM5(c) \) indicates the cohesive class \( C \), while the higher values indicate the lower class cohesion [10], [13].

Conceptual Cohesion of Classes (C3) is based on the measurement of textual similarity between class methods. The concept of Marcus et al. employs a latent semantic indexing approach [59] to determine this textual similarity. The definition of the metric is composed of several steps and is explained in [59].

**Ratio of Cohesive Interactions (RCI)** [26] is the ratio of the number of the actual interactions to the number of all possible interactions [60] between data declarations themselves or the methods. A data declaration \( a \) DD-interacts with another data declaration \( b \), if a change in \( a \)'s declaration or use may cause the need for a change in \( b \)'s declaration or use. We say that there is a DD-interaction between \( a \) and \( b \). There is a DM-interaction between data declaration \( a \) and the method \( m \), if \( a \)-DD-interacts with at least one data declaration of \( m \). \( CI(c) \) (CI for cohesive interactions) is the set of all DD- and DM-interactions. \( Max(c) \) is the set of all possible DD- and DM-interactions on the class interface [26]. For all classes \( c \in C \),

\[ RCI(c) = \frac{|CI(c)|}{|Max(c)|}. \]

RCI is to be in the interval [0, 1], and 0 indicates that no cohesive interactions were found, while 1 indicates that cohesive interactions were found. The RCI metric does not take into account indirect interactions [61].

There are three extended measures to RCI: Neutral RCI, pessimistic RCI and optimistic RCI. These capture the degree to which the software modules share features. The underlying idea is that cohesion only matters with respect to exported or public interfaces. A module has a low cohesion because its public procedures and data (properties/variables) have few interactions or between the data themselves. The definitions are:

**Neutral Ratio of Cohesive Interactions (NRCI)** is defined as follows. For all classes \( c \in C \), the set of known cohesive interactions \( CI(c) \) of a class is denoted by \( K(c) \), and the set of unknown cohesive interactions \( CI(c) \) by \( U(c) \), where \( Max(c) \) is explained in the definition of RCI. According to [62], the interaction is unknown if it is not detectable from the high-level design and is not signaled by the designers. In general, \( |Max(c)| \geq |K(c)| + |U(c)| \), since some interactions are not detectable from the high-level design and the designers explicitly exclude their existence [26], [62], [63]. Then, for all classes \( c \in C \),

\[ NRCI(c) = \frac{|K(c)|}{|Max(c)| - |U(c)|}. \]

**Pessimistic Ratio of Cohesive Interactions (PRCI)** is defined as follows without unknown cohesive interactions. For all classes \( c \in C \),

\[ PRCI(c) = \frac{|K(c)|}{|Max(c)|}. \]

where \( K(c) \) and \( Max(c) \) are explained in the NRCI and RCI definitions [26].

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
Optimistic Ratio of Cohesive Interactions (ORCI) can be used as another option to express the cohesion level of classes [26]. For all classes \( c \in C \),

\[
ORCI(c) = \frac{|K(c)| + |U(c)|}{\text{Max}(c)}.
\]

The definitions imply that if \( PRCI(c) \), \( NRCI(c) \), and \( ORCI(c) \) are not undefined, it can be shown that for all \( c \), it holds: \( 0 \leq PRCI(c) \leq NRCI(c) \leq ORCI(c) \leq 1 \). \( ORCI(c) \) and \( PRCI(c) \) provide the bounds for the admissible range for cohesion, and \( NRCI(c) \) takes a value in between [64]. According to Briand et al. [65] a low value of \( PRCI(c) \), \( NRCI(c) \), and \( ORCI(c) \) indicates that there is no cohesive interaction present, while a high value indicated all possible cohesive interactions present. If the unknown interactions become smaller, the interval \( [PRCI, ORCI] \) also decreases accordingly [62], [63]. However, in a situation with a single isolated object, the measures become \( PRCI(c) = 0 \), \( NRCI(c) = \text{undefined} \) and \( ORCI(c) = 1 \), that is, \( NRCI(c) \) is undefined when all interactions are unknown and there is no information available on cohesive interactions. In this case, with \( PRCI(c) = 0 \) and \( ORCI(c) = 1 \) the \( PRCI(c) \) and \( ORCI(c) \) do not provide stricter bounds than those provided by the interval for cohesion. The fact that \( NRCI(c) \) is not defined can be interpreted as the possibility that \( NRCI(c) \) can take any value in the interval \([0, 1]\).

7) Code Readability

The readability of the source code potentially enables easier maintenance and decreases a potential number of regression defects in the system [66]. Despite such a property of code that might be difficult to measure exactly, several attempts have been made for such a quantification [14], [60], [67].

Buse and Weimer’s Model (BW) employs quantification of structural aspects (e.g., number of branches, loops, etc.) to express the code readability [66]. No simple formula can be presented as a definition; refer to the original study for details.

Model by Scalabrino et al. (SCA) takes a different approach and expresses the readability of the code using the lexical analysis of the source code [67]. For details, see the original study.

Comment Percentage (CP) is calculated by the total number of comments divided by the total lines of code [14],

\[
CP = \frac{N_{\text{comment}}}{LOC} \cdot 100\%,
\]

where \( N_{\text{comment}} \) is the total number of comments and \( LOC \) denotes the number of code lines.

8) Security

Coding practices primarily affect the system that is prone to broken data integrity, compromised privacy, or even data breaches. Security metrics on how to measure them can be formulated.

Number of Security Antipatterns per Line of Code (SAP) express the density of security antipatterns in the source code:

\[
SAP = \frac{n_{\text{antipattern}}}{LOC},
\]

where \( n_{\text{antipattern}} \) denotes the number of security antipatterns and \( LOC \) stands for the number of code lines. Generally, antipatterns in the code can be manually searched based on their description [68], [69], or they can be automatically searched for [70]. Security antipatterns can be detected by analogy [71].

Number of Common Vulnerabilities and Exposures (CVE) measures the density of commonly known vulnerabilities and exposures (CVEs) currently known in the source code:

\[
CVE = \frac{n_{\text{CVE}}}{LOC},
\]

where \( n_{\text{CVE}} \) represents the number of CVEs in the code and \( LOC \) is the number of lines of code.

For instance, a database managed by non-profit organization MITRE that acts as top-level editor and numbering authority for CVEs can be used as a reference.

Number of Zero-day Vulnerabilities per Day (ZDV) measures the density of Zero-day Vulnerabilities [72] that were detected per day of system operation:

\[
ZDV = \frac{n_{\text{ZDV}}}{LOC \cdot t_{\text{operation}}},
\]

where \( n_{\text{ZDV}} \) represents the number of zero-day vulnerabilities found during \( t_{\text{operation}} \) days of system operation. \( LOC \) is the number of lines of code.

Attack Surface Size (AS) quantifies the extent of the accessible system for various types of cyberattacks [73]. Manadhata et al. bases this metric on an automata model and presents an abstract method to quantify the attack surface. For more details, refer to their study [73].

Stall Ratio (SR) expresses an extent to which the source code contains activities that are not necessary to achieve the goal of the program [74]. Chowdhury et al. defines \( SR \) for loops as:

\[
SR = \frac{NPLOC_{\text{loop}}}{LOC_{\text{loop}}},
\]

where \( NPLOC_{\text{loop}} \) denotes nonproductive lines of code in a loop and \( LOC_{\text{loop}} \) denotes lines of code in this loop.

Coupling Corruption Propagation (CCP) expresses how many methods can be impacted by an error in the original method if these methods rely on each other due to coupling in the object structure of a program [74]. The metric is defined as a number of child methods called with parameters based on the parameters of the original method call [74].

\[^3\text{https://cve.mitre.org/}\]
Critical Element Ratio (CER) measures how many critical data elements, whose corruption might cause a security breach, are present in a class:

\[ CER = \frac{n_{critical}}{n_{all}} \]

where \( n_{critical} \) denotes the number of critical data elements in a class and \( n_{all} \) denotes the number of all data elements in this class [74].

Ratio of Third-party Code (3PC) measures an extent of a code produced and/or maintained by an external party in a SUT. This extent can be measured in the number of lines of code, for instance, as:

\[ 3PC = \frac{LOC_{external}}{LOC_{total}} \cdot 100\% \]

where \( LOC_{external} \) is the number of lines of code produced by an external party and \( LOC_{total} \) is the total number of code lines.

Number of outdated code components (OCC) gives an actual indication of how many components of the SUT are out of date [75]. An outdated component might contain possible security flaws known in the cybersecurity community, or might potentially cause some integration defects during system deployment or production run. At this point, the more the component is outdated, the higher the risk, so to express the actual risk, a metric based on this aging time can be used, the \( ACC \) metric presented in more detail.

Aging of code components (ACC) express overall aging of code components that are relevant to be updated in a SUT:

\[ ACC = \frac{\sum_{i=1}^{\left| C \right|} time_{c_i}}{\left| C \right|} \]

where \( C \) is a set of components relevant to be updated and \( time_{c_i} \) denotes the time for which a component \( c_i \in C \) will already be updated.

9) Code Heterogeneity

Heterogeneity of source code, for instance, used programming languages and coding styles and conventions have a potential negative impact on maintainability. In an approximate way, their properties can be quantified.

Number of Different Programming Languages (PL) is a rough indication of the heterogeneity of an overall IoT system. With a growing number of different programming languages, one can expect that maintainability might be negatively impacted. However, other factors might also play a role.

Number of Different Programming Styles (PS) can be considered as the more useful metric that influences the maintainability of the system. Heterogeneity of coding styles and conventions inside one block of code (class, module, or library) negatively impacts the readability of the code and makes it prone to defects. However, this heterogeneity is difficult to express by an exact formula, as distinguishing code styles is difficult to quantify in the source code. Hence, we propose to rate this property on a scale no, low, medium, high, based on the judgment of a senior developer or architect during a code review.

Number of Different Communication Protocols (DCP) is another indicator that reflects the heterogeneity of an IoT system. The larger this number is, the more integration issues will have to be solved, and also the attack surface of the system will increase.

C. IMPACT OF METRICS TO SYSTEM QUALITY AND THEIR RELEVANCE TO SYSTEM LAYERS, CODE LEVELS, AND PROJECT PHASES
Source code quality, which can be expressed by metrics suggested in Section III-B, potentially impacts the quality, reliability, and security of an IoT system. This potential is significant, as the source code is present at many levels of a system, spanning from the device firmware to the server-side code.

Although it might be difficult to express an exact correlation between the quality of source code and the overall quality aspect of an IoT system (e.g., in terms of detected defects), this impact exists and can be analyzed. We do so in Table 1. Individual code quality metrics discussed in this study are linked to an IoT system’s general quality characteristics, where possible impact might likely be expected. A cross mark on the table captures this fact.

The quality characteristics discussed are a subset of quality characteristics defined in the product quality model of the ISO/IEC 25010:2011 standard, revised in 2017. From this standard, we have selected a set of quality characteristics that are relevant to be impacted by underlying source code quality, namely Analysability, Changeability, Stability and Testability from the Maintainability general category, Functional correctness from the Functional suitability general category, and Confidentiality and Integrity belonging to the Security general category.

Quality attributes from ISO/IEC 25010:2011 taxonomy are designed for general software systems. The programming code in IoT systems that is the subject of our study is a variant of a general software system, where this taxonomy can be used. Therefore, the ISO/IEC 25010:2011 taxonomy is relevant for use in this case.

Figure 2 provides an overview of ISO/IEC 25010:2011 quality characteristics. These, potentially impacted by code quality aspects discussed in this study, are distinguished by underlined bold.

Mapping given in Table 1 is the result of the following algorithm. (1) Based on the analysis of the literature, their experience with IoT systems, system architectures, code quality, and software development process, the authors provided individual opinions on mapping. In this process, the authors were encouraged to consult other independent experts in the field, and in total, nine more specialists from research and industry fields were involved in these consultations. (2) The data were consolidated into a shared table document. (3) If consensus was reached on a particular code quality metric and the quality characteristic of the system was affected in the sense that more than 75% of the participants had the same opinion, the relation was established as final. (4) If less than 75% of the participants had the same opinion, a discussion was started. In this discussion, arguments for and against were raised, and after consideration of the discussion participants, their opinion was expressed independently again. Then the result was consolidated. Approximately 15% of the mapping items were subjected to this discussion, and consensus was found after the completion of Step (4).

Regarding the literature analysis, which is one of the inputs to the mappings (see Step 1 of the previous algorithm), several sources have been utilized. Savchenko et al. propose the architecture for a maintenance metrics collection and analysis system, where the metrics are presented as a set of 8 scores, based on the quality characteristics of ISO/IEC 2501 (suitability, performance, compatibility, usability, reliability, security, maintainability, and portability). Motogna et al. uses several object-oriented metrics to propose an approach to assess the maintainability characteristic by its own sub characteristics as defined by ISO 25010. The metrics they use are Depth of Inheritance Tree (DIT), Coupling Between Objects (CBO), Weighted Methods per Class (WMC) (which is similar to the Cyclomatic Complexity per Unit that is described in this paper) and the Tight Class Cohesion (TCC). Colakoglu et al. recently performed a systematic mapping study, in which they mention both the quality characteristics, as defined in ISO 25010:2011, and the code quality metrics. They provide answers to several interesting RQs, such as which metric levels can be applicable for which application domains, or which quality models have been used in the papers to measure software quality.

Another question is to which principal layers of an IoT system the discussed metrics are relevant. An overview is provided in Table 2. Number of system layers can be discussed and in this overview we selected five principal types.

These types are device firmware code using non-object and object-oriented (OO) programming language, back-end server part application, where we assume usage of some object-oriented language, system user interface implemented as a web application, mobile application or a thick-client application, and network infrastructure part.

The overview given in Table 2 was carried out as a consensus of the expert group following the same algorithm as employed in the case of mapping the code quality metrics to the ISO/IEC 25010:2011 quality characteristics (Table 1).

Apart from the principal SUT layers, the code level where metrics can be applied is another relevant topic to be discussed. We do this analysis in the left-hand side of Table 3. We use three principal levels: (1) a project level, where the whole IoT project is considered comprehensively, (2) class level, focusing on an object structure of the code, and (3) function level, where we consider individual class methods or functions in the case of non-object-oriented code.

The right part of Table 3 shows the mapping of code quality metrics to the main phases of the development of the IoT project, where they can be applied. We consider four principal development life cycle phases of an IoT project: (1) technical design and architecture definition, (2) system coding, (3) system testing, and (4) system operation (live run in a production environment).

To construct the mappings given in Table 3, we employed the same method as was used in the case of previous analyses presented in Table 1 and 2.

https://www.iso.org/standard/35733.html
| Code property | Code quality metric                                                                 | Maintainability | Changeability | Stability | Testability | Functional correctness | Confidentiality | Integrity |
|---------------|-------------------------------------------------------------------------------------|-----------------|--------------|-----------|-------------|------------------------|----------------|----------|
| Size          | Lines of code (LOC)                                                                  | x               | x            |           |             |                        |                |          |
|               | Estimated rebuild value (ERV)                                                        | x               |             |           |             |                        |                |          |
|               | Average Unit Size (US)                                                                | x               | x           |           |             |                        |                |          |
|               | Unit Interface Size (UIS)                                                             | x               |             |           | x           |                        |                |          |
|               | Number of Distributed Architectural Components (DAC)                                  | x               |             |           |             |                        |                |          |
|               | Number of NoT Architectural Components (NAC)                                         | x               |             |           |             |                        |                |          |
| Redundancy    | Code Duplication (CD)                                                                | x               | x           | x         |             |                        |                |          |
|               | Percentage of Redundant Code (RC)                                                    | x               |             |           |             |                        |                |          |
| Complexity    | Cyclomatic Complexity (CYC)                                                           | x               |             |           | x           |                        |                |          |
|               | Cyclomatic Complexity per Unit (CCU)                                                 | x               |             |           | x           |                        |                |          |
|               | Depth of Inheritance Tree (DIT)                                                       | x               |             |           |             |                        |                |          |
|               | Number of Children (NOC)                                                              | x               |             |           | x           |                        |                |          |
|               | Number of Thing Interconnections (NTH)                                               | x               |             |           | x           |                        |                |          |
| Coupling      | Coupling Between Objects (CBO)                                                        | x               |             |           |             |                        |                |          |
|               | Response for a Class (RFC)                                                            | x               |             |           |             |                        |                |          |
|               | Number of incoming calls per module (INC)                                            | x               |             |           |             |                        |                |          |
| Unit Test     | Line Coverage (LC)                                                                    | x               | x           | x         |             |                        |                |          |
| Coverage and  | Branch Coverage (BC)                                                                  | x               |             |           |             |                        |                |          |
| Effectiveness | Method Coverage (MC)                                                                  | x               |             |           | x           |                        |                |          |
|               | Density of Assert Methods (DA)                                                        | x               |             |           |             |                        |                |          |
| Cohesion      | Lack of Cohesion of Methods (LCOM)                                                    | x               |             |           |             |                        |                |          |
|               | Lack of Cohesion of Methods 1 (LCOM1)                                                 | x               |             |           |             |                        |                |          |
|               | Lack of Cohesion of Methods 5 (LCOM5)                                                 | x               |             |           |             |                        |                |          |
|               | Conceptual Cohesion of Classes (C3)                                                   | x               |             |           |             |                        |                |          |
|               | Ratio of Cohesive Interactions (RCI)                                                   | x               |             |           |             |                        |                |          |
|               | Neutral Ratio of Cohesive Interactions (NRCI)                                         | x               |             |           |             |                        |                |          |
|               | Optimistic Ratio of Cohesive Interactions (ORCI)                                      | x               |             |           |             |                        |                |          |
| Code          | Buse and Weimer’s Model (BW)                                                          | x               | x           | x         | x           |                        |                |          |
| readability    | Scalabrino et al. Model (SCA)                                                         | x               | x           | x         | x           |                        |                |          |
|               | Comment Percentage (CP)                                                               | x               | x           | x         | x           |                        |                |          |
| Security      | Number of Security Antipatterns per Line of Code (SAP)                               | x               |             |           |             |                        |                |          |
|               | Number of Common Vulnerabilities and Exposures (CVE)                                 | x               |             |           |             |                        |                |          |
|               | Number of Zero-day Vulnerabilities per Day (ZDV)                                     | x               |             |           |             |                        |                |          |
|               | Attack Surface Size (AS)                                                              | x               |             |           |             |                        |                |          |
|               | Stall Ratio (SR)                                                                     | x               |             |           |             |                        |                |          |
|               | Coupling Corruption Propagation (CCP)                                                 | x               |             |           |             |                        |                |          |
|               | Critical Element Ratio (CER)                                                         | x               |             |           |             |                        |                |          |
|               | Ratio of Third-party Code (3PC)                                                       | x               |             | x         | x           |                        |                |          |
|               | Number of Outdated Code Components (OCC)                                              | x               |             | x         | x           |                        |                |          |
|               | Aging of Code Components (ACC)                                                       | x               |             | x         | x           |                        |                |          |
| Code          | Number of Different Programming Languages (PL)                                        | x               |             |           |             |                        |                |          |
| heterogeneity | Number of Different Programming Styles (PS)                                           | x               |             |           |             |                        |                |          |
|               | Number of Different Communication Protocols (DCP)                                     | x               |             |           |             |                        |                |          |
| Code property | Code quality metric | System layer to which the metric is relevant |
|---------------|---------------------|------------------------------------------|
|               |                     | Device firmware (non-object language)    |
|               |                     | Device firmware (OO language)            |
|               |                     | Backend/server application (OO language)  |
|               |                     | System user interface                    |
|               |                     | Network infrastructure                   |
| Size          | Lines of code (LOC) | x                                        |
|               | Estimated rebuild value (ERV) | x x x x |
|               | Average Unit Size (US) | x x x x |
|               | Unit Interface Size (UIS) | x x x x |
|               | Number of Distributed Architectural Components (DAC) | x x x x |
|               | Number of NoT Architectural Components (NAC) | x x x x |
| Redundancy    | Code Duplication (CD) | x x x x |
|               | Percentage of Redundant Code (RC) | x x x x |
| Complexity    | Cyclomatic Complexity (CYC) | x x x x |
|               | Cyclomatic Complexity per Unit (CCU) | x x x x |
|               | Depth of Inheritance Tree (DIT) | x x x x |
|               | Number of Children (NOC) | x x x x |
|               | Number of Thing Interconnections (NTI) | x x x x |
| Coupling      | Coupling Between Objects (CBO) | x x x x |
|               | Response for a Class (RFC) | x x x x |
|               | Number of Incoming calls per module (INC) | x x x x |
| Unit Test Coverage and Effectiveness | Line Coverage (LC) | x x x x |
|               | Branch Coverage (BC) | x x x x |
|               | Method Coverage (MC) | x x x x |
|               | Density of Assert Methods (DA) | x x x x |
| Cohesion      | Lack of Cohesion of Methods (LCOM) | x x x x |
|               | Lack of Cohesion of Methods 1 (LCOM1) | x x x x |
|               | Lack of Cohesion of Methods 5 (LCOM5) | x x x x |
|               | Conceptual Cohesion of Classes (C3) | x x x x |
|               | Ratio of Cohesive Interactions (RCI) | x x x x |
|               | Neutral Ratio of Cohesive Interactions (NRCI) | x x x x |
|               | Pessimistic Ratio of Cohesive Interactions (PRCI) | x x x x |
|               | Optimistic Ratio of Cohesive Interactions (ORCI) | x x x x |
| Code readability | Buse and Weimer’s Model (BW) | x x x x x x |
|               | Scalabrino et al. Model (SCA) | x x x x x x |
|               | Comment Percentage (CP) | x x x x x x |
| Security      | Number of Security Antipatterns per Line of Code (SAP) | x x x x x x |
|               | Number of Common Vulnerabilities and Exposures (CVE) | x x x x x x |
|               | Number of Zero-day Vulnerabilities per Day (ZDV) | x x x x x x |
|               | Attack Surface Size (AS) | x x x x x x |
|               | Stall Ratio (SR) | x x x x x x |
|               | Coupling Corruption Propagation (CCP) | x x x x x x |
|               | Critical Element Ratio (CER) | x x x x x x |
|               | Ratio of Third-party Code (3PC) | x x x x x x |
|               | Number of Outdated Code Components (OCC) | x x x x x x |
|               | Aging of Code Components (ACC) | x x x x x x |
| Code heterogeneity | Number of Different Programming Languages (PL) | x x x x x x |
|               | Number of Different Programming Styles (PS) | x x x x x x |
|               | Number of Different Communication Protocols (DCP) | x x x x x x |
TABLE 3. Relevance of discussed code quality metrics to the level of the code to principal IoT project development phases.

| Code property | Code quality metric | Level of the code | Development phases |
|---------------|---------------------|-------------------|-------------------|
| Size          | Lines of code (LOC) | x x x x x         | Project Class     |
|               | Estimated rebuild value (ERV) | x x x x x         | Function          |
|               | Average Unit Size (US) | x x x x x         | Technical design and architecture |
|               | Unit Interface Size (UIS) | x x x x x         | Coding            |
|               | Number of Distributed Architectural Components (DAC) | x x x x x         | Testing           |
|               | Number of Not Architectural Components (NAC) | x x x x x         | System operation  |
| Redundancy    | Code Duplication (CD) | x x x x x         |                   |
|               | Percentage of Redundant Code (RC) | x x x x x         |                   |
| Complexity    | Cyclomatic Complexity (CYC) | x x x x x         |                   |
|               | Cyclomatic Complexity per Unit (CCU) | x x x x x         |                   |
|               | Depth of Inheritance Tree (DIT) | x x x x x         |                   |
|               | Number of Children (NOC) | x x x x x         |                   |
|               | Number of Thing Interconnections (NTI) | x x x x x         |                   |
| Coupling      | Coupling Between Objects (CBO) | x x x x x         |                   |
|               | Response for a Class (RFC) | x x x x x         |                   |
|               | Number of incoming calls per module (INC) | x x x x x         |                   |
| Unit Test Coverage and Effectiveness | Line Coverage (LC) | x x x x x         |                   |
|               | Branch Coverage (BC) | x x x x x         |                   |
|               | Method Coverage (MC) | x x x x x         |                   |
|               | Density of assert Methods (DA) | x x x x x         |                   |
| Cohesion      | Lack of cohesion of Methods (LCOM) | x x x x x         |                   |
|               | Lack of cohesion of Methods 1 (LCOM1) | x x x x x         |                   |
|               | Lack of cohesion of Methods 5 (LCOM5) | x x x x x         |                   |
|               | Conceptual cohesion of Classes (C3) | x x x x x         |                   |
|               | Ratio of cohesive interactions (RCl) | x x x x x         |                   |
|               | Neutral ratio of cohesive interactions (NRCl) | x x x x x         |                   |
|               | Pessimistic ratio of cohesive interactions (PRCl) | x x x x x         |                   |
|               | Optimistic ratio of cohesive interactions (ORCl) | x x x x x         |                   |
| Code readability | Busen and Weimer’s Model (BW) | x x x x x         |                   |
|               | Scalabrino et al. Model (SCA) | x x x x x         |                   |
|               | Comment Percentage (CP) | x x x x x         |                   |
| Security      | Number of security antipatterns per line of code (SAP) | x x x x x         |                   |
|               | Number of common vulnerabilities and exposures (CVE) | x x x x x         |                   |
|               | Number of zero-day vulnerabilities per day (ZDV) | x x x x x         |                   |
|               | Attack surface size (AS) | x x x x x         |                   |
|               | Stall ratio (SR) | x x x x x         |                   |
|               | Coupling corruption propagation (CCP) | x x x x x         |                   |
|               | Critical element ratio (CER) | x x x x x         |                   |
|               | Ratio of third-party code (3PC) | x x x x x         |                   |
|               | Number of outdated code components (OCC) | x x x x x         |                   |
|               | Aging of code components (ACC) | x x x x x         |                   |
| Code heterogeneity | Number of different programming languages (PL) | x x x x x         |                   |
|               | Number of different programming styles (PS) | x x x x x         |                   |
|               | Number of different communication protocols (DCP) | x x x x x         |                   |
In construction of this mapping, several sources in the literature have been used. Sae-Lim et al. conducted an investigation study to find out how developers filter and prioritize code smells [79]. They found that the most important factors for filtering are 1) task relevance, 2) smell severity, and 3) task implementation cost. For the prioritization, the most important factors are 1) the module importance, 2) the task relevance, and 3) the testability. Moreover, individual source code metrics are mentioned as being used to detect code smells in several papers [80]–[83]. We have used analysis of these papers when constructing the map presented in Figure 3.

D. RELATION BETWEEN METRICS AND CODE SMELLS

An important point to mention and discuss is the relation of code smells [84] to the discussed code quality metrics. Our study does not explicitly focus on the problem from the code smells viewpoint. Programming styles for IoT systems might slightly differ by technology or in particular details implied by the type of system or hardware. However, general principles related to code quality and smells are generally valid for source code. As code metrics naturally overlap with code smells, this field is relevant to be discussed, especially in IoT systems.

According to Fowler [68] a code smell is a surface indication in the source code that usually corresponds to a deeper problem in the system. Code smells are not necessarily a problem, but rather an indicator of a problem. They can be seen as code structures that indicate a violation of fundamental design principles and negatively impact the quality of the design [85].

Some code smells are difficult to quantify by exact metrics; however, some can be considered closely related to the metrics discussed in this study.

Based on [86] urgent maintenance activities prioritizing features delivered over code quality often leads to code smells. Thus, code smells are codebase anomalies and do not necessarily impact the performance or correct functionality; they are patterns of bad programming practice that can affect a wide range of areas in a program, including reusability, testability, and maintainability. Code smells must be detected and managed by refactoring [68]. Similarly, Gupta et al. [87] stresses that it is essential to identify and control code smells during the design and development stages to achieve higher code maintainability and quality. Regarding the growing complexity of IoT systems and the potential problem with the number of various possible system configurations, code maintainability is especially important at this point.

In the reverse direction, the presence of code smells in the source code has a potential negative impact on system maintainability [84]. As observed, code smells are strongly influenced by LOC [84].

Gupta et al. [87] identifies 18 common code smells to be found and identified the interrelationship and driving power among these code smells. The motivation behind their work was to identify code smells with high driving power to improve overall code maintainability and to identify which smells are dependent. The effect of their dependency is that developers can refactor one of the dependent smells with higher driving power than to address all found smells and still significantly improve code maintainability. It is common to identify code smells in monolithic systems using code-analysis. For instance, tools like SpotBugs 5, FindBug 6, CheckStyle 7 or PMD 8 can detect code patterns that resemble a code smell. However, the common limitation is that a single code base can be fit. This can be limiting in the case of IoT systems, which typically consist of more components and individual system parts employing programming styles appropriate for different levels (practically from a device firmware to server-side enterprise application).

In a distributed environment, particularly for the microservice, commonly used for IoT middleware, multiple code smells have been identified. In [88], [89], these smells include improper module interaction, modules with too many responsibilities, or a misunderstanding of the microservice architecture. A good example is the existence of a legacy mediator module (similar to an enterprise service bus component in service-oriented architecture) to pass messages between modules, which is unintended. There may be too many standards involved across discrete teams of developers in the overall project when a single standard should be established for consistency across the modules. In the communication perspective, there might exist a wrong cut in the layers. There might be a missing manager component for connections between modules (API gateway), and rather, it used direct communication, which is a wrong practice. Hard-coded endpoints with IP addresses and ports are examples of malpractice. Regarding the development and design process, there might be no versioning of the API involved, or there exist too many MSA modules each for a rather small purpose. Another issue that can be found is shared persistence. This happens when using a single database for multiple modules instead of involving individual data storage. When one module accesses encapsulated data from another service, it can be seen as inappropriate service intimacy. Similarly, access to shared libraries is not preferred. Another smell to be found is a cyclic call dependency.

These smells can be detected manually, requiring assessment and a basic understanding of the system demanding considerable effort. However, these smells are easy to discover almost instantly with no previous system knowledge with code analysis instruments. Such a tool is not yet present. Similar to Microservices, these smells could be identified for IoT. However, this direction is an open research challenge.

To summarize, discussed metrics overlap with code smells to some extent. A poor value of a particular code quality metric can be considered a code smell in some cases.

---

5 https://spotbugs.github.io/
6 http://findbugs.sourceforge.net/
7 https://checkstyle.sourceforge.io/
8 https://pmd.github.io/
We analyze the situation in Figure 3. The discussed code properties (and hence code-quality metrics that express this property) are presented on a scale. The left side represents close relevance to code smell, and the right side of our opinion is that the relation is more remote. However, due to certain overlap but no exact relations established and different possible opinions on this problem, the overview presented in Fig. 3 might be subjective and also depends on the particular situation. To give an example, if we discuss Code heterogeneity, different programming languages (and different programming styles) might not necessarily imply problems if an IoT system consists of different devices, each having firmware written in the different programming language. Still, these source codes are well-written and reliable. On the contrary, when different programming styles are mixed in one logical block of code (e.g., one device), such a situation is usually considered as a code smell. Consequences of such a mixture of coding styles in an IoT system are typically lower maintainability of the complete source code base and potential issues with interoperability and integration when individual components are changed during system development and operation.

IV. DISCUSSION
This section discusses some issues arising from creating a consolidated view on code quality characteristics. A significant part of the metrics presented overlap with the metrics for standard software systems, which is logical: Software development for IoT employs current programming styles and programming languages. This also implies that the requirements for good source code overlap between software and IoT. Part of the metrics presented are specific to distributed IoT systems. As examples, we can give DAC, NTI, NAC, PL or PS.

The given overview may not be complete. Following our goal to include definitions and explanations of the metrics in the paper, a comprehensive overview of possible available metrics would be somewhat outside the limits of a standard journal article. However, due to our field’s best knowledge, we tried to select a representative sample of metrics useful for IoT system development.

Regarding the presented categorization of the metrics in code properties (see Table 1), some metrics might potentially belong to more code property categories, depending on the context. As an example, UIS can be given: this metric might belong to the Size as well as Complexity category. Another example are NTI and NAC, which could potentially be assigned to the Size as well as Complexity category. Also, scaling questions arise when applying the suggested metrics for an IoT system. In some parts of the IoT system, low-level languages such as C are used (and hence only some of the presented metrics are relevant). On the contrary, object-oriented languages are usually used for the back-end (BE) server parts of IoT systems. Typically, LOC for C code in IoT device has to be interpreted differently than the size of Java EE code on BE server part.

Regarding the code duplication, in our proposal, we have considered code blocks that are identical or identical after leaving comments and white spaces. However, more hidden code duplication can be present in code: the blocks might not be matching syntactically, but in the semantic sense, the code blocks might perform the same action. Recently, we observed this effect in the case of automated tests [90], for example. When one detects the code duplication by more advanced methods, also considering semantic duplication, practically the same metrics can be applied; only the duplicate blocks will be identified by a different method.

Generally, a PS metric may be subjective, as it may be difficult to distinguish individual coding styles; sometimes there is no clear boundary between the styles. To avoid this problem, we used a three-level rating of heterogeneity, which has to be determined by a specialist developer during a code review.

When discussing general quality characteristics in the ISO/IEC 25010:2011 taxonomy, we understand the possibility of effectively extending a discussed system as part of Changeability.

Unit test coverage and effectiveness (Section III-B5) is discussed as a relevant category for code quality, and a further detailed discussion can be conducted on the quality of the code of the created tests. As typical examples, unit tests code smells [91] and their fragility [92] can be expressed by metrics and further discussed. However, this field is outside the scope of our study. The unit test coverage metrics presented LC, BC, and MC shall be accompanied by the DA metric, because LC, BC and MC themselves do not explicitly capture a number of assertions in the unit tests, not talking about the relevance of the preformed unit tests in terms of a number of tested variants and data combinations.

Regarding the CVE metric, there are also alternatives for individual languages, for example, for C++ [93]. This proposal is based on the research of common vulnerabilities in C++ code. It considers generic code constructs such as the maximum nested cycles or the number of parameters of the methods and the amount of pointer arithmetic and other language-specific constructs. However, due to the language specificity, we have not included such proposals in the general overview provided in this study.

From a distributed perspective, we cannot expect metrics to add to each other when combined to obtain a holistic perspective. For instance, a SLOC of two modules or things of IoT can be easily measured; however, we cannot expect this to apply in general since one module or thing reliability summed with another does not behave that way. Queuing theory for static networks could be applied to this distributed perspective [35] to quantify reliability. Besides reliability, there are many dynamic metrics relevant to IoT and other distributed systems, such as maintainability, testability, security, safety, privacy, performance sustainability, fault tolerance, and resilience. Trust and security are even more challenging. In particular, in IoT, trust does not have an accepted definition, and security is unbounded as it suffers from unknown
V. THREATS TO VALIDITY

Several concerns may be raised about the completeness of the overview provided. This section discusses these possible limits and describes actions that shall mitigate these issues.

Impact of Metrics on System Quality Analysis we did (see Table 1 in Section III-C) could be subjective and not complete. The mapping was created using the algorithm described in Section III-C to minimize this effect. During this process, if a difference in opinion has been found, a discussion has been initiated, and the final version has been settled as a consensus. The same situation applies to the relevance of the metrics discussed to the main layers of an IoT system (see Table 2) and the mapping of the metrics to the code levels and the main phases of the project (see Table 3). In these cases, we took the same measures to minimize possible bias.

A similar concern might apply for the presented relation of discussed code properties to code smells (Figure 3), which is subjective and based on the analyzed literature and the experience of the authors. To minimize subjectivity, each author independently reviewed this positioning, and in case of different opinions, a consensus has been settled through a discussion.

Another concern can be raised regarding the completeness of the code quality metrics presented. However, a thorough search process and intensive discussions during the preparation of this study will minimize this possibility.

Another possible issue might be that some metrics may be included under a different name in the list. We have identified such an issue in the case of CYC, published under different names, but practically meaning the same, only in a different context. Other examples might probably be found for other metrics discussed in this study. However, we believe that the description and explanation provided for the individual metrics are sufficient to identify such possible ad hoc cases.

VI. CONCLUSION

This study provided a comprehensive overview of code quality metrics relevant to IoT systems. In this effort, we consolidated various code quality metrics from the available literature, revised them in the context of IoT systems, and further suggested our metrics to capture the properties of the source code of the IoT system. This overview categorized the metrics into nine major code property categories: size, redundancy, complexity, coupling, unit test coverage and effectiveness, cohesion, code readability, security, and code heterogeneity.

A significant part of the presented metrics overlaps with previously published metrics for software systems, which is logical. In IoT systems, software code is developed. Then, part of the metrics are more aimed at IoT and distributed systems.

The second part of the article analyzes the possible impact of the code quality aspects captured by the metrics on the general quality characteristics of an IoT system. To use a unified and established classification of general system quality characteristics, we used the ISO/IEC 25010:2011 standard, recently revised in 2017. Furthermore, the relevance of the analyzed metrics to the principal layers of an IoT system is discussed.

Finally, we discussed the code smells area and its relation to the presented metrics, as there are significant overlaps; in some cases, a lack of code quality expressed by a particular metric can be considered a code smell.

We believe that this overview will be helpful to researchers and IoT testing practitioners in planning quality assurance strategies and discussing the impact of code quality on the overall quality and reliability of a system.

ACKNOWLEDGMENT

The authors acknowledge the support of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 “Research Center for Informatics”. Bestoun S. Ahmed has been supported by the Knowledge Foundation of Sweden (KKS) through the Synergi Project AIDA - A Holistic AI-driven Networking and Processing Framework for Industrial IoT (Rek:20200067).
REFERENCES

[1] B. S. Ahmed, M. Bures, K. Frajtak, and T. Ceray, “Aspects of quality in internet of things (IoT) solutions: A systematic mapping study,” IEEE Access, vol. 7, pp. 13,788–13,790, 2019.

[2] E. J. Marisinnis, Y. Zorian, M. Konijnpenburg, C.-T. Huang, P.-H. Hsieh, P. Cockburn, J. A. K. van Rooyen, V. Rožič, B. Yang, and D. Singelée et al., “Iot: Source of test challenges,” in 2016 21th IEEE European Test Symposium (ETS). IEEE, 2016, pp. 1–10.

[3] M. Bures, M. Klima, V. Rechberger, B. S. Ahmed, H. Hindy, and X. Bellemk, “Review of specific features and challenges in the current internet of things systems impacting their security and reliability,” in Trends and Applications in Information Systems and Technologies, A. Rocha, H. Adeli, G. Dzemyla, F. Moreira, and A. M. Ramalho Correia, Eds. Cham: Springer International Publishing, 2021, pp. 546–556.

[4] A. u. R. Khan, q. f. Hassan, and S. Madani, Internet of Things: Challenges, Advances, and Applications, 01 2018.

[5] J. Kurthika and S. Khaddaj, “Software quality issues and challenges of internet of things,” in 2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCARES). IEEE, 2015, pp. 176–179.

[6] E. Cavalcante, J. Pereira, M. P. Alves, P. Maia, R. Moura, T. Batista, F. C. Delicato, and P. F. Pires, “On the interplay of internet of things and cloud computing: A systematic mapping study,” Computer Communications, vol. 89, pp. 17–33, 2016.

[7] J. Giraldo, E. Sarkar, A. A. Cardenas, M. Maniatakos, and M. Kantarcioglu, “Security and privacy in cyber-physical systems: A survey of 34 no. pp. 4, 2017–17, 2017.

[8] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, “A survey on security and privacy issues in internet-of-things,” IEEE Internet of Things Journal, vol. 4, no. 5, pp. 1250–1258, 2017.

[9] J. Pantichiana, M. Lanza, and G. Bavota, “Improving code: The (mis) perception of quality metrics,” in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018, pp. 80–91.

[10] O. V. Rapisaro, G. A. Mari, M. G. Marcus, and M. Di Penta, “On the impact of refactoring operations on code quality metrics,” in 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014, pp. 456–460.

[11] T. J. McCabe, “A complexity measure,” IEEE Transactions on software Engineering, no. 4, pp. 308–320, 1976.

[12] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” IEEE Transactions on software engineering, vol. 20, no. 6, pp. 476–493, 1994.

[13] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, “Cropping and cohesion (towards a valid metrics suite for object-oriented analysis and design),” Object oriented systems, vol. 3, no. 3, pp. 143–158, 1996.

[14] L. H. Rosenberg and L. E. Hyatt, “Software quality metrics for object-oriented environments,” Coursstalk journal, vol. 10, no. 4, pp. 1–6, 1997.

[15] R. Baggen, J. P. Correa, K. Schull, and J. Visser, “Standardized code quality and benchmarking for improving software quality,” Software Quality Journal, vol. 20, no. 2, pp. 287–307, 2012.

[16] M. A. Khan and K. Salah, “Iot security: Review, blockchain solutions, and open challenges,” Future Generation Computer Systems, vol. 82, pp. 395–411, 2018.

[17] S. Hannebo, F. I. Khan, and B. Hannebo, “Understanding security requirements and challenges in internet of things: A topic review,” Journal of Computer Networks and Communications, 2019, 2019.

[18] L. van der Aalst, E. Roodenrijns, J. Vink, and R. Baarda, TMap NEXT: business driven test management. Uitgeverij kleine Uil, 2013.

[19] T. Koomen, B. Broekman, L. van der Aalst, and M. Vroon, TMap next: for result-driven testing. Uitgeverij kleine Uil, 2013.

[20] I. Heitlager, T. Kuipers, and J. Visser, “A practical model for measuring maintainability,” in 6th international conference on the quality of information and communications technology (QUATIC’2007). IEEE, 2007, pp. 30–39.

[21] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), USA: Prentice Hall PTR, 2004.

[22] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martinez-Perez, and C. Soubrièver-Montalvo, “Source code metrics: A systematic mapping study,” Journal of Systems and Software, vol. 128, pp. 164 – 197, 2017.

[23] M. Bures, “Metrics for automated testability of web applications,” in Proceedings of the 16th International Conference on Computer Systems and Technologies, 2015, pp. 83–89.
I. Alexopoulos, S. Neophytou, and I. Kyriakides, “Identifying metrics for an iot performance estimation framework,” in 2021 10th Mediterranean Conference on Embedded Computing (MECO), 2021, pp. 1–6.

B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” 2007.

M. Bures, X. Bellekens, K. Frajatik, and B. S. Ahmed, “A comprehensive view on quality characteristics of the iot solutions,” in EAI International Conference on IoT in Urban Space. Springer, 2018, pp. 59–69.

D. Dvorak, “Nasa study on flight software complexity,” in AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference, 2009, p. 1882.

I. Siket, A. Beszédes, and J. Taylor, “Differences in the definition and calculation of the loc metric in free tools, technical report, 2014.”

C. Jones, “Software metrics: good, bad and missing,” Computer, vol. 27, no. 9, pp. 98–100, 1994.

M. Bystricky and V. Vranić, “Modularizing code by use cases and tests for better maintainability,” in Comparison to the First International Conference on the Art, Science and Engineering of Programming, 2017, pp. 1–1.

K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, “Is duplicate code more frequently modified than non-duplicate code in software evolution? an empirical study on open source software,” in Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IPSWES), 2010, pp. 73–82.

S. Turati and S. Rambout, “Coupling and cohesion metrics for object-oriented software: A systematic mapping study,” in Proceedings of the 11th Innovations in Software Engineering Conference, ser. ISE’C 18. New York, NY, USA: Association for Computing Machinery, 2018.

L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in Proceedings of the 36th international conference on software engineering, 2014, pp. 435–445.

H. Zhu, P. A. Hall, and J. H. May, “Software unit test coverage and adequacy,” Acm computing surveys (csur), vol. 29, no. 4, pp. 366–427, 1997.

J. Ingens, Software Architect’s Handbook: Become a successful software architect by implementing effective architecture concepts. Packit Publishing Ltd, 2018.

A. Marcus, D. Poshyvanyk, and R. Ferenc, “Using the conceptual cohesion of classes for fault prediction in object-oriented systems,” IEEE Transactions on Software Engineering, vol. 34, no. 2, pp. 287–300, 2008.

Heung Seok Chae and Yong Rae Kwon, “A cohesion measure for classes in object-oriented systems,” in Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No.98TB00026), 1998, pp. 158–166.

J. Al Dallal, “A design-based cohesion metric for object-oriented classes,” International Journal of Computer Science and Engineering, vol. 1, pp. 195–200, 2007.

L. Briand, R. Morasca, and V. Basili, “Defining and validating high-level design metrics, 1994.”

L. Briand, S. Morasca, and V. Basili, “Defining and validating measures for object-based high-level design,” Software Engineering, IEEE Transactions on, vol. 25, pp. 722 – 743, 10 1999.

F. Riguzi, “A survey of software metrics,” 2002.

L. C. Briand, S. Morasca, and V. R. Basili, “Measuring and assessing maintainability at the end of high level design,” in 1993 Conference on Software Maintenance, 1993, pp. 88–87.

R. P. Buse and W. R. Weimer, “Learning a metric for code readability,” IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 546–558, 2009.

S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, “Improving code readability models with textual features,” in 2016 IEEE 24th International Conference on Program Comprehension (ICPC). IEEE, 2016, pp. 1–10.

M. Fowler, Refactoring: Improving the Design of Existing Code. USA: Addison-Wesley Longman Publishing Co., Inc., 2018.

W. J. Brown, R. R. Malveau, H. W. McCormick III, and T. J. Mowbray, “Antipatterns: Refactoring software, architectures, and projects in crisis,” 1998.

N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “Decor: A method for the specification and detection of code and design smells,” IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20–36, 2010.

T. Nafees, N. Coull, I. Ferguson, and A. Sampson, “Vulnerability anti-patterns: a timeless way to capture poor software practices (vulnerabilities),” in 24th Conference on Pattern Languages of Programs. The Hillside Group, 2018.

A. Lamba, S. Singh, and S. Balvinder, “Mitigating zero-day attacks in insecure web services,” International Journal for Technological Research in Engineering, vol. 4, no. 1, 2016.

P. K. Manadhata and J. M. Wing, “An attack surface metric,” IEEE Transactions on Software Engineering, vol. 37, no. 3, pp. 371–386, 2011.

I. Chowdhury, B. Chan, and M. Zulkernine, “Security metrics for source code structures,” in Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems, ser. SESS ’08. New York, NY, USA: Association for Computing Machinery, 2008, p. 57–64. [Online]. Available: https://doi.org/10.1145/1370905.1370913

P. Ferrara, A. K. Mandal, A. Cortesi, and F. Spoto, “Static analysis for discovering iot vulnerabilities,” International Journal on Software Tools for Technology Transfer, vol. 23, no. 1, pp. 71–88, 2021.

D. Savchenko, T. Hyminnen, and O. Taipale, “Code quality measurement: Case study,” in 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, pp. 1455–1459.

S. Motogna, A. Vescan, C. Serban, and P. Tirban, “An approach to assess maintainability change,” in 2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2016, pp. 1–6.

I. Chowdhury, B. Chan, and M. Zulkernine, “Is duplicate code more harmful? an empirical study on iot performance estimation framework,” in 2018 41st International Convention on Information and Systems Research in Engineering (ICSE), 2019, pp. 60–71.
MATEJ KLIMA is currently a PhD. student in Software testing Intelligent Lab (STILL), dept. of Computer Science and Engineering, Czech technical University in Prague. In his research, Matej focuses on reliability of Internet of Things and critical systems, for instance model-based testing techniques aimed to situations of limited or disrupted network connectivity, or system component outage, IoT system quality and related metrics or IoT reliability models.

MIROSŁAW BURES leads the System Testing IntelLigent Lab (STILL) at the Dept. of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague where he is currently the Associate Professor of Computer Science.

His research interests include quality assurance and reliability methods, model-based testing, path-based testing, combinatorial interaction testing and test automation for software, Internet of Things, and mission-critical systems. He leads several projects in the filed of test automation for software and Internet of Things systems, covering the topics of automated generation of test scenarios as well as automated execution of the tests.

MICHAL TRNKA Michal Trnka obtained master’s degree in computer science from Faculty of Electrical Engineering at Czech Technical University in Prague and he is currently waiting for his Ph.D. defense at Czech Technical University in Prague. He is also a Fulbright scholar. His area of interest is software engineering and especially context-aware applications security. Lately, he has focused on using context information to enhance security rules, as well as methods to obtain context from trusted sources, such as devices on the Internet of Things.

KAREL FRAJTAK is lecturer and researcher with the System Testing IntelLigent Lab (STILL) at the Dept. of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague. He received his Master’s, and Ph.D. degrees from the Faculty of Electrical Engineering at the Czech Technical University in Prague.

VACLAV RECHTBERGER is PhD. student in Software testing Intelligent Lab (STILL), dept. of Computer Science and Engineering, Czech technical University in Prague. His focus is Model-based Testing, test automation and testing of Internet of Things systems.

XAVIER BELLEKENS (BSc, MSc, PhD, FHEA, MBCS, MIEEE) is the CEO and co-founder of Lupovis.io a company that provides dynamic cyber-deception, a Nonresident Senior Fellow of the Scowcroft Center for Strategy and Security at the Atlantic Council advising on critical infrastructures, maritime and naval cyber-security and a Lecturer (Assistant Professor) Chancellor’s Fellow in the Institute for Signals, Sensors and Communications with the Department of Electronic and Electrical Engineering at the University of Strathclyde. His current research interests include critical infrastructure protection, defence as well as cyber deception and deterrence.

TOMAS CERNY is a Professor of Computer Science at Baylor University. His area of research is software engineering, code analysis, security, aspect-oriented programming, user interface engineering, and enterprise application design. He received his Master’s, and Ph.D. degrees from the Faculty of Electrical Engineering at the Czech Technical University in Prague and his M.S. degree from Baylor University.

From 2009 to 2017, he was an Assistant Professor of Computer Science at the Czech Technical University, FEE, Prague, Czech Republic. In 2017, he was a Postdoctoral fellow at Baylor University, Texas, USA, and from the same year he continues as Assistant Professor with the concentration on Software Engineering.

BESTOUN S. AHMED obtained his Ph.D. from University Sains Malaysia (USM) in 2012. Currently, he is an Associate Professor at the department of mathematics and computer science, Karlstad University, Sweden, and a researcher with the Czech Technical University in Prague. His main research interest includes Combinatorial Testing, Search-ased software testing (SBST), Machine Learning Testing, Applied Soft Computing, and quality assurance of smart devices and IoT systems.