# Survey on software quality and dependability

Concepts, terminology and software fault tolerance techniques

João Almeida*, José Donato†
Departamento de Engenharia Informática
Universidade de Coimbra
*jlalmeida@student.dei.uc.pt, †donato@student.dei.uc.pt

***Abstract*—Software quality attributes are essential but commonly overlooked, leading to projects being riddled with faults (*bugs*). This paper aims to give a brief exposition of the encompassing concepts and techniques, and be a valuable resource for those wanting to get up to speed with them. Striving for a compromise between completeness and conciseness, references for each topic are provided to allow further exploration at the reader's discretion.**

***Index Terms*—Software Quality, Security, Dependability, Fault tolerance**

## I. Introduction

Software is inconceivably ingrained into our society. We depend on it for an enormous range of purposes: from automating the more menial day-to-day tasks to managing life-critical systems.

Sadly, too often we are only concerned with delivering the required functionality, disregarding essential quality attributes due to time/money/knowledge constraints.

And sure, performance is usually a big focus, but only because it directly leads to increased profits (less computing power needed), and likewise for usability (more *attractive* to customers).

The big problem is regarding attributes like (most notably) security, where we don't have such a direct incentive to ensure it, and as such it's where most *corners are cut* [15]. Only when the inevitable incident occurs (with potentially devastating consequences), do we realize our mistake actually happened when we first started the project, since such quality attributes aren't really implementable as an afterthought - they must be considered since its inception (*by design*).

With this in mind we present a condensed but complete view of software quality concepts, with particular focus on security/dependability issues and associated techniques. The **goal** is thus to provide an adequate introduction into the regularly overlooked facets of software quality.

The paper will be **structured** starting with the very broad concept of 'Software Quality' and its related terminology: its attributes and requirements, among others (Sec. II). We then follow Fig. 1's hierarchy of concepts (shown in a bigger scale in the appendix - Fig. 2): , beginning with security (Sec. III), progressing upwards into dependability (Sec. IV) and finally resilience (Sec. V). Section VI will be dedicated to faults and the approaches available to handle them, while Section VII will conclude the paper.

Further readings will be provided through the paper hoping to aid the reader in selectively deepening their knowledge.

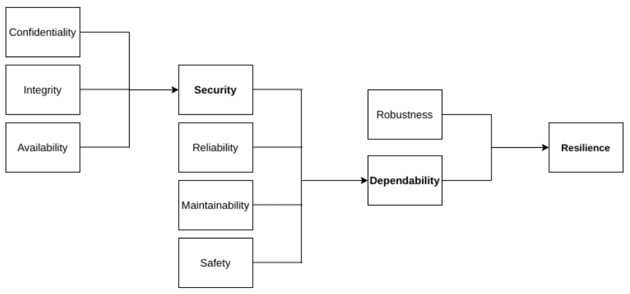

Fig. 1. Concept Hierarchy (adapted from [1])

## II. Software Quality

Providing adequate definitions for *simple* terms like 'software quality', is deceptively hard [11]. Even IEEE standardized glossary's [7] definition seems quite lackluster:

"The degree to which a system, component, or process meets specified requirements."

We'll then choose to formulate our own definition, based on [1] and [7]:

**Software Quality:** The degree to which the software satisfies requirements regarding **how** its functionalities are achieved and assured.

### A. *Software Views*

We typically look at software from two distinct points of view [1]:

- Functional View: **what** it does. Related to the software's functionalities.
- Non-functional View: **how** it is done. For example, If the system is *fast* or *reliable* at providing a certain functionality - related to its **quality**.

### B. *Quality Requirements*

The software's **requirements** consist of a set of statements specifying the user's needs. These may be subdivided according to each software **view**:

- Functional: are directly related to the functionalities, i.e., what the system should provide to the users
- Non-functional: constraints/goals on **how** how the system should provide those functionalities to the users, ie. **quality requirements**.

An example of a non-functional (quality) requirement could be that a certain functionality is only available to specific users, and as such requires their previous authentication.

### C. *Quality Attributes*

A **software attribute** is a characteristic affecting its quality [7], [8].

There are numerous such attributes, many of which we will discuss in the next sections since they are related with our paper's main focus - security and dependability:

- **Confidentiality**: III-A
- **Integrity**: III-B
- **Availability**: III-C (where **scalability** is also discussed)
- **Reliability**: IV-A
- **Maintainability**: IV-B
- **Safety**: IV-C
- **Robustness**: V-A

There are however many other quality attributes, most notably:

- **Performance**: related to the software's *speed*, particularly how it handles its workload.
- **Usability**: related to the user's ease of interaction with the software, ie. whether acceptable performance can be achieved for each task ([17]). It can be decomposed into many *sub-attributes* (learnability, efficiency,...[1]), for which guidelines are available ([16] is a good example).
- **Cost**: can be time and/or money. Even though this may seem quite out of place next to the other attributes we have enumerated, cost is in fact the biggest constraint to quality. The bigger the amount of resources we invest into the software [1], the more (quality) requirements we can afford to satisfy.

We might then question how these attributes can be *measured*? It certainly looks a necessity for assessing the quality of the software.

Well, in some cases, such (quality) **metrics** may be straightforward to identify. For example, with performance we may only care about the throughput (work per time unit [1]), easily ***benchmark****able* - where we define a common workload to be executed by different (versions) of the system, and for which the results will serve as comparison.

But how do we measure usability? We could test in on many people, but there would still be representativeness concerns... And likewise security is notoriously hard to measure [14], since a single vulnerability may be the cause of great harm done to a system and/or its **stakeholders**[1], and we can never be sure of their absence.

### D. *Quality Factors*

Quality attributes can be organized hierarchically, with those in higher levels being called **quality factors** [8].

Our paper is structured following this concept, focusing on increasingly encompassing quality factors: security, dependability, resilience (as shown in Fig. 2).

### E. *By Design*

Whereas functionalities can be generally added in an *ad hoc* manner, quality requirements require careful planning for their accommodation.

This planning must start from its inception (**by design**), since that is when we have the most flexibility to change its architecture/functional requirements. The further along we wait in the development *life cycle*, the bigger the cost that will ensue. Studies have shown that reworking represents 40%-50% of a project's efforts [15].

Security is a glaring example of this. It must be a consideration when implementing each functionality, imposing considerable constraints on its working (requirements), and how we structure the different components making up our software (architecture).

SQUARE [18] is a methodology to develop system requirements with security in mind, but many similar exist (cf. [20]).

[1]anyone benefitting from it [13]

## III. Software Security

To develop secure systems the confidentiality, integrity and availability of information system resources must be preserved [3]. According to NIST standard FIPS 199, these three attributes are the security goals for information systems [4].

In addition to those, there are also other important attributes that we must think about when developing secure systems: **accountability** and **authentication**. The former is related to mechanisms to link actions to their actors and the latter refers to mechanisms to make sure that an individual is who he claims to be.

We need to understand that we can never guarantee perfect security. However, the use of methodologies such as Security by Design and in-depth Security (several overlapping layers of defense mechanisms) will help us implement securer systems.

Sometimes it is hard for a system to combine all security attributes due to the fact that they can collide with one another. [26] is a good reference that studies exactly this, with one example being availability (Sec. III-C) vs. confidentiality (Sec. III-A).

In the following subsections, we will focus on the main security attributes. For further analysis about this topic we recommend the chapter: *Introduction To Software Security Concepts* of [19].

### A. Confidentiality

**Confidentiality**: consists on mechanisms to forbid unauthorized parties to access certain data. This can be achieved with authentication and role based access control approaches, for example.

This concept is quite simple, we want private data to remain private. To achieve this, encryption with robust algorithms and role based access controls must be used.

To dive into the encryption topic, [2] is recommended.

### B. Integrity

**Integrity**: make sure only authorized users can create, remove or change certain data.

This means that information can only be changed by users with a certain set of permissions. Also, the system needs to provide its functionalities without unauthorized manipulation.

Usual mechanisms to preserve integrity are: input and data validation (ie. not blindly trusting a user) and role based access controls. Due to the importance of this attribute, role based access controls is forced by General Data Protection Regulation in Europe. Recently, a Hospital from Portugal was fined in €400k because all the users in the system had the same level of privileges, i.e., employees or doctors without proper authorization could see and change medical records that should only be available to authorized doctors [22].

### C. Availability

**Availability**: ensure that users can use the services provided by the system when they want or need.

A common attack to this attribute is denial-of-service (abbr. DoS) because it does not require high skill level and the consequences can be severe (eg. consider the financial loss Amazon has for each period of downtime). To prevent this, companies like Cloudfare provide

> "fast, globally distributed and intelligent protection against sophisticated DDoS attacks."

This is possible because Cloudfare has a distributed, redundant network that absorb the *flood or superfluous requests* associated with this type of attacks. The reference [34] has tutorials on how to use this product.

A quality attribute that we can relate to availability, is **scalability** - the ability for a system to handle increasing load [9].

If this load is unpredictable, **elasticity** is a desirable property - adding computing resources when such need is detected [9]. However we need to be careful of this increasing load possibly being malicious, due to the possibility of a DoS.

## IV. Software Dependability

Dependability: builds on security, i.e., aggregates the security attributes we talked so far (confidentiality, integrity and availability) plus reliability, maintainability and safety. We can define dependability as the ability to deliver a service that can be trusted while avoiding service failures that can have more serious consequences than a certain limit [11].

Achieving a dependable system is a big challenge because it conflicts with other attributes important to the business side of a certain system such as usability. Also, another constraint is that it can be expensive. [2]

Nevertheless according to a NIST [30], the impact of faults in software, due to "inadequate infrastructures for testing" represented a cost of over \$59 billion annually, but reducible by over \$22 billion through feasible improvements. These values date to 2002, but we can only assume they have increased over time.

[2] [9]'s a great resource on building reliable/maintainable systems

### A. *Reliability*

**Reliability** is concerned with continuity of correct service [11]. Specifically, we wish to reduce the occurrence of **failures** - deviations of a system's service from its specification [10] - and are thus usually concerned with the mean time between them (MTBF [1]).

Our goal is then to prevent **faults** from *materializing* into failures. To do so we have a variety of approaches, which will be discussed in Section VI.

It's important to note that reliability should be a concern not only to those developing highly critical software (eg. nuclear power stations [9]). Even if failures don't pose a **safety** (IV-C) problem, they can nevertheless affect a company's reputation, financials,... [9].

### B. *Maintainability*

A software system may *be around* for a long time, requiring new functionalities and/or adjustments to existing ones (eg. bug fixing). Additionally, it may need to be deployed in a variety of scenarios, and the team responsible for the it may change over time.

**Maintainability** can be considered the ease with which such operations can be done (adaptation of the existing software [13]) typically powered by abstractions providing simplicity [9].

The contributions towards dependability include the ease (leading to reduced cost) in removing newly identified faults, along with not being prone to inadvertently introducing new ones.

### C. *Safety*

**Safety** consists of the absence of negative consequences from the software on its users/environment [11][13].

Important to note that safety will not be a consideration for every software system [11]. Returning to the nuclear power plant example, it'll certainly be (a failure will lead to devastating consequences), but for a generic website it will most likely not.

The amount of faults we will be able to *accept* (and handle by tolerating) will also be highly dependent on the *level* of safety required for our system. For critical systems, **formal methods** such as inspections and model checking are a must (VI-B), for others the associated cost is a deterrent. Similarly, it may lead to different process models having to be followed [13], needing to avoid the unstructuredness of agile methods.

Once again we can identify relationships between different quality attributes. In this case, the safety of a system is dependent not just on the mechanisms we implement to assure it, but also on the **availability** of these (useless otherwise).

## V. Software Resilience

**Resilience** *builds* on dependability, i.e., it is the combination of all dependability attributes plus robustness. If a system is facing changes, for example, in a non-static environment, it is resilient if it is able to maintain the persistence delivering its service and still be trusted.

> "The word "resilience", from the Latin verb resilire (re-salire: to jump back), means literally the tendency or ability to spring back, and thus the ability of a body to recover its normal size and shape after being pushed or pulled out of shape, and therefore figuratively any ability to recover to normality after a disturbance." [21]

This definition is interesting, we can transpose it directly to software. If we have a resilient system, we have a system that can easily adapt to environment changes without losing any of the attributes talked before: confidentiality, integrity, maintainability, etc. We already know that the addition of these attributes results in a system dependable but we still need one more attribute to achieve resilience and that is robustness. Robustness will be defined in the following subsection.

### A. *Robustness*

**Robustness**: is an attribute of resilience and it measures the behaviour of the system under non-standard conditions. A robust system should operate correctly even when it is presented with not expected inputs or extraordinary environment conditions.

With robustness testing we can discover faults (normally human cause, programming or design faults). If we find faults, the developer can can fix them and increase how robust the system is.

There are lot of strategies to perform these tests but the most relevant, in our opinion, are called **fuzzing** and **penetration testing**. In the former abnormal inputs are inserted with the goal of discovering vulnerabilities (by *triggering* errors) to fix them later. [24] is a good book about fuzzing, it describes interesting fuzzing areas such as: fuzzing methods or automation tools to perform this test and consequently achieve robustness. **Penetration testing** has the goal of identifying vulnerabilities but it is more related to malicious inputs (ie. we adopt an attacker's perspective).

[6] More about robustness in [6].

## VI. Faults

A **fault** is the potential cause of a deviation in the system state [27] (**error** [1]). This may in turn lead to a **failure**, in turn affecting its service (cf. IV-A).

Faults can be *in* [9]:

- hardware, not the focus of paper, but an example would be *wear and tear* of physical components
- software, commonly called ***bugs***. We need systematic methods to attempt to identify them (VI-B)
- humans, through **mistakes** when interacting with the system that activate *bugs* (usability is therefore important), but also when exploiting vulnerabilities (**malice**)

Software faults, which we will refer to by their more common name of *bugs*, can also be classified as:

- deterministic (*Bohrbugs*), lead to failure predictably [27].
- non-deterministic (*Heisenbugs*), harder to detect since failure (manifestation) is often dependent on another condition [29]
- specific cases, cf. [29].

A particular instance of (security-related) faults are **vulnerabilities**, that must be activated (exploited) through an attack. Interesting to note that the majority of these faults are due to few simple mistakes (cf. [32] for a frequency table).

Another interesting type of faults, and relevant in the context of dependability, are **Byzantines** faults. These are based on the "Byzantine generals problem" (introduced in [33]), where some generals may send incorrect information when planning an attack. Our system should provide correct service in spite of this, usually through redundancy [9].

Please refer to [11] for a complete taxonomy of faults.

The following sub-sections will go over the distinct approaches we have for handling faults in our software.

### *A. Fault Prevention*

When first designing the software, we can consider a set of measures in an attempt to minimize (but ideally prevent) the introduction of faults [11].

This is typically done by following best practices, like those provided by OWASP for security [31].

Coding styles, ensuring the consistency of the code that's written (airbnb example), can also be helpful in order to facilitate peer reviews.

Finally, a test-driven approach (cf. **TDD**) can be a valuable aid, particularly in avoiding the inadvertent introduction of bugs in existing features (regression testing, also explained in [28]).

### *B. Fault Removal*

However even following best practices, faults are bound to appear in the code. We thus need methods for identifying such faults, so that they can be removed.

> "At implementation time, we can increase the dependability of the system through fault removal techniques." [10]

Such techniques can be analysis-based or testing-based.

Without diving too deep, the first one means that the program is not running and we analyse the source code to identify faults, either through automated tools like static code analysers, or through *manual* methods for which there are various strategies: reviews, analysis, inspections, etc.

Regarding testing, tests are made in order to see if the code is doing what is expected to, i.e., we want to understand if the dynamic behaviour of the code is correct or not. This can be classified as:

- black-box if we base the tests solely off of the specification (does it comply?)
- white-box when using the source code as a reference, aiming for coverage of all possibilities.

There are multiple testing types that can help removing faults.[28] is a great resource to carry on on this topic.

### *C. Fault Tolerance*

We can't expect to identify every existing fault, due to both human error and shortcomings in the available automated tools (*recall* of 1 is infeasible). Dijkstra famously commented [12] on the "fantastically high" number of inputs that must be considered when testing, stating that we can never assure ourselves of the absence of *bugs* (faults). Additionally, manually fixing faults is costly, particularly so when maintainability is lacking.

Another *level of protection* is therefore needed, to what could be considered a defense in-depth approach.

**Fault tolerance** techniques are applied at execution time [10], and are meant to handle faults by reacting to their manifestation [21], providing correct service in spite of them. In some cases though, service may experience some degradation, even if we avoid system failure [9]. There are many such techniques, but we can typically categorize them into three groups:

- **Redundancy**: where we create several copies of the same software, so that failure of one does not lead to the failure of the system as a whole. A common example is N-version programming, whereby the same software is be developed by several independent teams, all following the same specification. This is then used to identify faults in one of them (eg. if we have two in disagreement - **comparison** [10]) and possibly make a decision in spite of this (eg. when we have three, with two in agreement - majority **voting** [10]). Obviously one needs to understand whether this technique should be applied to their case, given the high costs that are needed (a typical example would be for critical systems such as aircraft sensors).

- **Recovery**: through transformation of the system state into one that's fault *free* [11]. A common example is through **rollback**, that is, registering all operations undertook (*logging*) or saving current state (*checkpointing*), in a *stable storage* [23], so that the system can return to a certain state in case of failure (common in databases, for instance). Another that we could consider is a *watchdog* [27], detecting failures to then restart the software.
- **Acceptance** Test: where we test the result of the program [10]. If it fails we consider the existence of some fault, even though we can't identify its origin [10]. In some scenarios, parity checking could be an example. A common application are *one-way functions*, where it's *hard* to compute but *easy* to validate (therefore not much overhead in applying this).

Important to note though, that fault tolerance is not a *substitute* for the remaining mechanisms we have studied, and should be looked at as a final barrier between service and failure.

### D. *Fault Forecasting*

There are techniques for estimating the number of faults in our systems, and therefore attempt to quantify its dependability.

By far the most interesting would seem to be **fault injection** given its remaining benefits [27]. By introducing faults and seeing how the system reacts, we can test the fault tolerance mechanisms' efficiency. From here we can extrapolate the number of existing, and not yet handled, faults (assess our coverage [27]). [35] and [36] are two great starting points to assess mobile systems through fault injection.

Fault forecasting helps us assess the confidence we can place in our system [11]. For this end, Netflix famously employs *chaos monkeys* [25].

## VII. Conclusion

Defining software quality and dependability concepts is not a trivial task. With this paper we tried to provide brief descriptions based on reliable references for each topic while exposing some of our critical analysis. The definitions on this survey are a solid starting point into each topic and if the reader wants to dive into a bit deeper in one specific, we tried to always provide further readings for more in-depth analysis.

## References

[1] 'Software Quality and Dependability' slides

[2] William Stallings. Cryptography and Network Security: Principles and Practice. Pearson, 2017.

[3] Barbara Guttman, E Roback. An Introduction to Computer Security: the NIST Handbook. 1995

[4] National Institute of Standards and Technology. Standards for Security Categorization of Federal Information and Information Systems. 2004

[5] Walter L. Heimerdinger Charles B. Weinstock. A Conceptual Framework for System Fault Tolerance. 1992.

[6] Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, Nuno Antunes. Robustness Testing Techniques and Tools. Springer, Berlin, Heidelberg, 2012.

[7] 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology
https://ieeexplore.ieee.org/document/159342/definitions#definitions

[8] 1061-1998 - IEEE Standard for a Software Quality Metrics Methodology
https://ieeexplore.ieee.org/document/749159/definitions#definitions

[9] Kleppmann, Martin. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. " O'Reilly Media, Inc.", 2017.

[10] Heimerdinger, Walter L., and Charles B. Weinstock. A conceptual framework for system fault tolerance. No. CMU/SEI-92-TR-33. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1992.

[11] Avizienis, Algirdas, et al. "Basic concepts and taxonomy of dependable and secure computing." IEEE transactions on dependable and secure computing 1.1 (2004): 11-33.

[12] E. W. Dijkstra. Structured Programming.
http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF

[13] Sommerville, Ian. Software Engineering 10th edition. Peason Education, 2016.

[14] Bellovin, Steven M. On the brittleness of software and the infeasibility of security metrics. IEEE Security & Privacy 4.4 (2006): 96-96.

[15] Mead, Nancy. Security Requirements Engineering. 2006.
https://resources.sei.cmu.edu/asset_files/WhitePaper/2010_019_001_297315.pdf

[16] Ben Shneiderman. The Eight Golden Rules of Interface Design.
https://www.cs.umd.edu/users/ben/goldenrules.html

[17] 1023-2004 - IEEE Recommended Practice for the Application of Human Factors Engineering to Systems, Equipment, and Facilities of Nuclear Power Generating Stations and Other Nuclear Facilities
https://ieeexplore.ieee.org/document/1440988/definitions#definitions

[18] Mead, Nancy R., and Ted Stehney. ”Security quality requirements engineering (SQUARE) methodology.” ACM SIGSOFT Software Engineering Notes 30.4 (2005).

[19] Cotroneo, Domenico, ed. Innovative technologies for dependable ots-based critical systems. Springer, 2013.

[20] Fabian, Benjamin, et al. ”A comparison of security requirements engineering methods.” Requirements engineering 15.1 (2010): 7-40.

[21] Wolter, Katinka, et al., eds. Resilience assessment and evaluation of computing systems. Berlin, London: Springer, 2012.

[22] Visão — CNPD: Hospital do Barreiro multado em 400 mil euros por permitir acessos indevidos a processos clínicos https://visao.sapo.pt/exameinformatica/noticias-ei/mercados/2018-10-19-cnpd-hospital-do-barreiro-multado-em-400-mil-euros-por-permitir-acessos-indevidos-a-processos-clinicos/

[23] Treaster, Michael. ”A survey of fault-tolerance and fault-recovery techniques in parallel systems.” arXiv preprint cs/0501002 (2005).

[24] Sutton, Michael, Adam Greene, and Pedram Amini. Fuzzing: brute force vulnerability discovery. Pearson Education, 2007.

[25] The Netflix Simian Army - Netflix TechBlog https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116

[26] Wilson, Kelce S. ”Conflicts among the pillars of information assurance.” IT Professional 15.4 (2012): 44-49.

[27] Carreira, João, and João Gabriel Silva. ”Why do some (weird) people inject faults?.” ACM SIGSOFT Software Engineering Notes 23.1 (1998): 42-43.

[28] Ammann, Paul, and Jeff Offutt. Introduction to software testing. Cambridge University Press, 2016.

[29] Heisenbug - Wikipedia https://en.wikipedia.org/wiki/Heisenbug

[30] Planning, Strategic. ”The economic impacts of inadequate infrastructure for software testing.” National Institute of Standards and Technology (2002).

[31] OWASP Secure Coding Practices Quick Reference Guide https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf

[32] Barbosa, Raul, et al. ”The most frequent programming mistakes that cause software vulnerabilities.” arXiv preprint arXiv:1912.01948 (2019).

[33] Lamport, Leslie, Robert Shostak, and Marshall Pease. ”The Byzantine generals problem.” Concurrency: the Works of Leslie Lamport. 2019. 203-226.

[34] DDoS Protection By CloudFlare — CloudFlare DDoS Guide https://www.a2hosting.com/kb/add-on-services/cloudflare/using-cloudflare-to-defend-against-ddos-attacks

[35] Cotroneo, Domenico, et al. ”Dependability Assessment of the Android OS Through Fault Injection.” IEEE Transactions on Reliability (2019).

[36] Gawkowski, Piotr, et al. ”LRFI–Fault Injection Tool for Testing Mobile Software.” Emerging Intelligent Technologies in Industry. Springer, Berlin, Heidelberg, 2011. 269-282.

## Appendix

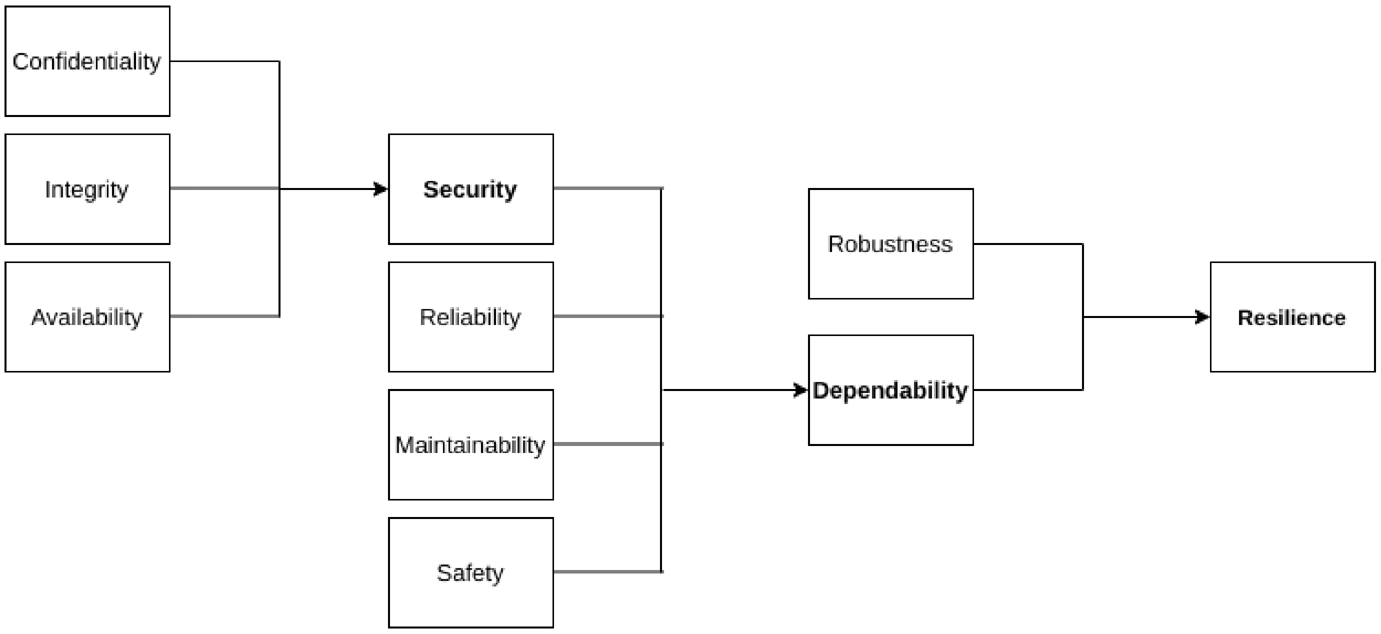

Fig. 2. Concept Hierarchy (adapted from [1])