UNIVERSITÄT
BERN

Cryptographic APIs

Evaluating the Usability of Java Cryptography Architecture

Bachelor Thesis

Sophie Gabriela Pfister
from
Bern, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

31. Dezember 2021

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari, Mohammadreza Hazhirpasand
Software Composition Group

Institut für Informatik
University of Bern, Switzerland

Abstract

Contents

1 Introduction

Cryptography is a fundamental part of our digital world. It provides techniques to ensure confidentiality, authenticity, and integrity of information. Yet, Buchanan described the internet as an unsafe place [3]. He complained that too little security was implemented in the services and protocols used. He insisted that "the next generation of the Internet [...] must be built in a trustworthy way" (Buchanan, 2017, [3], p. 1).
In practice, we witness a large number of vulnerabilities found in various software and protocols each year.[1] One of the disastrous weakness types is the ones concerning cryptography. Although there exist a great number of cryptography libraries for building secure applications by providing services such as hashing, symmetric, and asymmetric encryption, a series of recent studies have indicated that software developers have difficulty using cryptography correctly. Krüger *et al.* found at least one misuse of a cryptography API in 96% of 10'001 investigated Android applications [8]. Piccolboni *et al.* analyzed the 1780 most popular free apps from Google Play Store that make use of cryptography APIs and found that more that 99.7% used inappropriate sources for random number generation and 99.1% used broken hash functions [19].

One of the leading issues is that cryptography libraries lack usability. This issue has been studied in a number of well-known cryptography libraries. The results showed that libraries often do not support auxiliary tasks (*i.e.,* Mindermann *et al.* [10]), that they are not abstract enough (*i.e.,* Nadi *et al.* [12]), and that they lack documentation quality (*i.e.,* Mindermann *et al.* [10], Nadi *et al.* [12], Patnaik *et al.* [18]). Similarly, the results of

[1] http://www.exploit-db.com

Acar *et al.* imply that unusable cryptography libraries do not only prevent developers from writing functional code but also lead to the emergence of security vulnerabilities since developers are more likely to misuse APIs [1]. What's more, a good documentation is a strong predictor for both, functional and secure code. Acar *et al.* emphasized the importance of having an official documentation, which contains secure examples "to keep developers from searching for unvetted, potentially insecure alternatives" (2017, [1], p.167).

We believe that still some areas, such as context of API usability, documentation usability, API misuse and unsafe code, need closer investigations. Therefore, for the three aforementioned areas, we are interested to address the following research questions:

1. What issues do programmers face when implementing symmetric encryption using Java Cryptography Architecture?

2. What security risks can be found in code and advice shared on Stack Overflow referring to the implementation of symmetric encryption scenarios Java Cryptography Architecture?

3. To what extent are these issues linked to missing or inadequate documentation?

Since different cryptography libraries have different API design, it may manifest various and scattered types of issues if we had studied more than one cryptography libraries on Stack Overflow. As a result, we focused on one library (*i.e.,* Java Cryptography Architecture) and one use case (*i.e.,* symmetric encryption) to gain a deeper understanding of the issues, which provides us with more details compared to previous research that focused on a more general level. The Java Cryptography Architecture (JCA) has been examined in recent studies, *i.e.,* Nadi *et al.* [12] and Hazhirpasand *et al.* [6]. Moreover, JCA is the default cryptography API for Java developers and acquired FIPS-140 standards issued by the National Institute of Standards and Technology (NIST) to specify the requirements for cryptography libraries and modules. [2]

To answer our research questions, we analyzed 150 threads from Stack Overflow where at least one issue related to our scope was discussed. We used three queries and computed the sample size per query proportional to the number of posts returned by it. We also selected half of the sample from the newest threads and the other half from the most popular ones. To answer the first research question, we identified the issues the person asking the question (original poster) was facing and categorized them regarding technical aspects or requirements that have not been met. To answer the second research question, we checked the same sample for rule violations based on a predefined set of security rules for symmetric encryption scenarios. To answer the third research question,

[2] FIPS-140-3

we derived a set of prioritized questions from the previous findings and sought their answer in the documentation of JCA library. We also took notes regarding documentation quality in general.

Our key findings were that programmers struggled most with properly handling keys, encryption modes[3], and initialization vectors. This was found regarding both, the first and the second research question. Regarding the first research question, we also observed that developers failed to correctly configure dependencies between different properties. They also struggled with the API design of JCA, especially the dependency from providers and various method overloads. We also found more evidence that programmers lack domain knowledge to successfully implement symmetric encryption.
Regarding the second research question, we found that programmers frequently used unsafe encryption modes (ECB, CBC). The code snippets on Stack Overflow also made use of static values for keys and initialization vectors. Furthermore, password based key derivation was sometimes not implemented in a secure way.
Regarding the third research question, observed that most of the derived questions were covered by the documentation, especially the ones with higher priorities. We concluded that most of the issues could have been prevented if the original poster thoroughly read and understood the documentation. Yet, we issued some recommendations to improve it.

In this thesis, we first present the state of research in chapter 2. In chapter 3, we describe the methodical approaches of our study. We explain and interpret our results in chapter 4. Chapter 5 includes the conclusion, limitations, and conductive thoughts. In the ”Anleitung zum Wissenschaftlichen Arbeiten” (chapter 6), we describe Mayring’s guidelines for qualitative content analysis more precisely that were followed during our analyses.

[3] cipher block mode of operation

2
Related Work

2.1 API Usability

One of the most popular definitions for usability is the one by ISO 92411-11:1998: ”the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use” (ISO 9241-11, [4], p.2). Although the definition is very precise, it does not explain how to measure the usability of a product.

Past research on (API) usability approached the topic in different ways and the literature is therefore rather heterogeneous. Some researchers focused rather on programmers' needs and defined guidelines and heuristics to describe what a usable API should look like. Zibran conducted a meta analysis on API usability literature and described a set of 22 specific guidelines [24]. He considered an API as usable if it was ”(1) easy to learn, (2) easy to remember, (3) easy write client code, (4) easy to interpret client code, and (5) difficult to misuse” (2008, [24], p. 256).
Other researchers approached the problem from the perspective of software metrics. Rama & Kak proposed 8 metrics for API usability referring to method overloads and name confusion, method grouping, parameter list complexity and consistency, thread safety, and documentation [20]. Scheller & Kühn even defined an extensible framework to measure interface complexity automatically [22].
Another approach was to focus on the concept of usability and redefine it more precisely. Alonso-Ríos *et al.* investigated the concept and described it using a detailed taxonomy

[2]. Starting from usability, they organized a wide range of attributes in a hierarchy.

Mosquiera-Rey *et al.* combined several approaches [11]. They extended the usability model by Alonso-Rìos *et al.* with the context of use and mapped existing guidelines for API usability to the model's attributes. The first level attributes of these taxonomies are shown in figure 2.1. They found and described a total of 45 heuristics for API usability.

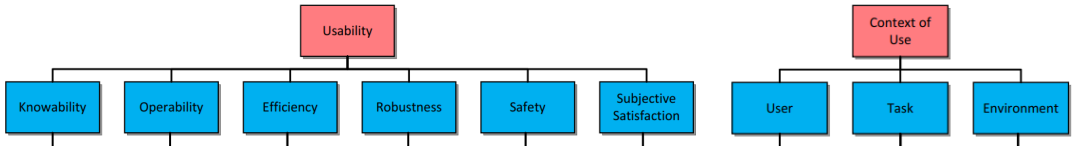


Figure 2.1: First Level Attributes of Usability and Context of Use Taxonomies (Mosquiera-Rey *et al.*, 2018, [11], p. 49 ff.)

2.2 Usability Criteria for Documentation

Mosquiera-Rey *et al.* located documentation quality within the *Knowability* attribute of an API [11]. They defined this property as the extent a programmer can "understand, learn, and remember how to use the system" (Mosquiera-Rey *et al.*, 2018, [11], p. 48). They described three documentation related heuristics:

- Documentation should not contain irrelevant information such as meta data or obsolete and redundant comments.
- Documentation should contain code samples for key scenarios.
- Documentation should identify deprecated methods, explain why these are deprecated, and propose alternatives.

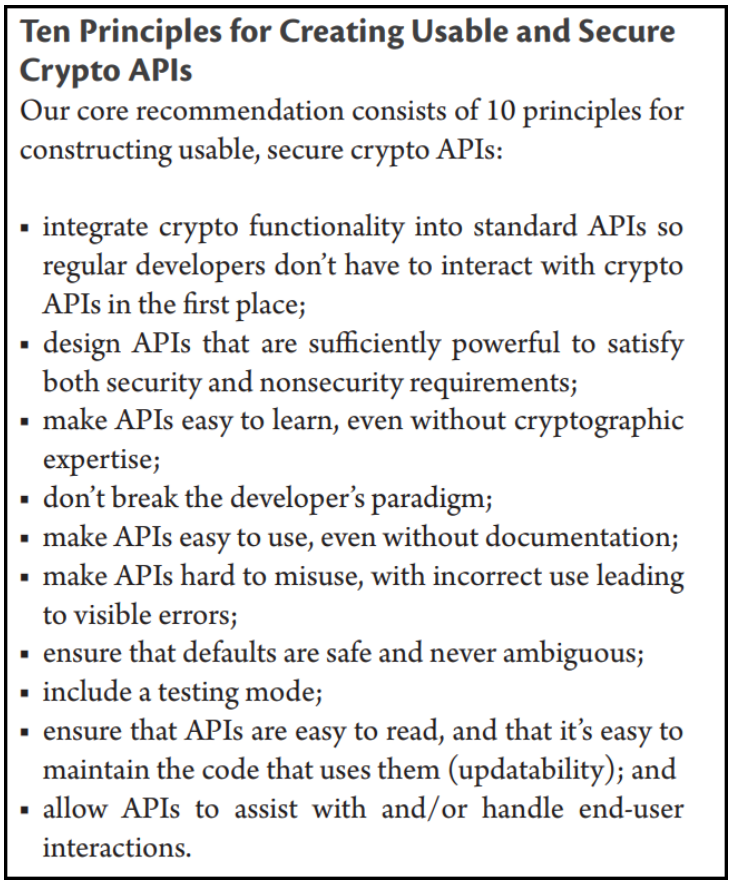
Robillard asked developers what they struggled with most when they had to learn a new API [21]. Their answers identified missing or unclear documentation as a major obstacle. Robillard concluded that API documentation must be complete and provide example code. Additionally, it should support a wide range of usage scenarios, include relevant design elements, and be organized in a convenient way.

Mindermann *et al.* who evaluated the usability of rust cryptography libraries also made recommendations how to improve the usability of such libraries and their documentation [10]. These were more specific for the cryptographic context. A good documentation for a cryptography API should additionally

- link to comprehensible resources that explain cryptographic concepts.
- mention closely related keywords (*i.e.,* block cipher mode of operation, cipher mode, encryption mode).
- describe in which scenarios an algorithm should be used.
- warn from weaknesses and vulnerabilities (*i.e.,* unsafe algorithms that are supported for legacy).
- explain all parameters.
- give advice when there are multiple options and explain the differences between them.

2.3 Usability of Cryptography Libraries

Green & Smith defined ten principles regarding the usability and security of cryptography libraries [5] . Their key idea was that security-related functionalities should be integrated into non-cryptographic APIs such that regular programmers[1] did not have to deal with cryptography APIs at all. The entire set of principles is shown in figure 2.2



Ten Principles for Creating Usable and Secure Crypto APIs

Our core recommendation consists of 10 principles for constructing usable, secure crypto APIs:

- integrate crypto functionality into standard APIs so regular developers don't have to interact with crypto APIs in the first place;
- design APIs that are sufficiently powerful to satisfy both security and nonsecurity requirements;
- make APIs easy to learn, even without cryptographic expertise;
- don't break the developer's paradigm;
- make APIs easy to use, even without documentation;
- make APIs hard to misuse, with incorrect use leading to visible errors;
- ensure that defaults are safe and never ambiguous;
- include a testing mode;
- ensure that APIs are easy to read, and that it's easy to maintain the code that uses them (updatability); and
- allow APIs to assist with and/or handle end-user interactions.

Figure 2.2: 10 Design Principles for Cryptography APIs by Green & Smith (2016, [5], p. 42)

Patnaik *et al.* extended these principles by defining usability smells [18]. They were looking for "telltale signs, that one of the ten usability principles is being violated" (2019, [18], p. 245). They examined a wide range of popular cryptography APIs: OpenSSL, NaCl, libsodium, Bouncy Castle, SJCL, Crypto-JS, and PyCrypto. They manually reviewed almost 2500 posts on Stack Overflow and tried to identify the issues

[1] without cryptography expertise

the programmers were facing. They identified 16 thematic issues of which 2 related to the programmers' lack of knowledge:

- *Passing the buck*: Questions on Stack Overflow that are answered in documentation.
- *Lack of Knowledge*: Questions implying that "the developer does not have foundation level cryptography knowledge" (Patnaik *et al.*, 2019, [18], p. 250).

They mapped the remaining issues to four usability smells:

Whiff	Issue
Need a super-sleuth	Missing Documentation Example code Clarity of documentation
Confusion reigns	Should I use this? How should I use this? Abstraction issues Borrowed mental models
Needs a post-mortem	What's gone wrong here? Unsupported feature API misuse Deprecated feature
Doesn't play well with others	Build issues Compatibility issues Performance issues

Figure 2.3: Usability Smells and Issues as defined by Patnaik *et al.* (2019, [18], p. 254)

Acar *et al.* also evaluated and compared five cryptography libraries for python [1].[2] Their aim was to understand the reasons for failure (or success) when implementing cryptography scenarios and to define a blueprint for a new, more usable cryptography libraries. They conducted a between-subjects online study where python programmers had to implement a symmetric or asymmetric encryption task using an assigned library. Both, task and library, were assigned randomly. The programmers also had to fill out an exit survey where they were asked about their backgrounds as well as their opinion regarding the assigned task and library. Acar *et al.* then examined the submitted code regarding functionality and security and controlled their findings for the participant's background.

[2] cryptography.io, Keyczar, PyNaCl, M2Crypto, and PyCrypto

They found that the strongest predictors for working code was the documentation quality and the availability of working code examples. When it comes to security, the programmers background was most important. Developers with a security background were more likely to produce secure code. Although "simpler" APIs[3] seemed to promote better security results, they did not completely solve security problems. The key issues regarding security were that libraries did not support auxiliary tasks (*i.e.,* key storage) and lacking documentation quality. Acar *et al.* also observed that a complex API with good documentation (*i.e.,* PyCrypto) was rated more usable than a simple API with bad documentation (*i.e.,* Keyczar) by the participants.

Mindermann *et al.* evaluated the usability of Rust cryptography libraries [10]. After determining the most popular libraries, they conducted an exploratory study where one of the authors completed a set of cryptography related task several times using a different library for each round. They afterwards compared two popular libraries, rust-crypto and ring, in a controlled experiment. Students had to complete a code skeleton by adding symmetric encryption logic.
They found that the older "low-level" but more powerful libraries (*i.e.,* rust-openssl, rust-crypto) lacked usability whereas others made a great effort to provide it (*i.e.,* rust-sodium, sodiumoxide). Similarly, high-level libraries use authenticated encryption by default, whereas Mindermann *et al.* considered that this was not advertised enough in low-level libraries. Default values were often avoided, but if present, they were secure. Yet, there were some security risks: Some libraries did not warn about broken algorithms or did not give any warnings when a nonce was accidentally reused. Documentation quality varied between and within the libraries.
Mindermann *et al.* also issued 12 recommendations to remedy present usability issues. These were more specific than the ones by Green & Smith as they only applied to Rust libraries.

Nadi *et al.* investigated the usability of Java cryptography libraries. They wanted to understand the underlying causes for misuse of the related APIs. They also aimed to identify the most common cryptography tasks and possible support tools.
Nadi *et al.* followed several approaches: They manually reviewed 100 posts on Stack Overflow, examined 100 GitHub repositories and conducted two surveys. Regarding the usability of Java cryptography libraries, they found that the biggest obstacles were the lack of documentation (especially code examples), the APIs' design (especially error messages, method overloading, and insecure default values), and lack of domain knowledge among programmers. The participants of the surveys explicitly asked for more abstract APIs and better documentation.

[3] more abstract, secure default values

2.4 Misuse of Cryptography APIs

Krüger *et al.* proposed CrySL, a definition language that allows to specify rules for secure usage of cryptography API [8]. It allows to specify rule sets classwise in separate files. Such file is structured in several sections that are identified by a certain keyword. An example is shown in figure 2.4. There are 6 mandatory sections for each CrySL file:

```
9  SPEC javax.crypto.KeyGenerator
10
11 OBJECTS
12   java.lang.String algorithm;
13   int keySize;
14   javax.crypto.SecretKey key;
15
16 EVENTS
17   g1: getInstance(algorithm);
18   g2: getInstance(algorithm, _);
19   GetInstance := g1 | g2;
20
21   i1: init(keySize);
22   i2: init(keySize, _);
23   i3: init(_);
24   i4: init(_, _);
25   Init := i1 | i2 | i3 | i4;
26
27   GenKey: key = generateKey();
28
29 ORDER
30   GetInstance, Init?, GenKey
31
32 CONSTRAINTS
33   algorithm in {"AES", "Blowfish"};
34   algorithm in {"AES"} => keySize in {128, 192, 256};
35   algorithm in {"Blowfish"} => keySize in {128, 192,
         256, 320, 384, 448};
36
37 ENSURES
38   generatedKey[key, algorithm];
```

Figure 2.4: CrySL Ruleset for JCA's `KeyGenerator` class (Krüger *et al.*, 2017, [8], p. 3)

- SPEC: the class to which the rules apply
- OBJECTS: the list of objects that will be used later in the rule set
- EVENTS: the list of methods that my contribute to the successful usage of the API. Similar methods (*i.e.,* overloads) may be aggregated (*i.e.,* lines 17-19: the `getInstance(...)` method is overloaded).
- ORDER: the order in which the events must be executed.

- CONSTRAINTS: accepted values for the specified object (”internal constraints”). We may also define dependencies (*i.e.,* lines 34 and 35: accepted key sizes depend on the algorithm)
- ENSURES: predicates to govern interactions between different classes.

The CrySL definition language additionally allows to define illegal events in the FORBIDDEN section and kill existing predicates in the NEGATES section.

Krüger *et al.* also implemented CogniCrypt, a compiler that translates CrySL rules into a static analysis which automatically checks a given Java application for rule violations [8]. To evaluate it, they defined a rule set for JCA library and analyzed 10,001 Android apps and compared the results to the findings of a manual review of 50 randomly chosen apps. They found at least one API misuse in 96 % of apps that made use of JCA library. CogniCrypt discovered 19,756 rule violations in 4,071 apps. Most of them referred to broken constraints, especially for the `MessageDigest` class. When Kühne *et al.* found in their manual analysis that some programmers still used MD-5 and SHA-1 hash functions although these are considered broken. CogniCrypt also identified a large number of misuses of the `Cipher` class, especially the use of broken algorithms (DES) and unsafe encryption modes (ECB). Another common misuse was that programmers forgot to clear the password at the end of the lifetime of a `PBEKeySpec` object.

Hazhirpasand *et al.* also used CogniCrypt to analyze 489 open source Java projects that made use of the JCA library [6]. Only 2 of them were considered as completely secure. Figure 2.5 shows the ratio of correct and incorrect usage for each of the investigated APIs. Although a few records were mistakenly marked as misuse according to the authors manual review, their findings showed that programmers especially struggled to use the classes `AlgorithmParmeters`, `SecretKeySpec`, `Signature`, `Cipher`,`KeyStore`, `MessageDigest`, and `IVParameterSpec` correctly. These classes are used to implement (symmetric) encryption, hashing, and digital signatures.
Hazhirpasand *et al.* also contacted 216 maintainers of the repositories to understand the reasons for API misuse. Their answers implied that developers often underestimated the impact of cryptography misuse in publicly accessible code. They were not aware that and were not aware that their publicly accessible code could influence other programmers who were looking for examples. Some maintainers also lacked security knowledge. They did not know how to use the API correctly and accepted security related pull request blindly. Another identified issue was that there were not enough security concerns in the official documentation. Sometimes, programmers also argued that although they use a cryptographic API, the code was not security related.

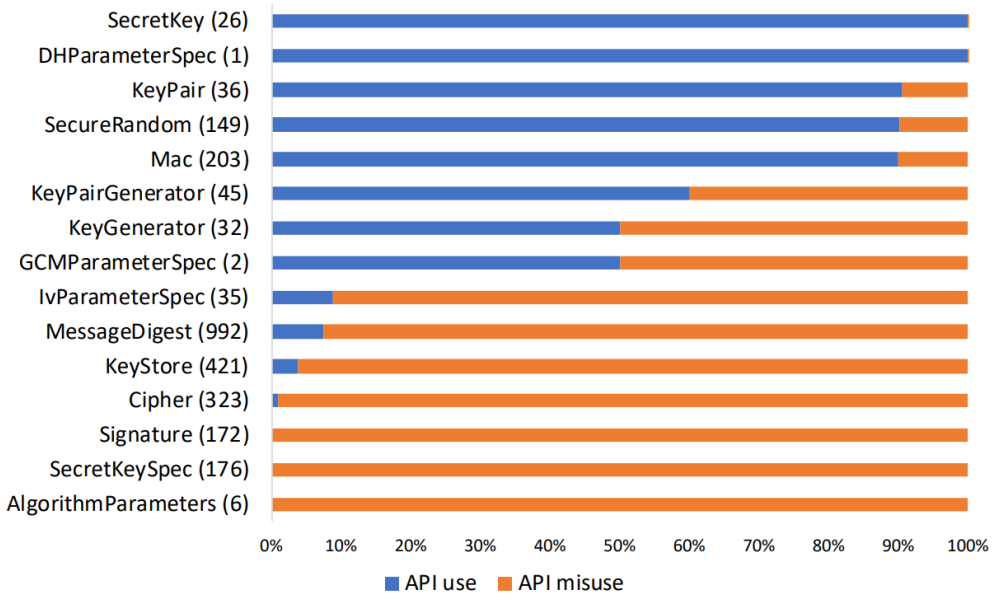


Figure 2.5: Correct API use vs. API Misuse (Hazhirpasand *et al.*, 2020, [6], p. 3)

Piccolboni *et al.* also developed a tool to check security related code for API misuse: CRYLOGGER [19]. Unlike CogniCrypt, CRYLOGGER conducts the analysis dynamically by logging the parameters that are passed to the cryptography APIs during the execution. It later checks their legitimacy using a list of security related rules (see figure 2.7). Piccolboni *et al.* used CRYLOGGER to analyze 1,780 Android apps.

ID	Rule Description	Ref.
R-01	Don't use broken hash functions (`SHA1`, `MD2`, `MD5`, ..)	[8]
R-02	Don't use broken encryption alg. (`RC2`, `DES`, `IDEA` ..)	[8]
R-03	Don't use the operation mode `ECB` with $>$ 1 data block	[5]
R-04 †	Don't use the operation mode `CBC` (client/server scenarios)	[12]
R-05	Don't use a static (= constant) key for encryption	[5]
R-06 †	Don't use a "badly-derived" key for encryption	[5]
R-07	Don't use a static (= constant) initialization vector (IV)	[5]
R-08 †	Don't use a "badly-derived" initialization vector (IV)	[5]
R-09 †	Don't reuse the initialization vector (IV) and key pairs	[46]
R-10	Don't use a static (= constant) salt for key derivation	[5]
R-11 †	Don't use a short salt ($<$ 64 bits) for key derivation	[14]
R-12 †	Don't use the same salt for different purposes	[46]
R-13	Don't use $<$ 1000 iterations for key derivation	[14]

ID	Rule Description	Ref.
R-14 †	Don't use a weak password (score $<$ 3)	[47]
R-15 †	Don't use a NIST-black-listed password	[48]
R-16	Don't reuse a password multiple times	[48]
R-17	Don't use a static (= constant) seed for PRNG	[49]
R-18	Don't use an unsafe PRNG (java.util.Random)	[49]
R-19	Don't use a short key ($<$ 2048 bits) for RSA	[13]
R-20 †	Don't use the textbook (raw) algorithm for RSA	[50]
R-21 †	Don't use the padding `PKCS1-v1.5` for RSA	[51]
R-22	Don't use HTTP URL connections (use HTTPS)	[16]
R-23	Don't use a static (= constant) password for store	[48]
R-24	Don't verify host names in SSL in trivial ways	[16]
R-25	Don't verify certificates in SSL in trivial ways	[16]
R-26	Don't manually change the hostname verifier	[16]

Figure 2.6: CRYLOGGER's Security Rules (Piccolboni *et al.*, 2020, [19], p. 5)

They found that rules 01 and 18 were violated very often ($> 90\%$). This implied that broken hash functions and unsafe sources for random number generation were used very frequently. They also found a rather high prevalence of violations for rules 04, 05, 06,

07, 09 and 22 ($> 30\%$) which refer to unsafe keys and initialization vectors, the reuse of key-IV pairs and the use of HTTP protocol.

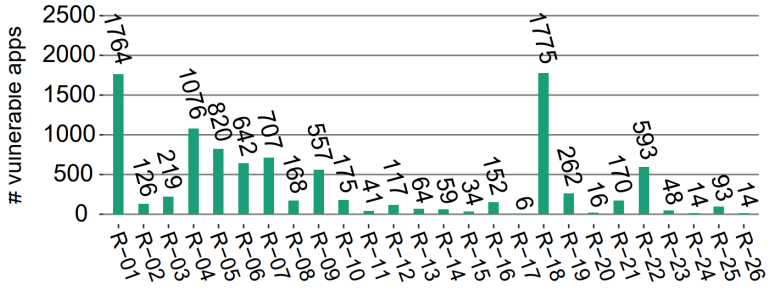


Figure 2.7: Rule Violations Detected by CRYLOGGER ($N = 1,780$) (Piccolboni *et al.*, 2020, [19], p. 12)

3 Methodology

To answer our research questions, we first had to identify the issues programmers face when implementing symmetric encryption using JCA. We derived them by analyzing 150 threads on Stack Overflow which is one of the most popular Q&A forums for programmers. We also scanned these threads regarding security risks. We then reprocessed the results to define a set of questions that should be answered in the documentation. Finally, we checked the documentation to see whether these questions are answered or not.

As we required several methodical approaches, this chapter is divided into five sections. The first section describes the sampling process. The second section refers to identifying issues from Stack Overflow posts. The third section explains how we checked the threads for security issues. The fourth and last section is about the analysis of the library's documentation.

3.1 Sampling

In JCA, symmetric encryption is implemented using the `Cipher` class. It supports a wide range of symmetric and asymmetric encryption algorithms. To search for suitable threads on Stack Overflow, we first defined a set of queries. We use the `[java]` tag combined with a minimal `Cipher.getInstance()` statement for each symmetric algorithm. This statement must be executed in all encryption scenarios using `Cipher` class.
As some of the symmetric algorithms supported by JCA are not very popular, the corresponding queries returned only a small amount of posts. We decided to exclude these algorithms and focused on the three most popular symmetric encryption algorithms: AES, DESede and DES.

Next, we calculated the sampling size using the sample size calculator by SurveyMonkey. To ensure a confidence level of 95% and a margin of error below 8%, we required 150 posts. Then, we computed sample size per query proportionally to the number of posts a it returns. The result can be found in table 3.1.

Query	#Posts	#P/T * 150	N
[java] Cipher.getInstance(AES)	3233	126.7	**126**
[java] Cipher.getInstance(DESede)	295	11.6	**12**
[java] Cipher.getInstance(DES) --DESede	300	11.8	**12**

Table 3.1: Computation of Sample Sizes

Finally, we selected the threads. We picked half of the sample from the newest[1] threads and the other half from the most popular[2] ones. This approach attempts to balance the ambition to detect present issues with the fact that most programmers first search for answers on Stack Overflow (generating views) before posting a question.
As the aim of the analysis was to reveal issues referring to the implementation of *symmetric encryption* using the *JCA* library, we excluded all posts that did not refer to this scope. We also did not include any posts of lacking quality. The complete list of reasons for exclusion can be found in appendix 8.

[1] based on the creation date of the question
[2] based on the view count

3.2 Analysis of Issues

The goal of the first analysis was to answer the first research question: *What issues do programmers face when implementing symmetric encryption using Java Cryptography Architecture?* We followed the guidelines for qualitative content analysis by Mayring [9]. They are a standard in humanities to evaluate text-based data and are especially well suited for method-integrative approaches that combine qualitative and quantitative elements. They allow to evaluate large amounts of material and quantify individual analysis steps - which is exactly what we need to answer our first research question. They also provide a set of criteria to evaluate the validity and reliability of a methodical approach.
In this section, we will only provide the information to understand the process. You can find the complete processing scheme in the ”Anleitung zu Wissenschaftlichen Arbeiten”, chapter 6.

We conducted the analysis in three rounds. We first applied *summarizing* to extract the relevant information (issues and questions) from the threads. Then we classified the issues in two rounds.

3.2.1 Summarizing

The goal of this first analysis step was to extract all relevant information and record it such that we did not have to re-read the entire threads during further evaluation. We considered the question post, posts marked as ”accepted answers” as well as comments to these posts. If there was not any accepted answer, we considered all posts and comments and tried to display the discussion in our records.
We conducted summarizing first on our own and discussed the results afterwards to create a consistent and more objective list of records. In particular, we eliminated records that did not refer to our scope. As an example, we excluded all issues that referred to the conversion of plain text or the cipher text as data conversion (or character encoding) is not only required in the context of cryptography.

For each thread, we recorded the set of issues and questions that the original poster[3] was facing. Then we tried to identify the reasons and solutions or answers. We aimed for a short issue description (e.g. an error message or a shortened form of a question) and more precise explanations for the reasons and solutions. To stay as near to the material as possible, we copy-pasted words, sentences or even paragraphs.

[3] author of the question post

3.2.2 Classification

As already mentioned, we classified the threads in two rounds regarding two different kind of aspects. We first classified all records regarding technical aspects and then classified the records regarding requirements the original poster was not able to meet. In each round, we assigned at most one category to a record. We also wanted to assign at least one category to each record. So in the end, a record should have one or two assigned categories.

Technical Aspects

In the first round of classification, we focused on the technical aspects of implementing symmetric encryption. As categories, we defined a set of tasks that programmers must take care of when implementing symmetric encryption using JCA. If they implement all tasks correctly, the code compiles and runs without rising an error and leads to the expected result. So if programmers ask questions on Stack Overflow about a technical aspect, they either implemented a task incorrectly or they have a question regarding one of these tasks. In this classification we asked "What implementation step was performed incorrectly causing the error?" or "What implementation step is targeted by the question?".

Identifying a set of main tasks was simple: *Cipher Object Instantiation, Generating Algorithm Parameters, Cipher Object Initialization, Transformation, and Transmitting Algorithm Parameters*. But as we wanted to have more insight, we also defined subcategories. To do so, we consulted the "Java Cryptography Architecture (JCA) Reference Guide" [15] which provides code examples as well as explications regarding the use of the different APIs. To allow an unambiguous category assignment, we also defined the dependencies as categories. As a result, the following technical categories and subcategories were derived:

- *Cipher Object Instantiation*: We assigned this category to all issues and questions referring to an inappropriate `Cipher.getInstance(...)` statement. As parameter, programmers must pass a transformation string consisting of:
  - **Algorithm** (mandatory)
  - **Encryption Mode** (optional)
  - **Padding Mode** (optional)

  Additionally, we defined the following subcategories:
  - **Dependency Encryption Mode - Padding**: The encryption modes determines whether padding is required or not. We assigned this category to all issues caused by an inappropriate specification of these two properties.

  - **Cipher Object Instantiation - Other** for issues and questions related to `Cipher` object instantiation but not any of the aforementioned aspects.

- *Generating Algorithm Parameters*: Depending on the specification of the cipher object, it requires different kind of parameters. For encryption, the programmer might need to perform the following tasks:

  - **Key Derivation** for issues and questions referring to random key generation, password based key derivation or key exchange protocols.
  - **Initialization Vector / Nonce Generation** for issues and questions referring to the generation of the IV or nonce used for the transformation.
  - **Generation of Other Algorithm Parameters**, e.g. a `GCMParameterSpec` object which contains additional parameters for GCM encryption mode.

- *Cipher Object Initialization*: We assigned this category to all issues caused by the misuse of the `init(...)` statement, e.g. not passing all required parameters. We defined the following subcategories:

  - **Dependency Algorithm - Key**: The algorithm determines what data type the key must be stored in. It also defines the allowed key sizes. We assigned this category to issues caused by passing an inappropriate key to the `init(...)` method or questions about this dependency.
  - **Dependency Algorithm & Encryption Mode - IV**: The encryption mode determines, whether an IV is required or not. For some encryption modes (e.g. "CBC") the IV must be the same size as the algorithms block size. As an example, an issue related to passing an IV of the wrong size to the `init(...)` method was assigned to this category.
  - **Cipher Object Initialization - Other**

- ***Transformation***: This category was assigned to all issues and questions targeting the actual transformation methods `update(...)` and `doFinal(...)`, e.g. passing the wrong input parameters or questions about the output.

- *Transmission of Parameters*: As all parameters from encryption must be reused for decryption, they must either be stored or transmitted. This category was assigned to all issues and questions referring to storing, restoring, or transmitting parameters. We defined the following subcategories:

  - **Key Transmission**: The key must be kept secret.
  - **Transmission of Other Parameters** such as the initialization vector. They can be transmitted along the cipher text as they do not have to remain secret.

- **Dependency Encryptor - Decryptor**: The `Cipher` objects used for encryption and decryption must be specified and initiated in the exact same way except for the parameter specifying the operation in the `init(...)` statement. We assigned this category to all issues caused by differing configurations.

You can find a list with concrete examples for each category in the appendix.

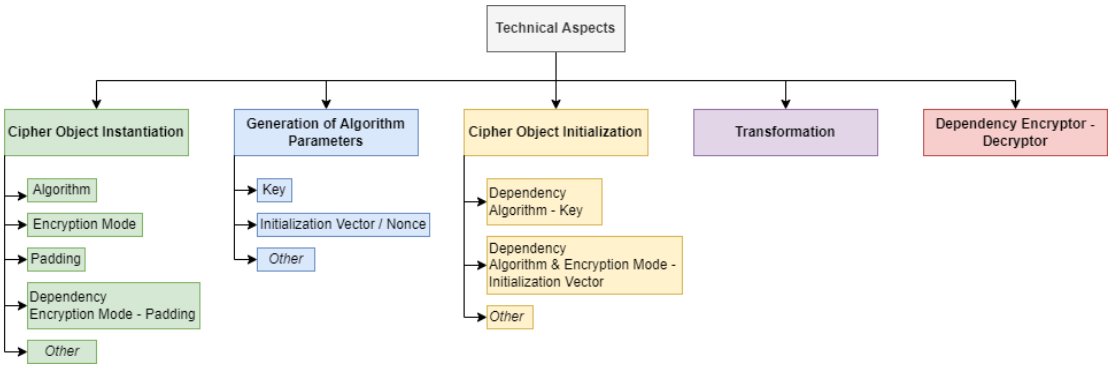


Figure 3.1: Hierarchy of Technical Aspects Categories

Requirements

As the first set of categories only targeted the implementation of symmetric encryption, we defined a second set of categories regarding the design of an application. We defined different kind of functional and non-functional requirements as categories. We consulted Sommerville [23] as a theoretical basis. During the analysis we asked ourselves ”Which requirements are not met?”. Not all requirements defined by Sommerville occurred in our analysis. We only assigned the following categories:

- **Functional Requirements** to issues where a programmer was not able to fulfill a certain requirement or tried to use encryption for an unsuitable use case.
- **Performance** to issues where an implementation was not as time-efficient as required.
- **Space** to code leading to an `OutOfMemoryException`.
- **Reliability** to situations where the implementation crashed frequently.
- **Portability** to implementations that behaved differently on different (Java) plattforms.

- **Interoperability** to issues where a developer was not able to decrypt a cipher text that was produced using another library or vice versa.
- **Security** to implementations containing security risks.

As we analyzed the discussion, we only assigned a category if either the original poster complained about not being able to meet a certain requirement or someone gave a hint. We conducted a broader examination of security risks in our second analysis (see chapter 3.3).

3.2.3 Evaluation

We conducted several forms of frequency analysis' on our data. We computed the absolute and relative frequencies for each category and subcategory. Finally, we also took a look at the combination of technical aspects and requirements categories. In our interpretation, we focused on the most occurring categories as these refer to more prevalent issues.
ToDo

3.3 Analysis of Security Risks

The goal of this second analysis was to answer the second research question: *What are common security risks in code and advice shared on Stack Overflow referring to the implementation of symmetric encryption scenarios Java Cryptography Architecture?*
We first defined a set of security rules regarding the implementation of symmetric encryption. Then we reprocessed the original sample checking them for rule violations.

3.3.1 Security Rules

We derived our rules from the rule sets used for CRYLOGGER tool (see Piccolboni *et al.* [19]) and the CrySL based compiler for Java applications by Krüger *et al.* [7]. We only considered the rules that were applicable to symmetric encryption and structured them using the categories from technical aspect's classification (see section 3.2). We also abstracted the context of the rules to facilitate their evaluation. As an example, R-04 of CRYLOGGER says " Don't use the operation mode CBC (client/server scenarios)" (Piccolboni *et al.*, 2020, [19], p.5). We abstracted the context and defined that a programmer should not use CBC at all. We often do not know in what context the original poster wanted to use the code. Additionally, if we think about someone naively copy-pasting code from Stack Overflow, the use of CBC is a potential security risk.
The resulting rules can be found in table 3.2,

3.3.2 Tracking Security Rule Violations

We reprocessed the original sample marking for each rule whether it was violated or not. We only considered the question post, the accepted answer post, and comments to one of these. We also distinguished between ”question” and ”answer” as well as ”code” and ”text”. We analyzed the four aspects independently and made a list for each one of them:

- **Question Code** to track security risks in code snippets of the question post
- **Question Text** to track security risks in the text of the question post as well as comments to it by the original poster
- **Answer Code** to track security risks in code snippets of the answer post
- **Answer Text** to track security risks in the text of the answer post as well as any comment by another person.

While analyzing the code snippets, we focused on the parts where encryption, decryption, key derivation, IV generation, and key storage were implemented. As an example, if someone is defining a static key in the main method and passes it to the encryption section as a parameter, we did not consider this a security risk. The encryption section can still be safe if an appropriately derived key is passed.
The answers were only analyzed if there was an accepted one.

3.3.3 Evaluation

We conducted a frequency analysis for each list. Our aim was to find out the most prevalent security risks to answer the research question. ToDo

3.4 Analysis of Documentation

The results from the precedent analyses formed the basis to analyze the documentation for JCA. As it spreads over several documents and sources, we only examined the most basic ones: the Java Cryptography Architecture (JCA) Reference Guide [15], the Java Security Standard Algorithm Name Specification [16], and the entire API documentation for `javax.crypto` package starting from the package overview [14]. These three documents are valid for all providers and therefore apply to a wide range of platforms.

To analyze the documentation, we first defined a set of question that should be answered. Afterwards, we sought the answers in the documentation. Our aim was to

Rule ID	*Rule - Cipher Object Instantiation*
R-01	Use AES or Blowfish algorithm.
R-02	Do not use ECB or CBC encryption mode.
Rule ID	***Rule - Generating Algorithm Parameters***
R-03	***Rules for Key Derivation***
R-03-a	Do not use a static (= constant) key.
R-03-b	Do not use static salt for key derivation.
R-03-c	Use at least 64 bits of salt for key derivation.
R-03-d	Use at least 1000 iterations for key derivation.
R-03-e	Do not use a weak password (score < 3)
R-03-f	Do not reuse passwords multiple times.
R-04	***Rules for IV / Nonce Generation***
R-04-a	Do not use a static (= constant) IV.
R-04-b	Do not use a static seed for IV generation.
R-04-c	Use SecureRandom for IV generation.
Rule ID	***Rule - Cipher Object Initailization***
R-05	Do not reuse the same key-IV pair.
Rule ID	***Rule - Parameter Transmission***
R-06	Do not use a static (= constant) password for store.

Table 3.2: Security Rules

answer the third research question: *To what extent are these issues*[4] *linked to missing or inadequate documentation?* The questions additionally gave us more insight in what programmers are struggling with. They helped us to answer then first research question more precisely.

3.4.1 Deriving Questions

As an API documentation should support the usage of the API and not educate the programmer, basic questions regarding cryptography should not be answered in it. However, a cryptography API should link reliable sources (*i.e.,* Mindermann *et al.* [10]). We therefore created two lists of prioritized questions: one with ”questions to documentation” and one containing ”general questions”.

We derived the questions from the results of the former analyses. We reprocessed

[4]obstacles in implementation, security risks in code

the records from the first analysis and tried to formulate a question for each one. If the question was new, we wrote it down and set its priority to one. If there was already a similar question, we increased its priority by one and sometimes reformulated the question.
The set of questions derived from the first analysis' results covered already a wide range of security related topics. We therefore did not have to add more questions based on the second analysis' findings. However, we adapted the priorities of these questions, setting them to the actual number of threads that included a related security risk.

3.4.2 Consulting Documentation

For each question on the list, we then tried to find an answer in the documentation. Depending on the question, we checked the different sources in another order. We aimed to find answers as efficiently as possible.
For "questions to documentation", we typically started with the reference guide to find general explanations and then consulted the related parts of the API documentation. For "general questions" we started in the standard algorithm name specification. Of course, we knew the documentation better after answering a set of questions and therefore optimized our search strategies.
Once we found an answer to the question, we recorded its source as well as some remarks regarding documentation quality.

3.4.3 Evaluation

We first wanted to find out, what percentage of questions remained unanswered. This gave us a starting point to answer the research question. If most questions are answered in a clear way, we cannot blame the documentation for the issues programmers face during implementation. We also cannot blame it for unsafe code if it contains enough hints regarding security.
In the further evaluation we focused on the unanswered questions and our comments regarding documentation quality to make suggestions regarding the improvement of the documentation.
We also had a closer look at the questions as they gave us more insight regarding the first research question.

4 Results and Interpretations

In this chapter, we discuss the results of our three analyses and their interpretation. There is a section for each of the three conducted analyses: The first section presents the issues programmers faced during the implementation of a symmetric encryption scenario using JCA library. The second section discusses the security risks we found in our sample data. The third section explains how the previous findings are linked to the documentation.

4.1 Implementation Issues

In our first analysis, we tried to identify all issues the original posters were facing and recorded them separately. Depending on what the original poster was struggling with, we afterwards classified the issues as relating to a *technical aspect* and/or a *requirement*. In total, we recorded 219 issues. We classified 197 (90%) of these records regarding technical aspects and 76 (35%) regarding requirements. 62 records were classified twice. We could not classify only one thread (and its relating record) to the lack of adequate information.

As shown in figure 4.1, the most common categories in the first round of categorization were *Generation of Algorithm Parameters* (50) and *Cipher Object Instantiation* (53). Both of these categories refer to specifying and generating the properties that are used during encryption or decryption. During the generation of algorithm parameters, programmers might derive a key, an initialization vector, and other algorithm parameters

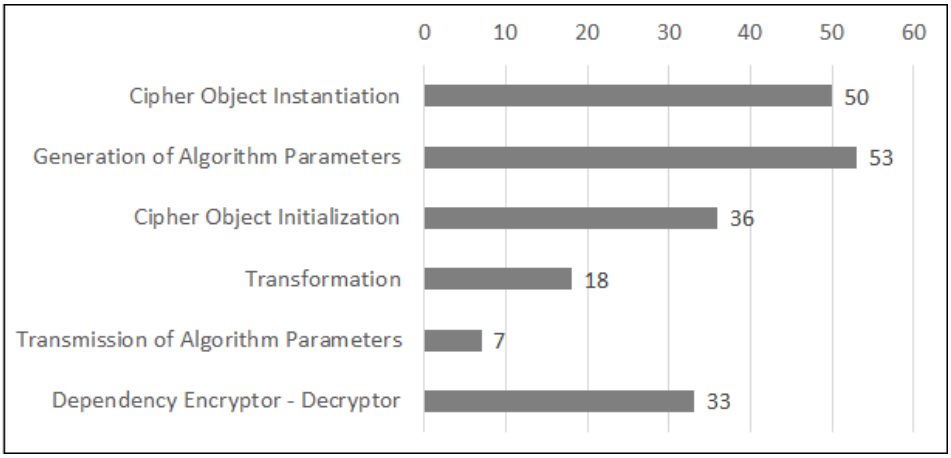


Figure 4.1: Number of Issues Assigned to a Technical Aspects Main Category ($N = 219$)

such as advanced authentication data. The original posters especially struggled with key derivation (36 records - 16.4%). This was not surprising as previous work has already revealed that programmers struggled with key handling. The set of issues referring to the key[1] made up more than $\frac{1}{5}$ of all issues.
During the instantiation of a `Cipher` object, programmers specify the algorithm, the encryption mode, and padding. The original posters were especially struggling with the latter two. 18 records referred to the encryption mode, 11 to padding and another 11 to the dependency of these properties

The third most common category was *Cipher Object Initialization* (36). In this implementation step, the generated parameters (*i.e.,* key, IV,...) are passed to the cipher object. Most issues were assigned to the *other* subcategory. They often refer to the original posters not passing all required parameters of the `init(...)`.
The fourth most common category was *Dependency Encryptor - Decryptor* (33). More than 27% of all issues referred to a dependency related subcategory. The high prevalence of issues referring to the dependency of encryptor and decryptor implies that people lack knowledge about (symmetric) encryption in general.
The fact that the cipher objects for encryption and decryption must use the exact same algorithms and parameters is *the* basic principle of symmetric encryption.

The other main categories and subcategories were not assigned that often. Some original posters were confused that there were 2 methods that perform *Transformation*: `update(...)` and `doFinal(...)`. They did not know which method must be called in their scenario.

[1] derivation, transmission, dependency from algorithm

Among issues referring to the *Transmission of Algorithm Parameters* (7), most referred to the key (5). The remaining records were related to the initialization vector (1) and the salt used for password based key derivation (1).

You can find the complete list of subcategories and their frequencies of assignment in table 4.1.

Subcategory	%	#	(Main Category)
Algorithm	3.20%	7	
Encryption Mode	8.22%	18	
Padding	5.02%	11	(Cipher Object Instantiation) (50)
Dependency Encryption Mode - Padding	5.02%	11	
Cipher Object Instantiation - Other	1.37%	3	
Key Derivation	16.44%	36	
IV / Nonce Generation	6.39%	14	(Generation of Algorithm Parameters) (53)
Generation of Other Algorithm Parameters	1.37%	3	
Dependency Algorithm - Key	3.65%	8	
Dependency Algorithm/Encryption Mode - IV	3.65%	8	(Cipher Object Instantiation) (36)
Cipher Object Initialization - Other	9.13%	20	
Transformation	8.22%	18	(Transformation) (18)
Key Transmission	2.28%	5	(Transmission of Algorithm Parameters) (7)
Transmission of Other Algorithm Parameters	0.91%	2	
Dependency Encryptor - Decryptor	15.07%	33	(Dependency Encryptor-Decryptor) (33)

Table 4.1: Number of Issues Assigned to a Technical Aspect Subcategory ($N = 219$)

Within the categories referring to requirements, *Security* was the dominating category (46 records - 21%). This seems natural as this is what cryptography is all about. However, original posters rarely asked about the security of their implementation. Most records related to security hints that we identified in the answer or comment body. Yet, there is massive under-reporting as our results from the security related analysis show (see section 4.2).

As shown in figure 4.2, the other requirement categories occurred less often. There were 12 issues related to the *Portability* of an application. They often refer to original posters not specifying all values themselves. As an example, several programmers only passed ”AES” to the `Cipher.getInstance(...)` method. In that case, default values are used for encryption mode and padding. These values however differ between different providers and therefore between different platforms.
Another 7 records were assigned to *Interoperability* category. These issues occurred if original posters implemented one task in different ways in both source codes. Sometimes, the other library was more abstract and used default values (*i.e.,* for padding) that were therefore not visible in the source code. As a result, the original posters instantiated or initialized the Java `Cipher` objects incorrectly. Some of these default values were also not

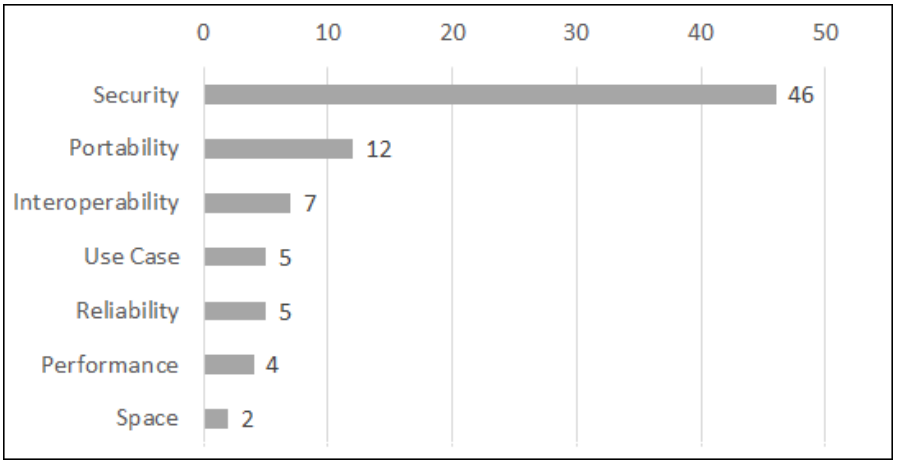


Figure 4.2: Number of Issues Assigned to a Requirement Category ($N = 219$)

supported by JCA (*i.e.,* ZeroPadding). Some programmers also used non-standardized functions (*i.e.,* SHA1PRNG for random number generation).
The 5 issues referring to *Use Case* category related to the misuse of encryption for an inappropriate use case (2), to a use case that was not supported by JCA library (2) and a use case that is just not possible to implement (1, mapping 16 B of data bijectively to a 12 digit number).
Most of the issues that were assigned to the *Reliability* category were caused by the original poster declaring `Cipher` objects statically in global space. The application crashed frequently because `Cipher` objects are not thread-safe.
Some programmers complained that the execution of `getInstance(...)` method was taking too long (lacking *Performance*). One original poster also observed that the encryption of a large file was time consuming. Other developers reported that an `OutOfMemoryException` occurred when they tried to encrypt a large file at once (lacking *Space* efficiency).

As previously mentioned, some records were classified twice, regarding technical aspects and requirements. The heat map in figure 4.3 shows the relative overlapping of categories. We observed the highest overlapping of categories for *Generation of Algorithm Parameters* and *Security*. 21% of all records assigned to *Generation of Algorithm Parameters* or *Security* category were assigned to both categories. Most of them referred to static values being used for key or initialization vectors. The second highest overlapping was between *Cipher Object Instantiation* and *Security* categories (16%). Almost all of them referred to the use of an unsafe encryption mode.
As we had less data for the other requirements categories, it is not surprising that we found much less overlapping for these. The issues that were assigned to the *Portability* or

Interoperability category and some other technical aspects category were mostly due to default values that vary between platforms and libraries. The overlapping with *Reliability* category indicated where the applications crashed: during *Cipher Object Initialization* or *Transformation*. The bi-classification of *Cipher Object Instantiation* and *Performance* implied that the execution of this function is particularly time consuming.

	Performance	Reliability	Portability	Interoperability	Security
Cipher Object Instantiation	0.06	0.00	0.08	0.04	0.16
Generation of Algorithm Parameters	0.00	0.00	0.05	0.05	0.21
Cipher Object Initialization	0.00	0.02	0.04	0.02	0.05
Transformation	0.00	0.09	0.00	0.00	0.00

Figure 4.3: Relative Overlapping of Technical Aspects and Requirements Categories

4.2 Security Risks

We manually checked 150 question, 84 answer posts, and related comments to find any violations against our rule set. We found a total of 331 security risks. Most of them (249 - 75%) stem from code snippets in question posts. The text of question posts included only 38 security risks. However, we observed that the questions commonly did not contain much text.
We found 35 rule violations in answers' code snippets and 9 in answers' text. We observed that some answers just fixed the functionality of a question related code section without improving its security. The resulting code therefore "inherited" the security risks from the question. Another common observation was that people correctly gave the advice that ECB was not considered safe. However, they suggested using CBC instead which must not be used in client-server scenarios. Such advice is therefore not safe, especially if we do not know in what kind of application the code is used.

Rule		Q_Code	Q_Text	A_Code	A_Text	Total
R-01	Use AES or Blowfish algorithm.	24	11	0	0	**35**
R-02	Do not use ECB or CBC encryption mode.	113	20	20	6	**159**
R-03-a	Do not use a static (= constant) key.	43	2	3	2	**50**
R-03-b	Do not use static salt for key derivation.	7	1	1	0	**9**
R-03-c	Do not use short salt for key derivation.	3	0	1	0	**4**
R-03-d	Do not use < 1000 iterations for key derivation.	6	0	1	0	**7**
R-03-e	Do not use a weak password (score < 3)	10	0	1	0	**11**
R-03-f	Do not reuse passwords multiple times.	3	0	1	0	**4**
R-04-a	Do not use a static (= constant) IV.	24	2	5	1	**32**
R-04-b	Do not use a static seed for IV generation.	1	0	0	0	**1**
R-04-c	Use SecureRandom for IV generation.	0	0	0	0	**0**
R-05	Do not reuse the same key-IV pair.	12	2	2	0	**16**
R-06	Do not use a static (= constant) password for store.	3	0	0	0	**3**

Table 4.2: Rule Violations per Check List

As we did not find much data to analyze for the other check lists, we focused on the evaluation of questions' code in the further analysis. On average, each question post contained 1.66 security risks in its code snippets. We observed that the average for the most popular (and older) posts (1.91) is slightly higher than the one for the newest posts (1.43). In total, there were 24 question posts did not contain any security risk in their code snippets.
The most often violated rule was *R-02: Do not use ECB or CBC encryption mode*. In more than 75% of question posts, the original poster used one of these unsafe block cipher modes. This is also due to ECB being the default encryption mode for most providers.

The second and third most violated rules were *R-03-a: Do not use a static (= constant) key.*, *R-04-a: Do not use a static (= constant) IV.* and *R-01: Use AES or Blowfish algorithm*. The number of posts using an unsafe algorithms is due to sample: We included 24 posts where DES or 3DES was used. Some original posters stated that they used static values only for Stack Overflow to simplify their code. Nevertheless, this is a potential security risk if we think of a programmer naively copy-pasting the code snippet. If an original poster used both, static key and IV, this led to the reuse of key-IV pairs (*R-05*) which is the fifth most often violated rule.
The remaining rules were rarely violated. However,total number of rule violations referring to password based key derivation *R-03-b to R-03-f* is 29. 14 posts used an unsafe key derivation procedure.
The least violated rules were R-04-c: Use SecureRandom for IV generation, R-04-b: Do not use a static seed for IV generation, and R-06: Do not use a static (=constant) password for store. As most original posters used ECB, which does not require an IV, and many used a static initialization vector otherwise, there were not many code sections showing IV generation. There were also hardly any question posts that showed how the key was stored.

4.3 Documentation

Based on the previous findings, we derived a total of 64 questions: 43 referring to documentation and 21 more general ones. After checking the documentation, 10 questions remained unanswered and for 15 questions, we considered the answer as incomplete, unclear, or even misleading.

4.3.1 Questions

Our first observation regarding the questions was that there are twice as many JCA-specific questions as general ones. Yet, the total priority of all ”general” questions was much higher than the total priority of the specific ones even before correcting the priority for the security relevant ones. This strongly indicates that programmers asking questions on Stack Overflow do not only lack knowledge about the JCA library but also about cryptography in general.

The results from the first analysis gave us insight into what tasks and requirements programmers are struggling with. The questions helped us to detect API related issues. Several questions targeted a specific platform or were related to providers, two of them were even among the three most prioritized questions to documentation. As an example, the default values and behavior of a `Cipher` object depends on the provider. However,

whether a provider is available or not and which provider is used by default depends on the platform.
Another 5 questions were related to method overloads. There are two methods that perform the data transformation and both of them are overloaded. The methods for instantiation and initialization are overloaded as well.

Question	Priority
What happens if I do not specify the IV although it is required?	9
How can I derive a key from a password?	7
What is the default value if I do not specify padding?	6
What kind of parameters do I have to pass to the decryption methods (update / doFinal)?	6
Which of the provided key derivation functions are standardized?	6
Are there any external dependencies for the implementation of AES-256?	4
What parameters does a Cipher object require for initialization?	4
When do I have to call update(), when do I have to call doFinal()?	4
Are Cipher objects thread safe?	4

Table 4.3: Top 9 JCA-Specific Questions

4.3.2 Missing and Unclear Answers

We were not able to find and answer to 3 JCA-specific and 7 general questions. They all had a rather low priority (¡ 4). The higher prevalence of unanswered general questions implies that JCA documentation does not provide (enough) links to trusted resources for general information about cryptography.
The following three unanswered questions represent documentation related issues:

- *How to specify PKCS#7 padding in Java?* PKCS#7 is a standardized padding for arbitrary block sizes that is supported by many cryptography libraries. JCA internally interprets PKCS#5 padding as PKCS#7 if it is required. However, this is not mentioned in the Standard Algorithm Name Specification nor in any other document that we examined. This might complicate the implementation of interoperability scenarios.
- *What properties does AES-256 require?* There is a link to the official specifications for most algorithms in the Standard Algorithm Name Specification. However, for AES there is not. What's more, specifications are hard to understand if one has not the required background. This is problematic since AES is one of only two symmetric encryption algorithms that are recommended.
- *Which symmetric encryption algorithms are safe to use?* JCA documentation does not provide enough hints regarding security.

For 13 JCA specific questions and 2 more general ones, we considered the answers as not clear enough. They often referred to method overloads. The documentation did not point out the differences clearly enough. It often seemed like copy pasted.
Another common issue was that there is not any code example for some key scenarios (e.g using a `KeyStore` or password based key derivation using PBKDF2). The available ones often did not work due to some missing parts: The code example for encryption did not show how the algorithm parameters were generated.

4.3.3 Recommendations

5 Conclusion and Future Work

In this chapter, we first aim to answer our research questions based on our finding from the previous chapter. We then discuss possible limitations of our study and suggest further research topics.

RQ 1: *What issues do programmers face when implementing symmetric encryption using Java Cryptography Architecture?* Our study provided answers to this question from different perspectives:
If we think of *tasks* that programmers are struggling with, most of them faced problems with generating algorithm parameters, especially with deriving the key from a password. The second most problematic task was instantiating a `Cipher` object. The original posters particularly failed to specify encryption mode and padding correctly. The third most problematic task was initializing the `Cipher` object. Most issues and questions related to this tasks referred to the programmer not passing all required parameters to the `init(...)` method.
Another aspect was the *design* of JCA library. One of the major issues in this context was that default behavior and values depended on the provider. The platform however determines which providers are available and what provider is chosen by default. This decreases the portability of applications. Another issue was that several key methods are overloaded, particularly `getInstance(...)` and `init(...)`. Additionally, there are two methods that perform transformation, `update(...)` and `doFinal(...)` which are both overloaded as well. Some original posters failed to chose the correct one.
This could also be related to the *documentation*. The method overloads are documented too similarly and there is not any advice which overload should be used in what scenario.

Also, there are not enough working code examples and it does not link to trusted, understandable resources for more general information about cryptography.
When we think of the *programmers*, our study added more evidence confirming the assumption that they lacked knowledge about cryptography. However, we would like to extend it: As the scope of our study was quiet limited, we excluded all threads that did not target it. To collect our sample, we had to exclude 296 threads. For each thread that (partly) belonged to our scope, there were two threads that did not. Of these excluded threads, 138 (ca. 47%) did either not refer to cryptography at all or they did not refer to the JCA library. The large amount of threads containing non-JCA-related issues let us conclude that programmers do also lack knowledge about computer science in general. We therefore should be aware that cryptography scenarios are implemented by programmers of all kind of backgrounds when we design a cryptography library.

RQ 2: *What are common security risks in code and advice shared on Stack Overflow referring to the implementation of symmetric encryption scenarios using JCA library?* In general, we found that security risks were particularly present in code snippets from the question body.

The most common security risk was the use of an unsafe encryption mode. This is also related to JCA using ECB as default block cipher mode of operation.
Other common security risks were the use of static values for either key, initialization vector, or both. Although some people explicitly stated that they only use them in their post on Stack Overflow to simplify the code section, it would become a security risk if someone just copy-pasted it. The procedures used for password based key derivation also contained security risks.
We however observed a slight improvement when we compared the older posts with the newer ones.

RQ 3: *To what extent are these issues linked to missing or inadequate documentation?* This research question is the most difficult one to answer. Most questions (¿60%) were clearly answered in the documentation, especially the higher prioritized ones. We found related information for almost 85%. We therefore cannot blame insufficient documentation for the struggles of developers. There are at least two other explanations: Either, the programmers who asked these questions did not consult the documentation at all or they did not understand it.
Yet, the documentation could be improved. It should provide more (working) code examples, security hints, and links to trusted resources for more information about cryptography. In the API documentation, the method overloads should be documented more specifically.

5.1 Limitations and Future Work

As this project followed a qualitative approach and performed an in-depth analysis, it is reasonable to have a limited scope. Future work must extend it by examining more use cases and libraries.

When it comes to the methodical approach, one threat to validity is that the population of Stack Overflow threads matching our scope cannot be described exactly. There is no guarantee that the used queries returned all posts referring to our scope. Additionally, we had to exclude almost $\frac{2}{3}$ of all threads as they did not refer to our scope. The sample can therefore not be considered as representative for the threads referring to our scope nor for the threads returned by our queries. This implies that the results must be verified with further investigations.
Also, we did not verify the intercoder reliability of our issue classification or our security check. It requires a certain expertise in cryptography and software security as well as experience with JCA library. Our data is however published on GitHub[1] and reanalyses are welcome.

5.2 Conclusion

Our study added more evidence confirming the findings of previous research. Working with a low-level library such as JCA requires expertise. Not all programmers working on security related code however have the required domain knowledge.

[1] ToDo

6 Anleitung zu wissenschaftlichen Arbeiten

This chapter describes the methodical approach of qualitative content analysis according Mayring [9]. Each section refers to a work step of the processing scheme which is first explained generally and then applied to the derivation of issues from Stack Overflow threads, see chapter 3.2.
Please note that the aim of this chapter is to specify the processing scheme more precisely. All kind of justification can be found in the methodology section of this thesis.

6.1 Presenting the Sample

Defining and presenting the sample is the first step in qualitative content analysis: We describe the population, the context of its creation and its formal characteristics as well as the sampling process.

6.1.1 Population

Stack Overflow is one of the most popular online forums for programmers. According to the developer survey from 2020, about 50 million people visit the platform per month. Approximately 50% of them are either professional developers or university-level students [17]. As we only want to examine threads that refer to the implementation of symmetric encryption using the Java Cryptography Architecture library, we define our population as the set of posts returned by at least one of the following queries:

- `[java] Cipher.getInstance("AES")`
- `[java] Cipher.getInstance("DESede")`
- `[java] Cipher.getInstance("DES") --DESede`

We do not know a lot about the context of our population. Especially, we do not know anything about the people writing the posts and comments. Based on the findings by Nadi *et al.* [12], we assume that the people asking questions are professional developers (industrial or freelance), researchers or students who must implement a symmetric encryption scenario either for work or education. However, their backgrounds regarding experience with Java as a programming language or cryptography are very heterogeneous. In Nadi *et al.*'s survey, the test persons declared that they "spend at least several hours reading through online resources" (2016, [12], p. 938) before asking on Stack Overflow.

All material is available online on Stack Overflow. It reflects a conversation between one or several users. For each post, we can see the username of the author as well as the date and time of its creation. Some posts have been edited after their creation. In that case, we can see the username and the date of the last edit. However, we cannot reproduce what has been changed. Sometimes, persons also delete their posts or comments. Such posts are not visible anymore.

6.1.2 Sampling Process

Based on the total number of posts returned by the queries (3828) we calculated that 150 threads should be enough to satisfy our requirements of a confidence level of at least 95 % and a margin of error below 10%. Next, we computed the number of threads per query proportionally to the number of posts it returned. Finally, we selected the threads for our sample. We picked half of the sample from the "newest" threads which was determined by the creation date of the question post. We took the other half from the most popular ones based on the view count of the question. Also, we only selected threads that addressed at least one issue within our scope. Other posts were excluded for at least one of the following reasons:

6.2 Specific Research Questions

The second step of qualitative content analysis is to describe precisely what the aim of the interpretation is. We must provide a set of well defined research questions that are theoretically differentiated.

The aim of our first analysis was to find an answer to the first research question of the thesis: *What issues do programmers face when implementing symmetric encryption using Java Cryptography Architecture?* This question can be answered from several perspective.
As we wanted to analyze posts from Stack Overflow and people typically ask a question there if their code does not compile, throws a run-time error, or shows some other unexpected behavior, we first had a look at the *technical aspects*. When implementing a symmetric encryption scenario, developers must take care of some properties (*i.e.,* algorithm, encryption mode, padding mode, key) and perform some actions with them (*i.e.,* derivation, storage, transmission). Additionally, some of these properties depend on each other (*i.e.,* encryption mode - padding) (*i.e.,* see Nakov [13]). If any of these properties, actions, and dependencies is not implemented correctly, the code does not run or shows some strange behavior (*i.e.,* see JCA's Reference Guide [15])
Once a system runs, it must meet various *requirements*. Sommerville distinguished between functional- and non functional requirements [23]. He described functional requirements as a description of a system's the behavior and the services it provides (what a system should do). He considered them as dependent from the type of software, the expected users, and the general approach of an organization when writing requirements. He therefore did not define any sub-requirements. Sommerville defined non-functional requirements as constraints on the services and functions of a system (*i.e.,* memory limitations, time resources for development, legislative requirements). As they are independent from the system, he presented a wide range of non-functional requirements (see figure 6.1).

As we aimed to cover both perspectives with our analysis, we formulated the following specific research questions regarding the implementation of symmetric encryption scenarios using JCA library:

- What technical aspects do programmers fail to implement correctly?
- What requirements do programmers struggle to meet?

6.3 Processing Scheme for Interpretation

Summarizing: The coding unit of this first round of analysis was a single word (e.g. an error message). The context unit was an entire post or a comment.

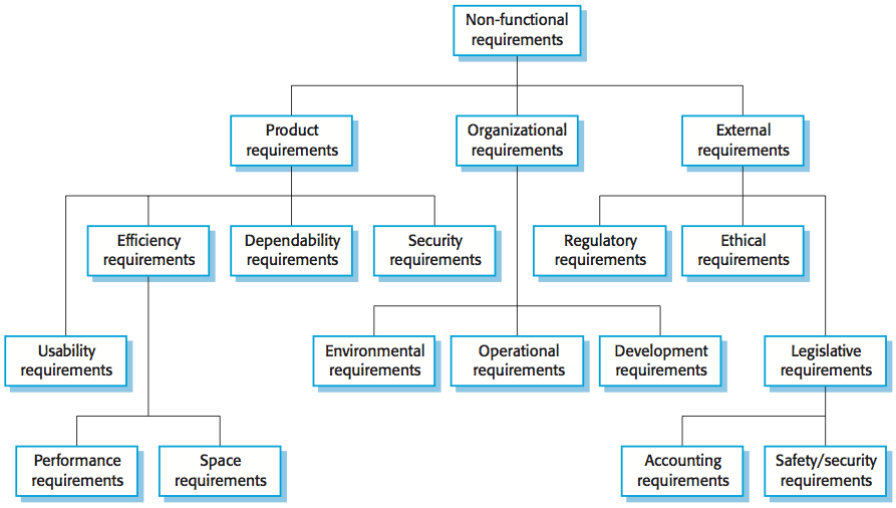


Figure 6.1: Non-Functional-Requirements as defined by Sommerville (2011, [23], p.88)

6.4 Evaluation of Specific Quality Criteria

Bibliography

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Dowoon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy*, pages 154–171. IEEE, 2017.

[2] David Alonso-Ríos, Ana Vázquez-García, Eduardo Mosqueira-Rey, and Vicente Moret-Bonillo. Usability: a critical analysis and a taxonomy. *International journal of human-computer interaction*, 26(1):53–74, 2010.

[3] William J. Buchanan. *Cryptography*. River Publishers, 2017.

[4] International Organization for Standardization. *ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs): Part 11: Guidance on usability*. 1998.

[5] Matthew Green and Matthew Smith. Developers are not the enemy! the need for usable security apis. *IEEE security & privacy*, 14(5):40–46, 2016.

[6] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. Java cryptography uses in the wild. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6. ACM/IEEE, 2020.

[7] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE transactions on software engineering*, pages 1–1, 2019.

[8] S Kruger, J Spath, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: Validating correct usage of cryptographic apis. *CoRR*, 2017.

[9] Philipp Mayring. *Qualitative Inhaltsanalyse : Grundlagen und Techniken*. Beltz, Weinheim, 12. edition, 2015.

[10] Kai Mindermann, Philipp Keck, and Stefan Wagner. How usable are rust cryptography apis? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 143–154. IEEE, 2018.

[11] Eduardo Mosqueira-Rey, David Alonso-Ríos, Vicente Moret-Bonillo, Isaac Fernández-Varela, and Diego Álvarez Estévez. A systematic approach to api usability: Taxonomy-derived criteria and a case study. *Information and Software Technology*, 97:46–63, 2018.

[12] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946, 2016.

[13] Svetlin Nakov. *Practical Cryptography for Developers*. GitBook, 2018.

[14] Oracle. Class cipher, 2021.

[15] Oracle. Java cryptography architecture (jca) reference guide, 2021.

[16] Oracle. Java security standard algorithm names specification, 2021.

[17] Stack Oveflow. 2020 developer survey, 2020.

[18] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability smells: An analysis of developers' struggle with crypto libraries. In *Fifteenth Symposium on Usable Privacy and Security*, pages 245–257. usenix, 2019.

[19] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P Carloni, and Simha Sethumadhavan. Crylogger: Detecting crypto misuses dynamically. 2020.

[20] Girish Maskeri Rama and Avinash Kak. Some structural measures of api usability. *Software: Practice and Experience*, 45(1):75–110, 2015.

[21] Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009.

[22] Thomas Scheller and Eva Kühn. Automated measurement of api usability: The api concepts framework. *Information and Software Technology*, 61:145–162, 2015.

[23] Ian Sommerville. *Software Engineering*. Pearson, 9th edition, 2011.

[24] Minhaz Zibran. What makes apis difficult to use? *International Journal of Computer Science and Network Security (IJCSNS)*, 8(4):255–261, 2008.

7 Acknowledgments

8 Appendix

A Sample and Analysis Related Data

B Reasons for Excluding Threads During Sampling

ToDo: C# Beispiele austauschen

- **too general**: posts referring to cryptographic concepts or cyber security in general rather than the targeted API
  Example: When will C# AES algorithm be FIPS compliant?
- **does not refer to cryptography**: issues occurring in a non-cryptographic context, i.e. establishing a network connection or file access
  Example: Syntax error
- **does not refer to symmetric encryption**: posts referring to other cryptographic concepts such as asymmetric encryption or hashing
  Example: RSA decryption (fails)
- **does not refer to JCA**: issues occurring during a symmetric encryption scenario but which are not due to Java Cryptography Architecture. Such issues can refer to another library (e.g. BouncyCastle) or to some other aspect such as character encoding.
  Example: 256bit AES/CBC/PKCS5Padding with Bouncy Castle
- **does not refer to targeted algorithm**: posts referring to other algorithms than the targeted ones
  Example: how to use password with `PBEKeySpec` hash
- **looking for an equivalent** or **interoperability issue**: looking for equivalents / counterparts in different programming languages
  Example: Change Rijndael decryption logic from C# to Python
- **negative votes** or **closed**: posts of poor quality
- **academic**: posts with a different focus than obstacles when using the API
- **duplicate**: Duplicates are often left unanswered (or only answered with the reference to a similar post). For posts belonging to the ”most popular” category, we included a duplicate if it had more views than the original.

C Example Records For Technical Aspects' and Requirements' Categories