b
UNIVERSITÄT
BERN

Jenny in Wonderland

Exploring the Difficulties of Symmetric Encryption

Bachelor Thesis

Sophie Gabriela Pfister
from
Bern, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

21 December 2021

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari, Mohammadreza Hazhirpasand

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Recent research revealed that a wide range of cryptography libraries lacked usability. Developers therefore misused them and produced insecure applications. A commonly observed source of obstacles was lack of documentation quality. Programmers consult other resources (*i.e.,* Stack Overflow) if they do not find the required information or code examples in the official documentation.

In this context, we aimed for an investigation on the API level to further clarify developers' obstacles. We focused on symmetric-encryption-related APIs from the Java Cryptography Architecture (JCA) library, in particular the `Cipher` class. We analyzed the content of 150 threads from Stack Overflow to identify the issues programmers faced when working with these APIs as well as common forms of API misuse causing security risks. We also sought links between these problems and JCA's documentation by formulating questions for each issue and seeking the answers in the documentation.

We observed that most of the identified issues related to the generation of parameters (*e.g.,* keys) or instantiating a `Cipher` object (*e.g.,* specifying encryption mode). About 20% of all issues were discussed regarding security. However, only 24 threads did not contain any potential security risks. The identified risks mainly related to the use of unsafe encryption modes and constant/static values as a key or initialization vector. We were able to reduce the issues and security risks to 64 questions. Most of them ($\sim$84%) were at least partly covered by the documentation. We concluded that most issues and cases of misuse could have been prevented if the original poster had read and understood the documentation. However, JCA's documentation is spread over several documents, and locating the required piece of information might therefore be difficult. Additionally, programmers may lack the required domain knowledge and find documentation hard to understand. As this study revealed several JCA-specific obstacles relating to its documentation or the library design, we recommend that future research continues evaluating cryptography libraries on the API level.

Contents

1 Introduction

Cryptography is a fundamental part of the digital world. It provides techniques to ensure confidentiality, authenticity, and integrity of information. Nonetheless, Buchanan described the internet as an unsafe place [3], citing that too little security was implemented in the services and protocols used. He argued that "the next generation of the Internet [...] must be built in a trustworthy way" (Buchanan, 2017, [3], p. 1).

In practice, there are numerous vulnerabilities found in software and protocols each year.[1] One of the most potentially disastrous weakness types concerns cryptography. Although there are a large number of cryptography libraries for building secure applications by providing services such as hashing, message authentication, as well as symmetric and asymmetric encryption, a series of recent studies indicated that software developers had difficulty correctly using cryptography. Hazhirpasand *et al.* analyzed 489 open-source Java projects and found that only two were completely secure [7].

One of the leading issues is that cryptography libraries lack usability, a problem which has been studied in various well-known cryptography libraries. The results showed that libraries often do not support auxiliary tasks (*e.g.,* Mindermann *et al.* [12]), that they are not abstract enough (*e.g.,* Nadi *et al.* [14]), and that they lack documentation quality (*e.g.,* Mindermann *et al.* [12], Nadi *et al.* [14], Patnaik *et al.* [19]). Similarly, Acar *et al.* indicated that unusable cryptography libraries not only prevented developers from writing functional code but also lead to the emergence of security vulnerabilities since developers were more likely to misuse the APIs [1]. Moreover, good documentation

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

was a strong predictor for both functional and secure code. Acar *et al.* emphasized the importance of having official documentation that 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 the context of API usability, documentation usability, API misuse, and unsafe code require closer investigation. Therefore, the following research questions address these areas:

1. Which issues do programmers face when implementing symmetric encryption using Java Cryptography Architecture (JCA)?
2. What security risks can be found in code and advice shared on Stack Overflow referring to the implementation of symmetric encryption scenarios using JCA?
3. To what extent are these issues due to missing or inadequate documentation?

Since different cryptography libraries have different API designs, studying more than one cryptography library on Stack Overflow may have revealed a multitude of issues. Thus, we focused on one library (*i.e.,* JCA) 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. JCA is the default cryptography API for Java developers, and it acquired FIPS-140 standards issued by the National Institute of Standards and Technology (NIST) specifying the requirements for cryptography libraries and modules.[2]

To answer the research questions, we analyzed 150 threads from Stack Overflow, where at least one issue related to the study's scope was discussed. To address the first research question, we identified the issues the original poster[3] was facing and categorized them regarding technical aspects that had been incorrectly implemented or requirements that had 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. For the third research question, we derived a set of prioritized questions from the previous findings and sought answers in the documentation of the JCA library. We also took notes regarding documentation quality in general.

[2] FIPS-140-3

[3] author of the question post in a thread

Regarding the first research question, we observed that most of the issues discussed referred to generating algorithm parameters (*i.e.,* key, initialization vector) (24.2%), and the `Cipher.getInstance(...)` method (22.8%). Many errors (27%) were caused by developers failing to correctly configure the dependencies between different properties involved in encryption (*e.g.,* encryption mode–padding, algorithm–key size). Some programmers even used different properties or parameters for encryption and decryption. This was the reason for 15% of all issues. We concluded that these developers lacked the domain knowledge to properly use a low-level cryptography library such as JCA. Other developers also struggled with the API design of JCA, especially the dependency from providers (*i.e.,* default behavior) and the high prevalence of overloaded methods.

Concerning the second research question, we found that programmers frequently used unsafe encryption modes (ECB, CBC). Indeed, 75.3% of all original posters used one of these modes. They also utilized static values for keys (28.7%) and initialization vectors (16.0%). Other original posters did not implement password-based key derivation in a secure way (7.3%). They used a weak password (6.7%), static salt (4.7%), too few iterations for key derivation,[4] (4%) short salt,[5] (2%) and reused passwords (2%). The answer posts contained much fewer security risks: only 27% of all accepted answers had security risks in their code snippets, and most of these were inherited from the original post as the person answering the question focused on producing functional code without thinking of security.

In relation to the third research question, we observed that most of the derived questions were covered by the documentation (84.4%), especially those with higher priorities. We concluded that most of the issues could have been prevented if the original poster had thoroughly read and understood the documentation. However, an answer might be difficult to find since the documentation is spread over several documents (*i.e.,* API documentation [16], Reference Guide [17], Standard Algorithm Name Specification [18]). Additionally, the results from the first and second research questions imply that some programmers lack domain knowledge. Thus, the documentation might be difficult for them to understand. We considered 27.8% of the answers as unclear or incomplete.

Among the unanswered questions, 70% targeted cryptography or software security in general. We observed that JCA documentation did not provide links to resources for comprehensible information about these topics. While it links the specifications for most algorithms, the most popular symmetric encryption algorithm, the AES, is missing. Additionally, specifications might be difficult to read for a reader who does not have a mathematical/technical background.

[4] $< 1,000$

[5] < 64 bits

This thesis presents the state of research in chapter 2. Chapter 3 describes the methodical approaches of the study. We explain and interpret our results in chapter 4. Chapter 5 includes the conclusion, limitations, and conductive thoughts. Lastly, in the "Anleitung zum Wissenschaftlichen Arbeiten" (chapter 6), we provide and explain a best-practice example for password-based encryption using the AES-GCM.

2
Related Work

2.1 API Usability

One of the most popular definitions of usability is from 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 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 heterogeneous. Some researchers focused on programmers’ needs and defined guidelines and heuristics to describe what a usable API should look like. As an example, Zibran conducted a meta-analysis on API usability literature and described a set of 22 specific guidelines [25]. He considered an API usable if it was “(1) easy to learn, (2) easy to remember, (3) easy [to] write client code, (4) easy to interpret client code, and (5) difficult to misuse” (2008, [25], p. 256).

Other researchers approached the problem from the perspective of software metrics. For instance, Rama and Kak proposed eight metrics for API usability referring to method overloading and name confusion, method grouping, parameter list complexity and consistency, thread safety, and documentation [21]. Scheller and Kühn even defined an extensible framework to measure interface complexity automatically [23].

Another approach was to focus on the concept of usability and redefine it more precisely. Alonso-Ríos *et al.* developed a detailed taxonomy [2]. Starting from usability, they organized a wide range of attributes in a hierarchy.

Furthermore, Mosquiera-Rey *et al.* combined several approaches [13]. 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 also identified and described a total of 45 heuristics for API usability.

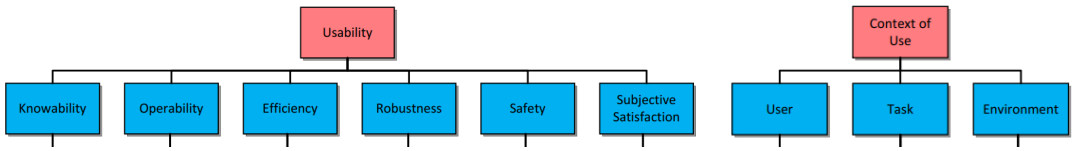


Figure 2.1: First Level Attributes of Usability and Context of Use Taxonomies (Mosquiera-Rey *et al.*, 2018, [13], p. 49 ff.)

2.2 Usability Criteria for Documentation

Mosquiera-Rey *et al.* located documentation quality within the *knowability* attribute of API usability [13]. They defined this property as the extent to which a programmer can "understand, learn, and remember how to use the system" (Mosquiera-Rey *et al.*, 2018, [13], p. 48). They further 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 [22]. 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 on how to improve the usability of such libraries [12]. They also asserted that good documentation for a cryptography API should

- 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 against 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 among them.

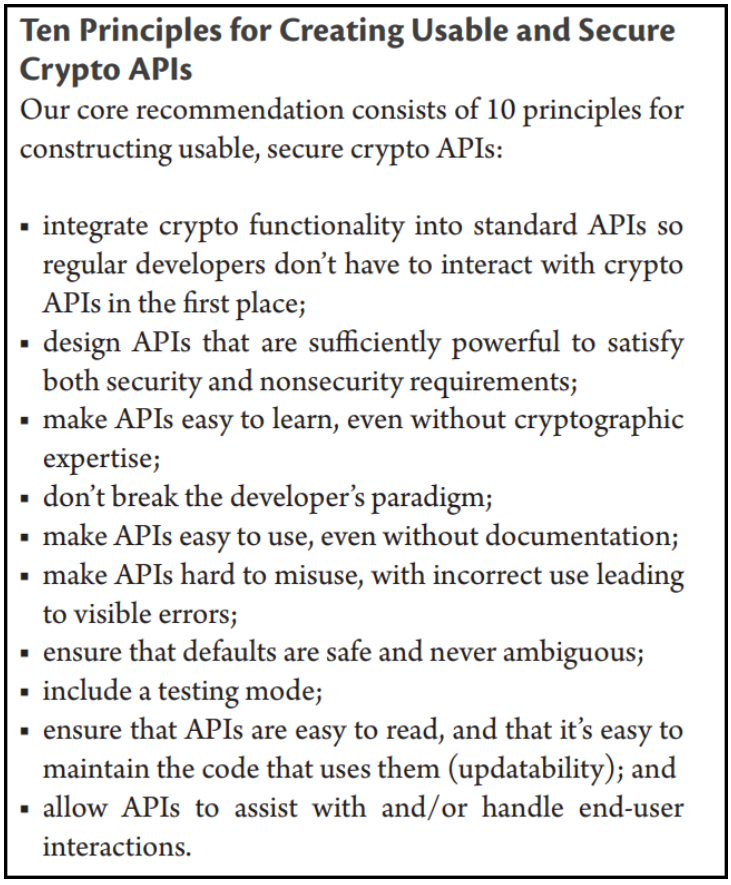
2.3 Usability of Cryptography Libraries

Green and Smith defined 10 principles regarding the usability and security of cryptography libraries [5] . Their main idea was that security-related functionalities should be integrated into non-cryptographic APIs so that regular programmers[1] do not have to deal with cryptographic APIs at all. The entire set of principles is shown in figure 2.2.

Patnaik *et al.* extended these principles by defining usability smells [19]. They were looking for "telltale signs that one of the ten usability principles is being violated" (2019, [19], p. 245). To do so, they examined a wide range of popular cryptography APIs: OpenSSL, NaCl, libsodium, Bouncy Castle, SJCL, Crypto-JS, and PyCrypto. They manually reviewed almost 2,500 posts on Stack Overflow and identified the issues the programmers were facing. They categorized 16 thematic issues, of which two related to the programmers' lack of knowledge:

- *Passing the buck*: Questions that are answered in the documentation
- *Lack of knowledge*: Questions implying that "the developer does not have foundation level cryptography knowledge" (Patnaik *et al.*, 2019, [19], p. 250).

[1] without cryptography expertise



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 (Green & Smith, 2016, [5], p. 42)

They subsequently mapped the remaining issues to four usability smells:

- "need a super-sleuth" (*i.e.,* missing, incomplete or unclear documentation)
- "confusion reigns" (*i.e.,* should I use this? how should I use this? abstraction issues, borrowed mental models)
- "needs a post-mortem" (*i.e.,* unclear error messages, unsupported features, API misuse, deprecated feature)
- "doesn't play well with others" (*i.e.,* build issues, compatibility issues, performance issues)

Similarly, Hazhirpasand *et al.* selected 20 popular cryptography libraries and evaluated 25 Stack Overflow posts for each, assigning a topic to every post (figure 2.3) [8]. They found that the prevalence of topics differed among the libraries. For example,

Theme	# of posts	Description
Encryption/Decryption	112	Technical problems, *e.g.,* modes of encryption, AES or IV, for encryption or decryption of a string
Library installation	111	Posts related to installation, compilation, and version mismatch
Certificate-related issues	74	Posts related to SSL, self-signed certificates, PEM, PKCS7, DER
Library interoperability	63	Posts related to working with more than a crypto library
Generate/store keys	45	Posts related to proper methods of loading or generating a crypto key
Hashing	37	Posts related to MD5, SHA, HMAC and other hashing algorithms
Digital signature	34	Posts related to how to sign or verify a signature
Sample implementation	19	Posts where a sample code was requested
Random number generator	3	Posts related to generating a true random number
Cryptography attacks	2	Concerns for cryptographic attacks in discussions

Figure 2.3: Topics Discussed on Stack Overflow Regarding Cryptography Libraries (Hazhirpasand *et al.*, 2019, [8], p. 4)

users of pyOpenSSL mainly struggled with certificates (17), mcrypt users had difficulties installing the library (16), and more than half of the posts referring to the CryptoJS library involved library interoperability (13). Nevertheless, many cryptography libraries share the same problems.

In another study, Acar *et al.* 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 new, more usable cryptography libraries. They conducted a between-subjects online study where Python programmers implemented a symmetric or asymmetric encryption task using an assigned library. Both task and library were assigned randomly. The programmers also completed an exit survey where they were asked about their backgrounds as well as their opinions 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.

They found that the strongest predictors for working code was the documentation quality and the availability of working code examples. Concerning security, the programmers' background was most important. Developers with a security background were more likely to produce secure code. Although "simpler" APIs (*i.e.,* more abstract, secure default values) 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 lacked documentation quality. Acar *et al.* also observed that a complex API with good documentation (*i.e.,* PyCrypto) was rated as more usable by the participants than a simple API with bad documentation (*i.e.,* Keyczar).

Mindermann *et al.* evaluated the usability of Rust cryptography libraries [12]. After determining the most popular libraries, they conducted an exploratory study where one of the authors completed a set of cryptography-related tasks several times using a different library for each round. Afterward, they compared two popular libraries, rust-crypto and ring, in a controlled experiment. For this, 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). As an example, high-level libraries use authenticated encryption by default. Default values were often avoided, but if present, they were secure. Nonetheless, Mindermann *et al.* identified several security risks: some libraries did not warn about broken algorithms or when a nonce was accidentally reused. Furthermore, documentation quality varied among and within the libraries.

Mindermann *et al.* also issued 12 recommendations to remedy present usability issues. These are more specific than the ones suggested by Green and Smith as they only applied to Rust libraries.

With a different focus, Nadi *et al.* investigated the usability of Java cryptography libraries to understand the underlying causes for misuse of the related APIs. They also

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

aimed to identify the most common cryptography tasks and possible support tools.

To do so, the researchers 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 (*i.e.,* code examples), the APIs' design (*i.e.,* error messages, method overloading, insecure default values), and lack of domain knowledge among programmers. The survey participants explicitly asked for more abstract APIs and better documentation.

2.4 Misuse of Cryptography APIs

Krüger *et al.* proposed CrySL, a definition language that allows the specification of rules for secure usage of cryptographic APIs [9]. It enables specifying rule sets class-wise in separate files. Krüger *et al.* also implemented CogniCrypt, a compiler that translates CrySL rules into a static analysis that automatically checks a given Java application for rule violations [9]. To evaluate it, Krüger *et al.* defined a rule set for the JCA library and analyzed 10,001 Android apps. They also reviewed 50 apps manually for comparison.

CogniCrypt detected the use of JCA in 4,071 apps. In 96% of them, CogniCrypt identified at least one issue. In total, CogniCrypt discovered 19,756 rule violations, most of which referred to broken constraints (*i.e.,* illegal values), especially for the `MessageDigest` class. In the manual analysis, Krüger *et al.* found 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 (*i.e.,* DES) and unsafe encryption modes (*i.e.,* 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 utilized the JCA library [7]. Only two of them were considered completely secure. Figure 2.4 shows the ratio of correct and incorrect usage for each of the investigated APIs.

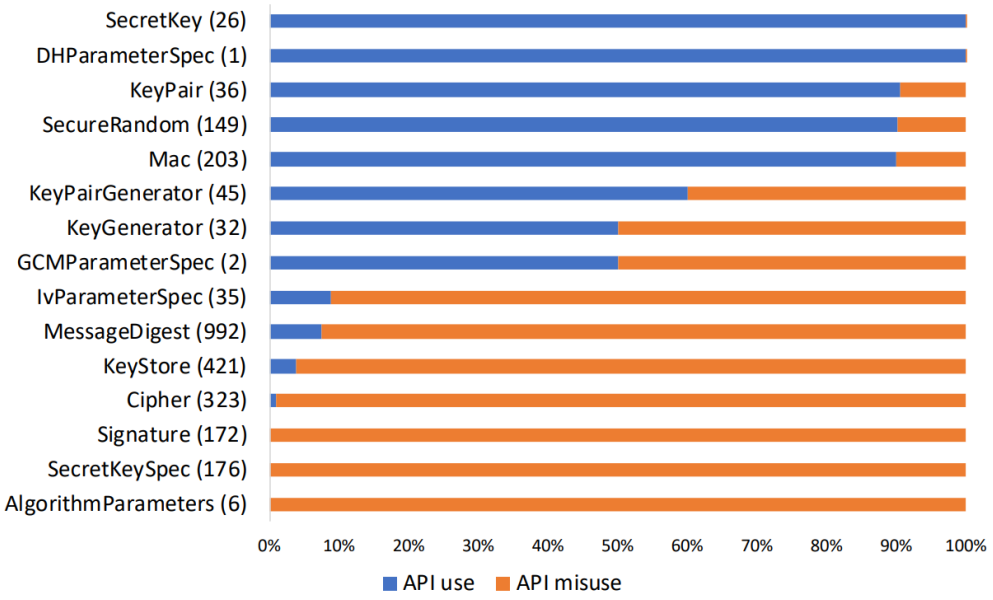


Figure 2.4: Correct API use vs. API Misuse (Hazhirpasand *et al.*, 2020, [7], p. 3)

Although a few records were mistakenly marked as misuse according to the authors' manual review, their findings showed that programmers especially struggled to correctly use the classes `AlgorithmParmeters`, `SecretKeySpec`, `Signature`, `Cipher`, `KeyStore`, `MessageDigest`, and `IVParameterSpec` correctly. These classes support (symmetric) encryption, hashing, and digital signatures.

In addition, Hazhirpasand *et al.* 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 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 correctly use the API and blindly accepted security-related pull requests. Another identified issue was that there were not enough security concerns in the official documentation. Programmers also argued that although they use a cryptographic API, the code was not security-related.

Piccolboni *et al.* further developed a tool, CRYLOGGER, to check security-related code for API misuse [20]. It conducts a dynamic analysis 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 (figure 2.5). Piccolboni *et al.* used CRYLOGGER to

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 2.5: CRYLOGGER's Security Rules (Piccolboni *et al.*, 2020, [20], p. 5)

analyze 1,780 Android apps. They found that rules 01 and 18 were violated very often ($> 90\%$), indicating that broken hash functions and unsafe sources for random number generation were frequently used. 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 (IVs), the reuse of key-IV pairs and the use of HTTP.

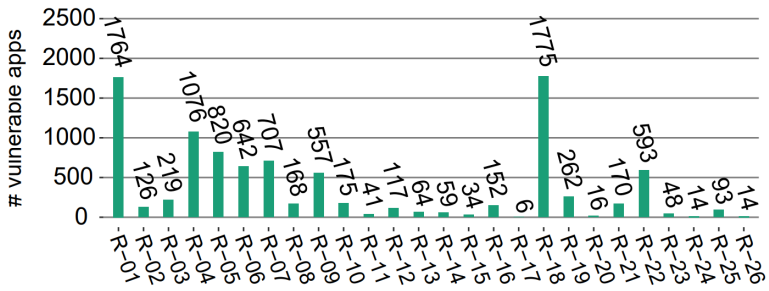


Figure 2.6: Rule Violations Detected by CRYLOGGER (Piccolboni *et al.*, 2020, [20], p. 12)

Finally, Hazhirpasand and Ghafari [6] followed a different approach and analyzed 173 vulnerability reports on the HackerOne bug bounty platform to understand the existing types of cryptography vulnerabilities. Most (33.5%) referred to the use of insecure SSL versions. Other common topics were the use of weak cryptography parameters (*i.e.,* broken hashing algorithms, short keys; 14.5%), OpenSSL bugs (14.5%), and mixing HTTP/HTTPS content (12.7%). In rarer cases, the reports involved miscellaneous attacks (6.9%), timing attacks (6.3%), the use of static keys or passwords (6,3%), or issues related to HTTP (5.2%).

3 Methodology

To answer the first research question,[1] we identified the issues that programmers face when implementing symmetric encryption using JCA. We derived a list of issues by analyzing 150 threads on Stack Overflow, one of the most popular Q&A forums for programmers. To address the second research question,[2] we also scanned these threads regarding security risks. Based on the elicited issues, we then defined a set of questions in order to observe the extent to which the the official documentation provided relevant answers. This helped to answer the third research question.[3]

As we required several methodological approaches, this chapter is divided into five sections. The first section describes the sampling process. The second refers to identifying issues from Stack Overflow posts. The third section explains how we checked the threads for security risks. The fourth describes the procedures for deriving questions from the previous findings and analyzing the library's documentation. The fifth section illustrates the evaluation processes.

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

[2] What are common security risks in code and advice shared on Stack Overflow referring to the implementation of symmetric encryption scenarios using the JCA library?

[3] To what extent are these issues due to missing or inadequate documentation?

3.1 Sampling

In Java, developers should use the `Cipher` class to accomplish a symmetric encryption task. The class 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 the `Cipher` class.

As some of the symmetric algorithms supported by JCA are not very popular, the corresponding queries returned only a small number of posts. We decided to exclude these algorithms, such as RC2, and focused instead on the three most popular symmetric encryption algorithms: AES, 3DES,[4] and DES. We therefore used three queries, which are shown in the left column of table 3.1.

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 needed to study 150 posts. Then we computed sample size per each query proportionally to the number of posts it returned. The results 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, to select the threads from Stack Overflow, we chose 50% of the sample size from the newest threads and the other half from the most popular ones. Using this approach, we attempted to balance the inclination of our sample size since the majority of developers first look for answers on Stack Overflow 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. A complete list of reasons for exclusion and some example threads are found in the appendix. However, a thread commonly consists of more than one issue. Therefore, we included the threads in which at least one issue, question, or piece of advice referred to the study's scope. As we also did not include any posts lacking quality, we excluded 296 threads.

[4] also TripleDES, DESede

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 JCA?* We conducted a manual qualitative content analysis following the guidelines presented by Mayring [11]. As the sample was large, we needed an approach that allowed us to efficiently evaluate the data. We also needed it to support method-integrative approaches that combine qualitative and quantitative elements. These requirements are all key characteristics of Mayring's guidelines.

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

In this step, the goal was to extract and record all relevant information so that we did not have to reread the entire threads during the further evaluation steps.

There were two coders involved in summarizing, who first summarized independently and then discussed the results together to create a consistent and more objective list of records. In particular, we eliminated records that did not refer to our scope. For example, we excluded all issues that referred to the conversion of plain text or cipher text (*i.e.,* character-encoding).

For each thread, we recorded the set of issues and questions that the original poster was facing. We sometimes also identified issues based on comments (*e.g.,* security hints). Then we tried to identify the reasons and solutions from the accepted answer post. If no answer was accepted, we derived it from the discussion (*i.e.,* based on a remark by the original poster indicating that a comment was helpful). However, we were not always able to find a possible explanation for the original poster's issues.

As a result, we aimed for one record per issue that must consist of a short description (*e.g.,* an error message, a shortened form of a question) and might include more precise explanations for the reason or solution.

3.2.2 Classification

All records that resulted from summarization were classified in two rounds: We first categorized all records regarding technical aspects and then regarding requirements the original poster was not able to meet. Per round, we assigned at most one category to each record. A list of examples for each category is provided in the appendix.

Technical Aspects

In the first round of classification, we focused on the technical aspects of implementing symmetric encryption, *e.g.,* mistakes in the original poster's code that lead to errors. We started with a set of predefined main categories that we inductively refined during the classification. Each time we defined a new category, we restarted the classification.

The main categories referred to the set of tasks that programmers must consider when implementing symmetric encryption using JCA: *Cipher Object Instantiation, Generating Algorithm Parameters, Cipher Object Initialization, Transformation, and Transmitting Algorithm Parameters*. We defined subcategories to obtain deeper insights (*i.e.,* if an issue targeted only one aspect of a main task), or to allow unambiguous classification (*i.e.,* dependencies between two properties). We also added one more main category resulting in the following *main* and **subcategories**:

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

  Additionally, we defined the following subcategories:
  - **Dependency Encryption Mode–Padding**: The encryption mode 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 none of the aforementioned aspects.
- *Generating Algorithm Parameters*: Depending on the specification of the `Cipher` object, different kinds of parameters are required. 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`).

- *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). For this, we defined the following subcategories:

  - **Dependency Algorithm–Key**: The algorithm determines which 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 algorithm's block size.
  - **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 further defined the following subcategories:

  - **Key Transmission**: The key must be kept secret.
  - **Transmission of Other Parameters** such as the IV. 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.

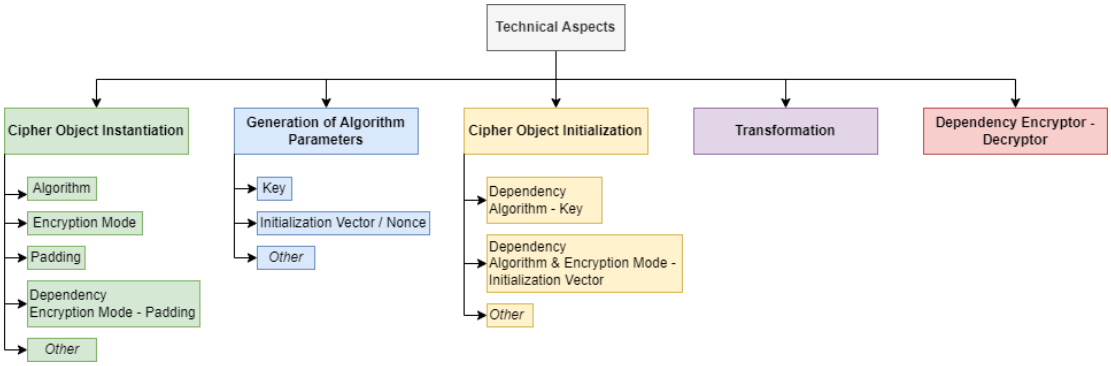


Figure 3.1: Hierarchy of Technical Aspect Categories

If all tasks are correctly implemented, the code compiles and runs without raising any errors. Therefore, 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. During this first classification round, we asked "What implementation step was performed incorrectly causing the error?" or "What implementation step is targeted by the question?"

Requirements

As not all issues related to technical aspects, we defined a second set of categories regarding the design of an application. We identified different kinds of functional and non-functional requirements as categories, consulting Sommerville [24] as a theoretical basis. During the analysis, we asked, "Which requirements are the original posters not able to meet?" Not all requirements defined by Sommerville occurred in our analysis, and therefore we only assigned the following categories:

- **Use Case** (functional requirements)
- **Performance**
- **Space**
- **Reliability**
- **Portability**
- **Interoperability**
- **Security**

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 warned that the shared code might cause issues regarding one of the requirements (*i.e.,* a security hint).

3.3 Analysis of Security Risks

The goal of the 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 using the JCA library?* We first defined a set of security rules regarding the implementation of symmetric encryption. Then we manually checked the threads from our sample for violations of these rules.

3.3.1 Security Rules

We derived our rules from the sets used for CRYLOGGER (Piccolboni *et al.* [20]) and CogniCrypt (Krüger *et al.* [10]). We only considered the rules that were applicable to symmetric encryption and structured them using the categories from the technical aspect classification (section 3.2). We also generalized them to simplify the evaluation. For example, R-04 of CRYLOGGER says "Don't use the operation mode CBC (client/server scenarios)" (Piccolboni *et al.*, 2020, [20], p. 5). As we often did not know in what context the original poster wanted to use the code, we inferred that CBC should not be used at all. The resulting rules can be found in table 3.2.

3.3.2 Tracking Security Rule Violations

We manually checked the original sample to observe any security rule violations. For this, we only considered the question post, the accepted answer post, and comments on 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:

- **Question–Code**
- **Question–Text**
- **Answer–Code**
- **Answer–Text**

While analyzing the code snippets, we focused on the parts where encryption, decryption, key derivation, IV generation, and key storage were implemented. For instance, if someone defined a key-`String` in the main method and passed it to the encryption

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

section as a parameter, we did not consider this a security risk. The encryption section can still be safe if an appropriately derived, non-static key is passed.

3.4 Analysis of Documentation

The results from the preceding analyses formed the basis on which to evaluate the documentation for JCA. As it is spread over several documents and sources, we only examined the most basic ones: JCA Reference Guide [17], the Java Security Standard Algorithm Name Specification [18], and the API documentation for the `javax.crypto` package [16]. 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 questions to be answered. Afterward, we sought the answers in the documentation. Our aim was to answer the third research question: *To what extent are these issues due to missing or inadequate documentation?* The questions additionally gave us more insight into issues with which programmers are struggling (first research question).

3.4.1 Deriving Questions

In general, the official documentation should support the usage of the API and not educate developers or address their basic questions regarding cryptography. However, the documentation of a cryptography library should link reliable and comprehensible resources that explain basic cryptographic concepts (Mindermann *et al.* [12]). We therefore created two lists of prioritized questions: one with *questions for documentation* and one containing *general questions*.

We derived the questions from the results of the first analysis. To do so, we reprocessed the records and formulated questions for each one. If a 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.

Then we adapted the priorities of the questions referring to security based on the results of the second analysis. We set the priority to the actual number of posts targeted by it.

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 resources in another order. In this manner, we aimed to find answers as time efficiently as possible. For questions for 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. As a benefit, 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 (section 2.2).

3.5 Evaluation

After the first analysis (section 3.2), we applied several forms of frequency analysis to identify the tasks and requirements with which most programmers were struggling. We similarly evaluated the results from the second analysis (section 3.3) to identify the most common security risks.

After the third analysis (section 3.4), we interpreted the results in several ways. To begin, we more closely reviewed the questions' content as well as their priorities. We also determined how many and which questions were not answered in the documentation. We tried to identify possible relationships between issues and security risks on one side and documentation (and API design) on the other. Finally, we made recommendations on how the documentation of JCA could be improved.

4 Results and Interpretations

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

4.1 Implementation Issues

In the first analysis, we identified all issues the original posters were facing and recorded them separately. Depending on what the original poster was struggling with, we classified the issues as relating to a *technical aspect* and/or a *requirement*.

In total, we recorded 219 issues, 197 (90%) of which we recorded as relating to technical aspects and 76 (35%) regarding requirements; 62 records were classified twice. We could not classify only one thread (and its relating record) due 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* (53) and *Cipher Object Instantiation* (50). Both of these categories refer to specifying and generating the properties used during encryption and decryption. During the generation of algorithm parameters, programmers might derive a key, an IV, and other algorithm parameters such as advanced authentication data. The original posters especially struggled with key derivation (36 records). This was

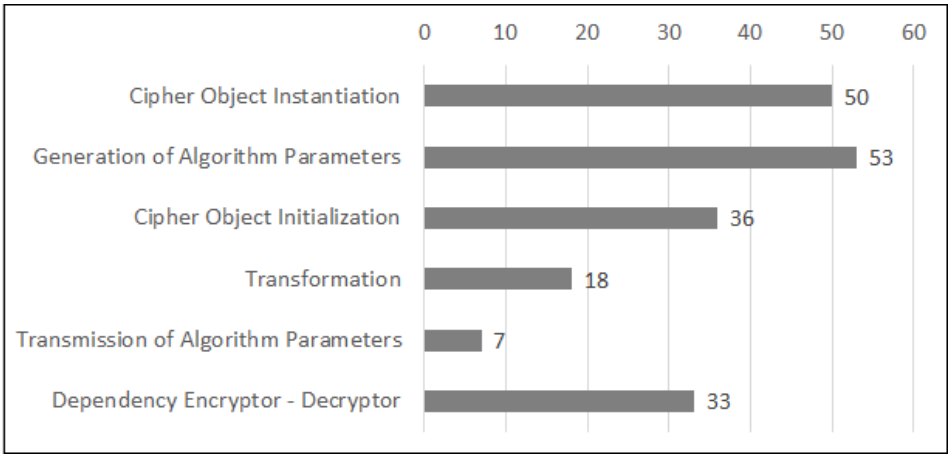


Figure 4.1: Number of Issues Assigned to a Technical Aspect Main Category ($N = 219$)

not surprising as previous work has revealed that programmers often had difficulty with key handling. The set of problems relating to the key[1] comprised more than one-fifth of all issues.

During the instantiation of a `Cipher` object, programmers specify algorithm, encryption mode, and padding. In this context, the original posters were especially struggling with the latter two aspects. Eighteen 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 (*e.g.,* key, IV) are passed to the cipher object. Most issues were assigned to the *other* subcategory and often referred 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 in this category implies that many programmers 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 very often. Some original posters were confused that there were two methods that perform transformation: `update(...)` and `doFinal(...)`. They did not know which one must be called in their scenario.

Among issues referring to the transmission of algorithm parameters (7), most referred

[1] derivation, transmission, dependency from algorithm

to the key (5). The remaining records were related to the IV (1) and the salt used for password-based key derivation (1).

Table 4.1 shows the complete list of subcategories and their frequencies of assignment.

Subcategory	%	#	(Main Category) (relativ frequency)
Algorithm	14.00%	7	
Encryption Mode	36.00%	18	
Padding	22.00%	11	(Cipher Object Instantiation) (22.83%)
Dependency Encryption Mode - Padding	22.00%	11	
Cipher Object Instantiation - Other	6.00%	3	
Key Derivation	67.92%	36	
IV / Nonce Generation	26.42%	14	(Generation of Algorithm Parameters) (24.20%)
Generation of Other Algorithm Parameters	5.66%	3	
Dependency Algorithm - Key	22.22%	8	
Dependency Algorithm/Encryption Mode - IV	22.22%	8	(Cipher Object Instantiation) (16.44%)
Cipher Object Initialization - Other	55.56%	20	
Transformation	100.00%	18	(Transformation) (8.22%)
Key Transmission	71.43%	5	(Transmission of Algorithm Parameters) (3.20%)
Transmission of Other Algorithm Parameters	28.57%	2	
Dependency Encryptor - Decryptor	100.00%	33	(Dependency Encryptor-Decryptor) (15.07%)

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

Within the categories referring to requirements, security was the predominant category (46 records). This seems logical as this is what cryptography is about. However, only three original posters asked about the security of their implementation: one asked whether the IV generated by default was safe, another whether it was secure to reuse algorithm parameters (e.g., key, IV) for several transformations, and the third person wanted to know whether encryption became safer if some kind of salt was added to the plain text. A more common security issue was that programmers were not able to run AES-256 due to missing security policy files (5 threads). However, most security records were identified based on security hints from people commenting on or answering the questions. However, there is massive underreporting as the results from the security-related analysis show (section 4.2).

As seen in figure 4.2, the other requirement categories occurred less often. There were 12 issues related to the portability of an application, and they often referred to original posters not specifying all values themselves. For 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 among different providers and therefore among different platforms.

Another seven records were assigned to the interoperability category. These issues

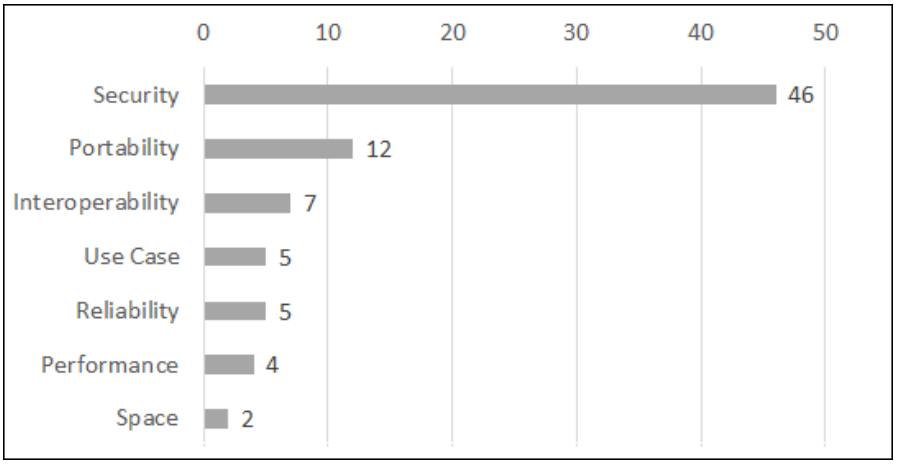


Figure 4.2: Number of Issues Assigned to a Requirement Category ($N = 219$)

often 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 (*e.g.,* for padding) that were therefore not visible in the source code. As a result, the original posters incorrectly instantiated or initialized the Java `Cipher` objects. Some of these default values were also not supported by JCA (*e.g.,* ZeroPadding). One programmer also used a non-standardized function (*i.e.,* SHA1PRNG) for random number generation).

The five 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).[2]

Most of the issues 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.

Moreover, some programmers complained that the execution of the `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 all at once (lacking space efficiency).

As previously mentioned, some records were classified twice, regarding technical aspects and requirements. The heat map (figure 4.3) shows the relative overlapping of categories. We observed the highest overlapping for *Generation of Algorithm Parameters* and *Security*; 21% of all records assigned to each category were also assigned to both. Most of them referred to static values being used for key or initialization vectors. The second-highest overlap was between *Cipher Object Instantiation* and *Security* categories

[2]mapping 16 B of data bijectively to a 12 digit number

(16%). Almost all of these 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 them. The issues assigned to the *Portability* or *Interoperability* category and some other technical aspects category were mostly due to default values that vary among platforms and libraries. The overlap 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 `getInstance(...)` 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 questions, 84 answer posts, and related comments to find any violations of the determined rule set and found a total of 331 security risks. Most (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 also found 35 rule violations in answers' code snippets and nine in answers' text. 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 employed 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 Checklist

The further analysis focused on the code snippets in the body of the questions as we did not find any additional security risks in the text or answers that were not present in or related to the question code. However, some original posters only shared the code section where the issue occurred (*e.g.,* an error was thrown) and did not show how auxiliary tasks (*e.g.,* key derivation) were implemented. We therefore must be aware that there might be even more vulnerabilities in their code.

On average, each question post contained 1.66 security risks in its code snippets. We noted that the average for the most popular (and older) posts (1.91) is slightly higher than for the newest posts (1.43). In total, 24 question posts did not contain any security risks 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 algorithm is due to the 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 a programmer naïvely copied and pasted 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, the total number of rule violations referring to password-based key derivation *R-03-b* to *R-03-f* was 29. Fourteen posts used an unsafe key derivation procedure. They often just hashed the password and used the first n bits[3] instead of using a safe key derivation function such as PBKDF2.

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 IV otherwise, there were not many code sections showing IV generation. There were also hardly any question posts that showed or explained how the key was stored.

[3] n = key size

4.3 Documentation

We were able to reduce the issues and security risks from the previous findings to 64 questions: 43 for documentation and 21 more general ones. After checking the documentation, 10 questions remained unanswered, and for 15 questions, we considered the answer incomplete, unclear, or even misleading.

4.3.1 Questions

The first observation regarding the questions was that there were twice as many JCA-specific questions as general ones. Still, 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 not only lack knowledge about the JCA library but also about cryptography in general.

The results from the first analysis provided insight into the tasks and requirements with which programmers were struggling. The JCA-specific 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 for documentation. For example, the default values and behavior of a `Cipher` object depend on the provider. However, whether a provider is available or not and which provider is used by default depends on the platform. This decreases the portability of an application, particularly if the code relies on default behavior.

An additional five questions were related to overloaded methods. Two methods perform data transformation, and both of them are overloaded. The methods for instantiation and initialization are overloaded as well.

The questions also revealed that programmers particularly had difficulty with password-based key derivation.

The general questions helped us to understand which knowledge the original posters were missing. As we only knew the accurate number for questions / issues referring to the security of code, most of the higher-prioritized questions also referred to that topic. As shown in table 4.4, the remaining higher-prioritized questions targeted the various dependencies. From this, we concluded that some original posters were not aware of cryptography at all.

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 Nine Questions to Documentation

Question	Priority
Which encryption modes are safe?	113
Which key derivation functions are safe?	40
What requirements must an IV / nonce / salt meet to be safe?	37
Which input data and specifications must be equal for encryption and decryption?	27
Which symmetric encryption algorithms are safe to use?	24
What are requirements for a safe password based key derivation function?	15
What encryption modes require padding?	9
Which encryption modes do require an IV or nonce?	6
What is the required size for an IV?	5

Table 4.4: Top Nine General Questions

Among the questions that did not relate to security or to the dependencies, we found the following topics:

- AES algorithm (four questions, total priority = 9)
- initialization vector (one, 4)
- use cases that symmetric encryption should be used for (one, 3)
- cipher text input for decryption (one, 2)
- padding (two, 2)
- message authentication (two, 2)

4.3.2 Missing and Unclear Answers

We were not able to find answers to three JCA-specific and seven general questions, most of which had a rather low priority (< 4). However, the higher prevalence of unanswered general questions implies that JCA documentation does not provide (sufficient) links to comprehensible resources for general information about cryptography. We observed that there were some explanations within the documentation, but they were not detailed.

The following three unanswered questions represented further documentation-related issues that violate the principles and recommendations described in section 2.2:

- *Which symmetric encryption algorithms are safe to use?* Firstly, this was the only higher-prioritized question (24) that was left unanswered. It can be seen as an example of JCA documentation not providing enough hints regarding security. The reference guide mentions that ECB is not safe. However, it does not give any advice on which encryption mode should be used. Moreover, there are not further security warnings for other aspects (*e.g.,* algorithm, password-based key derivation).

- *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. Thus, this might complicate the implementation of interoperability scenarios.

- *What properties does AES-256 require?* We found a link to the official specifications for most algorithms in the Standard Algorithm Name Specification. However, for AES, there was no link. This is problematic since AES is one of only two symmetric encryption algorithms that are recommended. Furthermore, specifications are rather hard to understand.

For 13 JCA-specific and two more general questions, we considered the answers not to be clear enough. They often referred to overloaded methods. The documentation did not distinguish the differences clearly enough and they were copied and pasted.

Another prevalent issue was that there were no code example for some common or important scenarios (*e.g.,* using a `KeyStore` or password-based key derivation utilizing PBKDF2). The available ones often did not work due to some missing parts. For instance, the example code for `Cipher` class did not show how the algorithm parameters were generated and only demonstrated how the `Cipher` class was used. There were other code sections that depicted how a key was randomly generated or how password-based encryption could be implemented. However, there were hardly any examples that worked all by themselves.

```
SecretKey myKey = ...
byte[] myAAD = ...
byte[] plainText = ...
int myTLen = ...
byte[] myIv = ...

GCMParameterSpec myParams = new GCMParameterSpec(myTLen, myIv);
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, myKey, myParams);
```

Figure 4.4: Extract From *Example 2-2 Sample Code for Using an AES Cipher with GCM Mode* [17]

5 Conclusions and Future Work

In this chapter, we first aim to answer the research questions based on the findings 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 JCA?* Our study provided answers to this question from different perspectives: Considering tasks with which programmers are struggling, most involved generating algorithm parameters, especially deriving the key from a password. The second most problematic task was instantiating a `Cipher` object. The original posters particularly failed to correctly specify encryption mode and padding. The third most problematic task was initializing the `Cipher` object. Most issues and questions related to this task referred to the programmer not passing all required parameters to the `init(...)` method.

Another aspect related to this question was the design of the 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 which provider is chosen by default. This decreases the portability of applications. An additional issue was the high number of overloaded methods, particularly `getInstance(...)` and `init(...)`. Moreover, two methods perform transformation, `update(...)` and `doFinal(...)`, which are both overloaded as well, and some original posters failed to choose the correct one.

This could also be related to the documentation: the overloaded methods are documented too similarly, and there is no advice about which overload should be used in which scenario. Finally, there are not enough working code examples, and the docu-

mentation does not link comprehensible resources for more general information about cryptography.

On the other hand, regarding the programmers, our study added more evidence to the existing literature confirming the assumption that they lack sufficient knowledge about cryptography.

RQ 2: *What are common security risks in code and advice shared on Stack Overflow referring to the implementation of symmetric encryption scenarios using the JCA library?* We found that security risks were particularly present in code snippets from question posts. Several answers gave advice regarding or even improved security, whereas others only focused on the functionality of the code. We also observed a slight improvement when we compared the older posted questions with the newer ones.

The most common security risk was the use of an unsafe encryption mode. This is also related to the most common providers using ECB as the default block cipher mode of operation. However, some answers suggested to use CBC instead of ECB, which is not safe either.

Other common security risks were the use of static values for either the key or IV, or both. Although some original posters explicitly stated that they only used them in their post to simplify the code section, it would become a security risk if another programmer just copied and pasted the code.

The procedures used for password-based key derivation also contained security risks. JCA supports PBKDF2 as a safe procedure for password-based key derivation. However, it was rarely used.

RQ 3: *To what extent are these issues due to missing or inadequate documentation?* This research question is perhaps the most difficult one to answer. We considered most of the derived questions ($> 60\%$) being clearly answered by the documentation, especially the higher-prioritized ones. We also found related information for almost 85% of the questions. We therefore cannot blame insufficient documentation for the struggles of developers.

There are several explanations. It is possible that some original posters did not consult the documentation at all. However, JCA's documentation is spread over several documents, and finding a required piece of information might be time-consuming. Another explanation is that some programmers lacked the domain knowledge to understand the documentation. For example, if someone must derive a key from a password, they must know what salt and iteration count are and which requirements these parameters must meet to be safe.

Based on our findings, we suggest the following improvements:

- Link comprehensible resources for more information about cryptography.
- Provide working code examples that cover all important scenarios, in particular state-of-the-art scenarios (*e.g.,* authenticated encryption). These examples should be found in one place.
- Provide advice regarding security. Warn against unsafe values.
- Document overloaded methods more specifically (*e.g.,* describe in what scenarios which overload should be used).

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. However, we found new issues that were specific to the JCA library. Future work should extend it by examining more use cases and libraries.

Regarding the methodical approach, a major threat to validity is that we did not verify the intercoder reliability of our issue classification or our security check. This requires a certain expertise in cryptography and software security as well as experience with the JCA library. We therefore did not succeed in finding a suitable reviewer. Nonetheless, our data is published on GitHub,[1] and reanalysis is welcome.

Another limitation 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 two-thirds of all threads as they did not fit into our scope. The sample therefore cannot be considered as representative for the threads referring to our scope nor for the threads returned by the queries. This implies that the results must be verified with further investigations.

This might also indicate that programmers do not only struggle with cryptographic APIs when they implement a cryptography scenario. Thus, an important research query could be to investigate the various backgrounds of programmers and develop tools that support them in acquiring knowledge about computer science in general as well as more specific topics such as cryptography.

[1] data directory

5.2 Implications

Software security requires expertise, especially while working with a low-level library such as JCA. Our results imply that programmers who lack the required domain knowledge fail to implement symmetric encryption. Cryptography and software security should therefore be an essential part in the education of future developers. In addition, the IT industry must also be aware of the importance of security and how difficult it is to implement. Researchers must find ways to put their findings and tools into practice.

6 Anleitung zu wissenschaftlichem Arbeiten

One obstacle we observed during the thesis project was that JCA documentation did not contain enough working code examples. We therefore decided to provide a best-practice example for password-based encryption (PBE) which is a common use case. Unlike the PBE example in the JCA Reference Guide [17], our example does not rely on the default behavior. This allows us to use the GCM encryption mode, which is the recommended block cipher mode of operation (*e.g.,* Nakov [15]).

The example is also available on GitHub.[1] It illustrates the encryption of `String` objects and files and can be used as a service class for *educational purposes*. A guide on how to use it can be found in the corresponding ReadMe file. The code should work if any providers are available that support the `Cipher`-transformation "AES/GCM/NOPADDING" and the key derivation function "PBKDF2WithHmacSHA256." We built and tested the implementation on Java 15.0.1 using the default SunJCE Provider (version 15) for both `Cipher` and `SecretKeyFactory` class.

[1] code directory

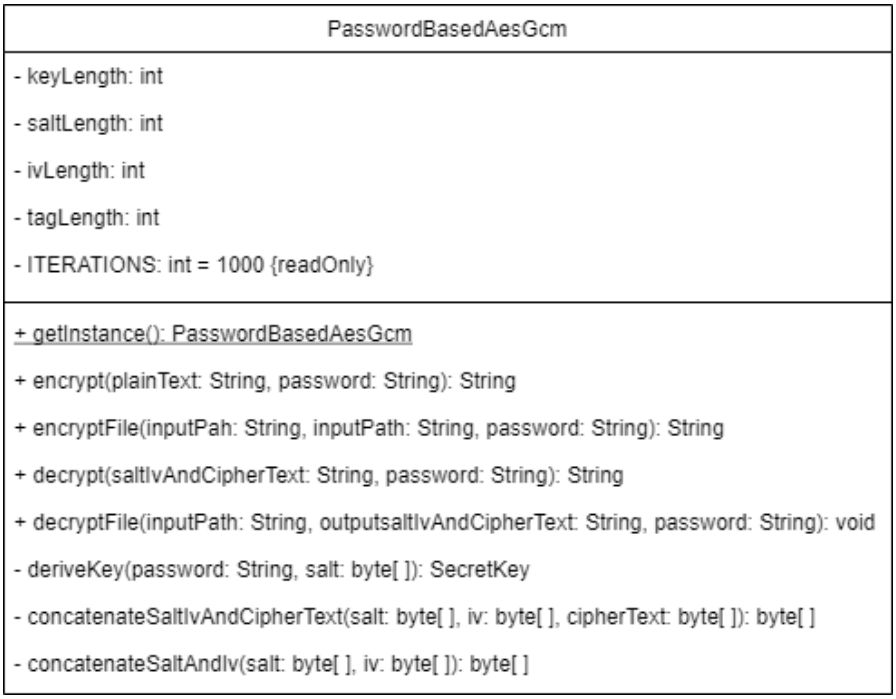


Figure 6.1: UML of the Example Service Class Showing the Most Important Class Variables and Methods

The aim of this chapter is to explain our example code. It describes and justifies each implementation step and also provides the most important security hints. We sometimes make use of class variables, namely the lengths of various algorithm parameters as well as the number of iterations performed during key derivation. Depending on the kind of application, they could all be constants (such as `ITERATIONS`). We kept them modifiable because we wanted to support a wider range of use cases. Figure 6.2 shows the class variables used in our example. The comments indicate which values are acceptable to have both a functional and a secure application.

```
private int keyLength;                  // keyLength in bits, keyLength in {128, 192, 256}
private int saltLength;                 // saltLength in Bytes, saltLength > 8
private int ivLength;                   // ivLength in Bytes, ivLength > 1
private int tagLength;                  // tagLength in bits, tagLength in {96, 104, 112, 120, 128}
private final int ITERATIONS = 1000;    // iterations for key derivation function, ITERATIONS > 1000
```

Figure 6.2: Class Variables and Their Constraints

6.1 Subroutines

deriveKey(String password, byte[] salt)

```
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, this.ITERATIONS, this.keyLength);
SecretKey key = keyFactory.generateSecret(keySpec); // the secret key
byte[] keyMaterial = key.getEncoded();
keySpec.clearPassword();
return new SecretKeySpec(keyMaterial, algorithm: "AES");
```

Figure 6.3: Algorithm for Key Derivation

This is probably the most crucial task of the implementation. As we want to derive the key from a password, we must instantiate a `SecretKeyFactory` object using a password-based key derivation function as algorithm. We selected "PBKDF2WithHmacSHA256" as it is both secure and standardized. It is therefore also suitable for scenarios where different platforms are involved. PBKDF2 can also be used with another pseudo random function (*i.e.,* HmacSHA1), depending on the specification of the other platform.

We then must instantiate a `PBEKeySpec` that holds the password, the salt, the number of iterations, and the length of the key. We can use the `keyFactory` to generate a `SecretKey`from the `PBEKeySpec`.

However, the factory transmits its algorithm to the generated key, but to use it with an "AES"-`Cipher` object, the key must hold "AES" as algorithm as well. We worked around this issue by fetching the key material from the generated key (`getEncoded()`) and instantiating a new one holding "AES" as algorithm. Since `SecretKey` is an interface, we instantiated a `SecretKeySpec` instead.

In addition to the statements shown in figure 6.3, the method includes some minimal error handling. For a safe application it is important to

- use a safe password
- use a safe key derivation function
- use random salt of at least 64 bits (eight bytes)
- clear the password from the `PBEKeySpec` after generating the (first) key

concatenateSaltIvAndCipherText(byte[] salt, byte[] iv, byte[] cipherText)

```
private byte[] concatenateSaltIvAndCipherText(byte[] salt, byte[] iv, byte[] cipherText) {
    byte[] result = new byte[this.saltLength + this.ivLength + cipherText.length];
    for (int i = 0; i < result.length; i++) {
        if (i < this.saltLength) {
            // fill in salt
            result[i] = salt[i];
        } else if (i < (this.saltLength + this.ivLength)) {
            // fill in IV
            result[i] = iv[i - this.saltLength];
        } else { // saltLength + ivLength <= i
            // fill in cipher text
            result[i] = cipherText[i - (this.saltLength + this.ivLength)];
        }
    }
    return result;
}
```

Figure 6.4: Subroutine to Concatenate Salt, IV, and Cipher Text

Since encryption and decryption must use the same salt and IV, we must transmit these values from the encryptor to the decryptor. However, they must not remain secret. We therefore decided to prepend these values to the cipher text when encrypting a `String`. As a `Cipher` object returns the cipher text as a `byte` array, we must create a larger array and correctly copy the values.

concatenateSaltAndIvA(byte[] salt, byte[] iv)

Similarly to the previous subroutine, this method joins two `byte` arrays. Since we are not returning any cipher text when encrypting a file, we only want to return the salt and the IV.

6.2 Encryption

Configuring the Cipher Object

Whether we are encrypting a `String` or a file, we must instantiate and initialize a `Cipher` object. The following lines of code are therefore part of both encryption methods, `encrypt(...)` and `encryptFile(...)`.

The first step is creating the required algorithm parameters. After generating some random `byte` values for salt using a `SecureRandom` object, we can call the previously described subroutine to derive a key from our password. Since we are using GCM encryption mode, we also require a `GCMParameterSpec` object. It holds the length of the authentication tag (`tagLength`) and a randomly generated IV.

```
SecureRandom generator = new SecureRandom();

// derive key from password
byte[] salt = new byte[this.saltLength];
generator.nextBytes(salt);  // salt must be random
SecretKey key = deriveKey(password, salt);

// generate random initialization vector
byte[] iv = new byte[this.ivLength];
generator.nextBytes(iv); // iv must be random

// prepare algorithm parameter
GCMParameterSpec params = new GCMParameterSpec(this.tagLength, iv);
```

Figure 6.5: Generating a `SecretKey` and `GCMParameterSpec`

Then we can instantiate the `Cipher` object and pass the parameters using the `init(...)` method.

```
// instantiate and initialize cipher object
Cipher encryptor = Cipher.getInstance("AES/GCM/NOPADDING");
encryptor.init(Cipher.ENCRYPT_MODE, key, params);
```

Figure 6.6: Instantiating and Initializing a `Cipher` Object for Encryption

Encrypting a String

`encrypt(String plaintext, String password)`

Since encryption is performed on bytes, we must convert the plain text into a `byte` array. This can be done using `String`'s `getBytes()` method. To ensure interoperability and portability, we recommend specifying the character set (*e.g.,* UTF-8) that should be used for decoding.

```
// convert plaintext to byte[] (character decoding)
byte[] plain = plainText.getBytes(StandardCharsets.UTF_8);
```

Figure 6.7: Converting Plain Text to `byte[]`: Character Decoding

The resulting `byte` array is passed to `Cipher`'s `doFinal(...)` method that performs encryption.

```
// encrypt data
byte[] cipher = encryptor.doFinal(plain);
```

Figure 6.8: Encrypting Plain Text

As previously mentioned, we want to transmit the salt and the IV along the cipher text. To do so, we prepend them to the encryption result. After joining the three arrays, we want to convert it back into a `String`. Each of the concatenated values potentially contains any bit sequence, and also sequences that are not included in a standard character set. We must therefore use base64 encoding as we might lose data otherwise.

```
// concatenate salt, iv and cipher text
byte[] result = concatenateSaltIvAndCipherText(salt, iv, cipher);

// convert to string (character encoding)
return Base64.getEncoder().encodeToString(result);
```

Figure 6.9: Encrypting Plain Text

Encrypting a File

`encryptFile(String inputPath, String outputPath, String password)`

The first step is to open a set of streams:

- a `FileInputStream` to read from the file specified by `inputPath`
- a `CipherInputStream` that encrypts anything read by the `FileInputStream` using the previously initialized `Cipher` object
- a `FileOutputStream` that writes the encryption results to the `outputPath`

```
// initialize streams
FileInputStream in = new FileInputStream(new File(inputPath));
FileOutputStream out = new FileOutputStream(new File(outputPath));
CipherInputStream c = new CipherInputStream(in, encryptor);
```

Figure 6.10: Initializing Streams for File Encryption

The streams can now be used to read and encrypt the file block-wise. To do so, the `CipherInputStream` must read (and encrypt) the next block of bytes and store it to an int variable. The value is then written to the output file by the `FileOutputStream`.

```
// read and encrypt block wise
byte[] b = new byte[256]; // Buffer - one block is 256 Byte
int i = c.read(b); // read and encrypt the first block
while (i != -1) {
    out.write(b, off: 0, i); // write the result to the output file
    i = c.read(b); // read and encrypt the next block
}
```

Figure 6.11: Using Streams to Encrypt the Content of a File Block-Wise

When the streams are done, we must close them.

```
// close streams
c.close();
in.close();
out.close();
```

Figure 6.12: Closing All Streams

Similarly to the encryption result from encrypting a `String`, we are returning the salt and the IV concatenated and base64-encoded.

```
// return salt and iv, concatenated as byte[] and base64 encoded
byte[] result = concatenateSaltAndIv(salt, iv);
return Base64.getEncoder().encodeToString(result);
```

Figure 6.13: Returning Base64-Encoded Salt and IV

Security Requirements for Encryption

- Use a safe algorithm (AES or Blowfish).
- Use a safe block cipher mode of operation (preferably CTR or GCM).
- Use a safely derived, secret key.
- Use a random IV that was generated using a secure source of randomness (*e.g.*, `SecureRandom`).
- Use a new key-IV pair each time performing encryption.

6.3 Decryption

Decrypting a String

`decrypt(String saltIvAndCipherText, String password)`

As we base64-encoded the encryption result, we first must decode it using the same format.

```
// convert saltIVAndCipherText to byte[] (character decoding)
byte[] data = Base64.getDecoder().decode(saltIVAndCipherText);
```

Figure 6.14: Base64 Decoding

Next, we must restore the parameters. As we prepended the salt and the IV to the cipher text, they can easily be restored using `saltLength` and `ivLength` class variables. We can then derive the key using the password parameter and the retrieved salt and instantiate another `GCMParameterSpec` using the `tagLength` class variable and the retrieved IV.

```
// separate salt, iv, and cipher text
byte[] salt = Arrays.copyOfRange(data, from: 0, this.saltLength);
byte[] iv = Arrays.copyOfRange(data, this.saltLength, (this.saltLength + this.ivLength));
byte[] cipherText = Arrays.copyOfRange(data, (this.saltLength + this.ivLength), data.length);

// restore the parameters
SecretKey key = deriveKey(password, salt);
GCMParameterSpec params = new GCMParameterSpec(this.tagLength, iv);
```

Figure 6.15: Restoring Algorithm Parameters

Finally, we can instantiate and initialize another `Cipher` object and ask it to decrypt the cipher text.

```
// decrypt the data
Cipher decryptor = Cipher.getInstance("AES/GCM/NOPADDING");
decryptor.init(Cipher.DECRYPT_MODE, key, params);
byte[] plain = decryptor.doFinal(cipherText);
```

Figure 6.16: Instantiating and Initializing a `Cipher` Object and Asking It for Decryption

In the end, we must just convert the `byte` array holding the plain text back to a `String` using the same character set that was used for decoding it before encryption (UTF-8).

```
// convert plain text to string
return new String(plain, StandardCharsets.UTF_8);
```

Figure 6.17: Character-Encoding the Plain Text

Decrypting a File

`decryptFile(String inputPath, String outputPath, String saltAndIv, String password)`

Similarly to `String` decryption, we first must restore the parameters. To do so, we base64-decode the `saltAndIv` parameter and separate the values using the `saltLength` and `ivLength` class variables. The salt is used with the password to derive the key. The IV is passed to a new `GCMParameterSpec` object along the `tagLength` class variable. Then we can instantiate and initialize a `Cipher` object.

```
// restore parameters
byte[] parameters = Base64.getDecoder().decode(saltAndIv);
byte[] salt = Arrays.copyOfRange(parameters, from: 0, this.saltLength);
byte[] iv = Arrays.copyOfRange(parameters, this.saltLength, parameters.length);

SecretKey key = this.deriveKey(password, salt);
GCMParameterSpec params = new GCMParameterSpec(this.tagLength, iv);

// initialize cipher object
Cipher decryptor = Cipher.getInstance("AES/GCM/NOPADDING");
decryptor.init(Cipher.DECRYPT_MODE, key, params);
```

Figure 6.18: Restoring the Algorithm Parameters for File Decryption and Creating a `Cipher` Object

The rest of the decryption is implemented in exactly the same way as encryption:

1. Open a `FileInputStream`, a `CipherInputStream`,[2] and a `FileOutputStream` (see figure 6.10).
2. Block-wise, read (and decrypt) the input file using the `CipherInputStream` and write the result to the output file using the `FileOutputStream` (see figure 6.11).
3. Close the streams (see figure 6.12).

As the plain text has already been written to the output file, this method does not have a return value.

[2] use the prepared `decryptor` instead of an encrypting `Cipher` object

6.4 Remarks on Parameter Transmission

We did not include any form of parameter transmission or storage in our example because this aspect strictly depends on the kind of application. In client-server scenarios, we might need to transmit the parameters via HTTPS. In some scenarios, it is sufficient to store them locally.

However, we want to emphasize the importance of keeping the password a secret. It is the only value not known by a potential intruder. If one needs to store or transmit it, it should be wrapped using (public key) encryption.

In case of customized algorithm parameter lengths (class variables), these must also be stored to ensure that all parameters can be restored at decryption. Our example class provides a simple, static `getInstance()` method to create a valid default `PasswordBasedAesGcm` object. However, there are setter methods to customize the class variables. They can be retrieved by calling the corresponding getter methods. The values may be transmitted as separate *POST* variables or stored in a database alongside the encryption result.

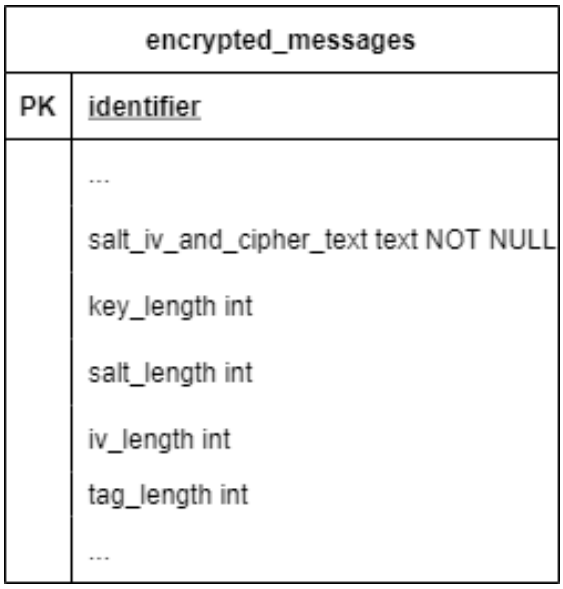


Figure 6.19: Draft of a Database Entity

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 and Mohammad Ghafari. Cryptography vulnerabilities on hackerone. *arXiv e-prints*, pages arXiv–2111, 2021.

[7] 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.

[8] Mohammadreza Hazhirpasand, Oscar Nierstrasz, and Mohammad Ghafari. Dazed and confused: What's wrong with crypto libraries? *arXiv e-prints*, pages arXiv–2111, 2021.

[9] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: Validating correct usage of cryptographic apis. *CoRR*, 2017.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] Oracle. Class cipher, 2021.

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

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

[19] 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.

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

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

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

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

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

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

Appendix

A Sample and Analysis-Related Data

All data related to sampling, issues classification, security risk tracking, and evaluating the documentation is available in the data directory of the thesis' GitHub repository.

B Reasons for Excluding Threads During Sampling

- **too general**: posts referring to cryptographic concepts or cyber security in general rather than the targeted API
  Example: How to check if a string is encrypted or not?

- **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: Decryption using blowfish failing

- **looking for an equivalent** or **interoperability issue**: looking for equivalents / counterparts in different programming languages
  Example: Encrypt in node and decrypt in java

- **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 Issue Classification

Technical Aspects

Category	Issue	Reason	Solution
Algorithm	TripleDES vs. DESede - what is the difference?	-	They are equivalent. TripleDES is the name for SunJCE Provider
Encryption Mode	Convert AES-128-CBC PHP to Java	different encryption modes (CBC vs. Java default = ECB)	Use “AES/CBC/PKCS5PADDING”
Padding	Why does doFinal() add extra bytes to my cipher text? How can I remove them?	-	doFinal() adds padding for block ciphers. It is removed automatically if the same padding is specified for en- and decryption
Dependency Encryption Mode - Padding	IllegalBlockSizeException : Input length must be multiple of 16 when decrypting with padded cipher	Transformation “AES” might be interpreted as “AES/ECB/NoPadding”. ECB is a block cipher and requires padding in most cases	add padding or switch encryption mode
Cipher Object Instantiation - Other	NoSuchAlghoritmExeption for “AES/ECB/PKCS5Padding” on Android	-	-
Key Derivation	InvalidKeyException: Invalid AES key length: 128 bytes	Key retrieved through Diffie-Hellman is 128 Bytes instead of bits	Generate key of correct size
IV / Nonce Generation	invalid IV length	IV = [00000000000000000] $\rightarrow$ length = 1	IV = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] $\rightarrow$ length = 16
Generation of Other Algorithm Parameters	AES-GCM: different output for AuthenticationTag for Java and JS	Using Timestamp as AAD	-
Dependency Algorithm - Key	Is this code AES-256 encryption?	-	Yes, key is of size 256b
Dependency Algorithm/Encryption Mode - IV	InvalidKeyException: Parameters missing (IV)	CBC requires IV	add IV or switch encryption mode
Cipher Object Initialization - Other	How is the IV generated (if it is not passed to Cipher.init())?	-	default values depend on provider
Transformation	BadPaddingException: pad block corrupted	you shouldn’t call doFinal() on every block, because doFinal() expects any padding at the end, which obviously won’t be there in intermediate blocks	Either (a) call update() on intermediate data, then doFinal() at the end, or (b) just arrange to have all your data in one buffer or byte array, and call doFinal() once on the whole job lot.
Key Transmission	Stored tat DES Secretkey into database converting it into String. Now i want to Convert that String to Secretkey.	-	You are seeing the object class and the hashcodes of 2 different instances sharing the same reference. If you want to confirm whether your key is getting decoded correctly, print the encoded version of the decoded key.
Transmission of Other Algorithm Parameters	how to decrypt cipher text encrypted with salt	-	pass salt along cipher text
Dependency Encryptor - Decryptor	BadPaddingException: Given final block not properly padded	OP generates new key for decryption	pass encryption key to decryption section as parameter

Requirements

Category	Issue	Reason	Solution
Use Case	inappropriate use case	OP want to store password encrypted	store hash instead
Performance	Encryption of large file is very slow (AES)	-	-
Space	OutOfMemoryException for large files	This appears to be an issue with the implementation of the GCM mode. I'm not sure that you can work-around it.	-
Reliability	Cryptograhy Service crashes about 1x/day	Cipher objects in global scope –¿ Cipher is not thread safe	-
Portability	Same code produces different cipher text on "native java" and android platform	default values depend on platform	fully specify transformation
Interoperability	Same key generation procedure results in different keys	SHA1PRNG is not standardized	use standardized RNG
Security	unsafe encryption mode (ECB)	default value	fully specify transformation