Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Java Interface for Secure Crypto Config

Lisa-Marie Teis

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Kai Mindermann M.Sc.
Commenced:	April 4, 2020
Completed:	December 17, 2020

Abstract

Context: Cryptography is mainly considered in the field of information security for the protection of digital data. But the right selection of a secure set of cryptographic algorithms and parameters can be difficult. Another problem is that provided cryptographic Application Programming Interface (API)s cannot change their default configurations, meaning that they get insecure over time.

Aim: The general aim is to create a cryptographic library that allows developers to easily use secure default configurations. Such a library should realize all security-relevant details internally by safe default configurations, which are adapting to changing security standards. To achieve this goal the Secure Crypto Config (SCC) can be used which ensures security, usability, maintainability and up-/downward compatibility.

Method: First, a draft for a future standardized Request for comments (RFC) was created. In addition, a sample implementation for the corresponding API in Java was developed. This implementation was evaluated by conducting a study that consists of live programming tasks and online questionnaires. The study should compare the Secure Crypto Config Interface (SCCI) with the standard cryptographic libraries of the Java Development Kit (JDK) and Google Tink.

Result: The evaluation has shown that the SCCI is more usable than JDK and Google Tink. By considering the number of security bugs the SCCI is also more secure than JDK. Unfortunately, there was no significant result by comparing the security of the SCCI and Google Tink. Furthermore, no significant difference in the maintainability between the SCCI and the other libraries could be shown. In terms of security and maintainability the SCCI was not significantly better according to statistical tests, nevertheless there are fewer security bugs with the usage of the SCCI.

Conclusion: The SCCI is a future-proof alternative to other cryptographic libraries as it has proven to be both more usable and more secure than other implementations. In the next steps, it is now necessary to drive the standardization process forward. Furthermore, implementations in other languages must follow.

Kurzfassung

Kontext: Die Kryptographie wird hauptsächlich im Bereich der Informationssicherheit zum Schutz digitaler Daten betrachtet. Die richtige Auswahl eines sicheren Satzes von kryptographischen Algorithmen und Parametern kann sich jedoch schwierig gestalten. Ein weiteres Problem besteht darin, dass bereits existierende kryptographische Application Programming Interface (API)s ihre Standardkonfigurationen nicht ändern können. Dies führt dazu, dass sie mit der Zeit unsicher werden.

Ziel: Das allgemeine Ziel ist es, eine kryptographische Bibliothek zu erstellen, die es Entwicklern ermöglicht, auf einfache Weise sichere Standardkonfigurationen verwenden zu können. Eine solche Bibliothek sollte alle sicherheitsrelevanten Details intern durch sichere Standardkonfigurationen realisieren, die sich an ändernde Sicherheitsstandards anpassen. Um dieses Ziel zu erreichen, kann die Secure Crypto Config (SCC) verwendet werden, welche die Sicherheit, Benutzerfreundlichkeit, Wartbarkeit und Auf- /Abwärtskompatibilität gewährleistet.

Methodik: Zunächst wurde ein Entwurf für einen zukünftigen standardisierten Request for comments (RFC) erstellt. Darüber hinaus wurde eine Beispielimplementierung für die entsprechende API in Java entwickelt. Diese Implementierung wurde durch die Durchführung einer Studie evaluiert, welche aus Live-Programmieraufgaben und Online-Fragebögen bestand. Diese Studie sollte dabei das Secure Crypto Config Interface (SCCI) mit den kryptographischen Standardbibliotheken des Java Development Kit (JDK) und Google Tink vergleichen.

Ergebnis: Die Auswertung hat gezeigt, dass das SCCI benutzerfreundlicher ist als JDK und Google Tink. Bei Betrachtung der Anzahl von Sicherheitsfehlern ist das SCCI zudem sicherer als JDK. Leider gab es kein signifikantes Ergebnis beim Vergleich der Sicherheit des SCCI und Google Tink. Außerdem konnte kein signifikanter Unterschied in der Wartbarkeit zwischen der SCCI und den anderen beiden Bibliotheken gezeigt werden. In Bezug auf Sicherheit und Wartbarkeit war das SCCI laut statistischen Tests nicht signifikant besser, dennoch gibt es weniger Sicherheitsfehler bei der Verwendung des SCCI.

Schlussfolgerung: Das SCCI ist eine zukunftsfähige Alternative zu anderen kryptografischen Bibliotheken, da es sich sowohl benutzerfreundlicher als auch sicherer als andere Implementierungen gezeigt hat. In den nächsten Schritten ist es nun notwendig den Standardisierungsprozess voran zu treiben. Zudem sollten Implementierung in anderen Sprachen folgen.

Contents

List of Figures

List of Tables

Listings

Acronyms

API	Application Programming Interface.
BSI	Bundesamt für Sicherheit in der Informationstechnik.
CFRG	Crypto Forum Research Group.
CMS	Cryptographic Message Syntax.
COSE	CBOR Encoded Message Syntax.
DSSC	Data Structure for the Security Suitability of Cryptographic Algorithms.
IANA	Internet Assigned Numbers Authority.
IDE	Integrated Development Environment.
IETF	Internet Engineering Task Force.
ISO	International Organization for Standardization.
Jar	Java ARchive.
JCA	Java Cryptography Architecture.
JDK	Java Development Kit.
JSON	JavaScript Object Notation.
NIST	National Institute of Standards and Technology.
RFC	Request for comments.
RSA	Rivest-Shamir-Adleman.
SCC	Secure Crypto Config.
SCCI	Secure Crypto Config Interface.
SUS	System Usability Scale.

1 Introduction

1.1 Motivation

Often it is the case that programmers have to deal with cryptography in their daily work to ensure data security. However, many of the programmers do not know enough about this area to be able to generate secure code correctly. Therefore, they make use of cryptographic APIs. These are an opportunity for programmers to make use of different cryptographic primitives. But many of them, do not offer a sufficient abstraction from the internal details of the provided primitives. Even in more abstract interfaces, it is often necessary to have cryptographic knowledge to use them securely. In many cases a large number of parameters must be chosen by the programmer, such as key-/salt-length or the algorithm to be used. For example, it is necessary to select 10 different parameters for the implementation of Argon 2 according to Biryukov et al. [BDKJ20]. If the expert knowledge is missing and the cryptographic library is too complex, this can lead to big security gaps. It can be seen that the secure selection of cryptographic algorithms and parameters is not sufficiently described within cryptographic standards at the moment. Even if the right parameters can be chosen there is no standardized way on how to store, distribute or retrieve these configurations. But this is necessary to guarantee software compatibility.

Furthermore, often programming libraries use insecure or outdated default settings (see paper by Egele et al. [EBFK13]), which are not detected by non-specialized programmers that are relying on the used library. There are approaches that want to guarantee more usability by defining secure default configurations but these are getting obsolete quickly. However, they cannot be changed in the interface anymore as applications using them are dependent on the specific functionality.

It can be seen that an automation and standardization of an adapting secure default configuration and the choice of a secure parameter set is missing at the moment.

1.2 Goal

The general goal is to create a cryptographic library that allows developers to easily use secure default configurations. This library should realize security-relevant details internally by safe default configurations, which are adapting to changing security standards. Also up- and downward compatibility must be guaranteed by this library. The approach of the SCC realizes all these aspects as it ensures security, usability, maintainability and up- /downward compatibility. It provides secure cryptographic default configurations in a standardized format for the main cryptographic primitives that are updated within a regular process.

1.3 Contribution

During this master thesis, a sample implementation for the corresponding API of the SCC in Java (SCCI) was developed (see source code [Tei20a]). It provides all important cryptographic primitives and makes them available to the users in an abstract way such that they have to specify as few cryptographically related parameters as possible. The up- and downward compatibility was guaranteed by using CBOR Encoded Message Syntax (COSE) (see RFC [RFC8152]). For the internally used default configurations, a JavaScript Object Notation (JSON) format with a standardized structure was defined. As a standardization for all processes important for the SCC an RFC draft was created (see RFC draft [MT20]) and provided to the Crypto Forum Research Group (CFRG). The main parts of the RFC are: the creation, structure, content of the necessary configuration files and how they should be kept up-to-date, but also the general structure of the API and important security aspects.

The resulting interface should enable all programmers to generate secure code without having cryptographic background. The interface is optimized regarding usability, maintainability and security. These aspects were evaluated by executing a pilot and a main study including programming tasks and a questionnaire. In this way, an alternative to other existing and too complex cryptographic Java libraries is given, which is easier for programmers to use and makes all decisions for secure parameters internally thus reducing the risk of insecure code.

2 Related work

There is one paper from Mindermann and Wagner [MW19b] that is directly related to the topic of this master thesis. In their work, the general idea and the process of the SCC are described. These contents will be discussed in more detail in section 3.1. A presentation that matches the content of this paper was also created and presented by Mindermann and Wagner [MW19a].

The implementation of cryptographically secure code is often difficult to realize. There are numerous papers such as the ones from Georgiev et al. [GIJ+] or Egele et al. [EBFK13] which point out various flaws in the implementation of secure code. Both static and dynamic analyses have often been used to detect these errors (e.g. see paper by Egele et al. [EBFK13]). These analyses show the actual bugs but not the core problem of why the bugs occurred. According to Acar et al. [ABF+17], the core problems that lead to the generation of insecure code are, among other things, the high complexity of the available APIs. These are expecting a high, complex selection of parameters (e.g. key-length) and also providing only insufficient documentation and code examples. The documentation aspect is beside the API complexity one of the most important aspects, according to Acar et al. [ABF+17]. Their results of a conducted study, in which different cryptographic tasks were performed with different libraries, show that the usage of complex libraries could also result in secure code if there was sufficient documentation. However, the complexity of the API still plays an important role in the creation of secure code. Furthermore, it was found that previous knowledge in cryptography is no guarantee for working and secure code. Whether secure code could be created was largely based on how complex or well documented the used API was. According to Acar et al. [ABF+17], the usability of crypto APIs has never been tested within a larger study. Although new cryptographic libraries are created from time to time which promise more usability, they are usually not evaluated in this aspect. The work from Acar et al. [ABF+17] is the first one with a broad usability testing of cryptographic APIs.

Many works show that the usability of cryptographic APIs is related to how secure the resulting code becomes (see paper by Acar et al. [ABF+17]). Also the work of Wijayarathna and Arachchilage [WA18a] points out that usability is related to the number of cryptographic errors. However, it is not always easy to evaluate or measure usability. In the paper from Wijayarathna and Arachchilage [WA18a], general usability test techniques of non-cryptographic APIs are shown and compared. From these results, the method which is best suited for cryptographic APIs was derived. Since security flaws are often caused by the wrong handling of the API by the programmers, the users should be involved by conducting a user study. This requires the definition of a programming task and a questionnaire, the recruitment of test persons and the evaluation of the results. According to the work from Wijayarathna and Arachchilage [WA18a] there are already general questionnaires for the evaluation of the API usability as defined in the work by Clarke [Cla04]. However, these do not take into account all aspects that are important for cryptographic APIs. One of the questionnaires recommended for the usability of cryptographic APIs is the one from Wijayarathna et al. [WAS17].

The questionnaire is based on the work of Clarke [Cla04] but was extended by dimensions and questions that are specific to cryptographic APIs. However, a major drawback of user studies, in general, is the very high time and resource consumption that is necessary to perform them.

As described above, some works deal with the measurement of API usability and some that investigate the relationship between usability and the generation of secure code. During this master thesis, a Java interface was implemented that should provide high usability, maintainability and low complexity such that secure code can be generated as easily and securely as possible. Therefore, it is necessary to have a look at works with a focus on Java cryptography APIs. In works like the one from Acar et al. [ABF+17] different APIs in Python were distinguished in terms of their usability. For Java, however, no comparable usability studies could be found in other papers. Therefore, there are no proven facts that demonstrated that cryptographic APIs in Java have a high or the best usability. By default, the so-called Java Cryptography Architecture (JCA)[1] is included in Java JDK. JCA contains packages such as java.security, javax.crypto. Also, there is Tink[2] which provides cryptographic primitives and already promises in its documentation to be easy to handle and hard to misuse. Although there are no papers that directly compare the usability of Java crypto APIs, there is a paper from Hazhirpasand et al. [HGK+19] that has analyzed Java projects that specifically use JCA packages. This work, however, was not primarily concerned with the usability of JCA packages, but rather with how they are used in practice and what influence the cryptographic experience and general programming experience has on the quality and security of the resulting code. It was found that general programming experience and cryptographic knowledge does not correlate with performance. With the paper by Nadi et al. [NKMB16] there is also a work that deals with problems of programmers in using Java cryptographic APIs in general. This paper examined StackOverflow[3] posts and various GitHub[4] repositories and conducted a study with some test persons to analyze the most common problems. About 1000 cryptography related posts were found on StackOverflow. The most frequently mentioned problems were according to Nadi et al. [NKMB16] in the area of symmetric encryption. In general, more questions about the use of the API than about general domain knowledge could be found. It can be seen that are several papers that focus on the investigation of Java cryptographic APIs with different focuses, but none of them compares different existing Java cryptographic APIs.

There are several other attempts to make it easier for programmers to write secure code. For example, the use of often complicated APIs should be simplified with the developed plugin of Krüger et al. [KNR+17]. This plugin called "CogniCrypt" should make it easier for the programmers to write secure code. First, a framework for the realization of a specific cryptographic primitive can be chosen. Afterward, the security of the code (with possible adjustments of the programmers) is supervised continuously through static code analysis. The programmers themselves have to answer a few questions when using CogniCrypt which does not require deep cryptographic knowledge. This plugin can be used in connection with the Integrated Development Environment (IDE) Eclipse as shown in the example of the work from Krüger et al. [KNR+17]. There are several static analysis tools but unlike CogniCrypt these often do not offer IDE integration possibility and only provide hard-coded security checks. Another attempt to guarantee the generation of secure code is done in the work of Krüger et al. [KSA+17]. It is shown that especially in the early development process

[1]https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html

[2]https://google.github.io/tink/javadoc/tink/1.1.1/

[3]https://stackoverflow.com/

[4]https://github.com/

it is difficult to find errors related to cryptography. To solve this problem "CrySL, a definition language for bridging the cognitive gap between cryptography experts and developers" [KSA+17] was developed. CrySL enables security experts to express in a formal language how secure code can be created using a specific cryptographic API. In addition, a compiler has been developed which translates the definitions formulated by CrySL into a static code analysis possibility. This static code analysis should help to find security gaps (according to the CrySL rules) in Java or Android applications. These rules have already been defined for JCA usage. The functionality to find these defined security gaps were evaluated in 10,000 Android apps. It could be shown that 95% of the examined applications had at least one security misuse of the defined CrySL rules. Also, in other works like from Egele et al. [EBFK13] about 12,000 Android applications were examined and in 88% at least one broken cryptographic rule could be found. That shows that in several works it was already possible to prove that the majority of applications contain security gaps that the programmers were unaware of. This should be avoided with the mentioned tools like CogniCrypt or CrySL.

Furthermore, some works show how the view on the cause of the frequent problems in generating secure code has changed. According to the work of Green and Smith [GS16], the problem of emerging vulnerabilities in implementations has often be seen as a problem of the programmers themselves. The programmers who wrote the code were held responsible for the vulnerability no matter how complicated the used cryptographic API was. Increasingly, this view has changed over time. As the main cause now, the cryptographic APIs were seen and no longer the programmers themselves. Not the programmer must adapt to the complicated API usage but the technology should adapt to the users and support the safe code generation with good usability. This view is also supported by the paper of Ellen and T. [ET96], which also promotes the usability of cryptographic systems. The works by Alma and D. [AD99] or Wijayarathna and Arachchilage [WA18b] also support the mentioned assumptions. For example, the paper of Alma and D. [AD99] shows, based on a conducted study, that secure code can be achieved more easily and with greater success when using cryptographic systems that have good usability, for example in the form of a clearly structured user interface. All these mentioned works support the statement that the basic problem of emerging security gaps are not the programmers themselves but the complexity of the used API or cryptographic system. Also, Hazhirpasand et al. [HGK+19] supports the statement by stating that there is no correlation between programming or cryptography experience and performance in secure code generation. Even inexperienced programmers can create secure implementations if the used API is good, or even experienced programmers can introduce security gaps if the API is too complex. The focus should be on making cryptographic systems in general as usable as possible.

All in all, it can be seen that the core problem of many security flaws is the high complexity of cryptographic APIs which are too difficult to use. Furthermore, missing, outdated or not enough documentation also contributes to security bugs. Therefore, an alternative with high usability is needed which helps the programmers to generate secure code as easily as possible. This alternative to solve the mentioned problem can be the SCC which will be explained in more detail in the following chapters.

3 Standardization

In the course of the master thesis, an RFC draft as standardization for the SCC was created. It already has been submitted to the CFRG and can be found at the Internet Engineering Task Force (IETF) datatracker (see RFC draft [MT20]). The formulations and the structure of this RFC were made in cooperation with the supervisor of the master thesis, Kai Mindermann. Since an RFC draft needs a specific structure before it can be submitted, the structure and formulations were discussed with the supervisor before the submission. It contains basics like the use cases and the motivation for using the SCC as well as all necessary processes needed to ensure the main characteristics of the SCC. This includes the structure and content of the SCC files and the corresponding process to guarantee the usage of current security standards. In addition, a description of a possible interface, which will be explained in more detail in section 4, is shown.

In the following section first, the basic process of the SCC will be explained to have the fundamental knowledge for understanding the main contents of the RFC. The other sections reflect the most important contents of the RFC. Therefore, some important sections and definitions were taken over or slightly adapted from the RFC. The sections only needed for the structure of a typical RFC without important information were omitted.

3.1 General process

In the following the main idea, the goal and the general process of the SCC will be shown. This description is based on the work of Mindermann and Wagner [MW19b].

For writing secure code it is often necessary to make a lot of different parameter choices (e.g. block mode or key length). For their choices, programmers rely on the tutorials or documentation they find on the web. The problem is that these sources are often outdated and therefore proposing insecure parameters. It would be much better to include a secure parameter set inside the cryptographic library itself such that no parameters must be chosen by the programmers themselves. One additional problem can be found in the needed updating process of these parameters. For a correct decryption for example the same parameters as in the encryption must be used. With updated parameters, a correct decryption would not be possible any longer. To solve this problem different layers to guarantee compatibility could be used, but it would be much easier to store the used parameters in addition to the actual outcome (e.g. ciphertext) of the cryptographic primitive. This can be realized by the SCC. The general process can be described as follows:

1) In a defined interval of time cryptographic specialized institutions like IETF, National Institute of Standards and Technology (NIST) or the Bundesamt für Sicherheit in der Informationstechnik (BSI) are deciding which parameters and algorithms are matching the current security standards for different security levels. These agreed on configurations are recorded inside the SCC
2) The SCC will be published at a defined official source in human and machine readable format
3) Default configurations of cryptographic libraries will be updated with the help of the machine readable format of the SCC
4) From the published SCC, companies can derive their own configurations according to their own needs

This process is illustrated in the following figure 3.1:

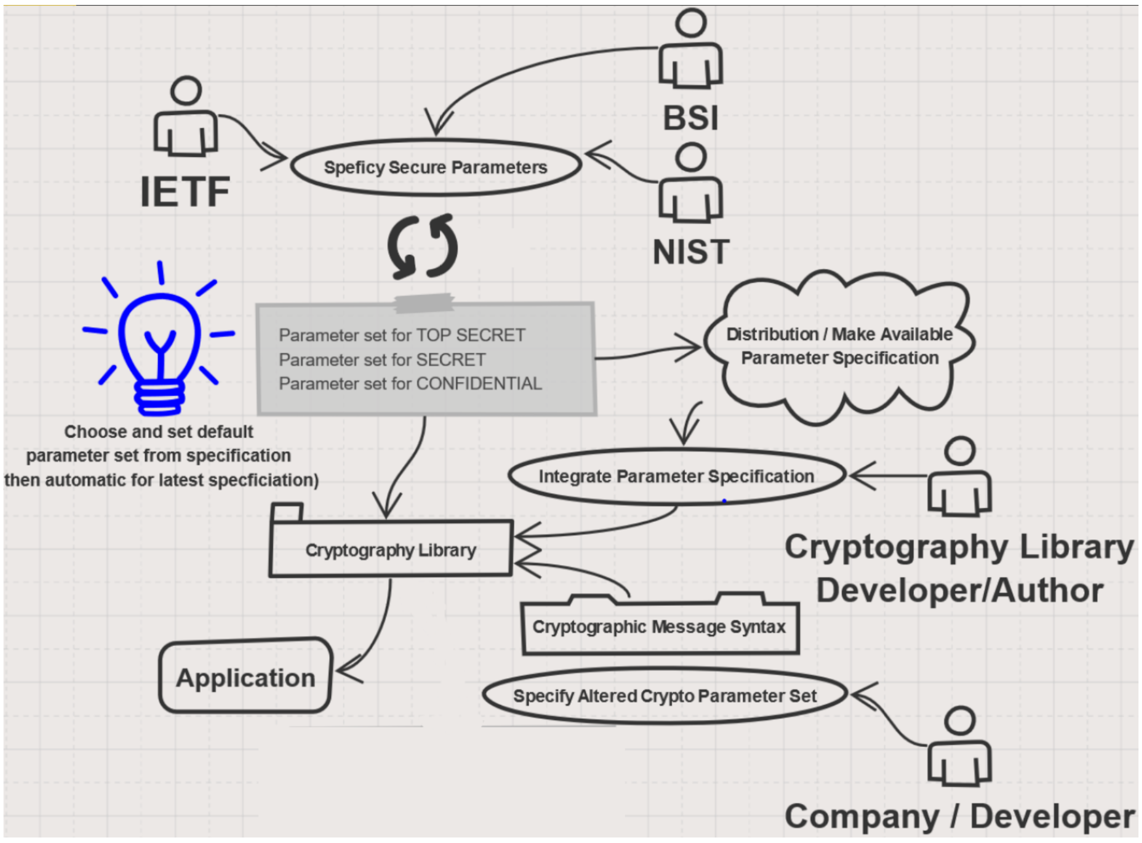

BSI
IETF
Speficy Secure Parameters
NIST
Parameter set for TOP SECRET
Parameter set for SECRET
Parameter set for CONFIDENTIAL
Distribution / Make Available Parameter Specification
Choose and set default parameter set from specification then automatic for latest specficiation)
Integrate Parameter Specification
Cryptography Library
Cryptography Library Developer/Author
Cryptographic Message Syntax
Application
Specify Altered Crypto Parameter Set
Company / Developer

Figure 3.1: Process of the SCC from Mindermann and Wagner [MW19b]

3.2 Use Cases

Secure Crypto Config Use Cases
The SCC has the following main use cases:

- Centralized and regularly updated single source of truth for secure algorithms and their parameters for most common cryptography primitives and use cases
- Both machine and human readable format to specify the above-mentioned cryptography algorithm and parameter configuration. The format is also extensible to allow others (e.g. governmental or commercial organizations) to define their own set of cryptography configurations
- Standardized cryptography API that uses the SCC for the selection of the most recent cryptography configurations and also uses a standardized cryptography operation output format to enclose the chosen parameters needed for up- and downward compatibility

Cryptography Use Cases
The SCC should support the most common cryptographic primitives. A cryptographic primitive describes in this context a generic building block that is used in the area of cryptography. These supported primitives are:

- Symmetric Encryption
  One of the most important cryptographic use cases. In contrast to the asymmetric primitive the encryption and decryption is much faster. Both the encryption and decryption is done with the same public key
- Asymmetric Encryption
  Besides symmetric encryption, one of the most important use cases. The encryption is done with a public key, the decryption with a private key
- Hashing
  Often used as part of other cryptographic use cases (e.g. password derivation)
- Password Hashing:
  To be able to store passwords safely, it is necessary to perform hashing. For password hashing the time needed to perform the hashing process is elementary. When performing password hashing, the hashing process must be very slow to prevent attackers from brute-force attacks or the guessing of the passwords from leaks. Therefore, only specific algorithms are suitable for password hashing
- Digital Signatures
  Based on the principle of asymmetric encryption. A private key is used to generate a signature that can be validated with the help of a public key
- Key Generation
  A key is needed in most of the cryptographic primitives. Therefore, it is necessary to generate keys that can be used securely

3.3 Requirements and assumptions

3.3.1 Requirements

In the following, all requirements are listed that regard the SCC or the SCCI.

General Requirements

- Interoperability with other standards/formats (e.g. [RFC8152])
- The SCC should cover most common cryptography primitives and their currently broadly available and secure algorithms
- The SCC should be protected against attackers as defined in section 3.4.2
- The SCC should prevent non-experts to configure cryptography primitives in an insecure way
- The SCC should not prevent experts from using or changing all parameters of cryptography primitives provided by a cryptography library/API

Security Level Requirements

- The SCC should define different security levels. E.g. information has different classification levels and longevity. Additionally, cryptography operations could not or much slower be performed on constrained devices, which should also be handled with the security levels. For each security level the consensus finding process and entities shall publish a distinct SCC

Consensus Finding Process and entities

- The SCC must be renewed regularly
- The SCC must be renewable on-demand
- There must be a guideline on which entities must agree to publish a new SCC
- There must be a guideline on which entities may participate in the consensus finding process and how they may do so
- There must be a guideline on how to determine broad availability of both cryptography algorithms and chosen parameters

Publication Format and Distribution Requirements

- General
  - The SCC must be easily publishable by the committee
  - Standardized unique and distinct names for (1) cryptography algorithms (2) their parameters and (3) the combination of the algorithm with set parameters. Otherwise, ambiguity would make it harder for developers and cryptography implementors to make correct and secure choices
  - There must be a versioning that allows distinguishing between different SCCs

  - There must be a deprecation process that ensures usage of outdated or insecure SCC cases
  - There must be an official source where this SCC is maintained and can be obtained from (e.g. via the WWW)
  - The official source format of the SCC must be cryptographically protected to ensure its integrity and authenticity
  - Other published formats derived from the source format (e.g. for human readability on a webpage) do not have to be cryptographically protected but should be generated automatically from the source format
  - The official source should also provide information about the SCCI that should be utilized for the application of the SCC
  - The SCC must specify how it can be extended (e.g. more security levels) and how derivatives work
- Human readable
  - The SCC must have a human readable format
  - The SCC must allow non-experts to find secure cryptography algorithms and appropriate parameters for common cryptography use cases
  - The SCC human readable publication format should only use easy to comprehend data structures like two-dimensional tables
- Machine readable
  - Cryptography libraries, regardless of the programming language, should be able to directly map (without extensive parsing) the SCC to their implementation
  - Must be easy to verify which SCC is used/was used (e.g. in Continuous Integration platforms)
  - Must be easy to verify the authenticity of the SCC (e.g. is this really what the CFRG has published?)

Cryptography library integration requirements

- Easy to integrate by cryptography libraries
- Experts should still be able to use or access the unaltered output of cryptographic primitives
- Recommendation what should be the default SCC for a cryptography library (e.g. should it be the one with the highest security level?)
- Recommendation what should a cryptography library do if it cannot support the parameters specified in the latest SCC. (E.g. the key size for Rivest-Shamir-Adleman (RSA) would be n^2 and the library supports only n)
- Recommendation on how a cryptography library should integrate the SCC so that it is up to date as soon as possible after a new SCC has been published

3.3.2 Assumptions

The SCC assumes that both the proposed algorithms and the implementations (cryptography libraries) for the cryptography primitives are secure. This also means that side-channel attacks are not considered explicitly. It is also assumed that programmers, software engineers and other humans are going to use cryptography. They are going to make implementation choices without being able to consult cryptography and security experts and without understanding cryptography related documentation fully. This also means that it is not considered best practice to assume or propose that only cryptography experts (should) use cryptography (primitives/libraries).

3.4 Security Levels

The SCC must be able to provide a secure parameter set for different security levels. These security levels depend on the following security constraints:

- Information classification (Secret, Confidential)
- Longevity (less than one day, more than a day)
- Constrained devices (constrained, not constrained)

They are defined in the following section 3.4.1 in more detail. The SCC provides 5 common security levels for which official algorithm/parameter choices are published.

3.4.1 Level description

Security Level 1 - Low: Confidential information, regardless of the other two constraints
Security Level 2: Secret information, less than one day longevity, constrained device
Security Level 3: Secret information, less than one day longevity, non-constrained device
Security Level 4: Secret information, more than a day longevity, constrained device
Security Level 5 - High: Secret information, more than a day longevity, non-constrained device

3.4.2 Security Level Constraints

Information Classification
Information classification within this document is about the confidentiality of the information. Not all information is equally confidential, e.g. it can be classified into different classes of information. For governmental institutions usually, three classes are used: Confidential, Secret, or Top Secret. The SCC considers only Confidential and Secret for its standardized security levels. Further levels with other classifications can be added by other organizations. Additionally, in common (non-governmental) use cases data is not labeled with an information class. Hence, often only one class is chosen for the cryptography processing of all data.

The SCC does not endorse a definition of the information classes, yet Secret information is to be considered to have higher confidentiality requirements than Confidential information.

Longevity
The time how long information has to be kept confidential can influence cryptography parameters a lot. Usually what you talked about with your friends should be kept confidential for a lifetime. Yet, a public trade transaction must only be confidential until the trade was executed which can happen in milliseconds. It directly influences a very important attacker resource: The time an attacker has to try to gain access to the confidential information. The SCC considers only two ranges of longevity for its standardized security levels: short longevity of less than one day and long longevity of a day or more than a day. Further levels with other longevity levels can be added by other organizations.

Constrained Devices
For cryptography often complex computations have to be executed. Yet, not all environments have the same hardware resources available. E.g. it is not always the case that the used processors have dedicated cryptography hardware or even specialized execution units or instruction sets like described by Gueron [Gue10]. Detailed discussion and definitions can be found in the RFC [RFC7228]. Yet, their definitions are too concrete to be used in the SCCs standardized security levels. Therefore, the SCC uses defines constraint devices not based on concrete processing power (e.g. 100k instructions per second):
A device is constrained when it has multiple orders of magnitudes fewer resources than a current (not a new one, but broadly in use at the time of publication of a SCC!) standard personal computer.

For example, if a current standard personal computer can encrypt with 1 GiB/s, constrained devices would be all devices that can only perform the same cryptography operation with less than 10 MiB/s. Resources can be everything important for cryptography like dedicated cryptography hardware, instruction sets, memory, power consumption, storage space, communication bandwidth, latency etc. The SCC considers only constrained and non-constrained for its standardized security levels. Further levels with other constrained resource definitions can be added by other organizations.

Attacker resources and capabilities
The SCC considers only the following same attacker resources and capabilities for all standardized security levels:

- The attacker knows all used algorithms and parameters except secrets according to Kerckhoffs's principle
- The attacker has access to the system used for cryptography operations and can utilize its cryptography operations apart from obtaining secrets
- The attacker can utilize very high-performance computing resources such as supercomputers and distributed computing (e.g. this includes very high memory, storage, processing and networking performance)

Further security levels with other attacker definitions can be added by other organizations.

3.5 Consensus finding process and entities

To provide a SCC, it is necessary to agree upon a secure and appropriate cryptographic parameter set for each security level (see section 3.4.1). This must happen in a common consensus finding process that takes place during regular intervals. The consensus finding process is based on the established RFC process during which the SCC Working Group decides in cooperation with the

CFRG and other institutions like the BSI or the NIST for a set of secure parameters. After the successful decision, the agreed on parameters can be added in the defined publication data structures (see section 3.6.5) and provided on the repository platform. There is a regular process and an emergency process to release SCCs.

3.5.1 Requirements for selection of cryptography algorithm and parameters

The SCC must only propose cryptography algorithms and parameters that fulfill the following requirements:

- Cryptography algorithms and parameters have stable implementations in at least two different programming languages
- Cryptography algorithms and parameters have a defined standard to store the algorithm and parameter identification alongside the result (e.g. like described in the RFC [RFC8152]). This is required to ensure cryptography operation results can be processed even if the default parameters have been changed or the information has been processed with a previous version of the SCC
- Cryptography algorithms that support parametrization to adapt to increased brute-force attack performance and to allow longevity of the algorithm especially for hardware optimized implementations

The SCC should only propose cryptography algorithms and parameters that fulfill the following requirements:

- Cryptography algorithms and parameters are defined in a globally accepted standard which was subjected to a standardization process
- Cryptography algorithms and parameters are licensed under a license that allows free distribution

3.5.2 Entities

Entities that make proposals should have significant cryptography expertise. Entities that participate in the consensus finding of new secure parameter sets must have significant cryptography expertise. Cryptographic expertise is defined by the SCC Working Group or the CFRG.

3.5.3 Regular process

Consensus must be found two years after the last consensus was found. This ensures that there is a new SCC every two years, even if the configuration itself has not changed.

The process has three phases that MUST be finalized within 2 years:

1) One year **Proposal phase** during which all participating entities must propose at least two cryptography algorithms and parameters per cryptography use case per security level

2) Six months **Consensus finding phase** during which all participating entities must agree on a common SCC
3) Six months **Publication phase** ensures the publication of the final SCC and allows the SCCI and other cryptography implementations to integrate the necessary changes

During the Proposal phase, the proposed algorithms and all necessary parameters should be submitted in table form for each security level and defined cryptographic use case as proposed. This table format is simply structured and is easy to read by humans as the Consensus finding phase can only be done manually. It is important that the parameters for each cryptographic use case (depending on its security level) can be found easily by the participants of the consensus finding process such that it is possible to get to an agreement faster.

3.5.4 Emergency process

In cases when a regular still valid SCC would become insecure regarding either a proposed algorithm or a proposed parameter choice it must be revised with the following process:

1) Determine the insecure configuration
2) Remove the insecure configuration
3) Publish the revised SCC with a new patch version
4) Mark the old (unrevised) SCC as deprecated

An applied emergency process results in the problem that currently used SCCI versions are no longer up-to-date because they are still supporting the no longer secure algorithms. Therefore the corresponding algorithms need to be marked as insecure. If e.g. a proposed algorithm gets insecure this can be marked inside the corresponding SCC Internet Assigned Numbers Authority (IANA) registry entry as no longer proposed to make the users aware of its insecurity. The Working Group itself can decide when to alter the SCC IANA registry.

3.6 Publication format and distribution

In general, the SCC is published via JSON files in an official repository. The SCC also utilizes IANA registries.

3.6.1 Versioning

The SCC is regularly published in a specific year. Therefore, the SCC format must use the following versioning format: **YYYY-PATCH**. YYYY is a positive integer describing the year (using the Gregorian calendar, and considering the year that has not ended in all time zones, cf. Anywhere on Earth Time) this specific SCC was published. PATCH is a positive integer starting at 0 and only increasing for emergency releases.

3.6.2 Naming

The naming of official released SCCs must follow this format:

SCC_**Version**_LEVEL_**Security Level Number**

E.g. a SCC for Security Level 5 release in 2020 the first time (so no patch version) would be named: *SCC_2020-00_LEVEL_5*.

The naming of files is not regulated, only the content is standard relevant. Yet, the SCC files should use the mentioned naming convention as well as adding a suffix (file type ending) *.json* to prevent ambiguity and remove implementation choices:

SCC_**Version**_LEVEL_**Security Level Number**.json

3.6.3 Secure Crypto Config IANA registry

The SCC requires one IANA registry with the following columns:

- SCC release version: YYYY-PATCH
- Distinct Algorithm-Parameter-Identifier that uniquely identifies the cryptography algorithm and the parameters
- Distinct and constantly available reference where all parameters are unambiguously defined
- (Optional) Short description of the parameters

Algorithm-Parameter-Identifier: MUST only consist of uppercase alphanumeric characters and underscores. Depending on the use case, the Algorithm-Parameter-Identifier can be constructed differently. We propose the following schemes:

- For symmetric encryption the name should look like:
  AlgorithmName_Mode_Padding_KeyLength_TagLength_NonceLength
  (e.g. AES_GCM_NoPadding_256_128_128)
- For hashing as:
  HashAlgorithmName_KeyLength
  (e.g. SHA3_256)
- For asymmetric encryption and digital signatures
  AlgorithmName_AuxiliaryAlgorithm_Padding_KeyLength
  (e.g. RSA_ECB_OAEP_4096)

A possible IANA registry for the SCC could look like this:

SCC Version	AlgParam ID	References	Description
2020-01	AES_GCM_256_128_128	[RFC8152]	AES 256 with GCM and 128 bit tag and random nonce

Table 3.1: SCC IANA registry

3.6.4 Utilized algorithm registries

The SCC can only propose cryptography algorithms and parameters that have been standardized. Therefore, it refers to the following IANA registries:

- CBOR Object Signing and Encryption (COSE)[1]
- AEAD Algorithms[2]
- Named Information Hash Registry[3]

Used registries must define all required parameters for an algorithm to implement it without ambiguity. E.g. implementations must not be able to choose other parameter values for a cryptography algorithm and parameter combination.

3.6.5 Data structures

For each defined security level, a distinct SCC JSON file must be provided. These files must adhere to the common schema that is shown in figure 3.2 and described in the following. The general idea for the data format and its components was based on the one described in the RFC [RFC5698] for the Data Structure for the Security Suitability of Cryptographic Algorithms (DSSC). The RFC [RFC5698] describes a structure for defining the security suitability of cryptographic algorithms that then enables the automatic analysis of them. Some fields that are needed inside this defined structure was also necessary and suitable for the SCC structure. Other existing fields were unnecessary for the SCC and were neglected. The SCC, therefore, contains only some fields that can be found in the DSSC and other additional ones that cannot be found in the DSSC structure.

- SecurityLevel: Contains the number of the corresponding Security Level of the SCC
- PolicyName: Contains the name of the corresponding SCC according to the naming schema defined in section 3.6.2
- Publisher: Contains an array of all parties that participated in the consensus finding process
  - name: Name of the participating party

[1] https://www.iana.org/assignments/cose/cose.xhtml

[2] https://www.iana.org/assignments/aead-parameters/aead-parameters.xhtml

[3] https://www.iana.org/assignments/named-information/named-information.xhtml#hash-alg

  - URL: Put in the official URL of the named publisher

- Version: Contains version in the format defined in section 3.6.1
- PolicyIssueDate: Date at which the SCC was published in the format: YYYY-MM-DD
- Expiry: Date at which the SCC expires in the format: YYYY-MM-DD
- Usage: Contains an array of objects for each cryptographic use case defined in section 3.2
  - For each cryptographic use case, at least two agreed upon algorithms (see section 3.5) with necessary parameters are included. Each of these algorithms with its parameters is specified with its unique identification name defined in an IANA registry used by the SCC

This format allows custom algorithm/parameter definitions both by overwriting use cases completely or by adding only specific algorithm identifiers via custom configurations.

```
{
    "PolicyName": "SCC_SecurityLevel_Security Level Number",
    "Publisher": [
        {
            "name": "Publisher name",
            "URL": "URL corresponding to publisher"
        }
    ],
    "SecurityLevel" : "Security Level Number",
    "Version": "YYYY-Patch",
    "PolicyIssueDate": "YYYY-MM-DD",
    "Expiry": "YYYY-MM-DD",
    "Usage": {
        "SymmetricEncryption": [
            "Algorithm 1",
            "Algorithm 2"
        ],
        "AsymmetricEncryption": [
            "Algorithm 1",
            "Algorithm 2"
        ],
        "Hashing": [
            "Algorithm 1",
            "Algorithm 2"
        ],
        "PasswordHashing": [
            "Algorithm 1",
            "Algorithm 2"
        ],
        "Signing": [
            "Algorithm 1",
            "Algorithm 2"
        ]
    }
}
```

Figure 3.2: General JSON format

3.6.6 Human readable format

The SCC cannot only be used automatically but also provide the cryptography algorithms and parameters for humans. The human readable format must be derived from the JSON files both to protect from copy-paste-errors and to validate the cryptographic signatures. Yet, the human readable format or publication page itself must not be cryptographically protected. There should be one accessible resource, e.g. a webpage, where the source format (JSON files) is automatically used for displaying them in appropriate ways (e.g. tables with various sorting and searching options).

3.6.7 Official Secure Crypto Config repository

Location
The needed SCCs should be published at an official Github repository. There, all current versions will be provided during the interval of the Publication phase (see section 3.5.3). Additionally, all previously published files are still stored at this location even if new versions are published.

Format of SCC Repository
The SCCs are expected to be in any folder hierarchy below the folder *configs*-folder. Each JSON file should be accompanied by corresponding signature files that have the same filename without extension as the JSON file, suffixed by *signatureX* where *X* is a counter starting at 1.

```
scc-repo
- configs
  - 2020
    - 00
      - SCC_2020-00_LEVEL_1.json
      - SCC_2020-00_LEVEL_1.signature1
      - SCC_2020-00_LEVEL_1.signature2
      - SCC_2020-00_LEVEL_2.json
      - SCC_2020-00_LEVEL_2.signature1
      - SCC_2020-00_LEVEL_2.signature2
      - SCC_2020-00_LEVEL_3.json
      - SCC_2020-00_LEVEL_3.signature1
      - SCC_2020-00_LEVEL_3.signature2
      - SCC_2020-00_LEVEL_4.json
      - SCC_2020-00_LEVEL_4.signature1
      - SCC_2020-00_LEVEL_4.signature2
      - SCC_2020-00_LEVEL_5.json
      - SCC_2020-00_LEVEL_5.signature1
      - SCC_2020-00_LEVEL_5.signature2
    - 01
    - 02
  - 2021
  - 2022
  - 2023
  - 2024
```

```
scc-repo
- configs
  - a
      - b
      - 0c1
      - ReallySecure
  - 0x1111
    - SCC_2020-00_LEVEL_1.json
    - SCC_2020-00_LEVEL_1.signature1
    - SCC_2020-00_LEVEL_1.signature2
    - SCC_2020-00_LEVEL_2.json
    - SCC_2020-00_LEVEL_2.signature1
    - SCC_2020-00_LEVEL_2.signature2
    - SCC_2020-00_LEVEL_3.json
    - SCC_2020-00_LEVEL_3.signature1
    - SCC_2020-00_LEVEL_3.signature2
    - SCC_2020-00_LEVEL_4.json
    - SCC_2020-00_LEVEL_4.signature1
    - SCC_2020-00_LEVEL_4.signature2
    - SCC_2020-00_LEVEL_5.json
    - SCC_2020-00_LEVEL_5.signature1
    - SCC_2020-00_LEVEL_5.signature2
  - asdf
  - afd
  - af
```

Figure 3.3: Example for SCC repository content (left) and SCC repository with custom naming scheme (right)

Integrity/Signing process
Each SCC JSON file should be accompanied by at least two signatures. Both signatures are stored in different files on the same level as their corresponding SCC file to reduce the parsing effort. The signatures should be generated by entities defined by the SCC Working Group. They are responsible to publish and renew the used public keys. For signing of the corresponding SCC JSON files, *openssl*[4] could be used. The public keys needed for validation are published in the official repository of the SCC.

3.7 Secure Crypto Config Interface

This section describes the programming interface that provides the application of the SCC. The SCCI is generic and describes the API that should be used by each programming language. The general idea of the SCCI as a layer between the SCC and the already existing library functionalities is visualized in figure 3.4. See section 4 for implementation details of a created SCCI in Java.

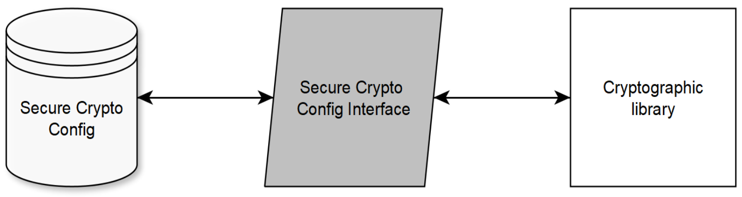


Figure 3.4: General idea of the SCCI

3.7.1 Semantic versioning

The implementation of the SCCI must follow *Semantic Versioning*[5], which specifies a version format of **X.Y.Z** (Major.Minor.Patch) and semantics when to increase which version part. It would be beneficial if the release of a new interface version gets synchronized with the publication of a new SCC. It should be possible to support the newly defined parameters of SCC in the interface as soon as possible.

3.7.2 Deployment of (custom) Secure Crypto Config with Interface

There are two different possibilities to work with the SCC:

- The preferred option is to use the SCCs that will be delivered within the interface. In each new interface version, the current SCCs will be added such that always the latest SCCs at the time of the interface release will be supported. Previous SCCs will remain inside the interface such that also older ones can still be used

[4] https://www.openssl.org/
[5] https://semver.org/

- Another option is to define a specific path to your own/derived versions of the SCCs with the same structure of the files as described in section 3.6.5 but with other values than in the official ones

The interface will process the SCCs as follows:

1) Check if the path to the SCCs is a valid one
2) Check if the *configs* folder exists
3) For each folder following configs in the hierarchy look inside that folder and check the existence of JSON files that need to be considered. This check will happen recursively for all folders inside the hierarchy
4) For every JSON file found, look if there exists a signature. If one is given, check if the signature is valid for the corresponding file
5) Every file with a valid signature will be parsed and further processed

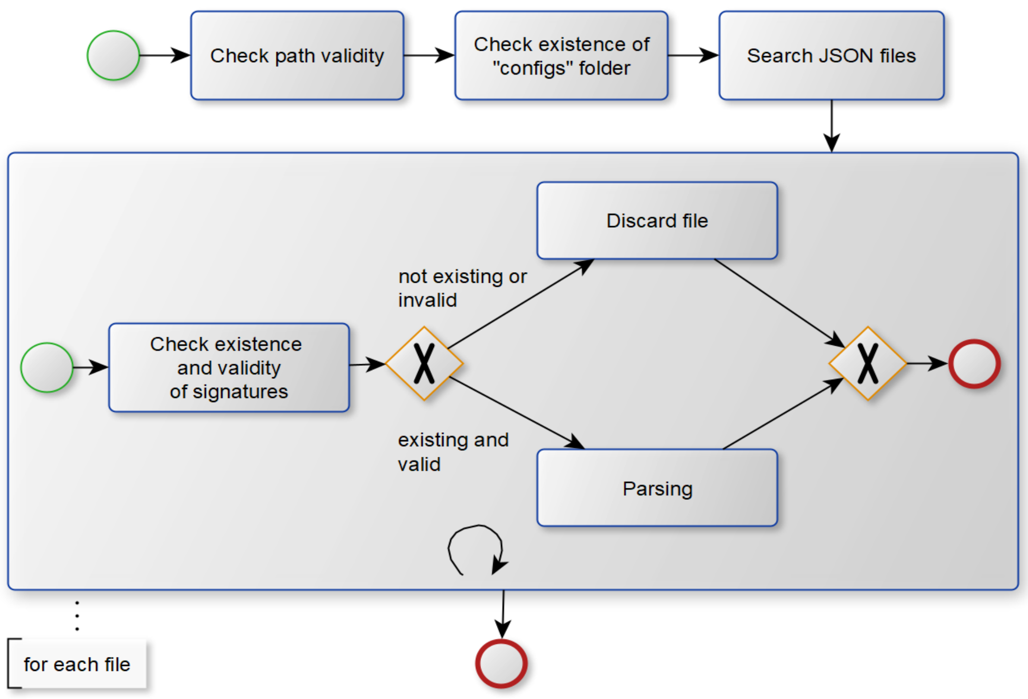


Figure 3.5: Processing of SCCs

The parsing of each valid JSON file must be done as follows:

1) Read out all information. The information of each file is stored in a corresponding object. With this procedure, all JSON files need to be read only once which will contribute to the performance
2) Parsing of security level: Check if it is a positive integer. All files not containing a (positive) integer number as security level value will be discarded
3) Parsing of the version of all files: All files with values in the wrong format (see section 3.6.1) will be excluded from further processing.
4) Parsing of algorithm identifiers: Only the algorithm identifiers that are supported by the interface will be considered and stored inside the corresponding object. The supported algorithms are specified inside the interface (e.g. with an enum)
5) From all SCC files find the latest (according to version) with the highest appearing security level (determined in the previous step). The path to this file will be used as the default path used for each cryptographic use case if nothing else is specified by the user. If two or more files with identical levels and version number are found, only the first one will be used, others are discarded

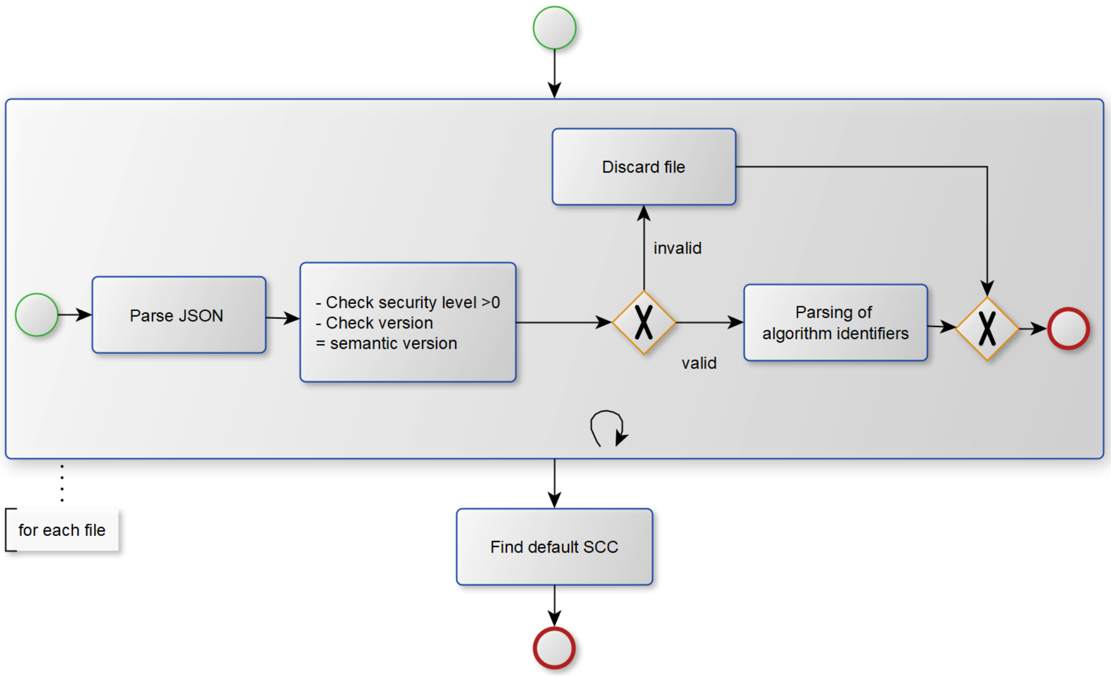


Figure 3.6: Parsing process

The unique algorithm identifiers for the execution of a specific cryptographic use case will be fetched from the corresponding object (representing the JSON file determined beforehand) at the time the user invokes a method for a specific cryptographic use case. The Interface will also provide a possibility to choose a specific algorithm (out of the supported ones) for executing the desired use case. In this case, the specified algorithm is used. The identifiers will be compared with the supported ones in order of their occurrence inside the file. If one matching identifier is found, it will be used for execution. If it is not a matching one, the value will be skipped and the next one will be compared. If none of the algorithms inside the selected SCC can be found an error will occur.

3.7.2.1 Delivery of Secure Crypto Config with interface

Each SCCI must be published in such a way that it uses (a copy of) the recent SCC repository.

The SCC will be stored inside the subfolder *scc-configs* which should be located in the interface *src*-folder if existent. The structure of the scc-configs folder will be the same as in the described hierarchy of the GitHub repository. In any new version of the interface, the latest published SCC and its signatures must be used.

If new SCCs will be published for which no published version of the Interface is available, the custom repository approach can be used as described in the following.

3.7.2.2 Using a custom Secure Crypto Config repository

It is also possible to use a different path to the SCCs. As also derived versions of the SCC for specific needs should be supported it will also be feasible to define a path to own or derived files that differentiate from the default *src/configs* folder. In this case, a method for setting and using a specific path must be provided by the interface.

3.7.2.3 Integrity check

The check for the valid signature of the SCCs is always made before every actual usage of the interface functionalities. In this way, it is possible to guarantee that the entity using the interface only works with valid SCCs and circumvents the risk of forged file contents. The public key needed for validity can be found in the official GitHub repository. If own derived SCCs are created then it can be possible that no validation process is needed for these files.

3.7.3 Application Programming Interface (API)

Methods and Parameters
Intended methods and parameters included in the interface are described in listing 90 for a Java interface example.

Supported Algorithm Parameter Types
Cryptography algorithms require different parameters. The SCCI considers the following types of parameters:

- Parameter Size (e.g. key length in bit)
- Parameter Counter Content (e.g. nonce)

- Parameter Secure Random Content (e.g. nonce)
- Parameter User Defined Content (e.g. plaintext and key for symmetric encryption)
- Parameter Compound Parameter Content (e.g. counter + random = nonce)

Output of readable Secure Crypto Config
A SCCI must offer the following additional methods regarding the configuration:

- A method that returns a human readable version of the currently used SCC
- A method that returns the currently used cryptography algorithm and parameters for a given use case
- A method that validates the content of a SCC JSON file and one or more signatures

3.8 Cryptography library implementation specification

Cryptography libraries should provide the above mentioned SCCI. Until a common cryptography library provides the SCCI itself, there should be wrapper implementations that provide the SCCI and make use of the programming languages' standard cryptography library. Such an implementation in Java can be seen in section 4.

3.9 Cryptography algorithm standards recommendation

When new cryptography algorithm and/or parameter/mode/etc. standards are created, they should contain a section mentioning the creating of the proposed secure parameter sets in the above mentioned IANA registries. This ensures that new cryptography algorithms and parameter sets are available faster for the SCCI implementations to use.

3.10 Security considerations

Consensus Finding
Only trustworthy and cryptographic specialized entities should participate in the publication process of the SCC. Otherwise, a SCC with a weak and insecure parameter set could be provided.

Publication Format
The operators of the SCC must ensure that potential unauthorized parties are not able to manipulate the parameters of the published SCC. Countermeasures to this are in place by utilizing gits gpg[6] signatures and integrity as well as signatures for the published SCC files as well.

[6] https://gnupg.org/

Cryptography library implementation

- Integrity must be ensured if potential users want to fetch the provided SCC from the corresponding platform over the network e.g. by using a signatures
- Users should only trust SCC issued from the original publisher with the associated signature. Users are responsible to verify the provided signatures

Special Use Cases and (Non-)Security Experts
The SCC does not apply to all use cases for cryptography and usage of cryptography primitives. It is meant to provide secure defaults for the most common use cases and non-expert programmers. Additionally, non-experts may still implement vulnerable code by using the SCC. Yet, it should reduce the vulnerabilities from making the wrong choices about parameters for cryptography primitives.

Security of Cryptography primitives and implementations
The SCC assumes that both the proposed algorithms and the implementations (cryptography libraries) for the cryptography primitives are secure as long as they are used with the correct parameters, states and orders of function calls.

Security Guarantees
The SCC makes the best effort to be as up-to-date with recent discoveries, research and developments in cryptography algorithms as possible. Following this, it strives to publish cryptography algorithms and corresponding parameter choices for common use cases.

Yet, the SCC and the involved parties working on and publishing it do not guarantee security for the proposed parameter configurations or any entity making use of it. E.g. a new algorithm that can do brute-force attacks exponentially faster could be existing or published right after the publication of the most recent SCC was published itself.

Threat Model / Adversaries:
There are different possibilities in which a potential adversary could intervene during the creation as well as after the publication of the SCC. These attack scenarios must be considered and prevented.

- *Process*
  During the creation process, selected institutions must agree on a secure parameter set. It could be possible that one party wants to influence this process in a bad way. As a result, it could be agreed on weaker parameter sets than originally intended
- *Publication*
  After the publication of the SCC a potential attacker could gain access to the provided files on the corresponding platform and change the content to an insecure parameter set
- *Content*
  Depending on the distribution method of the SCC, it is also possible that an attacker could change the content of the SCC as a man-in-the-middle. Especially if an http connection is used to obtain the SCC, this will be a serious problem

3.11 Current state

The RFC draft described in the previous sections has already been submitted and some feedback could already be received. There was a comment of a woman (owning a website to explain cryptographic terms) who was very interested in the concept of the SCC. In a digital meeting, the concept of the SCC was explained to her in more detail. Further, the SCC was also accepted as a topic in the 109th IETF meeting and was presented there. Of course, in the future further iterations of the created RFC will be needed to get a detailed and approved standardization for the SCC.

Inside the previous sections the concept of the SCC was described in a standardized way. The next step is to look at the actual usage of the SCC. Therefore, the next chapter will describe the implemented SCCI in more detail.

4 Secure Crypto Config Interface

One of the main tasks was to implement a SCCI in Java in order to show that the concept of the SCC is suitable for daily use. In this section, all important details about this implementation will be given. First, the general idea of why a SCCI is needed will be explained. Afterward, the structure of the created Java project will be described. Furthermore, it will be shown how the SCCI can be distributed, how the compatibility aspect can be guaranteed and in which way the SCC files are parsed. Next, the generated documentation of the interface will be described and finally, an explanation for the handling of SCC files and the specification of the algorithms to use will be given.

4.1 General idea

The general goal for the future would be that cryptographic libraries are providing the SCCI itself. But as this is not the case yet and a direct change in existing libraries is not easy possible a layer between the SCC and the functions provided by cryptographic libraries is need. This can be realized with the help of the SCCI as additional layer in between. This can be imagined as shown in figure 4.1:

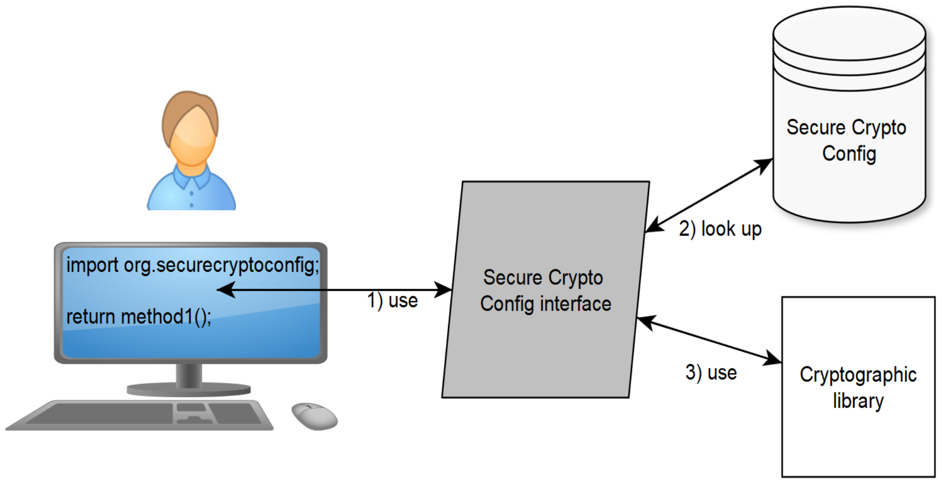


Figure 4.1: Process of working with the SCCI

The SCCI works as follows: the interface provides different methods to its users. The methods in the concrete implementation of the SCCI can be seen in A.2. The user imports the SCCI and calls the desired method. The interface encapsulates all cryptographic related details and hides this complexity from its users. Depending on the called method and the cryptographic primitive that should be realized, the unique identifier will be looked up in the SCC. According to the specified identifier it is determined which parameters and algorithms must be used to realize the cryptographic primitive securely. After this, the specific primitive and the corresponding parameters are realized with the functionalities provided from the cryptographic library (here JDK: javax.crypto, java.security). The result will be finally returned to the user.

4.2 Java project structure

In the following, the structure of the implemented SCCI will be described. For reference, the whole project can be found at [Tei20a].

4.2.1 Language selection

For the actual implementation of the SCCI the programming language Java was chosen. One reason why this language was selected is that it is widely used. This fact can be seen for example on the website [Git16]. On this website, one can see a comparison of which languages are the most used ones in GitHub repositories. The most common was JavaScript followed by Java and Python. It can be seen that Java is the second most spread language after JavaScript in GitHub which is with 50 million developers a very huge platform. This gives a first impression of how important a language is. This fact is also shown on the website [Car20]. On this source, an index for each language is determined depending on how often corresponding language tutorials are searched on Google. Here, Java is only on the second place after Python.

One point why Java was preferred to Python was the aspect of up- and downward compatibility that needs to be considered while implementing the SCCI (see section 4.4). For Java, there was already an implementation of COSE with relatively many different implemented cryptographic primitives and algorithm/parameter combinations. For Python there also exists such an implementation[1] that could be used inside the SCCI but this supports fewer primitives and combinations. All in all, it seems more convenient to work with Java instead of Python and according to the mentioned sources, Java seems to be a very common language. But of course, the aim for the future is that the SCC will be usable in all languages, not only in Java.

4.2.2 Classes

Different classes make up the SCCI. This section should give a short overview of which classes were implemented and what the main functionalities are. All classes and the corresponding interface are organized inside the package *src.main.java.org.securecryptoconfig*. The following classes are realizing the different cryptographic primitives:

[1]https://github.com/setrofim/COSE-PYTHON

SecureCryptoConfigInterface	Interface defining methods that can be used to perform different cryptographic use cases (see section A.2). Most of them are implemented inside the SecureCryptoConfig class.
SecureCryptoConfig	Starting point for performing every cryptographic use case. The class contains methods for performing symmetric or asymmetric en-/decryption, (password) hashing and signing. It implements the SecureCryptoConfigInterface. **To perform the desired use case it is only needed to create a new SecureCryptoConfig object and call the specific method.**
PlaintextContainer	Class for a container representing the plaintext processed in cryptographic use cases. A PlaintextContainer contains the plaintext as byte[] representation. The class provides various cryptography operations that can be performed on the plaintext (e.g. encryption, signing).
SCCCiphertext	Class representing a container for the ciphertext (as result of encrypting a plaintext). SCCCiphertext contains a byte[] representation of a specific COSE message. This message contains the ciphertext as well as all the parameters used during encryption. The inclusion of the used parameters, except the key, ensures that before the decryption no used cryptographic details must be specified by the user, but can parse it from the COSE message.
SCCSignature	Class representing a container for a digital Signature. SCCSignature contains a byte[] representation of a COSE message. The byte[] contains the signature as well as all the parameters used during signing.
SCCHash	Class representing a container for a cryptographic Hash. SCCHash contains a byte[] representation of a COSE message. The message contains the hash as well as all the parameters used during hashing.
SCCPasswordHash	Class representing a container for a cryptographic Password Hash. SCCPasswordHash contains a byte[] representation of a COSE message. The byte[] contains the password hash as well as all the parameters used during hashing.
SCCKey	Class representing a container of a key used for cryptography operations like symmetric or asymmetric encryption. SCCKey contains a byte[] representation of a key as well as different parameters like the type (SCCKey.KeyType) and the used algorithm for key creation.

Table 4.1: Main classes and their functionality

The next classes are only needed for the processing of the SCC file content (e.g. parsing, see section 4.6).

JSONReader	Class for handling/parsing SCC file content in JSON format.
SCCInstance	Class for SCC file processing. Needed for parsing the SCC files. The SCC file content gets transformed into a Java SCCInstance object such that the file must only be read once and is then stored in a corresponding object.
SCCInstanceUseCase	Class for SCC file processing. Needed for parsing the SCC files. Auxiliary class for SCCInstance as multiple different cryptographic primitives with different algorithms (use cases) can be specified inside the SCC.
SCCInstancePublisher	Class for SCC file processing. Needed for parsing the SCC files. Auxiliary class for SCCInstance as multiple publishers can be specified inside the SCC (see section 3.6.5).

Table 4.2: Classes needed for parsing and their main functionality

Test classes
To be able to test the main functionalities, that are previously described, there are also J-Unit classes defined in the package *src.test.java.org.securecryptoconfig*. For each cryptographic use case, there is one class in which all possible use cases are tested. Only the test for hashing and password hashing is contained within one class. Also, there are additional classes to test the PlaintextContainer and the JSONReader functionalities. With the help of this, it should be possible to take a look if all implemented functionalities work as intended.

4.2.3 Provided functionalities

All methods that can be used when working with the SCCI can be found in the *SecureCrypto-ConfigInterface* interface. All provided methods can be seen in section A.2. This includes for example methods for symmetric and asymmetric en-/decryption, signing and validation, hashing and password hashing.

4.3 Usage

In the future, the SCCI will be made available via dependency management systems (e.g. Maven, Gradle). In this way, it will be easily possible for developers to use the provided functionalities of the SCCI with minimal effort. The usage of dependency management systems is the simplest way to distribute the SCCI and thus make the use of defined secure default parameters available to everyone. At the moment the SCCI is not available via dependency management systems. The SCCI is only provided as Java ARchive (Jar) that can be imported inside the corresponding project. This fact should be changed in the future.

4.4 Compatibility with COSE

One important aspect while implementing the SCCI was the up- and downward compatibility. This means that for example encryption, which is carried out with certain parameters, has to use exactly the same parameters again for a correct decryption. To guarantee this, even if the standard encryption algorithms adapt over time with the changing security standard, the corresponding parameters must be saved in addition to the actual output (e.g. ciphertext) of the cryptographic primitive.

This can be realized with an already existing standard in form of COSE (see RFC [RFC8152]). There are also other already existing standards like Cryptographic Message Syntax (CMS) (see RFC [RFC5652]) that defines a specific syntax for storing cryptographically related content. As there is already an existing Java project[2] realizing COSE, the usage of this standard (based on version 1.1.0) was chosen. The implemented classes of this project were used in the implementation of the SCCI to realize the compatibility aspect. COSE in general is a standardized format on how to store all necessary parameters used in a cryptographic primitive besides the actual output.

One problem of the Java COSE realization is that not all needed combinations of algorithms and parameters are implemented at the moment. If a specific combination of algorithm and parameter (e.g. specific salt-length) should be used inside the SCCI, it could be the case that it was not implemented in regard to COSE. Also, at the moment, not all cryptographic primitives that should be supported in the SCCI are implemented within the used COSE project. The implementations of hashing, password hashing and asymmetric encryption are missing. To be able to use COSE even in the case of missing parameter configurations and cryptographic primitives, the Java COSE project was adapted and new classes were added to realize missing cases with COSE that are needed for the SCCI. Classes adapted and added to the already existing COSE project could be found inside a specific folder *COSE* inside the SCCI implementation (see [Tei20a]). Classes that are added for digital signature and asymmetric encryption are based on the already implemented classes that are realizing symmetric encryption in the context of COSE.

By deciding to use the COSE standard for compatibility reasons and its already existing implementation for Java, a dependency between the possible algorithms and parameters to provide in the SCCI and the already implemented algorithms and parameters appeared. If a specific combination of algorithm and parameter should be supported also an implementation of this configuration with regard to COSE must be made. For example, if the SCCI should support asymmetric encryption with RSA then a COSE message storing all necessary information must be created to guarantee a correct decryption with the same parameters in the future. This specific message then contains beside the actual outcome in form of the ciphertext, for example, the used algorithm and padding which will later be used for decryption. If there is no support of RSA inside the implementation of COSE then it is not possible to create the described message because this implementation is

[2] https://github.com/cose-wg/COSE-JAVA

responsible for generating COSE messages. Therefore, it would be beneficial if a cooperation between the developers of the COSE realization and the SCCI could be possible in the future to solve the mentioned problem.

Because of the additional use of COSE it is assumed that the compatibility aspect of the SCCI is better than in other Java cryptography libraries like Google Tink or JDK (java.security/javax.crypto). With the help of COSE up- and downward compatibility can be guaranteed as additional parameters used for a cryptographic primitive are stored beside the actual outcome. This makes it easy to e.g. decrypt something with the right parameters even if the security standards and corresponding algorithms are changing. If for example a symmetric decryption should be performed then the COSE message generated during the encryption process is needed. Inside this message, all necessary information is read out (e.g. ciphertext, algorithm, key length). This information will be used for the correct decryption. Without this additional information stored inside the COSE message, it can not be reconstructed how a specific ciphertext was created and therefore, a correct decryption can not be guaranteed. In the other libraries, only the actual outcome of the cryptographic primitives is returned, which would make up- and downward difficult if the corresponding parameters are not stored properly. However, this statement was not further evaluated during the thesis.

4.5 Supported algorithms

The SCCI should support all common cryptographic primitives (see section 3.2). For each supported primitive (Symmetric/Asymmetric en-/decryption, Signing, (Password) Hashing) different realization based on the unique algorithm identifiers (see section 3.6.3) including different secure parameter sets are supported. The supported algorithm identifiers, the corresponding COSE identifier and its realization in Java can be seen in section A.3.

For adding new algorithms that should be supported by the interface the following methods must be done:

1) Add the desired unique algorithm identifier to the enum *SCCAlgorithm* inside the *Secure-CryptoConfig.java*.
2) Define the new algorithm identifier also inside the method realizing the corresponding cryptographic primitive. E.g. a new identifier for symmetric en-/decryption must be specified as choice inside the method *encryptSymmetric*.
3) As there is no collaboration between COSE and the SCCI yet it is also necessary to define the realization of the identifier in Java inside the corresponding COSE classes (e.g. for symmetric en-/decryption inside *EncryptCommon.java*). The identifier must also be added to the *AlgorithmID.java*. This is only necessary if COSE does not provide an implementation for the new desired identifier.

4.6 Secure Crypto Config parsing

The detailed parsing process of the SCC files was already mentioned in section 3.7.2. Inside the concrete SCCI *jackson*[3] was used for parsing the JSON SCC files. *Jackson* is a library for processing contents of JSON files in Java easily. Every valid file found in the hierarchy was only read once and was transformed into a Java object (SCCInstance) such that the later processing is faster. Which classes are responsible for the parsing process can be seen in table 4.2 as already described.

4.7 Documentation

The documentation is beside the general complexity of the API one of the most important aspects as already described by Acar et al. [ABF+17]. Further, by Mindermann and Wagner [MW18] it is shown that the usage of cryptographic code examples has a positive effect on the effectiveness and the security aspect of the resulting code. Therefore, a detailed documentation with example code for the SCCI was generated. All provided methods and created classes were described within Javadoc comments. Also, a detailed package description that lists the general characteristics and the usage of the SCCI was created. The documentation also contains coding example showing how specific methods can be used. An html version of the documentation can be found at a specific branch[4] inside the GitHub repository that contains the implementation of the SCCI. As a starting point to take a look at the documentation the *package-summary.html*[5] can be used. This documentation was also improved during the processing time of the master thesis with the help of the feedback from the conducted studies (see section 5).

In addition, a user guide was created that should show potential users how the SCCI must be used and how specific cryptograhic primitives can be realized. This guide also contains many coding examples to make it as easy as possible to get started. This guide is added as *README.md* to the SCCI project (see [Tei20a]).

4.8 Handling of Secure Crypto Config files

As already described the internal execution of a cryptographic use case is done according to SCC files. These files contain for each supported use case unique algorithm ids which represent specific algorithm and parameter choices. By default, this information is parsed out of the SCC files which are provided within the interface. But it is also possible to give a custom path to own (derived) versions of the SCC files. This can be done with the *setCustomSCCPath(Path path)* method by simply giving the path to the directory as a parameter. However, the internal structure of the own files must be similar to the original ones (see section 3.6.5).

[3] https://github.com/FasterXML/jackson

[4] https://github.com/secureCryptoConfig/secureCryptoConfigInterface/tree/gh-pages

[5] https://github.com/secureCryptoConfig/secureCryptoConfigInterface/blob/gh-pages/org/securecryptoconfig/package-summary.html

By default, the files provided by the interface are used for parsing. There are files for different security levels. The higher the security level of a file the more confidential the data must be handled. By default, the interface will use the algorithm ids from the most recent (according to its version) file with the highest security level. If the user wants a specific file to be parsed the *setSCCFile(String policyName)* method with the corresponding PolicyName of the desired file as a parameter can be used. The PolicyName is defined inside each file (see section 3.6.5). Also one can use the *setSecurityLevel(int level)* method. As a result, the most recent file with the specified security level number will be used.

The PolicyName of the SCC file used for processing can be shown with *getUsedSCC()*. If the user has previously set a specific file for usage or a custom path it is also possible to go back to the default settings with the *setDefaultSCC()* method. All the mentioned methods for SCC handling are defined inside the *SecureCryptoConfig* class.

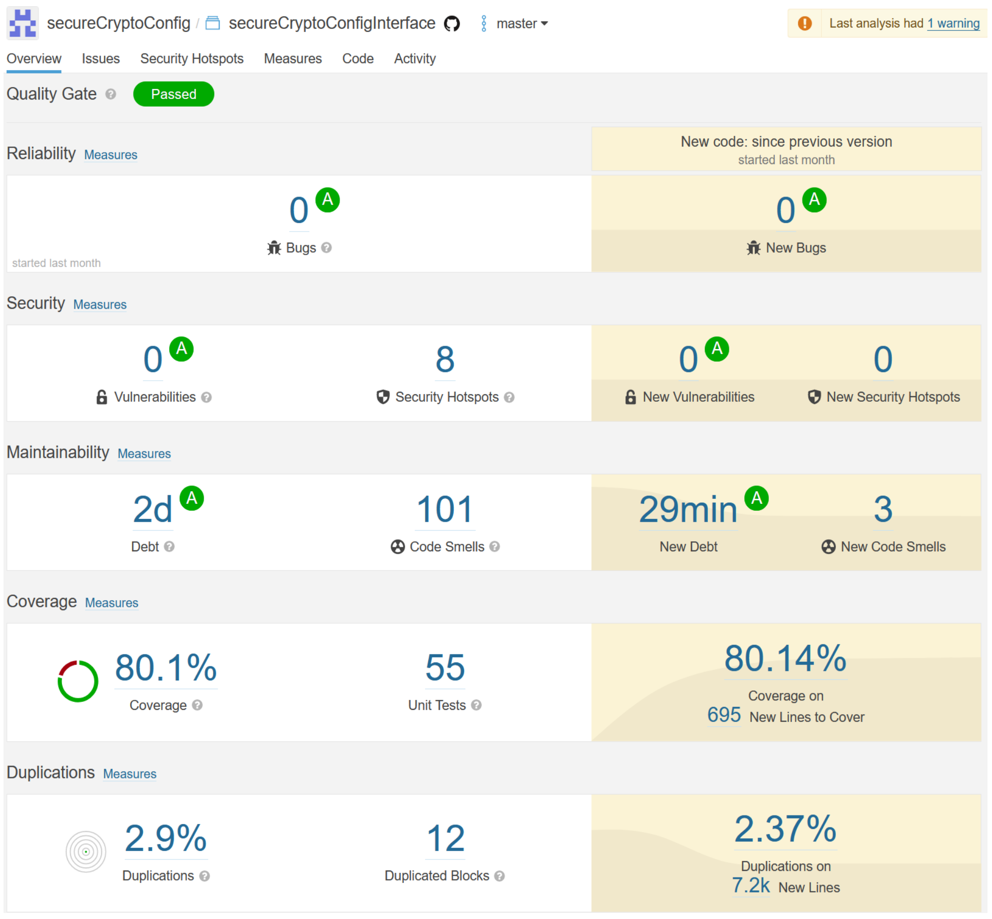
4.9 Specification of algorithms

By default, the algorithm used for executing the specified cryptographic use case is determined by the currently used SCC file. It is also possible to choose a specific algorithm from all the supported ones with the method *setAlgorithm(SCCAlgorithm algorithm)*. *SCCAlgorithm* contains the unique algorithm identifiers of all currently supported algorithms for all use cases. To be able to perform a specific use case (e.g. hashing) a suitable algorithm identifier must be chosen. This algorithm will be used for all further invoked methods. If the default choice of the SCC should be used call *defaultAlgorithm()* or for changing to a specific algorithm call *setAlgorithm(SCCAlgorithm algorithm)* again. All the mentioned methods for the specification of a specific algorithm are defined inside the *SecureCryptoConfig* class.

4.10 Quality assurance

To ensure the quality of the SCCI code, a connection to SonarCloud[6] was created for the associated GitHub repository. With every new push into the repository, SonarCloud creates an analysis of the existing code in the background. It examines reliability (bugs), security (vulnerabilities), maintainability (debt), coverage (through unit tests) and duplications. All in all, SonarCloud should improve the code quality. Different quality gates can be set to meet specific quality goals. In this case, these are: reliability-, maintainability-, security rating is A, duplicated lines $< 3\%$ and coverage $\geq 80\%$. The code of the project was adapted to meet these established quality gates. The SCCI satisfies the previously mentioned quality gate. The following figures are showing the concrete values of the SCCI for each aspect measured by SonarCloud (see figure 4.2) and the concrete coverage measurement of each SCCI class (see figure 4.3).

[6] https://sonarcloud.io/



secureCryptoConfig / secureCryptoConfigInterface master

Last analysis had 1 warning

Overview Issues Security Hotspots Measures Code Activity

Quality Gate Passed

New code: since previous version
started last month

Reliability Measures

0 A Bugs		0 A New Bugs	

started last month

Security Measures

0 A Vulnerabilities	8 Security Hotspots	0 A New Vulnerabilities	0 New Security Hotspots

Maintainability Measures

2d A Debt	101 Code Smells	29min A New Debt	3 New Code Smells

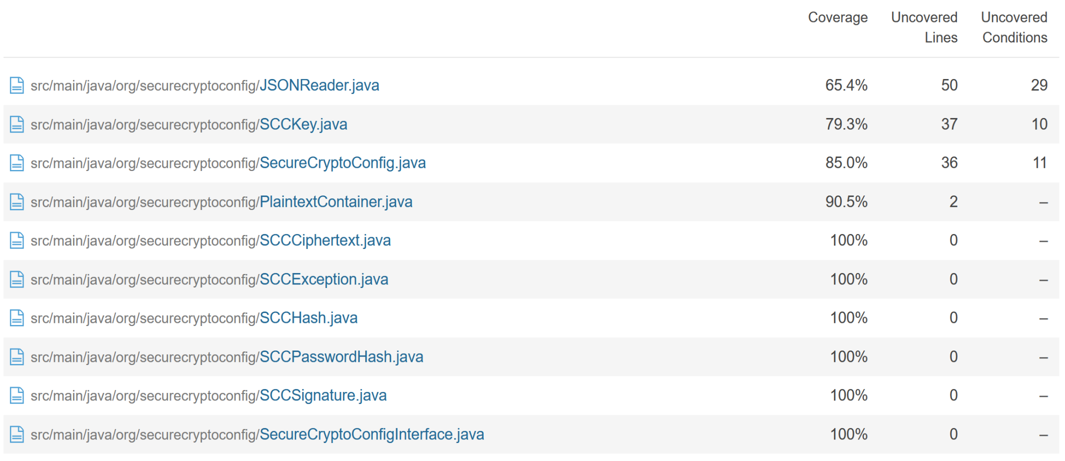
Coverage Measures

80.1% Coverage	55 Unit Tests	80.14% Coverage on 695 New Lines to Cover

Duplications Measures

2.9% Duplications	12 Duplicated Blocks	2.37% Duplications on 7.2k New Lines

Figure 4.2: SonarCloud measurements of the SCCI



	Coverage	Uncovered Lines	Uncovered Conditions
src/main/java/org/securecryptoconfig/JSONReader.java	65.4%	50	29
src/main/java/org/securecryptoconfig/SCCKey.java	79.3%	37	10
src/main/java/org/securecryptoconfig/SecureCryptoConfig.java	85.0%	36	11
src/main/java/org/securecryptoconfig/PlaintextContainer.java	90.5%	2	–
src/main/java/org/securecryptoconfig/SCCCiphertext.java	100%	0	–
src/main/java/org/securecryptoconfig/SCCException.java	100%	0	–
src/main/java/org/securecryptoconfig/SCCHash.java	100%	0	–
src/main/java/org/securecryptoconfig/SCCPasswordHash.java	100%	0	–
src/main/java/org/securecryptoconfig/SCCSignature.java	100%	0	–
src/main/java/org/securecryptoconfig/SecureCryptoConfigInterface.java	100%	0	–

Figure 4.3: SonarCloud coverage measurement of each SCCI class

5 Evaluation

In this section, the evaluation of the implemented SCCI will be described. In the following, first, the goals and hypotheses are explained. Afterward, the study design and the structure of the conducted pilot and main study will be described. Furthermore, the created tasks and questions of the studies are explained. Finally, the results will be investigated.

5.1 Goals and Hypotheses

5.1.1 Hypotheses

The quality of a software product is composed of different aspects. According to the International Organization for Standardization (ISO) 25010 quality model, 8 different aspects can be considered (see [ISO25010]), which make up the quality of the software product. This product quality model can be illustrated as follows:

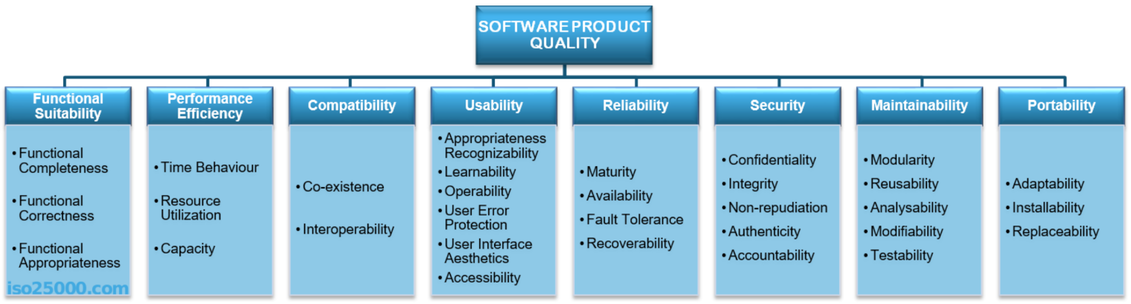


Figure 5.1: Product quality model from source [ISO25010]

Based on this quality model the SCCI was investigated for its usability, security and maintainability aspects to evaluate the quality of the implemented interface. The goal was to reject or validate the following established hypotheses. The index of these hypotheses is composed as follows: A - alternative, T - Tink, J - JDK, U - usability, S - security, M - maintainability.

General:

- H_A: The Secure Crypto Config Interface is better in terms of usability, security, maintainability than the stand-alone standard cryptographic library (JDK : java.security/javax.crypto) and Google Tink

Usability:

- H_{AJU}: The Secure Crypto Config Interface is more usable than JDK
- H_{AJT}: The Secure Crypto Config Interface is more usable than Tink

Security:

- H_{AJS}: The Secure Crypto Config Interface is more secure than JDK
- H_{ATS}: The Secure Crypto Config Interface is more secure than Tink

Maintainability:

- H_{AJM}: The Secure Crypto Config Interface is more maintainable than JDK
- H_{ATM}: The Secure Crypto Config Interface is more maintainable than Tink

These hypotheses were evaluated upon the study results. The general H_0 hypothesis is:

H_0: The Secure Crypto Config Interface is less or equal usable/secure/maintainable than library x

Since the aim was to investigate different aspects of the SCCI in comparison to already existing libraries, only hypotheses comparing the SCCI with another library (JDK or Tink) were established and investigated. Comparisons between JDK and Tink were therefore not further compared.

5.1.2 Library selection

As the implementation of the SCCI was made in Java (see section 4.2.1) it was necessary to look for other Java cryptography libraries to be able to compare them to the SCCI. Therefore, possible libraries needed to provide at least the same main cryptographic primitives as the SCCI. First, it was important that the SCCI is compared to the default Java cryptography library. Therefore, one library considered for the evaluation was the stand-alone standard cryptographic library JDK (java.security/javax.crypto, Java-SE 10). The consideration of this library should give an expression if the SCCI is better than the Java default library. It should be determined if the SCCI is better in the terms of usability, security and maintainability than the standard.

Further, it should be identified if the SCCI is even better than a more advanced library than the standard one. One library that claims in their documentation to be more usable was Google Tink. JDK does not make any claims about its usability aspect. With the help of the comparison between the SCCI and Google Tink (version 1.4.0) the aim was to look if the SCCI is not only better than the standard library but maybe even better than a more improved library.

5.2 Organization and Overview

5.2.1 Organization

The evaluation took place in two parts. First, a pilot study with only four selected participants out of a known circle was conducted. The first pilot study serves as a test of the designed tasks and questionnaire. It also shows possible problems in the functionality of the developed interface. Thus, all possible complications both in the understanding of the tasks and in the usage of the interface can be eliminated for the main study. After this, the main study is conducted with as many participants as possible and will eventually validate the hypotheses. The main study is similar to the pilot study, but with some changes based on the pilot study feedback.

5.2.2 Pilot study

The pilot study was conducted as a *Think aloud* study. That means, that the participants were supervised during the conduction. Expressed comments during the study were audio-recorded. This kind of study was chosen because additional information about the feelings of the participants while using the different cryptographic libraries could be collected. Also, it is possible to see and hear what kind of questions from the questionnaire were unclear or which formulations from the programming tasks must be adapted. This information would also be feasible to gain with questions inside the questionnaire, but with the help of *Think aloud* studies, it is possible to probably get aspects that were not considered beforehand and also get direct and live feedback from the statements or behavior of the participants. The participants should come from different areas in order to get a wide range of feedback from different perspectives.

5.2.3 Main study

The main study was not conducted as *Think aloud* study because more participants were planned. In general, the evaluation of results from Think aloud studies is very high. As in the pilot study, only a few participants were planned, the effort for doing this kind of survey was manageable, but for more participants it would be inconvenient. Therefore, the main study was conducted as *online* study. This online study was accessible for all possible participants via a special link. This link was then provided over different channels to reach as many participants as possible. To convince as many as possible to participate, a raffle among the participants was advertised.

5.2.4 Overview

The whole process of the planned evaluation as described in the previous sections can be summarized as follows:

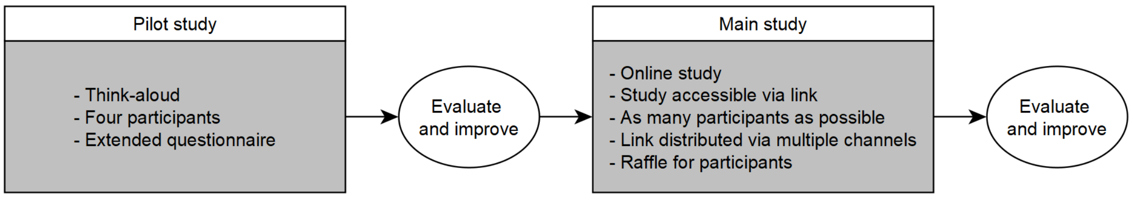


Figure 5.2: Process of the planned evaluation

5.3 Study design

The evaluation was carried out by conducting a survey created with *limesurvey*[1]. Only one survey (containing the same questions and programming tasks) for all cryptographic libraries under test was designed. As cryptographic libraries, JDK (java.security/javax.crypto), Google Tink and of course the developed Secure Crypto Config Interface were considered. For each of these libraries, there is a block with the same questions and one programming task. The specific programming tasks must be performed only with the help of the corresponding library of the block. All of the three blocks are contained within the same survey. In this way, the participants do not have to switch between different surveys and the answers can be mapped more easily to one participant. The general structure of the survey can be illustrated as follows:

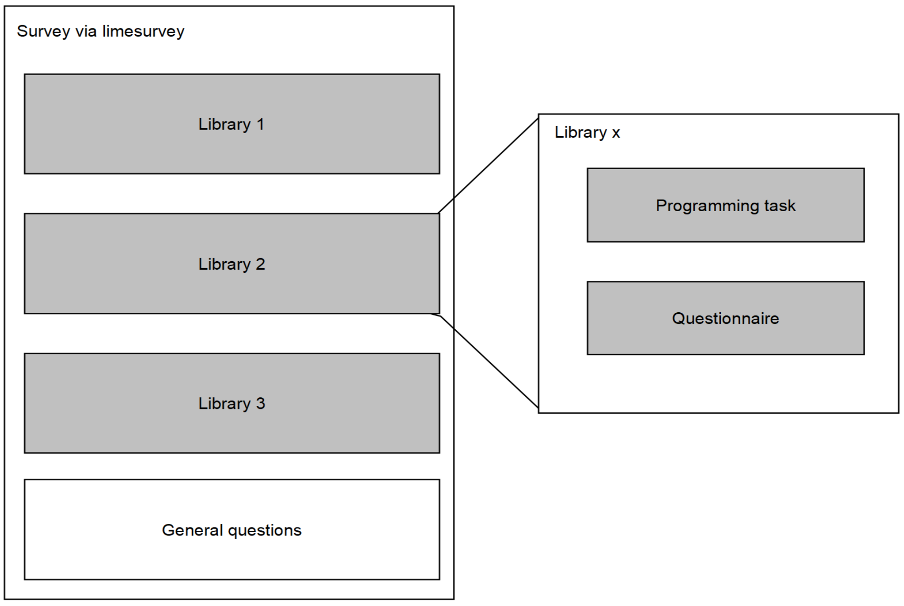


Figure 5.3: General survey structure

[1]https://www.limesurvey.org/de/

It was planned to let the participants conduct at least the block of JDK and the SCCI. In this way, it would be possible to compare the self-implemented SCCI with the default cryptographic library of the JDK. Depending on the estimated time the participant would need to solve the tasks, the block of Google Tink was planned as optional. The whole time for execution should not be higher than one hour. The order of the first two obligatory library blocks was JDK followed by the SCCI. Google Tink was the last optional block.

5.3.1 Programming task

The final programming task in the version as shown in the main study can be found in section A.4.1. In the pilot study, a previous version of this formulation was used. According to the feedback of the pilot study (see section 5.5.2), the task formulation was slightly adapted to the version that was used inside the main study. The same programming task was prepared for the three different libraries (JDK, Tink, SCCI). The only difference was the links to the specific library documentation and the library name in the description of what library to use.

As a programming task, a small Java application was developed. The goal was to create a small application that makes use of different cryptographic primitives. In this way, it should be avoided to let the participants only implement these primitives without context. Also, this should be beneficial for the motivation of the participants because they can see possible use cases of the cryptographic primitives. By creating the programming task, it was also important that the project gives some kind of output to the participants such that they can see the correctness of their implementations. They should also have a good feeling of motivation after each completed task.

To give the participants a feeling for the specific library, more than one primitive was included in the programming task. But as the time for conducting the study should not be too huge for the participants, it was decided to include two different primitives. The participants should implement symmetric en-/decryption and signing/validation.

The structure of the programming task was as follows:
First, the programming task is explained and then a link to a *GitPod*[2] will follow in which the participants have the opportunity to solve the programming task in the browser-based workspace. This should make the effort for the participants less because they do not have to import the project in their local IDE. The usage of GitPods is free but needs an account on GitHub, GitLab or BitBucket. If a participant does not have one of these mentioned accounts, he also had the possibility to download the programming task in one of the defined GitHub repositories containing the needed files. Therefore, for each library block one repository was created to be able to provide the code to the participants (see JDK: [Tei20b], Tink: [Tei20d], SCCI: [Tei20c]). In addition, there are also repositories that contain the solution for each library to be able to compare the results of the participants afterward more easily. These repositories with the solution were private to guarantee that the participants cannot find the solution on the internet.

[2]https://www.gitpod.io/

5.3.2 Questionnaire

First, different questions from Wijayarathna et al. [WAS17] were chosen and added to the survey. These questions are selected out of the so-called *Cognitive Dimension Questionnaire*, that covers different aspects to rate the usability of security-specific applications. Because Wijayarathna et al. [WAS17] considers a very huge number of questions in their provided questionnaire only a smaller selection of them was chosen to minimize the time effort for the participants. This selection is depicted in section A.4.3. Because, of timing reasons that could be seen during the pilot study (see section 5.5.2), these questions were only considered in the pilot and not in the main study questionnaire. These questions should give additional feedback for different usability aspects of the corresponding libraries.

In addition, the System Usability Scale (SUS) was used to collect usability metrics in the pilot and main study. To use this metric a specific set of questions (see work by Klug [Klu17]) using a five-point Likert scale (see section A.4.2) was included.

The next question block (for both studies) also contains statements that should be rated with a five-point Likert scale from the participants. These questions should give some feedback about the perceived security and maintainability aspects. These questions can be found in section A.4.4. The first two questions are considering the perception of the participants about the security aspect of the used library. The last three questions should help to get an expression of the maintainability aspect of the library. These questions cover the main aspects that maintainability consists of testability, modifiability and modularity (see figure 5.1).

One last block was added to get some additional information about the participants and the demographics in general (see section A.4.5). The whole structure of the questionnaire is shown in figure 5.4.

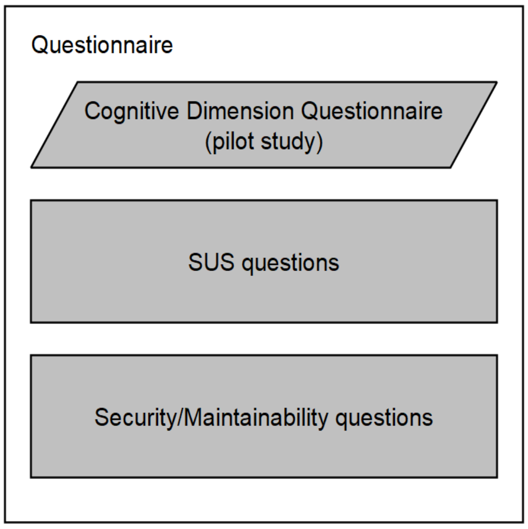

Questionnaire
Cognitive Dimension Questionnaire (pilot study)
SUS questions
Security/Maintainability questions

Figure 5.4: General questionnaire structure

The only difference in the questionnaire between pilot and main study was the exclusion of the selected questions by Wijayarathna et al. [WAS17].

5.4 Evaluation of aspects

5.4.1 Usability

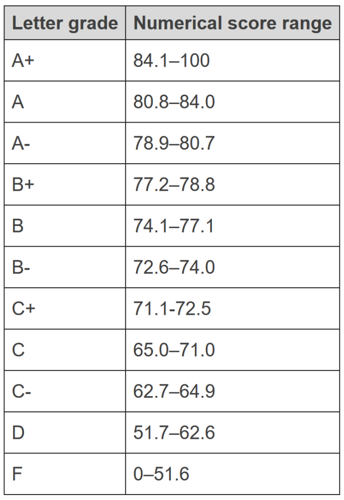
To test the usability of the mentioned cryptographic libraries, the advice of Wijayarathna and Arachchilage [WA18a] was followed. They propose the following procedure:

1) Designing tasks to employ programmers
2) Recruiting participants and conducting the evaluation
3) Collecting feedback from participants
4) Identifying usability issues from participants' feedback

For the pilot study, different questions from the work of Wijayarathna et al. [WAS17] with their Cognitive Dimension Questionnaire were chosen to get an impression on how usable the libraries are perceived. In addition, the study design of the pilot study as *Think aloud* gives the possibility to gain an even better impression. Further, the SUS was used to collect usability metrics. The SUS is the main scale for the main study to evaluate the usability aspect as it is conducted as *online* study and without the questions of the Cognitive Dimension Questionnaire [WAS17].

The SUS can be determined, according to Klug [Klu17], with the following formula:

$$2.5 \cdot (\sum(\text{odd numbered questions rate } -1) + \sum(5 - \text{even numbered questions rate}))$$



Letter grade	Numerical score range
A+	84.1–100
A	80.8–84.0
A-	78.9–80.7
B+	77.2–78.8
B	74.1–77.1
B-	72.6–74.0
C+	71.1-72.5
C	65.0–71.0
C-	62.7–64.9
D	51.7–62.6
F	0–51.6

Figure 5.5: Interpretation of SUS with corresponding grades from Klug [Klu17]

According to Klug [Klu17] a value between 65.0 – 71.0 is an acceptable value every value above is a very good result. The rating of the questions can be determined with the help of the five-point Likert scale. Every question rated with *Strongly disagree* gets the value of 1. Answers with *Strongly agree* get the value of 5. The questions in between are getting a value between 2 and 4. The interpretation of the SUS is illustrated in figure 5.5.

5.4.2 Security

For evaluating the security aspect, the most important part is to look at the implemented code of the participant and look for possible security flaws. For this purpose, the static code analysis tool Xanitizer[3] with a Open Source Project License was used. There are also other static code analysis tools like FindSecBugs (based on SpotBugs[4]), but this one could not be used for the evaluation of the SCCI and Tink. FindSecBugs has no capability to look for security flaws in used libraries and JARs of an application, but this would be necessary to show that there is no security bug when using the SCCI or Tink (imported as JAR in the own application).

Also, some questions (see section A.4.4, questions 1, 2) were formulated which should show how the participants perceive the security aspect of the library.

5.4.3 Maintainability

The perceived maintainability is evaluated by formulated questions from the questionnaire that can be seen in section A.4.4 (questions 3-5).

To get an additional impression on the actual maintainability of the implemented code SonarQube[5] was used. SonarQube defines the maintainability by looking at the number of code smells and the technical debt (measured in days/hours). Depending on the determined debt, a maintainability rating is calculated by SonarQube (A (best) to F (worst)). With the help of this, it should be possible to get an impression not only about the perceived but also the actual maintainability of the resulting code from the programming task.

5.4.4 Hypotheses

After the different ratings for the previously mentioned aspects are determined, it will be necessary to validate or reject the established hypotheses (see section 5.1.1). To be able to do this the following procedure will be performed: First, it will be determined with the help of the Shapiro test whether the data samples for the investigated aspect are normally distributed. A value in this test ≥ 0.05 means normally distributed a value of < 0.05 shows that there is no normal distribution. According to this, either a T-test (normally distributed) or the Wilcoxon test (not normally distributed) will be performed to calculate a corresponding p-value. With this p-value it can be determined if the H_0 hypothesis (Null-Hypothesis) can be rejected in favour of H_A (Alternative-Hypothesis) or not. A required confidence level of 95% (required significance level 0.05) was specified. This means that if the determined p-value is < 0.05 the data suggest to reject the H_0 hypothesis in favour of H_A. The result is significant and the hypothesis can be validated. With a p-value > 0.05 it is not possible to either reject or validate the H_0 hypothesis. In this case, the result is not significant and no assumptions can be made.

[3] https://www.rigs-it.com/xanitizer/

[4] https://spotbugs.github.io/

[5] https://www.sonarqube.org/

5.5 Pilot study analysis and interpretation

5.5.1 Participants

For conducting the pilot study four participants were invited and a virtual meeting was planned with them to perform the *Think aloud* study. The known participants all had a university degree (at least bachelor). One was still student, two were working in the industry and one at the university. Therefore different kinds of perspectives could be taken into account. None of them had specific knowledge in cryptography and no one was working especially in the field of security. The participants were 26 up to 59 years old (average: 34.5). Three of them had a general programming experience of more than five years and only one of less than one year, but only one of them had a job with a focus on programming.

5.5.2 Feedback

The following feedback is a summary of oral feedback but also from the questions of the Cognitive Dimension Questionnaire. The detailed evaluation of the different considered aspects usability, maintainability and security will be described in section 5.5.5 up to 5.5.7. For timing reasons (further described later on) only the SCCI could be considered in the pilot study evaluation.

Problems

1) Time
   The conduction of the survey with the first participant took more time than expected. To not exceed the planned execution time of one hour too much, it was decided to execute only the block about the SCCI as this was the most important thing to evaluate. The timing factor was also very long during the other trials. Therefore, all of them only made the block of the SCCI. It was seen that the survey must be adapted such that the effort for conducting the study is not too large in the main study. As a result, the number of questions was adapted for the main study. Because of the time aspect, the questions from the work of Wijayarathna et al. [WAS17] were excluded from the main study like already describe in section 5.3.2.

2) Format and spelling
   There were also a few spelling and grammatical aspects that were pointed out by the participants. Also, three of them indicated that the captions of the describing figures were too close to the next following figure such that they thought the caption of the one figure was the headline of the next one. Further, some explaining statements on the arrows of the shown figures were missing in the understanding of the participants. These aspects were fixed in the survey version of the main study.

3) Task descriptions
   The task descriptions were in general perceived as understandable by the participants. But one problem pointed out by all participants was the missing explanation on how to check if the implemented functionality from the subtask is working or not. The classes of the application provided to the participants give back output in form of console statements which should help and show the implemented functionalities. Therefore, it was thought that it is self-describing and unnecessary to explain this output even further, but the feedback from the

participants showed that more explanations of the application output were needed and that more information on how to check the results of the participants' implementation was needed. That was also a problem that leads to the previously mentioned time problems because the participants were unsure if they implemented the tasks the right way and took too much time in looking if their results are right or not. Therefore, there was a problem with the not enough self-describing application output that must be added to the task description.

4) Exceptions
While doing the programming task, it was necessary to handle some occurring exceptions that were thrown by the used methods. All of the participants asked if they should only throw or catch the exceptions in the corresponding method. Therefore, in the main study additional comments inside the programming task itself and also inside the task description were added which shows that the exceptions should be caught in the method they are appearing.

5) Documentation
At first, no link to the library to use was provided inside the survey, because the participants were allowed to use the internet to look for explicit questions. Also in the case of the SCCI it was thought that the automatically appearing documentation inside the own IDE or GitPod would be enough. During the execution of the study and from the feedback it was seen that the participants think it would be beneficial to directly give a link to the corresponding library documentation. In this way, they can easily find explanations for all packages and methods. This would also contribute to the time factor because by adding a link to the desired documentation the participants are able to find the necessary information more quickly. As a result, in all blocks, a link to the official documentation of the library was added.
At the time of the pilot study there was already a documentation for the SCCI methods and classes, but it was seen that some class or method descriptions were not precise enough. Also, a detailed package description was required by one of the participants. In general, the documentation should be extended with concrete coding examples, which was not the case during the conduction. The mentioned points were adapted and added to the documentation.

6) GitPods
All participants decided to use the GitPods for solving the programming tasks. All of them had no experience in using them. It was seen that the handling of some GitPod features is not that self-describing. Inside the pilot study, there were no additional remarks on how the workspace of a GitPod can be used. The participants needed some time to find out where to start the main methods of a class and how to end the execution. To make it easier for them the survey was extended with remarks on how to make basic things like running or terminating inside the GitPods. This also can lead to less time effort in the conduction of the later main study.

7) Import of programming task
One of the participants pointed out that the version of the minimal needed Java version for doing the programming task inside its own IDE should be pointed out. Otherwise, the participants not using GitPods directly can run into problems during the main study.

7) Methods of SCCI
During the processing of the programming task, it could be seen that some classes and methods are not used in the intended way. For example, the SCCCiphertext constructor should transform the byte representation of an already existing SCCCiphertext in an object

of the corresponding class. The intention was that the participants should use the encrypt method to get a new SCCCiphertext, but instead the byte representation that would need to be encrypted was used as a parameter for the mentioned constructor. It could be seen that the constructors need to be replaced by methods with more expressive names and that more coding examples for the usage are needed inside the documentation.

Another problem appeared in the usage of the *toString()* method. This method is part of the Java package *Java.lang.Object*. If it is necessary to get a String representation of the self-defined objects inside the SCCI it is necessary that possible users are not using the default Java *toString()* method, but the corresponding method inside the SCCI to get the right representation. Therefore, the SCCI needs to overwrite this method. The problem for this is that it is not known which charset (e.g. UTF-8) the user wants to use beforehand. For this reason, the SCCI only defined a *toString(Charset c)* method at first. The problem was that the participants had not looked inside the documentation on how to get the right String representation and simply used the *Java.lang.Object* method by habit. It could be seen that it was necessary to overwrite this method because it cannot be deleted or circumvented in another way. In the case of overwriting the problem of the not in advance known charset remains.

Positive feedback

1) GitPods
All of the participants pointed out that the use of GitPods was a good idea. With the help of this browser-based workspace, the effort for the participants could be reduced. They find it positive that they did not have to import the application in advance before starting the actual programming task.

2) Abstraction
The participants all gave positive feedback regarding the general idea and design of the SCCI. They pointed out that it is easy to realize a cryptographic primitive without much knowledge of cryptography. One participant also stated that he thinks the usage is easier than other cryptographic libraries he had used before because by working with the SCCI he does not have to define so many parameters related to cryptography. Another also gave the feedback: "[...] all operations have been abstracted away, thus I don't need to do any decisions or changes in regard to the actual cryptographic stuff" which also supports the opinion of the previously mentioned participant.

5.5.3 Improvements for Main Study

Based on the feedback of the participants, the following adaptions were made in the survey for the main study as already described in section 5.5.2:

- Spelling/ grammatical errors were corrected.
- Figures from the programming task were adapted and some explaining statements for the arrows inside were added. The distance between the two figures was enlarged to get a better structure of the task description.
- A link to the corresponding library documentation was added in each block.

- A remark on how to handle occurring exceptions in the programming task was added inside the survey and the code.
- Some additional remarks on how to use GitPod were added.
- Questions from Wijayarathna et al. [WAS17] were removed for time reasons.
- Documentation of the SCCI was adapted with a detailed package description and additional coding example. Unclear explanations mentioned by the participants were reformulated.
- The constructors of affected classes were replaced by methods with a more expressive name showing the real intention of the corresponding method. The default *toString()* method was overwritten and marked for the users as *deprecated* such that the users can see directly that the use of this method is not recommended. The alternative method *toString(Charset c)* has remained and is called internally by the overwritten *toString()* method with the chosen default charset UTF-8.

5.5.4 Functionality

All participants were able to process the programming task completely. There were only small issues and questions the participants were asking during the processing. The main questions were asked about the usage of the GitPods that were used by all of the participants and the handling of exceptions that were raised by used methods. Also some questions about the usage of the constructor of the SCCCiphertext and the toString() method were appearing. The mentioned problems are also described in section 5.5.2. All in all, it was relatively easy for most participants to complete the programming task. This fact can possibly also be attributed to the good SUS and thus usability values which are shown in the following.

5.5.5 Usability

The usability aspect was mainly evaluated by the SUS metric as already described in section 5.4.1. The mean SUS of the SCCI was with 76.2 relatively high. According to figure 5.5 this would correspond to the grade B and is better than the average. The single values for each participant are depicted in the following table:

ID 1	ID 2	ID 3	ID 4
55	80	90	80

Table 5.1: SUS values of pilot study participants

It can be seen that the first SUS value in comparison to the other three is relatively low. This could be explained by the fact that the first participant did not get a direct link to the documentation of the library directly at the beginning. That was the case because it was thought that the automatically displayed comments within the IDE are sufficient and the documentation of the classes and methods can also be seen in the class files included in the project. Therefore, the participant had problems using the SCCI at first. As it could be seen that the participant needs more documentation the link to the corresponding documentation was given after a short period of time. The link then was also provided to the next pilot study participants and added to the main study as an improvement. This problem at the beginning could be an explanation of the comparatively low first SUS value. The following values are relatively high: between grade A+ and A-.

5.5.6 Security

The security aspect was measured with the help of the found security bug number inside the implementation of the participants and the first two question of the corresponding question block that is shown in section A.4.4 about the perceived security.

The code was analyzed with Xanitizer and no security bugs could be found inside the implementations. Security bugs resulting from missing logging in form of using *error.printStackTrace()* while catching occurring exceptions were omitted because this usage was not related to the functionality of the used library itself. This low security bug number could be attributed to the fact that the participants do not need to specify anything related to security in the usage of the SCCI and therefore can make fewer mistakes in the parameter selection. This is not the case while using JDK.

The mean perceived security values for each participants can be seen in the following:

ID 1	ID 2	ID 3	ID 4
3.5	5	4	4.5

Table 5.2: Security values of pilot study participants

For these values, the score according to the five-point Likert scale was determined for the corresponding questions, summed up and the mean was calculated. The closer the value to 5 the better is the perceived security. It can be concluded that the perceived security is relatively high. Therefore, the participants seem to have a good feeling about the security aspect of the SCCI. These good values are also seen in the not existing security bugs inside their implementations.

5.5.7 Maintainability

The perceived maintainability aspect of the SCCI was evaluated with the last three questions of the corresponding question block that can be found in section A.4.4. The ratings of the participants are depicted in the following:

ID 1	ID 2	ID 3	ID 4
4.33	4.33	4.33	4.67

Table 5.3: Maintainability values of pilot study participants

For these values, the score according to the five-point Likert scale was determined for the corresponding questions, summed up and the mean was calculated. The closer the value to 5 the better is the perceived maintainability. The values are all relatively close to the best value of 5. Therefore, the participants rated the SCCI good in terms of testability, modifiability and modularity which were covered inside these three maintainability questions.

By looking at the number of code smells found by SonarQube only some from the project itself were found. No code smells inside the implementation from the participants could be found. Therefore, the maintainability score (rated with *A*) and the technical debt was the same for all participants. This could be concluded to the fact that the code created by the participants was too short to see any differences in the maintainability with SonarQube.

5.6 Main study analysis and interpretation

The main study had the same structure as the pilot study. Only the in section 5.5.3 mentioned things were changed and improved based on the pilot study feedback.

5.6.1 Participants

There were 79 people starting the survey, but only 13 of them finished the whole survey with and without the optional part about Google Tink. As a result, it can be seen that only 16.45% of all people finished the study at least with the obligatory parts about the SCCI and JDK. Further, only 7 from the 79 (8.86%) completed also the optional part. All 66 people that have not completed the survey seem to have only read the introduction page of the survey and/or the first description of the programming task because all of them have not given any answer. Therefore, it can be concluded that they do not stop in the middle of the survey, but before even really processing it. This can be an indicator that this unfortunately low number of participants can be attributed to the high effort the participants see in the processing of the survey. In the introduction of the survey, an estimated duration of one hour was given, this could be the first reason for some participants to stop further looking at it because the timing effort seems too huge for them. It could even be the case that they further read the first programming task description and saw that the survey is more effort than usually estimated by them. Unfortunately, a survey that should compare different cryptographic libraries and their functionality need besides a questionnaire additional programming task. This leads to the fact that this kind of survey comprises more time effort than other studies only consisting of a short questionnaire. Not many people seem to be willing to participate in studies with to huge timing effort, but the content of the survey could not be minimized further as at least the testing of the implemented SCCI with one other library (in this case JDK) was needed to be able to compare the results.

Most participants were still students with a bachelor's degree (12). Only one had finished university with a master's degree. The mean age of the participants was therefore 24.8 years. Only three of the 13 participants had less than 5 years of general programming experience which can be attributed to the fact the all of them studied a computer-science based course of study and needed to program there. Also, three of the participants stated to have programming as primary job. Only three of the participants said they have a security background. As only a few of the participants have advanced knowledge in cryptography it is assumed that the participants can be classified as non-security-experts, which is the target group of the SCC/SCCI.

5.6.2 Functionality

All participants were able to complete the programming task with all processed libraries in the intended way. Only one participant tried to decrypt the wrong variable and therefore got a *Null-PointerException* during running its implementation. As in manual review of the implementation revealed, the code and all used methods were completely right and only the wrong variable for decryption (in all libraries) was presumably used by accident. It could be seen that it was no problem in the usage of the libraries. As a result, this implementation was counted as functional.

5.6.3 Usability

Analysis

The SUS metric for evaluating the usability aspect was determined as already described in section 5.4.1. The following figure shows the boxplots for all libraries and the determined SUS value of each participant:

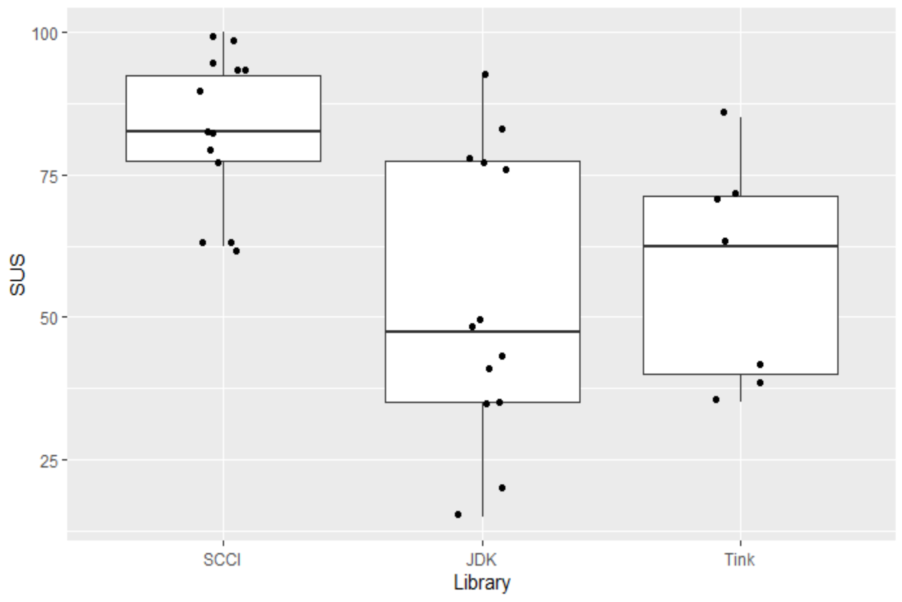


Figure 5.6: Boxplot for each cryptographic library and its SUS scores

The SUS values of the first boxplot of the SCCI are beginning at a relatively high rating. The lowest value of the SCCI was 62.5 while the ones of JDK and Tink were 15.0 and 35.0. This shows that the general rating of the SCCI started at a higher level than the other ones. The best rating of the SCCI was the full score of 100 while it was 82.5 for JDK and 85.0 for Tink. The mean SUS value for SCCI was with 82.9 (A) the highest followed by Tinks mean with 57.9 (D) and at least JDK with only 53.1 (D). This shows that the differences between the means of the libraries are relatively high. While the mean SUS of the SCCI is over average (65.0 - 71.0) the other ones are worse than the average and seem therefore not to be very usable for the participants on average. The previously mentioned values are summarized in the following table:

	SCCI	JDK	Tink
Minima	62.5	15.0	35.0
Maxima	100.0	92.5	85.0
Mean	82.9	53.1	57.9
Median	82.5	47.5	62.5

Table 5.4: Minima, Maxima and mean SUS for each cryptographic library

Further, it can be seen that the SUS values of the SCCI are less scattered than the other ones. All values are more closely together at an relatively high SUS level. In JDK and Tink there are few participants that perceive the library as good usable and the other bigger number of participants that perceives the opposite. In general it is very difficult to make concrete assumptions of Tink as only very few participants processed the corresponding part of the survey.

All in all, it can be concluded that the SCCI is rated very well in terms of usability especially by comparing it to the values of the JDK. Also, the mean SUS score is the highest for the SCCI.

With the determined SUS scores it is now possible to make conclusions according to the established hypotheses (see section 5.1.1). This will be done as already described in section 5.4.4 by first looking if the data samples are normally distributed with the Shapiro test and then by performing the T-test or Wilcoxon test for calculating a corresponding p-value. The following tables are showing the results of the calculations:

	SCC	JDK	Tink
Shapiro	0.08928	0.3237	0.386
Distribution	normal	normal	normal

Table 5.5: Shapiro test for SUS scores

	SCC/JDK	SCC/Tink
T-test, p-value	**0.0006929**	**0.007063**

Table 5.6: T-test for SUS scores

Interpretation

The H_0 hypothesis in this case is:
The Secure Crypto Config Interface is equally or less usable than JDK/Google Tink.

1) H_{AJU}: The Secure Crypto Config Interface is more usable than JDK

   With a p-value of 0.0002351 (< 0.05) the data suggest to reject H_0 in favour of H_{AJU}. As a result, it can be said that the SCCI is more usable than JDK.

2) H_{ATU}: The Secure Crypto Config Interface is more usable than Google Tink

   With a p-value of 0.01986 (< 0.05) the data suggest to reject H_0 in favour of H_{ATU}. As a result, it can be said that the SCCI is more usable than Tink.

5.6.4 Security

Perceived security - Analysis

The perceived security was determined based on the questions of the questionnaire (see section A.4.4, questions 1, 2). In the following figure, the boxplots for all libraries and the determined perceived security score of each participant is illustrated:

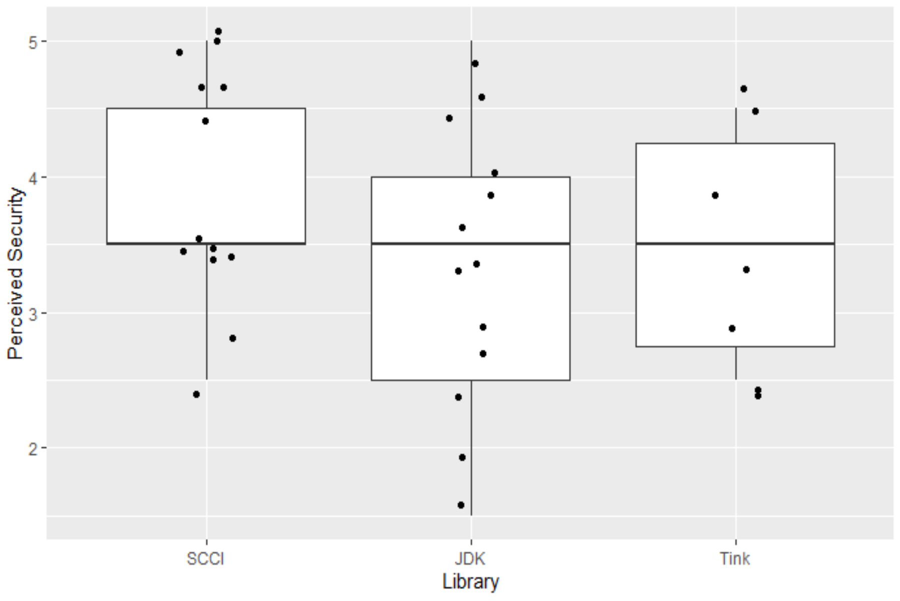


Figure 5.7: Boxplot for each cryptographic library and its Security question score

In all three boxplots, it can be seen that the security scores are widely scattered. In the case of the SCCI, the minimum can be found by 2.5 and ends with the highest possible rating of 5.0. Nearly the same can be found by looking at JDK (minimum: 2.0, maximum: 5.0) and Tink (minimum: 2.5, maximum: 4.5). By looking at the mean security scores (3.96, 3.38, 3.5) it is shown that all of them are relatively close together. The SCCI gets the highest rating followed by Tink and JDK. There seems to be no big difference in the perception of the security of the different libraries when looking at the relatively close mean values. This equally perceived security could be based on the fact that maybe too few questions for covering this aspect were asked and the participants are not informed enough how the SCCI works internally. More questions to investigate this aspect were considered, but because of the already high survey time effort, the number of questions was not increased any further. That there is no big difference in the scores can also be seen by the equal median from all of the boxplots. The previously mentioned values are summarized in the following table:

	SCCI	JDK	Tink
Minima	2.5	2.0	2.5
Maxima	5.0	5.0	4.5
Mean	3.96	3.38	3.5
Median	3.5	3.5	3.5

Table 5.7: Minima, Maxima and mean perceived security score for each library

However, in the case of the SCCI it is shown that 11 of the participants gave a value above the average of 3.0. While in JDK only 8 participants rated above 3.0. Therefore, it could be concluded that most of the participants perceived the security as positive in the usage of the SCCI in comparison to JDK. More participants would be needed to strengthen or weaken this impression.

With the determined scores, it is now possible to make conclusions according to the established hypotheses (see section 5.1.1). The results of the used statistical tests (see section 5.4.4) is depicted in the following tables:

	SCC	JDK	Tink
Shapiro	0.08722	0.8492	0.2402
Distribution	normal	normal	normal

Table 5.8: Shapiro test for Security question scores

	SCC/JDK	SCC/Tink
T-test, p-value	0.2263	0.2308

Table 5.9: T-test for Security question scores

Security bugs

The actual security was evaluated by looking at the made security bugs in the implementations of the programming tasks. For this, the static code analysis tool Xanitizer was used. With the help of Xanitizer only security bugs in the implementations with JDK could be found. There were no security bugs in the SCCI and Tink implementations found. Like already done in the pilot study security bugs resulting from missing logging in form of using *error.printStackTrace()* while catching occurring exceptions were omitted because this usage cannot be related to the functionality of the used library itself. The main bug categories Xanitizer could find in the JDK implementations were:

"Cipher with no integrity", "Weak hash function", "Insecure mode". These are all bugs that occur because of insecure parameter choices. Only two participants had no security bugs. Inside 10 implementations two bugs could be found and one participant had four security bugs.

By considering the number of found security bugs it is now possible to make conclusions according to the established hypotheses (see section 5.1.1). In the following, the results of the used statistical tests (see section 5.4.4) are shown:

	JDK
Shapiro	0.0003129
Distribution	not normal

Table 5.10: Shapiro test for security bugs

	SCC/JDK
Wilcoxon, p-value	**0.001586**

Table 5.11: Wilcoxon test for security bugs

For the comparison of the SCCI and Tink, no statistical test could be calculated because both had an equal security bug number of zero.

Interpretation

The H_0 hypothesis in this case is:
The Secure Crypto Config Interface is equally or less secure than JDK/Google Tink.

1) H_{AJS}: The Secure Crypto Config Interface is more secure than JDK

   Perceived security
   With a p-value of 0.2263 (> 0.05) the data suggest **not** to reject H_0. As a result, it is not possible to make any further assumptions about H_{AJS}.

   Security bugs
   With a p-value of 0.001586 (< 0.05) the data suggest to reject H_0 in favour of H_{ATS}. As a result, it can be said that the SCCI is more secure than JDK.

2) H_{ATS}: The Secure Crypto Config Interface is more secure than Google Tink

   Perceived security
   With a p-value of 0.2308 (> 0.05) the data suggest **not** to reject H_0. As a result, it is not possible to make any further assumptions about H_{ATS}.

   Security bugs
   Because no security bugs could be found either in the implementations of JDK or in the ones of Tink, the data suggest **not** to reject H_0. As a result, it is not possible to make any further assumptions about H_{ATS}.

5.6.5 Maintainability

The perceived maintainability was determined based on the questions of the questionnaire (see section A.4.4, questions 3-5). The following figure shows the boxplots for all libraries and the determined perceived maintainability score of each participant:

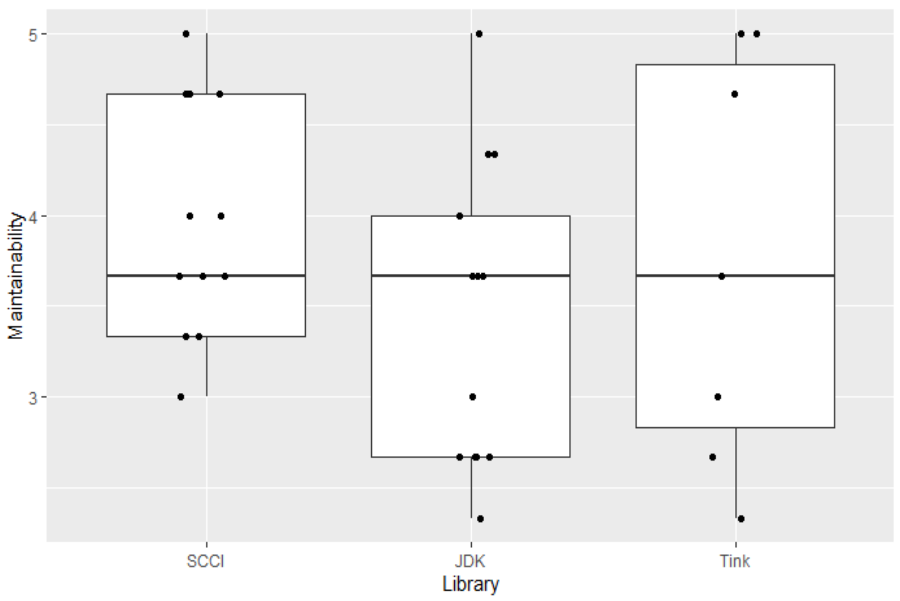


Figure 5.8: Boxplot for each cryptographic library and its Maintainability question scores

By looking at the boxplots it can be seen that the values in all libraries are widely scattered, there seems to be no huge difference in the perception for the maintainability aspect of the different libraries. All libraries are perceived as equally maintainable as the mean values with 3.92, 3.44 and 3.76 are very close together. Also, the median of 3.66 is equal for all libraries. There seems to be no tendency for a specific library for this aspect. The highest given values for the SCCI, JDK and Tink are also equal with 5.0. The minimum values are 2.0, 2.33 and 2.33. The previously mentioned values are summarized in the following table:

	SCCI	JDK	Tink
Minima	3.0	2.33	2.33
Maxima	5.0	5.0	5.0
Mean	3.92	3.44	3.76
Median	3.66	3.66	3.66

Table 5.12: Minima, Maxima and mean perceived maintainability score for each library

These equal values can probably be attributed to the fact that the participants may not know too much about the internal processing of the SCCI and therefore cannot know the beneficial points of the SCCI with respect to the maintainability. Because of this, they may not realize e.g. that they do not have to change used algorithms if the security standard changes in the usage of SCCI in comparison to JDK or Tink.

With the determined scores, it is now possible to make conclusions according to the established hypotheses (see section 5.1.1). The results of the used statistical tests (see section 5.4.4) is shown in the following:

	SCC	JDK	Tink
Shapiro	0.1949	0.2216	0.2217
Distribution	normal	normal	normal

Table 5.13: Shapiro for Maintainability question scores

	SCC/JDK	SCC/Tink
T-test, p-value	0.05409	0.3686

Table 5.14: T-test for Maintainability question scores

By looking at the number of code smells found by SonarQube only those from the project itself were found. No code smells inside the implementation from the participants could be found in the implementations of the SCCI and JDK. Therefore, the maintainability score (rated with A) and the technical debt was the same for all participants for these two libraries. This could be concluded to the fact that the code created by the participants was too short to see any differences in the maintainability with SonarQube. In the case of Tink besides the project-related code smells also some additional ones were found in the code created by the participants. All of the participants implemented the code with already as *deprecated* marked methods for their implementation. This fact is shown as a code smell inside SonarQube. However, this number of found code smells seems to be too small to get a worse overall maintainability score than in the cases of the SCCI and Tink. The overall maintainability score was with A still rated the same as in the implementations of SCCI and JDK despite the higher number of code smells. As a result, it can be concluded that the maintainability of all libraries by looking at the SonarQube indicators is equally.

Interpretation

The H_0 hypothesis in this case is:
The Secure Crypto Config Interface is equally or less maintainable than JDK/Google Tink.

1) H_{AJM}: The Secure Crypto Config Interface is more maintainable than JDK

   With a p-value of 0.05409 (> 0.05) the data suggest **not** to reject H_0. As a result, it is not possible to make any further assumptions about H_{AJM}.

2) H_{ATM}: The Secure Crypto Config Interface is more maintainable than Google Tink

   With a p-value of 0.3686 (> 0.05) the data suggest **not** to reject H_0. As a result, it is not possible to make any further assumptions about H_{ATM}.

5.7 Conclusion

As additional feedback, only two additional comments were given by the participants which pointed out that there was some inconsistency inside the SCCI documentation. There was one method where the parameters inside a code example were in the wrong order. This point was corrected afterward. However, I do not think that this error had a major impact on the study results. Even if the participants have chosen the wrong order of parameters according to the documentation, the used IDE will show such errors automatically during programming and the automatic error correction can be used. In this way, the error correction should be quickly possible and should not have had a major impact on the aspects under consideration. It is also possible that the participants have worked with the automatic completion of the IDE, which already displays and completes the method with the correct parameter order.

All in all, it can be concluded that the SCCI is more usable than JDK and Google Tink. By looking at the security aspect it can only be validated that the SCCI is more secure than JDK when looking at the actually found security bugs. Unfortunately by comparing the SCCI and Tink about their security no further assumptions could be made either in the perceived or the actual security. Also, the participants seem to see no big difference in the maintainability of the different libraries as the corresponding data suggests not to reject the corresponding Null-Hypotheses. All established hypotheses and whether their results were significant (hypothesis can be validated) or not (no further assumptions about the hypothesis possible) are summarised in table 5.15.

Hypotheses	Description	result
H_{AJU}	SCCI is more usable than JDK	✓ significant
H_{ATU}	SCCI is more usable than Tink	✓ significant
H_{AJS}	SCCI is more secure than JDK	× perceived: not significant ✓ bugs: significant
H_{ATS}	SCCI is more secure than Tink	× perceived: not significant × bugs: not significant
H_{AJM}	SCCI is more maintainable than JDK	× not significant
H_{ATM}	SCCI is more maintainable than Tink	× not significant

Table 5.15: Summary of hypotheses results

6 Summary and Outlook

6.1 Summary

In the course of the master thesis, the first step was to write an RFC draft which contributes to the standardization of the SCC. This created RFC draft covers all SCC specific processes and details. Since this draft has already been submitted, some feedback could already be received. Further, the SCC was also accepted as a topic in the 109th IETF meeting.

In the next step, a SCCI in Java was implemented to show the practical usage of the SCC. First of all, the usage of the SCCI solves the problems of insecure parameter choices by using secure parameter sets for different security levels defined by cryptographic specialized institutions. Furthermore, the implementation makes use of COSE to ensure up- and downward compatibility. This guarantees to stay compatible also with previous configurations. In addition, by using SCC JSON files created by conducting a frequently repeated and standardized process to provide currently secure cryptographic parameters, it is possible to solve the problem of insecure default configurations of common existing cryptographic libraries.

The last main part of the thesis was the evaluation of the implemented SCCI. In this course, a pilot and a main study were conducted to investigate the usability, security and maintainability of the interface in contrast to the default library of JDK and Google Tink. For these studies, a survey including a questionnaire and a programming task was created. Based on the collected data, established hypotheses were investigated by doing different statistical tests. The results of these studies showed that the SCCI is more usable than JDK and Google Tink. Also, by looking at the actual security bugs the SCCI was more secure than JDK. However, it was not possible to determine whether the security of the SCCI was better than in Google Tink. Unfortunately, no further assumptions about the hypotheses corresponding to the maintainability aspect could be made as the results based on the collected data was not significant.

All in all, it can be said that the SCCI seems to be a good alternative especially in comparison to the default JDK library. In particular, the usability of the SCCI based on the evaluation data seems to be significantly better than in the other two compared libraries.

6.2 Outlook

The next steps for the SCC should be first, to publish the SCCI as Maven artifact such that it will be possible for programmers to use the SCCI for their implementations easily.

Also, a cooperation between the SCCI and the implementors of the used COSE project is needed. With this, the supported algorithms inside COSE and the SCCI could be synchronized and no further unnecessary programming effort for using COSE inside the SCCI implementation is needed. Furthermore, this aspect will make it unnecessary to import an extended version of the COSE project when using the SCCI.

In addition, further iterations of the created RFC will be needed to get a detailed and approved standardization for the SCC. Furthermore, the commitment of standardization entities to work on the SCC will be needed in the future such that the general process of the SCC can be lived. To be able to do this, it will also be necessary to spread the knowledge about the SCC and make it known. Promoting the concept of the SCC will continue to be an important topic in the future such that more people get aware of the advantages of the SCC and participate in its further development.

The general goal for the SCC will be that in the future all common cryptographic libraries implement the SCCI themselves. As a result, it will be possible for the SCC to become the standard for cryptographic code. In this way, all programmers that need to implement secure code can benefit from the advantages of the SCC one day.

A Appendix

A.1 JSON Secure Crypto Config

A.1.1 Example of the Secure Crypto Config

```
{
    "PolicyName" : "SCC_SecurityLevel_5",
    "Publisher" : [
        {
            "name" : "Crypto Forum Research Group",
            "URL" : "https://irtf.org/cfrg"
        },
        {
            "name" : "BSI",
            "URL" : "https://BSI"
        }
    ],
    "SecurityLevel" : "5",
    "Version" : "2020-0",
    "PolicyIssueDate" : "2020-04-20",
    "Expiry" : "2023-04-21",
    "Usage" : {
        "SymmetricEncryption" : [
            "AES_GCM_256_96",
            "AES_GCM_192_96"
        ],
        "AsymmetricEncryption" : [
            "RSA_SHA_512",
            "RSA_SHA_256"
        ],
        "Hashing" : [
            "SHA3_512",
            "SHA_512"
        ],
        "Signing" : [
            "ECDSA_512",
            "ECDSA_384"
        ],
        "PasswordHashing" : [
            "PBKDF_SHA_512",
            "SHA_512_64"
        ]
    }
}
```

Figure A.1: Example for format of the SCC file

A.2 Java Interface using Secure Crypto Config

In the following all interfaces and classes from the SCC API can be seen:

```java

package org.securecryptoconfig;

import java.nio.charset.Charset;
import java.security.InvalidKeyException;
import org.securecryptoconfig.SCCKey.KeyType;
import org.securecryptoconfig.SCCKey.KeyUseCase;

import COSE.CoseException;

public abstract interface SecureCryptoConfigInterface {

//Symmetric
public AbstractSCCCiphertext encryptSymmetric(AbstractSCCKey key, PlaintextContainerInterface plaintext)
    throws SCCException;

public AbstractSCCCiphertext encryptSymmetric(AbstractSCCKey key, byte[] plaintext)
    throws SCCException;

public AbstractSCCCiphertext reEncryptSymmetric(AbstractSCCKey key, AbstractSCCCiphertext ciphertext)
    throws SCCException;

public PlaintextContainerInterface decryptSymmetric(AbstractSCCKey key, AbstractSCCCiphertext sccciphertext)
    throws SCCException;

// Asymmetric
public AbstractSCCCiphertext encryptAsymmetric(AbstractSCCKey key, PlaintextContainerInterface plaintext)
    throws SCCException;

public AbstractSCCCiphertext encryptAsymmetric(AbstractSCCKey key, byte[] plaintext)
    throws SCCException;

public AbstractSCCCiphertext reEncryptAsymmetric(AbstractSCCKey key, AbstractSCCCiphertext ciphertext)
    throws SCCException;

public PlaintextContainerInterface decryptAsymmetric(AbstractSCCKey key, AbstractSCCCiphertext ciphertext)
    throws SCCException;

// Hashing
public AbstractSCCHash hash(PlaintextContainerInterface plaintext)
    throws SCCException;

public AbstractSCCHash hash(byte[] plaintext)
    throws SCCException;

public AbstractSCCHash updateHash(PlaintextContainerInterface plaintext, AbstractSCCHash hash)
    throws SCCException;

public AbstractSCCHash updateHash(byte[] plaintext, AbstractSCCHash hash)
    throws SCCException;

public boolean validateHash(PlaintextContainerInterface plaintext, AbstractSCCHash hash)
    throws SCCException;

public boolean validateHash(byte[] plaintext, AbstractSCCHash hash)
    throws SCCException;

// Digital Signature
public AbstractSCCSignature sign(AbstractSCCKey key, PlaintextContainerInterface plaintext)
    throws SCCException;

public AbstractSCCSignature sign(AbstractSCCKey key, byte[] plaintext)
    throws SCCException;

public AbstractSCCSignature updateSignature(AbstractSCCKey key, PlaintextContainerInterface
plaintext)
    throws SCCException;

public AbstractSCCSignature updateSignature(AbstractSCCKey key, byte[] plaintext)
    throws SCCException;

public boolean validateSignature(AbstractSCCKey key, AbstractSCCSignature signature)
    throws SCCException;

public boolean validateSignature(AbstractSCCKey key, byte[] signature)
    throws SCCException;

// Password Hashing
public AbstractSCCPasswordHash passwordHash(PlaintextContainerInterface password)
    throws SCCException;

public AbstractSCCPasswordHash passwordHash(byte[] password)     throws SCCException;

public boolean validatePasswordHash(PlaintextContainerInterface password,
AbstractSCCPasswordHash passwordhash)
    throws SCCException;

public boolean validatePasswordHash(byte[] password, AbstractSCCPasswordHash passwordhash)
    throws SCCException;
}
```

Listing A.1: SecureCryptoConfigInterface

```java

abstract interface PlaintextContainerInterface {

public abstract byte[] toBytes();

public abstract String toString(Charset c);

@Override
public abstract String toString();

public abstract boolean validateHash(AbstractSCCHash hash)
    throws SCCException;

public abstract boolean validatePasswordHash(AbstractSCCPasswordHash passwordHash)
    throws SCCException;

public abstract AbstractSCCCiphertext encryptSymmetric(AbstractSCCKey key)
    throws SCCException;

public abstract AbstractSCCCiphertext encryptAsymmetric(AbstractSCCKey key)
    throws SCCException;

public abstract AbstractSCCSignature sign(AbstractSCCKey key)
    throws SCCException;

public abstract boolean validateSignature (AbstractSCCSignature signature, AbstractSCCKey key)
    throws SCCException;

public abstract AbstractSCCHash hash()
    throws SCCException;

public abstract AbstractSCCPasswordHash passwordHash()
    throws SCCException;
}
```

Listing A.2: PlaintextContainerInterface

```java

abstract class AbstractSCCCiphertext {

byte[] msg;

protected AbstractSCCCiphertext(byte[] msg) {
this.msg = msg;
}

public abstract byte[] toBytes();

@Override
public abstract String toString();

public abstract PlaintextContainerInterface decryptSymmetric(AbstractSCCKey key)
    throws SCCException;

public abstract PlaintextContainerInterface decryptAsymmetric(AbstractSCCKey key)
    throws SCCException;

public abstract AbstractSCCCiphertext reEncryptSymmetric(AbstractSCCKey key)
    throws SCCException;


public abstract AbstractSCCCiphertext reEncryptAsymmetric(AbstractSCCKey key)
    throws SCCException;
}
```

Listing A.3: AbstractSCCCiphertext

```

abstract class AbstractSCCKey {

KeyType type;
byte[] privateKey, publicKey;
String algorithm;

protected AbstractSCCKey(KeyType type, byte[] publicKey, byte[] privateKey, String algorithm)
{
this.type = type;
this.publicKey = publicKey;
this.privateKey = privateKey;
this.algorithm = algorithm;
}

public abstract byte[] toBytes()
    throws SCCException;

public abstract byte[] getPublicKeyBytes()
    throws SCCException;

public abstract byte[] getPrivateKeyBytes()
    throws SCCException;

public abstract KeyType getKeyType();

public abstract String getAlgorithm();
}
```

Listing A.4: AbstractSCCKey

```
abstract class AbstractSCCHash {

byte[] hashMsg;

protected AbstractSCCHash(byte[] hashMsg) {

this.hashMsg = hashMsg;
}

public abstract byte[] toBytes();

@Override
public abstract String toString();

public abstract boolean validateHash(PlaintextContainerInterface plaintext)
    throws SCCException;

public abstract boolean validateHash(byte[] plaintext)
    throws SCCException;

public abstract AbstractSCCHash updateHash(PlaintextContainerInterface plaintext)
    throws SCCException;

public abstract AbstractSCCHash updateHash(byte[] plaintext)
    throws SCCException;
}
```

Listing A.5: AbstractSCCHash

```
abstract class AbstractSCCPasswordHash {

byte[] hashMsg;

protected AbstractSCCPasswordHash(byte[] hashMsg) {
this.hashMsg = hashMsg;
}

public abstract byte[] toBytes();

@Override
public abstract String toString();

public abstract boolean validatePasswordHash(PlaintextContainerInterface password)
    throws SCCException;

public abstract boolean validatePasswordHash(byte[] password)
    throws SCCException;
}
```

Listing A.6: AbstractSCCPasswordHash

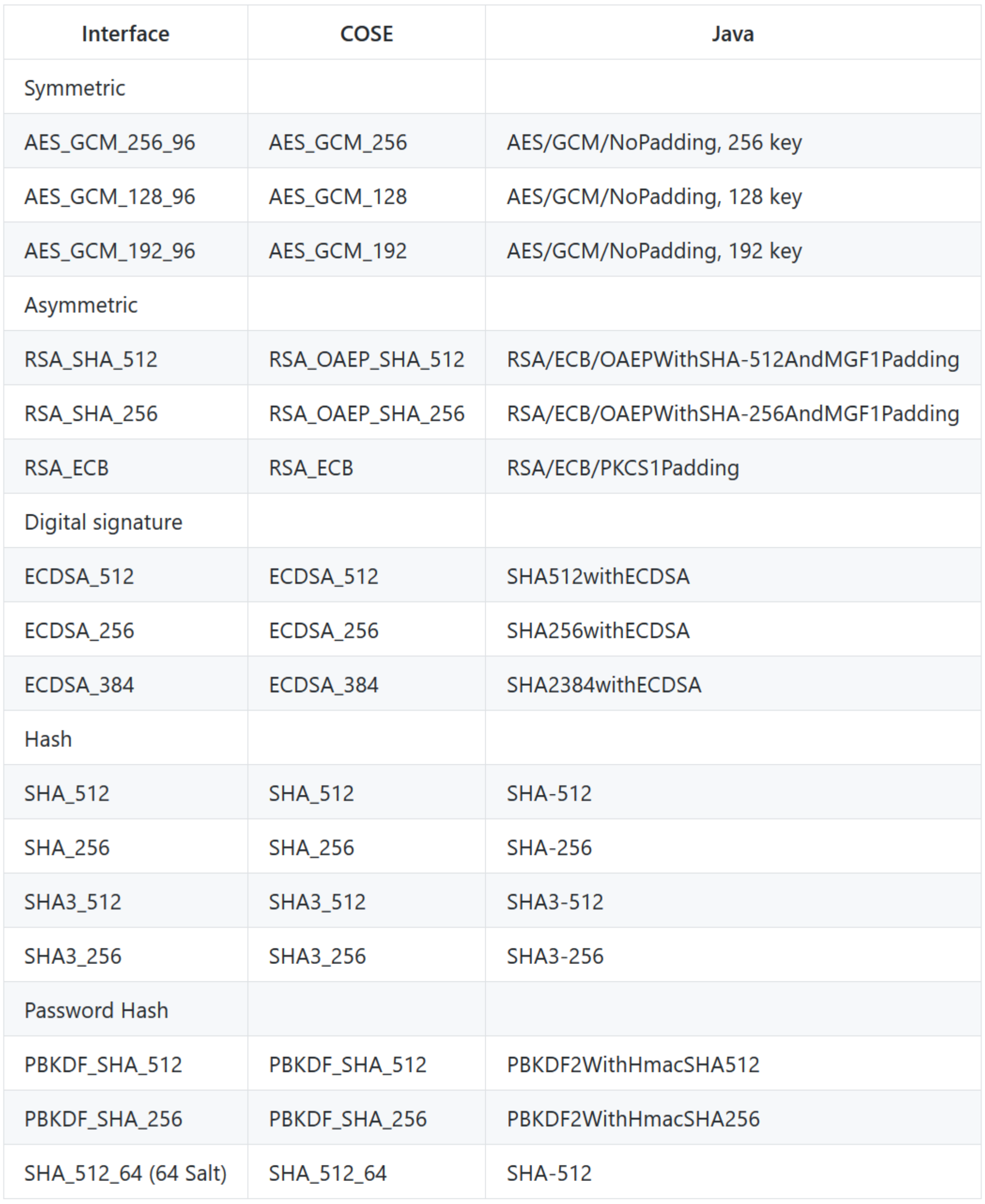
```
abstract class AbstractSCCSignature {

byte[] signatureMsg;

protected AbstractSCCSignature(byte[] signatureMasg) {
this.signatureMsg = signatureMasg;
}

public abstract byte[] toBytes();

```

```java
@Override
public abstract String toString();

public abstract boolean validateSignature(AbstractSCCKey key)
    throws SCCException;

public abstract AbstractSCCSignature updateSignature(PlaintextContainerInterface plaintext,
AbstractSCCKey key)
    throws SCCException;
}
```

Listing A.7: AbstractSCCSignature

A.3 Supported algorithms

In the following all supported algorithms of the SCCI can be seen:



Interface	COSE	Java
Symmetric		
AES_GCM_256_96	AES_GCM_256	AES/GCM/NoPadding, 256 key
AES_GCM_128_96	AES_GCM_128	AES/GCM/NoPadding, 128 key
AES_GCM_192_96	AES_GCM_192	AES/GCM/NoPadding, 192 key
Asymmetric		
RSA_SHA_512	RSA_OAEP_SHA_512	RSA/ECB/OAEPWithSHA-512AndMGF1Padding
RSA_SHA_256	RSA_OAEP_SHA_256	RSA/ECB/OAEPWithSHA-256AndMGF1Padding
RSA_ECB	RSA_ECB	RSA/ECB/PKCS1Padding
Digital signature		
ECDSA_512	ECDSA_512	SHA512withECDSA
ECDSA_256	ECDSA_256	SHA256withECDSA
ECDSA_384	ECDSA_384	SHA2384withECDSA
Hash		
SHA_512	SHA_512	SHA-512
SHA_256	SHA_256	SHA-256
SHA3_512	SHA3_512	SHA3-512
SHA3_256	SHA3_256	SHA3-256
Password Hash		
PBKDF_SHA_512	PBKDF_SHA_512	PBKDF2WithHmacSHA512
PBKDF_SHA_256	PBKDF_SHA_256	PBKDF2WithHmacSHA256
SHA_512_64 (64 Salt)	SHA_512_64	SHA-512

Figure A.2: Supported algorithm identifiers, the corresponding COSE identifiers and the realization in Java

A.4 Survey

A.4.1 Programming task

Scenario - Read all before starting
We provide a small application that simulates the interaction of a client with a stock-server (see figures below). The client can either send sell (SellStock) or buy (BuyStock) stock messages to the server. Also, it can request that the server sends it back all previously made orders (GetOrders). If a client communicates with the server it must be guaranteed that these messages are not manipulated. For this reason, the order is signed by the client-side with a private key. The server receives the corresponding public key of the client in advance and stores it. With the public key of the client, the incoming order and signature can now be validated with the public key on the server-side. If the signature is correct, the client's order should be stored symmetrically encrypted at a specific location. This ensures that only authorized persons can read the stored orders of the client. As clients can also request all their previously sent orders from the server it must also be possible to decrypt the stored data correctly.

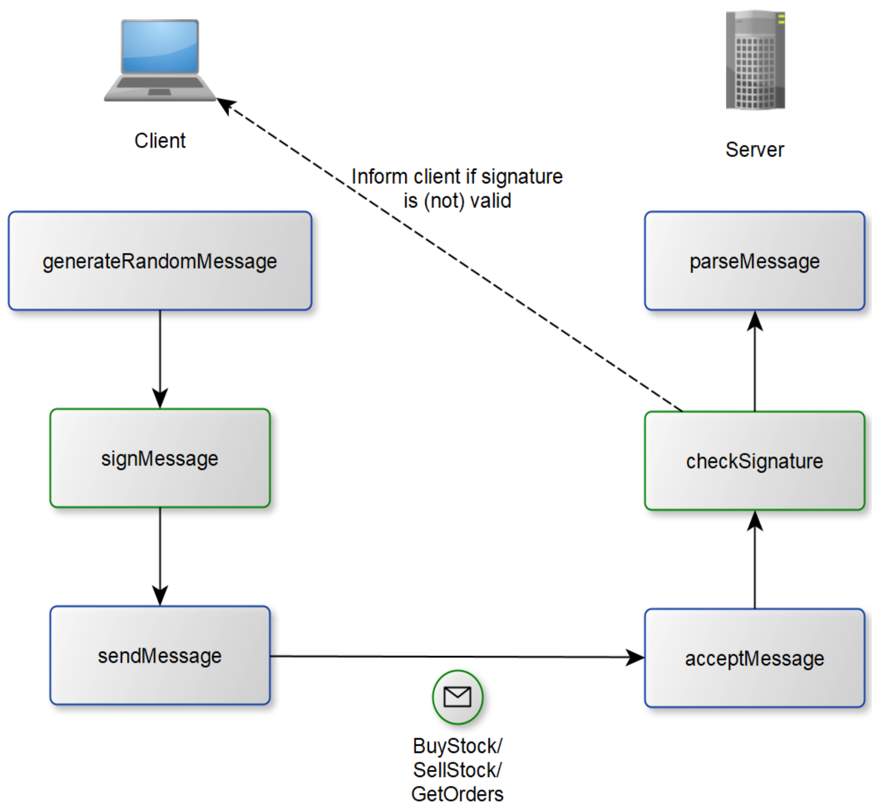


Figure A.3: Overview of the used methods for signing/validating messages at the client and server

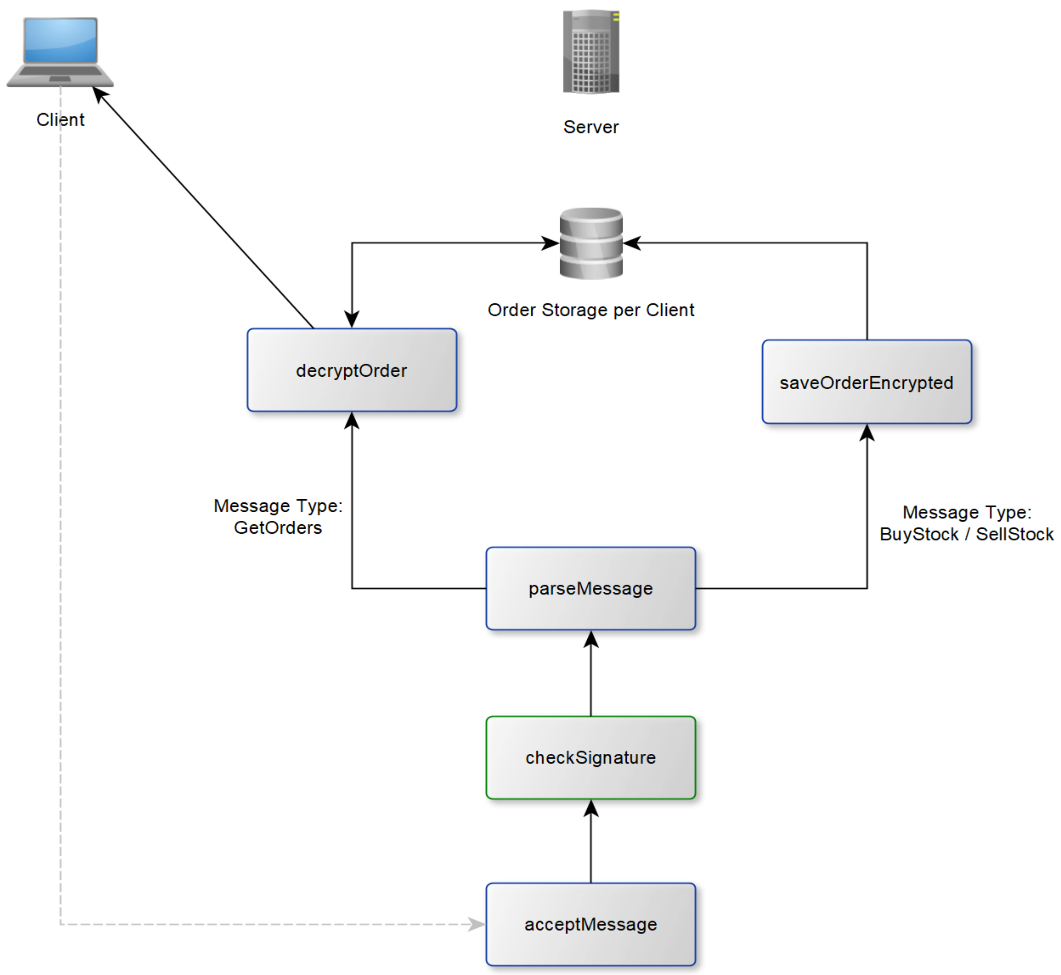


Figure A.4: Overview of only the server-side for encryption/decryption of order messages after validating the message signature

Java project
Given is a Java project. Several classes exist in it: “AppMain”, “Client”, “Server”, “Message”, “SignedMessage”. These classes implement only the functionalities relevant to the study. Further processing and the concrete storage of the encrypted data were omitted for simplification. One client will be created which sends automatically generated messages of different types (SellStock, BuyStock, GetOrders) to the server. The “MainApp.java” can be executed and shows both the messages sent to the server as well as the response of the server back to the client.

Procedure (Browser IDE based on GitPod or download ZIP file)

- If you have a “Gitlab”, “Github” or “Bitbucket” account you can perform the following tasks in this prepared snapshot of a GitPod. When you finished the tasks, create a snapshot of your solution. This can be generated using the round button in the upper right corner. There, select “Share Workspace Snapshot”. Copy that link and insert it as an answer to the following task description.

- Alternatively, fork the GitHub project or download the code as ZIP file and import it into a development environment you prefer (prerequisite: at least Java-SE 10). Upload the changed java files (Client.java, Server.java) with your solutions in the following "upload" field.

Your Tasks - Read all before starting
There is no time limit and remember we are not testing you but the library. Feel free to use any help you might need (e.g. web searches). Before starting, please read all tasks carefully.

When using GitPod open the following Link: *GitPod Link for specific library*

For the following tasks, use only functionalities from the *Library to be used*. You must not use any other cryptographic libraries.

Documentation for this library can be found at the following link: *Link to documentation of specific library*

Handle possible occurring exceptions in the method in which they appear.

1) Start the AppMain.java.
For running the program in GitPod make a right-click on the corresponding class and select "Run" in the context menu. For stopping a running execution type "Strg + C" into the console.

2) In the class "Client" you find the method "signMessage(...)": Create a digital signature for the given (Order-)Message of the client with the given key and return it. The corresponding place to insert your implementation contains a "//TODO" hint.

You can check the functionality by running "AppMain.java". If you see the message: *client 0: signature is (base64 encoded): xxxxxx* a signature was created (null otherwise)

3) In the class "Server" you will find the method "checkSignature(...)": Validate the digital signature of a client's SignedMessage with the given client's public key. The corresponding place to insert your implementation contains a "//TODO" hint.

You can check the functionality by running "AppMain.java". If you see the message: *server: message signature is valid* you correctly validated the message

4) In the class "Server" you will find the method "saveOrderEncrypted(...)": Perform a symmetric encryption of the client's order with the given key. The corresponding place to insert your implementation contains a "//TODO" hint.

You can check the functionality by running "AppMain.java". If you see the message: *server: encryptedOrder is (base64 encoded): xxxx* you were able to encrypt the message (null otherwise)

5) In the class "Server" you will find the method "decryptOrder(...)": Perform a symmetric decryption of the stored encrypted client's order with the given key. The corresponding place to insert your implementation contains a "//TODO" hint.

You can check the functionality by running "AppMain.java". Compare the content of server: *GetOrders, client 0: result from server: {"senderType":"Server","messageType":"ServerSendOrders","messageParameters": xxxx }* with the content of the previously sent client buy/sell messages. If they are equal the decryption was successful

If you are finished with all tasks, please create a GitPod snapshot and share the URL/Link with us as an answer to this question. This can be generated using the round button in the upper right corner. There, select “Share Workspace Snapshot”. Copy that link and insert it as an answer to the following task description.

If you did not use GitPod, please upload the changed java files (Server.java, Client.java) of your solution.

At next, the participant could find one text-field to insert its GitPod link or a question in which he could upload his solution files.

A.4.2 System Usability Scale

How much do you agree with the following statements? (Five-point Likert scale from Strongly Agree to Strongly Disagree)

- I think that I would like to use this system frequently.
- I found the system unnecessarily complex.
- I thought the system was easy to use.
- I think that I would need the support of a technical person to be able to use this system.
- I found the various functions in this system were well integrated.
- I thought there was too much inconsistency in this system.
- I would imagine that most people would learn to use this system very quickly.
- I found the system very cumbersome to use.
- I felt very confident using the system.
- I needed to learn a lot of things before I could get going with this system.

A.4.3 Cognitive Dimension Questionnaire

Following questions from Wijayarathna et al. [WAS17] were considered:

- Do you feel that you had to understand the underlying implementation to be able to use the API?
- Did you had to learn about different components exposed by API before starting to do anything useful related to your task? What are the components that you had to learn?
- Do you think, if you had previous knowledge of any specific area, it would have been easier to use the API? What are they?
- Does the amount of code required for each subtask in this scenario seem just about right, too much, or too little? Why?
- How easy is it to stop in the middle of the scenario and check the progress of work so far?

- When you are working with the API, can you work on your programming task in any order you like, or does the system force you to think ahead and make certain decisions first?
- Were you able to find enough information to distinguish between different methods and classes while you work on your programming task? If not, what is the information you think is missing?
- Did the API (including its documentation) provide enough information about the security relevant specifics related to the task you completed? What is the information that was missing or you had to find by referring to external sources?
- When you need to make changes to previous work, how easy is it to make the changes? Why?
- Were there different parts of the API that mean similar things? Is the similarity clear from the way they appear? Please give examples.
- Are there places where some things ought to be similar, but the API makes them different? What are they?
- When using the API, is it easy to know what classes and methods of the API to use when writing code?
- Did the types exposed by the API map directly onto the types and concepts you expected? If not, please mention the types you expected and how it was supported in the API?
- Have you came up with incidents where you incorrectly used the API and then identified the correct way of doing that? Did API give any help to identify that you used the API incorrectly? If there any similar incidents, please explain?
- Did the API provided any guidance on how to test your application?

A.4.4 Questions for maintainability and security aspects

Questions to get additional feedback for the evaluation of the maintainability and security aspects.

How much do you agree with the following statements? (Five-point Likert scale from Strongly Agree to Strongly Disagree)

1) I think it is easy to use a secure cryptography algorithm in the application with this library.
2) I think it is easy to keep the application code up-to-date with new releases of cryptography algorithms.
3) I think it is easy to test the cryptography related code in the application.
4) I think it is easy to modify the cryptography related code in the application (e.g. if security standards changed).
5) I think it is easy to structure the cryptography related code in the application.

A.4.5 Final general questions

How much do you agree with the following statements? (Five-point Likert scale from Strongly Agree to Strongly Disagree)

- I gave my best during the study.
- I had difficulties understanding the task.
- I had difficulties understanding the purpose of the application.
- I had difficulties working with the provided development environment.
- I participated in this study to win the raffle.
- I participated in this study because the topic is interesting.

- How long have you been programming in Java?
  (Choose one of the following answers: Less than 1 year, 1-2 years, 2-5 years, More than five years, No answer)
- How long have you been coding in general?
  (Choose one of the following answers: Less than 1 year, 1-2 years, 2-5 years, More than five years, No answer)
- How did you learn to code?
  (Check all that apply: Self-taught, Online class, College, On-the-job training, Coding Camp)
- Is programming your primary job?
  (Choose one of the following answers: Yes, No, No answer)
- Do you have an IT-security background?
  (Choose one of the following answers: Yes, No, No answer)
- Please tell us your highest degree of education.
  (Choose one of the following answers: Abitur/higher education entrance qualification, Bachelor, Master, Phd/Doctorate, Other, No answer)
- How old are you?
- What is your occupation?
- Do you have any further comments regarding the study?

Bibliography

[ABF+17] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, C. Stransky. "*Comparing the usability of cryptographic apis*". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 154–171 (cit. on pp. 14, 15, 43).

[AD99] W. Alma, T. J. D. *Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0*. 1999. DOI: 10.1.1.152.6298 (cit. on p. 16).

[BDKJ20] A. Biryukov, D. Dinu, D. Khovratovich, S. Josefsson, eds. *The memory-hard Argon2 password hash and proof-of-work function*. Mar. 2020. URL: https://tools.ietf.org/html/draft-irtf-cfrg-argon2-10#section-3.1 (cit. on p. 12).

[Car20] P. Carbonnelle, ed. *PYPL PopularitY of Programming Language*. 2020. URL: http://pypl.github.io/PYPL.html (cit. on p. 38).

[Cla04] S. Clarke. *Measuring api usability*. Ed. by 2. Doctor Dobbs Journal. 2004 (cit. on pp. 14, 15).

[EBFK13] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel. "*An Empirical Study of Cryptographic Misusein Android Applications*". In: *Security (CCS 2013). ACM, 2013*. 2013 (cit. on pp. 12, 14, 16).

[ET96] Z. M. Ellen, S. R. T. *User-centered Security*. 1996. DOI: 10.1.1.136.7754 (cit. on p. 16).

[GIJ+] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, V. Shmatikov. *The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software* (cit. on p. 14).

[Git16] GitHut.info. *GitHut: A Small Place to discover languages in GitHub*. 2016. URL: http://githut.info (cit. on p. 38).

[GS16] M. Green, M. Smith. "*Developers are not the enemy!: The need for usable security apis*". In: *IEEE Security & Privacy* 14.5 (2016), pp. 40–46 (cit. on p. 16).

[Gue10] S. Gueron. *Intel Advanced Encryption Standard (AES) Instruction Set White Paper*. Ed. by Intel. May 2010 (cit. on p. 23).

[HGK+19] M. Hazhirpasand, M. Ghafari, S. Krüger, E. Bodden, O. Nierstrasz. "*The Impact of Developer Experience in Using Java Cryptography*". In: (Aug. 5, 2019). arXiv: 1908.01489v1 [cs.CR] (cit. on pp. 15, 16).

[ISO25010] ISO. *ISO/IEC 25010*. Ed. by I. Standards. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (cit. on p. 46).

[Klu17] B. Klug. "*An overview of the System Usability Scale in library website and system usability testing*". In: *Weave: Journal of Library User Experience* 1.6 (2017) (cit. on pp. 51, 52).

[KNR+17] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, et al. "*CogniCrypt: supporting developers in using cryptography*". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 931–936 (cit. on p. 15).

[KSA+17] S. Krüger, J. Späth, K. Ali, E. Bodden, M. Mezini. "*CrySL: Validating Correct Usage of Cryptographic APIs*". In: (Oct. 2, 2017). arXiv: 1710.00564v1 [cs.SE] (cit. on pp. 15, 16).

[MT20] K. Mindermann, L. Teis. *Secure Crypto Config*. Internet-Draft draft-kaimindermann-securecryptoconfig-00. Work in Progress. Internet Engineering Task Force, Oct. 2020. 40 pp. URL: https://datatracker.ietf.org/doc/html/draft-kaimindermann-securecryptoconfig-00 (cit. on pp. 13, 17).

[MW18] K. Mindermann, S. Wagner. *Usability and Security Effects of Code Examples on Crypto APIs - CryptoExamples: A platform for free, minimal, complete and secure crypto examples*. July 2018 (cit. on p. 43).

[MW19a] K. Mindermann, S. Wagner. *Secure Crypto Config. Towards a central, distributed and secure default cryptography parameter*. 30th Crypto Day presentation, 2019 (cit. on p. 14).

[MW19b] K. Mindermann, S. Wagner. "*Towards a central, distributed and secure default cryptography parameter set*". In: *crypto day matters 30* (2019) (cit. on pp. 14, 17, 18).

[NKMB16] S. Nadi, S. Krüger, M. Mezini, E. Bodden. "*Jumping Through Hoops: Why do Java Developers struggle With Cryptography APIs?*" In: *2016 IEEE/ACM 38th IEEE International Conference on Software Engineering*. 2016 (cit. on p. 15).

[RFC5652] R. Housley. *Cryptographic Message Syntax (CMS)*. Tech. rep. 5652. Internet Requests for Comments, Sept. 2009. DOI: 10.17487/RFC5652. URL: https://www.rfc-editor.org/info/rfc5652 (cit. on p. 41).

[RFC5698] T. Kunz, S. Okunick, U. Pordesch. *Data Structure for the Security Suitability of Cryptographic Algorithms (DSSC)*. Tech. rep. 5698. Internet Requests for Comments, Nov. 2009. DOI: 10.17487/RFC5698. URL: https://www.rfc-editor.org/info/rfc5698 (cit. on p. 27).

[RFC7228] C. Bormann, M. Ersue, A. Keranen. *Terminology for Constrained-Node Networks*. Tech. rep. 7228. Internet Requests for Comments, May 2014. DOI: 10.17487/RFC7228. URL: https://www.rfc-editor.org/info/rfc7228 (cit. on p. 23).

[RFC8152] J. Schaad. *CBOR Object Signing and Encryption (COSE)*. Tech. rep. 8152. Internet Requests for Comments, July 2017. DOI: 10.17487/RFC8152. URL: https://www.rfc-editor.org/info/rfc8152 (cit. on pp. 13, 20, 24, 27, 41).

[Tei20a] L. Teis. "secureCryptoConfig/secureCryptoConfigInterface: Release 0.0.1-beta". In: (Dec. 2020). DOI: 10.5281/zenodo.4303983 (cit. on pp. 13, 38, 41, 43).

[Tei20b] L. Teis. "secureCryptoConfig/study-task-jdk: Release v0.0.1". In: (Dec. 2020). DOI: 10.5281/zenodo.4304002 (cit. on p. 50).

[Tei20c] L. Teis. "secureCryptoConfig/study-task-scc: Release v0.0.1". In: (Dec. 2020). DOI: 10.5281/zenodo.4303998 (cit. on p. 50).

[Tei20d] L. Teis. “secureCryptoConfig/study-task-tink: Release v0.0.1”. In: (Dec. 2020). DOI: 10.5281/zenodo.4304000 (cit. on p. 50).

[WA18a] C. Wijayarathna, N. A. G. Arachchilage. “*A methodology to Evaluate the Usability of Security APIs*”. In: *IEEE International Conference on Information and Automation for Sustainability (ICIAfS), 2019* (Oct. 11, 2018). arXiv: 1810.05100v1 [cs.CR] (cit. on pp. 14, 52).

[WA18b] C. Wijayarathna, N. A. G. Arachchilage. “*Why Johnny Can’t Store Passwords Securely? A Usability Evaluation of Bouncycastle Password Hashing*”. In: *22nd International Conference on Evaluation and Assessment in Software Engineering, 2018* (May 24, 2018). arXiv: 1805.09487v1 [cs.CR] (cit. on p. 16).

[WAS17] C. Wijayarathna, N. A. G. Arachchilage, J. Slay. “*A Generic Cognitive Dimensions Questionnaire to Evaluate the Usability of Security APIs*”. In: (Mar. 29, 2017). arXiv: 1703.09846v1 [cs.CR] (cit. on pp. 14, 51, 52, 54, 57, 81).

All links were last followed on December 06, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature